DERS KİTABI

Algoritmalar

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

Algoritmalar

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

Bu kitap, şu adreste yayımlanan ders notlarının basılı hâlidir:
`sercankulcu.github.io/teaching/algorithms/`

Sunumlar, çalışma soruları ve simülatörler o sayfadadır. Bölümlerin kodu `github.com/sercankulcu/algorithms-java` deposundadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 14 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Algoritma, bir problemi çözmek ya da bir hesabı yapmak için sonlu ve iyi tanımlanmış adımlar dizisidir. Bu kitap temel algoritma tasarım tekniklerini, algoritmaların doğruluğunu, zaman ve bellek karmaşıklığını ve farklı çözümler arasındaki ödünleşimleri inceler. Kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen Algoritmalar dersinin notlarından oluşur ve bölümleri dersteki sırayla izler. Örnekler Java ile yazılmıştır.

Kitap beş kısımdır. Kısımlar sırasıyla şunlardır: **Temeller** (Bölüm 1–3), **Çizge ve Dizgi Algoritmaları** (Bölüm 4–5), **Tasarım Paradigmaları** (Bölüm 6–8), **Zor Problemler** (Bölüm 9–11) ve **Diğer Hesaplama Modelleri** (Bölüm 12–14).

Bazı bölümler birden çok ders notundan oluşur. Bu bölümler kısa bir açılış sayfasıyla başlar; notlar ardından 4.1, 4.2 gibi numaralı alt bölümler olarak gelir.

Her bölüm aynı düzeni izler. Başta öğrenme çıktıları yer alır. Biçimsel tanımlar, teoremler ve çözülmüş örnekler adlandırılmış kutulardadır. *Sık yapılan hata* kutuları, öğrencilerin en sık düştüğü yanılgıları gösterir. Kod izlemeleri, bir programın ya da algoritmanın adım adım nasıl çalıştığını gösterir. Bölüm; bir özet, bir kavram listesi, cevaplarıyla birlikte *Kendini Dene* soruları, ileri okuma önerileri ve kaynakçayla kapanır. Kitabın sonundaki terim sözlüğü, her İngilizce terim için notlarda kullanılan tek Türkçe karşılığı verir.

Bölümlerin kodu derse eşlik eden `github.com/sercankulcu/algorithms-java` deposunda, her bölüm için ayrı bir klasördedir; klasörün adı bölümün başında yazılıdır. Sunumlar, çalışma soruları ve algoritmaları adım adım çalıştıran simülatörler ders sayfasındadır.

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · TEMELLER

1 Giriş ve Algoritma Karmaşıklığı 4

1.1 Giriş 5
1.2 Algoritma Karmaşıklığı 43

2 Sıralama Algoritmaları 91

2.1 Sıralama problemi ve değerlendirme ölçütleri 92
2.2 Kabarcık sıralaması 96
2.3 Eklemeli sıralama 102
2.4 Seçimli sıralama 106
2.5 Shell sıralaması 111
2.6 Birleştirme sıralaması 114
2.7 Hızlı sıralama 121
2.8 Heap sort 129
2.9 Doğrusal zamanlı sıralama 137
2.10 Timsort 145
2.11 Algoritmaların karşılaştırılması 147
2.12 Sık yapılan hatalar 149
2.13 Özet 150
2.14 Kavram listesi 151
2.15 Kendini Dene 151
2.16 İleri okuma 153
2.17 Kaynaklar 154

3 Arama Algoritmaları 155

3.1 Arama problemi 156
3.2 Doğrusal arama 158
3.3 İkili arama 165
3.4 Hash ile arama 175
3.5 Ara değer arama 185
3.6 Atlayarak arama 189
3.7 Üstel arama 192
3.8 Karşılaştırma ve seçim 196
3.9 Sık yapılan hatalar 198
3.10 Özet 199
3.11 Kavram listesi 200
3.12 Kendini Dene 200
3.13 İleri okuma 202
3.14 Kaynaklar 203

KISIM II · ÇİZGE VE DİZGİ ALGORİTMALARI

4 Çizge Algoritmaları **205**

4.1 Çizge Gezinme Algoritmaları 206
4.2 En Kısa Yol Algoritmaları 264
4.3 Minimum Kapsayan Ağaç Algoritmaları 318
4.4 Ağ Akışı 373

5 Dizgi Algoritmaları **436**

5.1 Dizgi Eşleştirme ve Sıkıştırma 437
5.2 İleri Dizgi Algoritmaları 495
5.3 Trie Veri Yapısı 560

KISIM III · TASARIM PARADİGMALARI

6 Böl ve Fethet Yaklaşımı **616**

6.1 Böl ve fethet paradigması 617
6.2 Yineleme bağıntıları 623
6.3 Temel örnekler 627
6.4 Dışbükey örtü ve Quickhull 633
6.5 En yakın nokta çifti 643
6.6 Karatsuba çarpımı 650
6.7 Strassen matris çarpımı 654
6.8 Değerlendirme 657
6.9 Sık yapılan hatalar 658
6.10 Özet 658
6.11 Kavram listesi 659
6.12 Kendini Dene 659
6.13 İleri okuma 661
6.14 Kaynaklar 661

7 Açgözlü Algoritmalar **662**

7.1 Açgözlü yaklaşım 663
7.2 Para üstü problemi 668
7.3 Etkinlik seçimi 673
7.4 Son tarihli iş sıralama 679
7.5 Kesirli sırt çantası 683
7.6 Minimum kapsayan ağaçta açgözlü seçim 687
7.7 Huffman kodlama 691
7.8 Diğer açgözlü algoritmalar 700
7.9 Karşılaştırma ve sık yapılan hatalar 701
7.10 Özet 702
7.11 Kavram listesi 703
7.12 Kendini Dene 703
7.13 İleri okuma 706
7.14 Kaynaklar 706

8 Dinamik Programlama 707

8.1 Temel kavramlar 708
8.2 Fibonacci: bellekleme ve tablolama 714
8.3 0/1 sırt çantası 719
8.4 Sınırsız sırt çantası 728
8.5 Alt küme toplamı 732
8.6 En uzun artan alt dizi 736
8.7 En uzun ortak alt dizgi 744
8.8 En uzun ortak alt dizi 747
8.9 Diğer klasik DP problemleri 751
8.10 Sık yapılan hatalar 756
8.11 Özet 757
8.12 Kavram listesi 758
8.13 Kendini Dene 758
8.14 İleri okuma 760
8.15 Kaynaklar 761

KISIM IV · ZOR PROBLEMLER

9 Geri İzleme Algoritmaları 763

9.1 Zor problemler: P, NP ve NP-zor 764
9.2 Geri izlemenin temel fikri 767
9.3 n-vezir problemi 778
9.4 En az çakışma sezgiseli 791
9.5 Sudoku 795
9.6 Çizge boyama 802
9.7 Hamilton yolu 808
9.8 Kelime arama 813
9.9 Diğer uygulamalar 816
9.10 Analiz 819
9.11 Sık yapılan hatalar 821
9.12 Özet 822
9.13 Kavram listesi 823
9.14 Kendini Dene 823
9.15 İleri okuma 825
9.16 Kaynaklar 826

10 Dal Sınır Algoritmaları 827

10.1 Dal sınır fikri 828
10.2 Paradigma 831
10.3 0/1 sırt çantası 837
10.4 Gezgin satıcı problemi 848
10.5 Atama problemi 859
10.6 Tam sayılı doğrusal programlama 862
10.7 Maksimum klik 864
10.8 Diğer problemler ve A* ile ilişki 868

10.9 Tasarım paradigmalarının karşılaştırması 869
10.10 Sık yapılan hatalar 870
10.11 Özet 871
10.12 Kavram listesi 871
10.13 Kendini Dene 872
10.14 İleri okuma 874
10.15 Kaynaklar 874

11 Yaklaşım Algoritmaları 876

11.1 Yaklaşım algoritmaları 877
11.2 Gezgin satıcı problemi 882
11.3 Yerel arama: 2-opt 892
11.4 İş çizelgeleme 900
11.5 Düğüm kapsama 906
11.6 Sırt çantası 914
11.7 Kuvvet yinelemesi 916
11.8 Karşılaştırma ve sık yapılan hatalar 922
11.9 Özet 923
11.10 Kavram listesi 923
11.11 Kendini Dene 924
11.12 İleri okuma 926
11.13 Kaynaklar 926

KISIM V · DİĞER HESAPLAMA MODELLERİ

12 Rastgele Algoritmalar 928

12.1 Rastgelelik ve algoritmalar 929
12.2 Las Vegas ve Monte Carlo 933
12.3 Karıştırma 936
12.4 BogoSort 940
12.5 Rastgele QuickSort 944
12.6 Rastgele seçme (QuickSelect) 951
12.7 Monte Carlo ile π 954
12.8 Miller–Rabin asallık testi 957
12.9 Karger minimum kesim algoritması 963
12.10 Diğer uygulamalar 969
12.11 Sözde rastgele sayılar 970
12.12 Sık yapılan hatalar 971
12.13 Özet 973
12.14 Kavram listesi 973
12.15 Kendini Dene 974
12.16 İleri okuma 976
12.17 Kaynaklar 976

13 Çevrim İçi Algoritmalar 977

13.1 Çevrim içi problem modeli 978

13.2 Rekabetçi analiz 982
13.3 Kayak kiralama 984
13.4 Sayfalama 991
13.5 Liste güncelleme 1008
13.6 Veri akışları 1012
13.7 Artımlı öğrenme 1021
13.8 Problemler ve sonuçlar 1023
13.9 Sık yapılan hatalar 1023
13.10 Özet 1024
13.11 Kavram listesi 1025
13.12 Kendini Dene 1025
13.13 İleri okuma 1027
13.14 Kaynaklar 1027

14 Paralel Algoritmalar 1028

14.1 Paralel algoritma fikri 1029
14.2 Paralel hesaplama modelleri 1032
14.3 Başarım ölçütleri 1035
14.4 PRAM ile toplama 1041
14.5 Paralel önek toplamı 1046
14.6 Boru hattı 1051
14.7 Boru hatlı ağaç 1056
14.8 Paralel algoritma türleri ve Java 1058
14.9 Paylaşılan durum ve yarış 1066
14.10 Sık yapılan hatalar 1071
14.11 Özet 1072
14.12 Kavram listesi 1073
14.13 Kendini Dene 1073
14.14 İleri okuma 1075
14.15 Kaynaklar 1075

EKLER

Ek A Terim Sözlüğü 1077

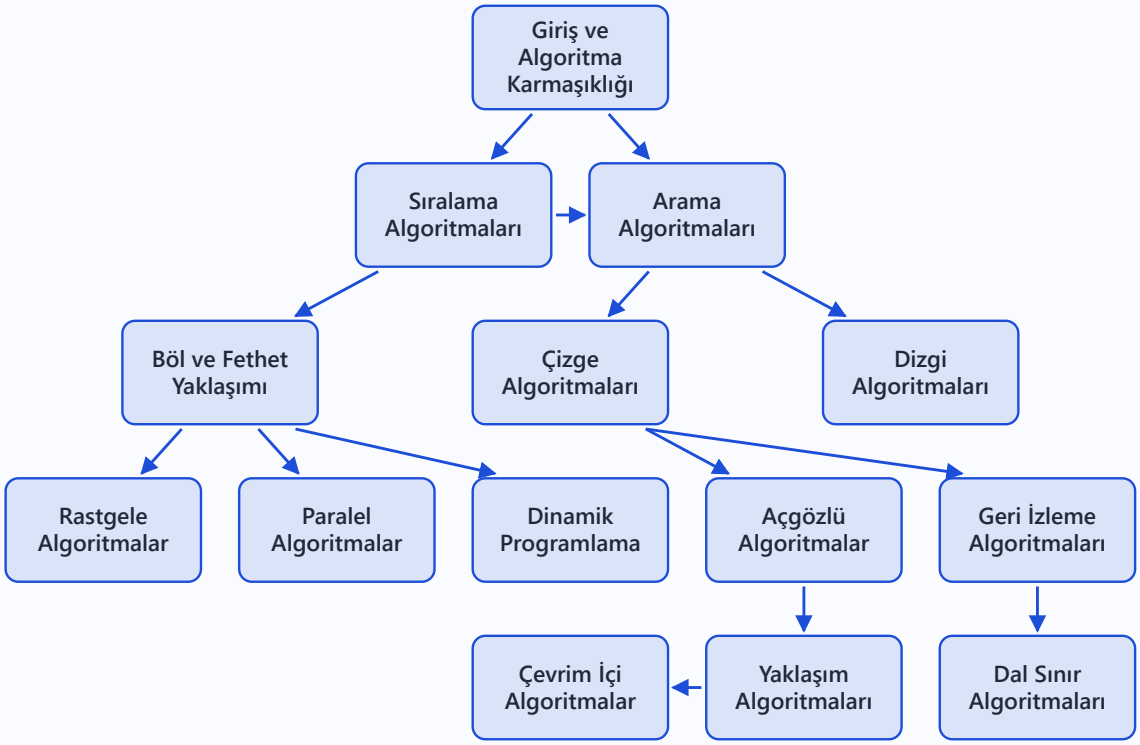
Ek B Algoritmalar İçin Ön Bilgiler 1083

Ek C Algoritmalarda Önemli İsimler 1106

ALGORİTMALAR

Dersin Haritası

Bu sayfa dersin bütün bölümlerini tek bir haritada birleştirir. Her bölüm bir düğümdür; bir ok, sonraki konunun önceki konunun hangi fikrine dayandığını gösterir. Derse başlamadan önce okunursa konuların nereye gittiğini, ders sırasında okunursa bulunulan yeri gösterir.



Şekil 1: Dersin haritası. Oklar, bir konunun hangi konunun üzerine kurulduğunu gösterir. Temel problemlerden tasarım paradigmalarına, oradan zor problemlere ve diğer hesaplama modellerine geçilir.

Şekil 1 konular arasındaki bağları gösterir. Her düğüm bir bölümdür; bir ok, sonraki konunun önceki-nin hangi fikrine dayandığını belirtir. Haritada yalnızca en yakın bağlar çizilmiştir; bir bölümün dayandığı bütün bölümler Tablo 1'dedir.

Konu (bölüm)	Dayandığı konu	Bağ
Giriş ve Algoritma Karmaşıklığı (1)	—	Algoritma kavramı ve çalışma süresinin girdi boyutuyla büyümesi: asimptotik gösterimler.
Sıralama Algoritmaları (2)	Giriş ve Algoritma Karmaşıklığı	İlk büyük problem sınıfı; $O(n^2)$ yöntemlerden $O(n \log n)$ yöntemlere geçiş analizle ölçülür.

Konu (bölüm)	Dayandığı konu	Bağ
Arama Algoritmaları (3)	Giriş ve Algoritma Karmaşıklığı, Sıralama Algoritmaları	Sıralı dizide ikili arama; hash ile ortalama O(1) arama.
Çizge Algoritmaları (4)	Giriş ve Algoritma Karmaşıklığı, Arama Algoritmaları	Arama çizgeye taşınır: DFS ve BFS, güçlü bağlantılı bileşenler, en kısa yollar.
Dizgi Algoritmaları (5)	Giriş ve Algoritma Karmaşıklığı, Arama Algoritmaları	Metinde örüntü arama: KMP, Boyer-Moore, Rabin-Karp; sıkıştırma ve dizi hizalama.
Böl ve Fethet Yaklaşımı (6)	Giriş ve Algoritma Karmaşıklığı, Sıralama Algoritmaları	Birleştirme sıralaması ve hızlı sıralamanın ortak şablonu; yineleme bağıntısı ve Ana Teorem.
Açgözlü Algoritmalar (7)	Çizge Algoritmaları, Böl ve Fethet Yaklaşımı	Her adımda en iyi görünen seçim; Prim, Kruskal ve Dijkstra açgözlü çizge algoritmalarıdır.
Dinamik Programlama (8)	Dizgi Algoritmaları, Böl ve Fethet Yaklaşımı	Böl ve fethetin örtüşen alt problemli hâli: bellekleme ve tablolama; dizgilerde LCS ve düzenleme uzaklığı.
Geri İzleme Algoritmaları (9)	Çizge Algoritmaları	Durum uzayı ağacında derinlik öncelikli arama; P, NP ve NP-zor problemlere giriş.
Dal Sınır Algoritmaları (10)	Geri İzleme Algoritmaları	Geri izlemeye sınır eklenir: en iyi çözümü içeremeyecek dallar budanır.
Yaklaşım Algoritmaları (11)	Açgözlü Algoritmalar, Geri İzleme Algoritmaları	NP-zor problemlerde kanıtlanmış oranlı çözüm: MST tabanlı 2-yaklaşım, LPT, düğüm kapsama.
Rastgele Algoritmalar (12)	Sıralama Algoritmaları, Böl ve Fethet Yaklaşımı	Rastgele pivotlu hızlı sıralama ve QuickSelect; Las Vegas ve Monte Carlo türleri.
Çevrim İçi Algoritmalar (13)	Açgözlü Algoritmalar, Yaklaşım Algoritmaları	Geleceği bilmeden karar: rekabet oranı yaklaşım oranının çevrim içi karşılığıdır; sayfalama.
Paralel Algoritmalar (14)	Giriş ve Algoritma Karmaşıklığı, Böl ve Fethet Yaklaşımı	Bağımsız alt problemler paralel çözülür: önek toplamı, paralel birleştirme sıralaması, iş ve derinlik.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **analiz → temel problemler → tasarım paradigmaları → zor problemler → diğer modeller**. Sıralama ve arama, sonraki bütün bölümlerin hem aracı hem örneğidir. Böl ve fethet sıralamadan doğar ve dinamik programlamaya, rastgele ve paralel algoritmalara açılır. Çizge algoritmaları açgözlü yönteme ve durum uzayı aramasına örnek verir. Geri izleme ile dal sınır en iyi çözümü üstel zamanda arar; yaklaşım algoritmaları bunun yerine kanıtlanmış bir oranla yetinir. Çevrim içi algoritmalar aynı oran fikrini bilinmeyen geleceğe uygular.

KISIM I

Temeller

Bölüm 1–3

KISIM I · BÖLÜM 1

Giriş ve Algoritma Karmaşıklığı

Algoritma, bir problemi sonlu sayıda ve iyi tanımlanmış adımla çözen yöntemdir. Bölümün ilk kısmında algoritma kavramı gündelik örneklerle tanıtılır; sabit, doğrusal, karekök, kübik, faktöriyel ve üstel zaman alan örnekler üzerinden çalışma süresinin girdi boyutuyla nasıl büyüdüğü gösterilir ve Öklid algoritmasının neden hızlı olduğu açıklanır. İkinci kısımda adım sayma, en iyi, en kötü ve ortalama durum, alan karmaşıklığı ile Büyük O, Omega ve Teta gösterimleri ele alınır. Döngülerin toplam formülleriyle çözümlenmesi, yineleme bağıntılarının kurulup Ana Teorem ile çözülmesi örnek fonksiyonlar üzerinde işlenir; amortize analize de kısaca değinilir.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum01/src`

BU BÖLÜMÜN NOTLARI

1.1 Giriş

1.2 Algoritma Karmaşıklığı

KISIM I · BÖLÜM 1.1

Giriş

Bilgisayarın yaptığı her iş, birinin önceden yazdığı bir adımlar listesine dayanır. Bu listeye algoritma denir. Aynı problemi çözen birçok algoritma bulunabilir ve bunlar hız bakımından birbirinden çok farklı olabilir. Biri bir milyon öğede milisaniyeler içinde biter, öteki aynı işi yıllarca sürdürür. Not, algoritmanın ne olduğunu ve iyi bir algoritmanın hangi özellikleri taşıdığını anlatarak başlar. Ardından çalışma süresinin saniyeyle değil, girdi boyutuna bağlı işlem sayısıyla nasıl ölçüldüğünü gösterir. Sabit, doğrusal, karekök, kübik, faktöriyel ve üstel zamanlı kısa Java kodları satır satır izlenir. Son konu, bilinen en eski algoritmalardan biri olan Öklid algoritmasıdır.

ÖĞRENME ÇIKTILARI

Algoritma
Algoritmanın tanımını ve özelliklerini (girdi, çıktı, açıklık, sonluluk, etkinlik, doğruluk) açıklamak; günlük bir işi algoritma olarak yazmak ve bir yöntemin hangi özelliği sağlamadığını göstermek.

Analiz
Çalışma süresini girdi boyutu n'nin fonksiyonu $T(n)$ olarak ifade etmek; kısa bir kodun işlem sayısını satır satır saymak; büyüme tablosundan n büyürken sürenin nasıl değiştiğini okumak.

Karmaşıklık örnekleri
Kısa kodların karmaşıklık sınıfını (1, n, $\sqrt{n}$, n^3, $n!$, 2^n) gerekçesiyle bulmak; belleklemenin üstel süreyi nasıl doğrusala indirdiğini açıklamak.

Öklid algoritması
EBOB hesabını adım adım izlemek; algoritmanın doğruluğunu ve $O(\log \min(a, b))$ süresini gerekçelendirmek; en kötü girdinin neden ardışık Fibonacci sayıları olduğunu açıklamak.

1.1.1 Algoritma kavramı

"Algoritma" sözcüğü, 9. yüzyılda Bağdat'ta yaşamış matematikçi Muhammed bin Musa el-Harezmî'nin adının Latinceleşmiş biçiminden gelir. Harezmî'nin Hint rakamlarıyla hesap yapmayı anlatan kitabı, Avrupa'da Latince çevirisiyle tanınmıştır. Çeviri "Dixit Algorizmi" (Harezmî der ki) sözleriyle başlar.

Zamanla bu ad, belirli bir hesabı adım adım yapma yönteminin genel adı olmuştur. Bugün bir bilgisayar programının çözdüğü her problemin arkasında bir algoritma vardır: arama motorunun sonuçları sıralaması, haritanın en kısa yolu bulması, bir fotoğrafın sıkıştırılması.

1.1.1.1 Algoritma nedir?

Bir yemek tarifi, bir mobilya kurulum kılavuzu ya da elle uzun bölme yapma yöntemi aynı ortak yapıyı taşır. Hepsinde başlangıçta elde bazı şeyler vardır: malzemeler, parçalar, iki sayı. Ardından sırası belli adımlar gelir. Adımlar bittiğinde istenen sonuç ortaya çıkar: yemek, dolap, bölüm ve kalan. Algoritma, bu yapının bilgisayar biliminde kesin kurallarla tanımlanmış hâlidir.

Tanım 1.1.1

Algoritma (*algorithm*), bir problemi çözmek ya da bir görevi yerine getirmek için izlenen, sonlu sayıda ve her biri açıkça tanımlı adımlardan oluşan yönergeler dizisidir. Algoritma girdi verilerini alır, bunlar üzerinde işlemler yapar ve bir **çıktı** üretir.

Tanımda üç parça öne çıkar. Birincisi *girdi ve çıktı*: algoritma bir şey alır ve bir şey verir. İkincisi *adımların açıklığı*: her adım tek bir biçimde anlaşılır. Üçüncüsü *sonluluk*: adımlar sonsuza kadar sürmez. Bu üç parçanın her biri, bir sonraki alt kesimde ayrı bir özellik olarak ele alınır.

Algoritmanın bir programlama dilinde yazılmış olması gerekmez. Türkçe cümlelerle, numaralı adımlarla, akış şemasıyla ya da sözde kodla (*pseudocode*) anlatılabilir. Program ise algoritmanın belirli bir dilde, belirli bir makinede çalışacak biçimde yazılmış hâlidir. Aynı algoritma Java, Python ya da C ile yazılabilir; her biri ayrı bir programdır ama hepsi aynı algoritmayı izler. Bu ayrım, algoritma analizinin temelidir. Bir algoritmanın iyi ya da kötü olduğuna, onu yazan dilden ve çalıştıran makineden bağımsız olarak karar verilir.

1.1.1.2 Bir algoritmanın taşıması gereken özellikler

Her adımlar listesi algoritma sayılmaz. Donald Knuth, algoritmanın beş özelliğini sayar: girdi, çıktı, açıklık, sonluluk ve etkinlik. Bunlara pratikte bir altıncısı, doğruluk eklenir. Tablo 1.1.1 bu altı özelliği kısa açıklamalarıyla verir; ardından her biri tek tek ele alınır.

Özellik	Anlamı
Girdi	Sıfır ya da daha fazla değer dışarıdan alınır.
Çıktı	En az bir sonuç üretilir; sonuç girdiyle ilişkilidir.
Açıklık	Her adımın tek bir anlamı vardır; yoruma yer bırakmaz.
Sonluluk	Her girdi için sonlu sayıda adımdan sonra durur.
Etkinlik	Her adım temel ve gerçekten yapılabilir bir işlemdir.
Doğruluk	Her geçerli girdi için doğru çıktıyı verir.

Tablo 1.1.1: Bir algoritmanın özellikleri.

Girdi (*input*). Algoritma çalışmaya başlamadan önce ya da çalışırken dışarıdan değer alır. Girdi sayısı sıfır da olabilir. Örneğin "ilk 100 asal sayıyı yazdır" görevinin dışarıdan aldığı bir değer yoktur; yine de geçerli bir algoritmadır. Çoğu algoritmada ise girdi, problemin kendisidir: sıralanacak dizi, aranan sayı, iki şehir arasındaki yol ağı.

Çıktı (*output*). Algoritma en az bir sonuç üretir ve bu sonuç girdiyle ilişkilidir. Hiçbir şey üretmeyen bir adımlar listesi, ne kadar uzun çalışırsa çalışsın bir işe yaramaz. Çıktı her zaman ekrana yazılan bir değer olmak zorunda değildir. Bir metodun döndürdüğü değer, değiştirilmiş bir dizi ya da "evet/hayır" biçiminde bir karar da çıktıdır.

Açıklık (*definiteness*). Her adım kesin olarak tanımlanır ve yalnız bir biçimde anlaşılır. "Biraz tuz ekle" bir insan için yeterli olabilir; bir makine için ise anlamsızdır. Algoritma, adımı okuyan herkesin aynı işi yapmasını gerektirir. Bu yüzden "biraz", "yeterince", "uygun bir değer" gibi ifadeler algoritmada yer almaz.

Sonluluk (*finiteness*). Algoritma her girdi için sonlu sayıda adımdan sonra durur. Kodu yazılmış ama bazı girdilerde hiç bitmeyen bir döngü, o girdiler için algoritma değildir. Sonluluğu bozan en yaygın hata, döngü koşulunun hiçbir zaman yanlış olmamasıdır. Örneğin bir sayıyı her turda 2 azaltıp sıfıra ulaşınca duran bir döngü, tek sayılarda sıfırı atlar ve sonsuza dek döner.

Etkinlik (*effectiveness*). Her adım, kâğıt kalemle ya da makinede sonlu sürede yapılabilecek kadar temel bir işlemdir. "Sayının asal çarpanlarını bul" tek bir adım olarak yazılamaz, çünkü bu kendi başına bir problemdir; ancak daha temel adımlara ayrıldıktan sonra algoritmanın parçası olur. "Bütün gerçel sayıları tek tek dene" ise hiç yapılamaz, çünkü gerçel sayılar sayılamayacak kadar çoktur.

Doğruluk (*correctness*). Algoritma her geçerli girdi için doğru çıktıyı verir. Birkaç örnekte doğru sonuç vermek yetmez; doğruluk, bütün geçerli girdiler için gösterilmesi gereken bir özelliktir. Bu gösterim çoğu zaman bir kanıt gerektirir. Öklid algoritmasının doğruluğu, bu notun son kesiminde böyle bir kanıtla gösterilir.

Örnek 1.1.1

Aşağıdaki yöntem, bir tam sayının basamak sayısını bulmak için yazılmıştır: "1. Sayaç = 0. 2. Sayı 0'dan büyük olduğu sürece sayıyı 10'a böl (tam sayı bölmesi) ve sayacı 1 artır. 3. Sayacı yaz." Bu yöntem hangi özellikleri sağlar?

1. **Girdi ve çıktı:** bir tam sayı alır, bir sayaç değeri yazar. İkisi de vardır.
2. **Açıklık:** her adım tek anlamlıdır: atama, karşılaştırma, tam sayı bölmesi, artırma.
3. **Etkinlik:** bu işlemlerin hepsi makinede tek adımda yapılabilir.
4. **Sonluluk:** pozitif bir sayı 10'a her bölündüğünde küçülür ve sonunda 0 olur. Örneğin 4075 → 407 → 40 → 4 → 0; dört bölmede durur, sayaç 4 olur. Sayı 0 ya da negatifse döngüye hiç girilmez ve yöntem hemen durur.

5. **Doğruluk:** pozitif sayılarda sonuç doğrudur. Ama 0 için çıktı 0 olur; oysa 0 sayısı tek basamaklıdır. Negatif sayılar için de çıktı 0'dır. Yöntem bu girdilerde yanlış sonuç verir. Ya girdi kümesi pozitif sayılarla sınırlanmalı ya da bu durumlar ayrıca ele alınmalıdır.

Örnek, doğruluğun girdi kümesine bağlı olduğunu da gösterir. Bir algoritma "her geçerli girdi için" doğru olmalıdır; hangi girdilerin geçerli olduğu ise problemin tanımında baştan söylenir. Problem tanımı belirsiz kalırsa algoritmanın doğru olup olmadığı da belirsiz kalır.

1.1.1.3 Günlük hayattan: çay hazırlama

Algoritma kavramını somutlaştırmanın kolay bir yolu, iyi bilinen bir günlük işi adım adım yazmaktır. Çay hazırlamanın girdisi su, çay yaprakları ve isteğe bağlı olarak şeker ile süttür; çıktısı bir fincan çaydır. Sunumdaki algoritma altı adımdan oluşur:

1. Suyu kaynat.
2. Kaynamış suyu demliğe koy.
3. Demliğin içine çay yapraklarını ekle.
4. Çayın demlenmesini bekle.
5. Demlenen çayı fincana süzerek koy.
6. İsteğe bağlı olarak şeker ve süt ekle.

Bu kısa liste, algoritmanın üç önemli niteliğini gösterir.

Sıra önemlidir. Algoritma, sıralı bir adımlar listesidir. Bazı adımların yeri değişebilir: 2. ve 3. adımlar yer değiştirse, yani önce yapraklar demliğe konup sonra kaynar su eklense, sonuç değişmez. Ama 1. adım ile 5. adımın yeri değişemez. Su kaynamadan çay süzülemez; süzülecek bir şey yoktur. Bir algoritma yazılırken hangi adımın hangisinden önce gelmesi gerektiği, yani adımlar arasındaki *bağımlılık* dikkatle düşünülür.

Açıklık her okur için ayrı sınanır. "Çayın demlenmesini bekle" adımı bir insan için yeterlidir; deneyimli biri çayın rengine bakıp karar verir. Bir bilgisayar için ise bu adım belirsizdir: ne kadar beklenecektir? Adım makine için "10 dakika bekle" biçiminde yazılmalıdır. Açıklık özelliği tam olarak bunu ister: adımı okuyan herkes, yorum yapmadan aynı işi yapabilmelidir.

Algoritmalar karar içerir. 6. adım bir koşul taşır: "şeker isteniyorsa şeker ekle, süt isteniyorsa süt ekle." Algoritmanın izlediği yol girdiye göre değişebilir. Programlamada bu, `if` deyimine karşılık gelir. Benzer biçimde "su kaynayana kadar bekle" gibi bir adım bir döngüdür. Sıralı adımlar, kararlar ve döngüler, her algoritmanın yapı taşlarıdır.

Sık yapılan hata

Günlük dildeki bir tarifi olduğu gibi algoritma saymak. "Göz kararı", "kıvamına gelene kadar", "uygun bir sayı seç" gibi ifadeler açıklığı bozar. Bir adımın açık olup olmadığı, onu hiç bilmeyen birinin (ya da bir makinenin) aynı işi yapıp yapamayacağıyla sınanır.

1.1.1.4 Algoritmadan koda: ortalama bulma

Üç sayının ortalamasını bulma problemi, algoritma ile program arasındaki ilişkiyi göstermek için uygun, küçük bir örnektir. Girdi üç sayıdır (Sayı1, Sayı2, Sayı3), çıktı bunların ortalamasıdır. Algoritma üç adımdan oluşur:

1. Toplam = Sayı1 + Sayı2 + Sayı3
2. Ortalama = Toplam / 3
3. Ortalama değerini ekrana yazdır.

Bu algoritmanın Python ile yazılmış hâli şöyledir:

```python
def ortalama_hesapla(sayi1, sayi2, sayi3):
    ortalama = (sayi1 + sayi2 + sayi3) / 3
    return ortalama

sayi1 = float(input("Birinci sayıyı girin: "))
sayi2 = float(input("İkinci sayıyı girin: "))
sayi3 = float(input("Üçüncü sayıyı girin: "))
ort = ortalama_hesapla(sayi1, sayi2, sayi3)
print("Girilen sayıların ortalaması:", ort)
```

Aynı algoritmanın Java'daki karşılığı da aşağıdadır. Kod, bir sınıfın `main` metodu olarak yazılmıştır; `Scanner` sınıfı için dosyanın başında `import java.util.Scanner;` satırı gerekir.

```java
public static void main(String[] args) {
  Scanner giris = new Scanner(System.in);
  System.out.println("Birinci sayıyı girin: ");
  double sayi1 = giris.nextDouble();
  System.out.println("İkinci sayıyı girin: ");
  double sayi2 = giris.nextDouble();
  System.out.println("Üçüncü sayıyı girin: ");
  double sayi3 = giris.nextDouble();
  double ortalama = (sayi1 + sayi2 + sayi3) / 3;
  System.out.println("Sayıların ortalaması: "
                     + ortalama);
}
```

İki program farklı görünür. Python'da hesap bir fonksiyona ayrılmıştır ve girdiler `input` ile okunup `float`'a çevrilir. Java'da her şey `main` içindedir, girdiler `Scanner` nesnesiyle `nextDouble()` metoduyla okunur ve değişkenlerin tipi (`double`) açıkça yazılır. Ama iki program da aynı üç adımı izler: üç sayı toplanır, toplam 3'e bölünür, sonuç yazılır. Örneğin girdi 4, 7 ve 10 ise iki program da 7,0 yazar.

Bu gözlem, algoritmanın **dilden bağımsız** olduğunu gösterir. Program, algoritmanın belirli bir dildeki karşılığıdır. Bir algoritmayı anlamak ve analiz etmek için onun hangi dilde yazıldığı önemli değildir. Bu notta kodlar Java ile verilir, ama her kodun arkasındaki fikir başka bir dile de aynen taşınabilir.

> **Sık yapılan hata**
>
> Java'da değişkenleri `int` olarak tanımlayıp `(sayi1 + sayi2 + sayi3) / 3` yazmak. İki tam sayı bölündüğünde Java kesirli kısmı atar: 4, 7 ve 9 için sonuç 6,67 yerine 6 olur. Algoritma doğrudur, ama programdaki tip seçimi onu yanlış uygular. Sunumdaki kod bu yüzden `double` kullanır; tam sayılarla çalışırken de `/ 3.0` yazmak yeter.

1.1.1.5 Algoritma türleri

Algoritmalar, çözdükleri probleme ya da izledikleri tasarım fikrine göre gruplanır. Bu ders, aşağıdaki grupların her birinden temel algoritmaları ele alır.

- **Sıralama algoritmaları** öğeleri bir düzene koyar. Eklemeli sıralama (*insertion sort*), birleştirme sıralaması (*merge sort*) ve hızlı sıralama (*quick sort*) bu grubun en bilinen üyeleridir. Sıralı veri, aramayı ve birçok başka işlemi hızlandırdığı için sıralama, bilgisayar biliminin en çok çalışılmış problemlerinden biridir.
- **Arama algoritmaları** bir veri kümesinde belirli bir öğeyi bulur. Doğrusal arama öğeleri tek tek inceler. İkili arama sıralı dizide aralığı her adımda yarıya indirir. Hash tablosu ise öğenin yerini doğrudan hesaplar.
- **Çizge algoritmaları**, düğümler ve onları bağlayan kenarlardan oluşan yapılar üzerinde çalışır. Dijkstra algoritması iki nokta arasındaki en kısa yolu bulur. Minimum kapsayan ağaç algoritmaları bütün düğümleri en az toplam maliyetle bağlar. Ford–Fulkerson yöntemi ise bir ağdan geçebilecek en büyük akışı hesaplar.
- **Dizgi ve sıkıştırma algoritmaları** metinle çalışır. Bir metinde bir sözcüğü aramak, sözcükleri trie adlı ağaçta saklamak ve dosya boyutunu küçültmek (Huffman kodlaması, gzip) bu gruba girer.
- **Tasarım teknikleri** tek bir problemi değil, birçok probleme uygulanabilen genel bir çözüm fikrini anlatır. Böl ve fethet (birleştirme sıralaması, ikili arama, hızlı üs alma), dinamik programlama, açgözlü algoritmalar, geri izleme ve dal sınır bu tekniklerin başlıcalarıdır.
- **Rastgele ve yaklaşık algoritmalar**, zor problemler için iki ayrı yol sunar. Rastgele algoritma adımlarında rastgele seçim yapar. Yaklaşık algoritma ise en iyi çözümü bulmanın çok uzun süreceği problemlerde, en iyiye kanıtlanmış bir oranda yakın bir çözümü kısa sürede bulur.

Bu gruplar birbirini dışlamaz. Birleştirme sıralaması hem bir sıralama algoritmasıdır hem de böl ve fethet tekniğinin örneğidir. Hızlı sıralama, pivotu rastgele seçen biçimiyle aynı zamanda bir rastgele algoritmadır.

1.1.1.6 Deterministik ve rastgele algoritmalar

Algoritmaları ayırmanın bir başka yolu, aynı girdi verildiğinde hep aynı yolu izleyip izlemediklerine bakmaktır.

> **Tanım 1.1.2**
>
> **Deterministik** (*deterministic*) algoritma, aynı girdi için her çalıştırmada aynı adımları izler ve aynı çıktıyı verir. **Rastgele algoritma** (*randomized algorithm*) ise adımlarından bazılarında rastgele seçim yapar; bu yüzden aynı girdide farklı çalıştırmalar farklı yollar izleyebilir.

Bir dizinin en büyük öğesini bulan algoritma deterministiktir: dizi aynıysa hangi öğelerin hangi sırayla karşılaştırılacağı da aynıdır. Pivotu rastgele seçen hızlı sıralama ise her çalıştırmada diziyi farklı noktalardan bölebilir. Bu algoritma bir anlamda zar atar. Ama sonuç değişmez; dizi her seferinde doğru sıralanır. Değişen yalnız izlenen yol ve dolayısıyla çalışma süresidir.

Rastgeleliğin bir algoritmaya ne kazandırdığı ilk bakışta açık değildir. Hızlı sıralamada pivot hep ilk öğe seçilirse, zaten sıralı bir dizi algoritmayı en yavaş çalışma biçimine sokar. Rastgele pivot ise "kötü girdi" kavramını ortadan kaldırır: hiçbir girdi her çalıştırmada kötü sonuç vermez ve beklenen süre her girdide aynı ölçüde kısa kalır. Bazı rastgele algoritmalar ise çok küçük bir olasılıkla yanlış cevap vermeyi göze alarak büyük hız kazanır. Bu notun geri kalanındaki bütün algoritmalar deterministiktir.

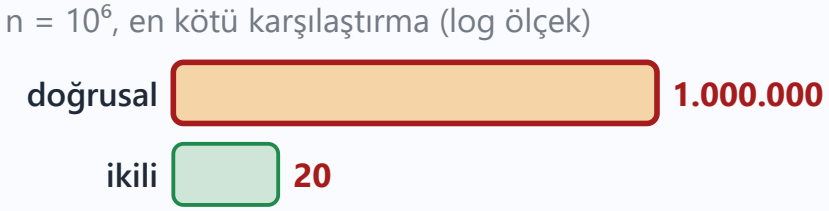
1.1.2 Algoritma analizi

Bir problemi çözen algoritmayı bulmak işin yarısıdır. Aynı problemi çözen birden çok algoritma varsa hangisinin seçileceğine de karar vermek gerekir. Bu kararın aracı algoritma analizidir. Bu kesim, analizin neyi ölçtüğünü, ölçümün neden saniyeyle değil işlem sayısıyla yapıldığını ve n büyürken farklı algoritmaların ne kadar ayrıştığını anlatır.

1.1.2.1 Neden analiz?

> **Tanım 1.1.3**
>
> **Algoritma analizi** (*algorithm analysis*), bir algoritmanın işlemci zamanı ve bellek gibi kaynakları ne ölçüde kullandığını girdi boyutuna göre inceler. Zaman gereksinimine **zaman karmaşıklığı** (*time complexity*), ek bellek gereksinimine **alan karmaşıklığı** (*space complexity*) denir.

Analizin değerini küçük bir örnek açıkça gösterir. Bir milyon (10^6) öğeli sıralı bir dizide bir sayı aranıyor olsun. Doğrusal arama dizinin başından başlar ve öğeleri tek tek inceler. Aranan sayı dizide yoksa ya da en sondaysa bir milyon karşılaştırma yapar. İkili arama ise ortadaki öğeye bakar ve aranan sayı daha küçükse sol yarıda, büyükse sağ yarıda aramayı sürdürür. Her karşılaştırma aralığı yarıya indirdiği için en fazla 20 karşılaştırma yapar, çünkü bir milyon yaklaşık 2^{20}'dir. Şekil 1.1.1 bu iki sayıyı logaritmik ölçekte karşılaştırır.



Şekil 1.1.1: Bir milyon öğeli sıralı dizide en kötü durumdaki karşılaştırma sayısı. Çubuk uzunlukları logaritmik ölçektedir; gerçek ölçekte ikili aramanın çubuğu görünmeyecek kadar kısa olurdu.

İki algoritma da doğrudur; ikisi de sayıyı bulur. Aralarındaki fark yalnız kaynak kullanımındadır ve bu fark elli bin kattır. Analiz, böyle bir farkı programı yazmadan, yalnız algoritmanın adımlarına bakarak önceden görmeyi sağlar. Hangi algoritmanın seçileceği, çoğu zaman bu tür bir karşılaştırmayla belirlenir.

1.1.2.2 Girdi boyutu ve temel işlem

Bir algoritmanın ne kadar sürdüğünü ölçmenin akla gelen ilk yolu, programı çalıştırıp saati tutmaktır. Ama bu ölçüm birçok şeye bağlıdır: işlemcinin hızı, programlama dili, derleyici, aynı anda çalışan öteki programlar. Aynı kod bir dizüstü bilgisayarda 2 saniye, bir sunucuda 0,3 saniye sürebilir. Bu sayılar algoritma hakkında değil, makine hakkında bilgi verir.

Algoritma analizi bu yüzden süreyi saniyeyle ölçmez. Onun yerine algoritmanın yaptığı **temel işlemleri** sayar. Temel işlem, makinede sabit sürede yapılan işlemdir: iki sayıyı toplamak ya da karşılaştırmak, bir dizi hücresine erişmek, bir değişkene değer atamak. İşlem sayısı makineden makineye değişmez. Hızlı bir makine her işlemi daha kısa sürede yapar, ama işlem sayısı aynı kalır.

İşlem sayısı ise girdiye bağlıdır. Beş öğeli bir diziyi toplamak, bir milyon öğeli diziyi toplamaktan az iş ister. Bu yüzden işlem sayısı, girdinin büyüklüğünü gösteren bir sayının fonksiyonu olarak yazılır.

Tanım 1.1.4

Girdi boyutu (*input size*) n, problemin büyüklüğünü gösteren sayıdır: dizideki öğe sayısı, matrisin satır sayısı, çizgedeki düğüm sayısı gibi. Algoritmanın n boyutlu bir girdide yaptığı temel işlem sayısı $T(n)$ ile gösterilir ve algoritmanın **çalışma süresi** (*running time*) olarak anılır.

Girdi boyutunun ne olduğu probleme göre seçilir. Bir dizi işleyen algoritmada doğal seçim öğe sayısıdır. İki $n \times n$ matrisi çarpan algoritmada n'dir. Tek bir sayıyı girdi olarak alan algoritmada ise iki seçenek vardır: sayının değeri ya da sayının kaç basamakla yazıldığı. Bu iki seçenek çok farklı sonuçlar verir; asallık testi incelenirken bu ayrıma ayrıca dönülür.

Aynı boyuttaki iki girdi de farklı sayıda işlem gerektirebilir. Doğrusal arama, aranan öğe dizinin ilk hücresindeyse tek karşılaştırmayla biter; dizide hiç yoksa n karşılaştırma yapar. Bu yüzden analizde çoğu zaman **en kötü durum** (*worst case*) incelenir: aynı boyuttaki girdiler arasında en çok işlem gerektireni. En kötü durum bir garanti verir. Algoritma hiçbir girdide bu sayıdan fazla işlem yapmaz.

1.1.2.3 Satır satır sayım: dizi toplamı

İşlem saymanın nasıl yapıldığını görmek için n öğeli bir dizinin toplamını hesaplayan kısa koda bakalım. Sunumdaki kodun her satırının yanında o satırın kaç kez çalıştığı yazılıdır.

```
int toplam = 0;                  // 1 kez
for (int i = 0; i < n; i++)      // n+1 karşılaştırma
  toplam += dizi[i];             // n kez
return toplam;                   // 1 kez
```

Sayım satır satır yapılır.

- `int toplam = 0;` döngüden önce bir kez çalışır: **1** işlem.
- `for` satırındaki `i < n` karşılaştırması, döngüye her girişte bir kez yapılır. i değeri 0, 1, …, $n - 1$ iken koşul doğrudur ve döngü gövdesi çalışır; bu n karşılaştırma eder. $i = n$ olduğunda koşul bir kez daha sınanır, yanlış çıkar ve döngü biter. Toplam $\boldsymbol{n + 1}$ karşılaştırma.
- `toplam += dizi[i];` döngü gövdesidir ve $\boldsymbol{n}$ kez çalışır.
- `return toplam;` bir kez çalışır: **1** işlem.

Bu sayıların toplamı çalışma süresini verir:

> $T(n) = 1 + (n + 1) + n + 1 = 2n + 3$

Örnek 1.1.2

`dizi = {4, 1, 3}` için sayım adım adım yapılırsa ($n = 3$):

1. `toplam = 0` : 1 işlem.
2. $i = 0$: `0 < 3` doğru (1 karşılaştırma), `toplam = 4` (1 işlem).
3. $i = 1$: `1 < 3` doğru, `toplam = 5` .
4. $i = 2$: `2 < 3` doğru, `toplam = 8` .
5. $i = 3$: `3 < 3` yanlış (1 karşılaştırma); döngü biter.

6. `return 8` : 1 işlem.
7. **Toplam:** 1 + 4 karşılaştırma + 3 toplama + 1 = 9 işlem. Formül de $T(3) = 2 \cdot 3 + 3 = 9$ verir.

Bu sayımda `i++` artırmaları ve `dizi[i]` erişimi ayrı işlem olarak sayılmadı. Onlar da sayılsaydı sonuç $4n + 3$ gibi başka bir ifade olurdu. Hangi işlemlerin sayılacağı bir seçimdir ve bu seçim katsayıları değiştirir. Ama bir şey değişmez: işlem sayısı her durumda n'nin bir katı artı bir sabittir. Dizi iki katına çıkarsa iş de yaklaşık iki katına çıkar.

1.1.2.4 Sabitler neden önemsizleşir?

$2n + 3$ ifadesindeki 2 ve 3 sayıları makineye ve sayım biçimine bağlıdır. Bir makinede karşılaştırma toplama işleminden iki kat yavaş olabilir; bir derleyici iki işlemi tek komutta birleştirebilir. Bu sabitleri kesin olarak bilmek hem zordur hem de pek bir işe yaramaz. Önemli olan, n büyürken $T(n)$'nin hangi hızla büyüdüğüdür.

$n = 10$ iken $2n + 3 = 23$'tür ve sabit terim 3 toplamın yaklaşık sekizde birini oluşturur. $n = 10^6$ olduğunda ise $2n + 3 = 2.000.003$'tür ve 3'ün katkısı milyonda bir düzeyine iner. Büyük n için yalnız en hızlı büyüyen terim belirleyicidir. Katsayısı da önemsizleşir, çünkü katsayı her n için aynı oranda etki eder: süreyi iki katına çıkarır, ama büyüme biçimini değiştirmez.

Bu düşünce, algoritmaları büyüme hızlarına göre sınıflamaya götürür. $2n + 3$, $5n$ ve $n/2 + 100$ fonksiyonlarının hepsi n ile doğru orantılı büyür. Bunlara **doğrusal** (*linear*) denir ve kısaca $\Theta(n)$ diye yazılır. Θ (teta) gösterimi, sabit katsayıları ve küçük terimleri atıp yalnız büyüme hızını bırakır. Benzer biçimde $O(n)$ (büyük O) gösterimi "en fazla n ile orantılı büyür" anlamına gelir; yani bir üst sınır verir. Bu notta iki gösterim bu sezgisel anlamlarıyla kullanılır. $T(n) = 2n + 3$ için hem $T(n) = \Theta(n)$ hem de $T(n) = O(n)$ yazılabilir.

Örnek 1.1.3

$T(n) = 5n^2 + 100n \log_2 n + 7$ fonksiyonunun büyüme hızı nedir? Hangi terim ne zaman baskındır?

1. Terimler: $5n^2$, $100n \log_2 n$ ve 7. Sabit 7, büyük n'de hemen önemsizleşir.
2. $n = 100$: $5n^2 = 50.000$, $100n \log_2 n \approx 100 \cdot 100 \cdot 6{,}64 \approx 66.439$. Bu noktada ikinci terim daha büyüktür, çünkü katsayısı 20 kat fazladır.
3. $n = 1000$: $5n^2 = 5.000.000$, $100n \log_2 n \approx 996.578$. Artık birinci terim beş kat büyüktür.
4. İki terimin oranı $5n^2 / (100n \log_2 n) = n / (20 \log_2 n)$'dir ve n büyüdükçe sınırsız artar. Hesap, birinci terimin $n = 144$'ten itibaren hep önde olduğunu gösterir.
5. Sonuç: $T(n) = \Theta(n^2)$. Büyüme hızını katsayısı büyük olan terim değil, n büyüdükçe en hızlı artan terim belirler.

1.1.2.5 Büyüme tablosu

Algoritmaların çalışma süreleri birkaç tanıdık fonksiyon ailesine düşer: sabit (1), logaritmik ($\log n$), karekök ($\sqrt{n}$), doğrusal (n), log-doğrusal ($n \log n$), karesel (n^2), kübik (n^3), üstel (2^n) ve faktöriyel ($n!$). Bu fonksiyonların n büyürken nasıl davrandığını görmenin en iyi yolu, değerlerini yan yana koymaktır. Tablo 1.1.2 sunumdaki büyüme tablosunu verir.

f(n)	n = 10	n = 100	n = 1000	n = 10^6	süre (n = 10^6)
1	1	1	1	1	1 ns
$\log_2 n$	3,3	6,6	10,0	19,9	19,9 ns
$\sqrt{n}$	3,2	10	31,6	1.000	1 µs
n	10	100	1.000	1.000.000	1 ms
$n \log_2 n$	33,2	664,4	9.966	$2{,}0 \cdot 10^7$	19,9 ms
n^2	100	10.000	1.000.000	$1{,}0 \cdot 10^{12}$	16,7 dk
n^3	1.000	1.000.000	$1{,}0 \cdot 10^9$	$1{,}0 \cdot 10^{18}$	≈ 32 yıl
2^n	1.024	$1{,}3 \cdot 10^{30}$	$\approx 1{,}1 \cdot 10^{301}$	$\approx 10^{301030}$	evrenin yaşından çok
$n!$	3.628.800	$9{,}3 \cdot 10^{157}$	$\approx 10^{2567}$	$\approx 10^{5565708}$	evrenin yaşından çok

Tablo 1.1.2: Büyüme tablosu. Son sütun, saniyede 10^9 işlem yapan bir bilgisayarda $n = 10^6$ için gereken süredir.

Tabloyu satır satır okumak birkaç önemli gözlem verir.

Sabit ve logaritmik büyüme neredeyse hiç büyümez. n, 10'dan 10^6'ya, yani yüz bin katına çıktığında $\log_2 n$ yalnız 3,3'ten 19,9'a çıkar. Logaritma, "n'yi kaç kez ikiye bölersem 1'e inerim?" sorusunun cevabıdır. Bir milyonu yirmi kez ikiye bölmek onu 1'e indirir. İkili aramanın 20 karşılaştırması bu sayıdan gelir.

Karekök, doğrusaldan çok yavaş büyür. $n = 10^6$ iken $\sqrt{n}$ yalnız 1000'dir. Bir algoritmayı n'den $\sqrt{n}$'ye indirmek, bir milyonluk girdide işi bin kat azaltır.

***n log n*, doğrusala çok yakındır.** Bir milyon öğede $n \log_2 n$ yaklaşık $2 \cdot 10^7$'dir; doğrusal sürenin yalnız yirmi katı. Saniyede 10^9 işlem yapan bir makinede bu 20 milisaniye eder. Verimli sıralama algoritmaları bu sınıftadır.

Polinomlar arasında bile büyük uçurumlar vardır. Aynı bir milyonluk girdide n^2 işlem 16,7 dakika, n^3 işlem yaklaşık 32 yıl sürer. Karesel bir algoritma küçük girdilerde sorun çıkarmaz, ama girdi büyüdükçe kullanılamaz hâle gelir.

Üstel ve faktöriyel büyüme her sınırı aşar. 2^n, $n = 100$'de bile $1{,}3 \cdot 10^{30}$'a ulaşır. Saniyede 10^9 işlemle bu, yaklaşık $4 \cdot 10^{13}$ yıl eder; evrenin yaşı ise yaklaşık $1{,}4 \cdot 10^{10}$ yıldır. $n!$ daha da hızlıdır: $n = 10$'da 3.628.800'e, $n = 100$'de 158 basamaklı bir sayıya ulaşır. Bu iki sınıftaki algoritmalar yalnız çok küçük girdilerde kullanılabilir.

Tablodaki sıralama, bu dersin geri kalanında sürekli kullanılır:

> $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$

Buradaki "<" işareti, yeterince büyük *n* için soldaki fonksiyonun sağdakinden küçük kaldığını, üstelik aradaki oranın sınırsız büyüdüğünü anlatır. Küçük *n* değerlerinde sıra bozulabilir: $n = 10$'da $\sqrt{n} \approx 3{,}2$, $\log_2 n \approx 3{,}3$'ten küçüktür. $n = 16$'da ikisi de 4'e eşittir ve bundan sonra $\sqrt{n}$ hep öndedir.

1.1.2.6 Hızlı bilgisayar mı, iyi algoritma mı?

Yavaş bir programı hızlandırmanın iki yolu vardır: daha hızlı bir bilgisayar almak ya da daha iyi bir algoritma bulmak. Büyüme tablosu, bu iki yolun etkisinin çok farklı olduğunu gösterir.

Daha hızlı bir bilgisayar, her işlemi daha kısa sürede yapar. Bu, *T(n)*'deki sabit katsayıyı küçültmekle aynıdır. Bin kat hızlı bir makine, her algoritmayı bin kat hızlandırır. Ama büyüme biçimini değiştirmez. Daha iyi bir algoritma ise büyüme hızını değiştirir. Bir milyon öğede n^2 işlem yapan bir sıralama yerine $n \log n$ işlem yapan bir sıralama kullanmak, süreyi 16,7 dakikadan 20 milisaniyeye indirir. Bu yaklaşık elli bin katlık bir kazançtır ve *n* büyüdükçe daha da artar.

Aynı karşılaştırma ters yönden de yapılabilir: belirli bir sürede çözülebilecek en büyük girdi ne kadardır? Tablo 1.1.3, saniyede 10^9 işlem yapan bir bilgisayarın bir saniyede çözebileceği en büyük *n* değerlerini verir.

f(n)	Koşul	En büyük n
n	$n \le 10^9$	10^9
n log₂ n	$n \log_2 n \le 10^9$	$\approx 3{,}96 \cdot 10^7$
n^2	$n^2 \le 10^9$	31.622
n^3	$n^3 \le 10^9$	1000
2^n	$2^{29} \approx 5{,}4 \cdot 10^8 \le 10^9 < 2^{30}$	29
n!	$12! \approx 4{,}8 \cdot 10^8 \le 10^9 < 13!$	12

Tablo 1.1.3: Saniyede 10^9 işlemle bir saniyede çözülebilecek en büyük *n*.

Şimdi bilgisayarın bin kat hızlandığını düşünelim. Doğrusal algoritmada çözülebilen girdi de bin kat büyür. Karesel algoritmada yalnız $\sqrt{1000} \approx 31{,}6$ kat, kübik algoritmada 10 kat büyür. Üstel algoritmada ise $2^{10} \approx 1000$ olduğu için *n* yalnız 10 artar: 29'dan 39'a. Bin kat hızlı bir makine, üstel bir algoritmaya on öğe daha işleme imkânı verir. Bu yüzden algoritma tasarımında asıl kazanç, donanımdan değil, büyüme hızını düşüren fikirlerden gelir.

> **Sık yapılan hata**
>
> Bir algoritmanın hızını tek bir ölçümle, saniye cinsinden değerlendirmek. "Programım 0,2 saniyede bitti, demek ki hızlı" yargısı yalnız o makine ve o girdi boyutu için geçerlidir. Girdi on kat büyüdüğünde doğrusal bir algoritma 2 saniye, kübik bir algoritma 200 saniye sürer.

Analiz, saniyeye değil, *n*'ye bağlı işlem sayısına ve onun büyüme hızına bakar.

1.1.3 Karmaşıklık örnekleri

Büyüme tablosundaki her satırın arkasında gerçek algoritmalar vardır. Bu kesimde her sınıftan kısa bir Java kodu incelenir. Her kod için aynı soru sorulur: girdi boyutu *n* nedir, hangi işlem sayılır ve bu işlem *n*'ye bağlı olarak kaç kez yapılır? Kodların çoğu satır satır izlenir ve ara değerleri bir tabloda verilir.

1.1.3.1 Sabit zaman: ilk öğeyi yazdırma

En hızlı algoritmalar, girdi ne kadar büyük olursa olsun aynı sayıda işlem yapanlardır. Bir dizinin ilk öğesini yazdıran metot bu türdendir.

```
void ilkElemaniYazdir(int[] dizi) {
  if (dizi.length > 0)
    System.out.println("İlk eleman: "
                       + dizi[0]);
  else
    System.out.println("Dizi boş");
}
```

Metot önce dizinin boş olup olmadığını `dizi.length > 0` koşuluyla sınar. Dizi boş değilse `dizi[0]` hücresine erişip değeri yazar; boşsa “Dizi boş” iletisini yazar. Java'da bir dizinin uzunluğu dizi nesnesinde saklanır; `length` değerini okumak öğeleri saymayı gerektirmez. Dizinin herhangi bir hücresine erişmek de, hücrenin adresi başlangıç adresinden doğrudan hesaplandığı için tek adımdır.

Böylece dizi ister 10 öğeli, ister 10 milyon öğeli olsun, metot bir karşılaştırma, bir dizi erişimi ve bir yazdırma yapar. Örneğin `{5, 8, 12, 4}` dizisiyle çağrıldığında “İlk eleman: 5”, boş diziyle çağrıldığında “Dizi boş” yazar. İşlem sayısı *n*'den bağımsızdır: $T(n) = O(1)$.

Tanım 1.1.5

Çalışma süresi girdi boyutundan bağımsız, bir sabitle sınırlı olan algoritmaya **sabit zamanlı** (*constant time*) denir ve süresi $O(1)$ ile gösterilir. Süresi *n* ile doğru orantılı büyüyen algoritmaya **doğrusal zamanlı** (*linear time*) denir: $O(n)$.

$O(1)$ “tek işlem” demek değildir. Üç işlem yapan bir metot da, bin işlem yapan bir metot da, bu sayı *n*'ye bağlı değilse sabit zamanlıdır. Önemli olan, girdi büyüdüğünde işlem sayısının değişmemesidir.

1.1.3.2 Doğrusal zaman: en büyük öğeyi bulma

Sırasız bir dizinin en büyük öğesini bulmak için her öğeye en az bir kez bakmak gerekir. Bakılmayan bir öğe, en büyük öğe olabilir. Sunumdaki `enBuyukBul` metodu bu işi tek geçişte yapar.

```
Integer enBuyukBul(int[] dizi) {
  if (dizi.length == 0)
    return null;            // dizi boşsa
  int enBuyuk = dizi[0];
  for (int eleman : dizi)
    if (eleman > enBuyuk)
      enBuyuk = eleman;
  return enBuyuk;
}
```

Metodun dönüş tipi `int` değil `Integer`'dır. Bunun nedeni boş dizi durumudur: boş bir dizinin en büyük öğesi yoktur ve metot bunu `null` döndürerek bildirir. `int` tipi `null` değeri alamaz; `Integer` ise bir nesne referansıdır ve alabilir.

Dizi boş değilse `enBuyuk` değişkeni ilk öğeyle başlatılır. Ardından `for (int eleman : dizi)` döngüsü dizinin bütün öğelerini sırayla gezer. Bu "her biri için" biçimindeki döngü, indis kullanmadan dizinin öğelerini baştan sona verir; ilk öğe de bu gezintiye dahildir. Her öğe `enBuyuk` ile karşılaştırılır. Öğe daha büyükse `enBuyuk` güncellenir. Döngü bittiğinde `enBuyuk`, o ana kadar görülen bütün öğelerin en büyüğüdür; döngü bütün diziyi gezdiği için de dizinin en büyüğüdür.

Tablo 1.1.4, metodun `{5, 8, 12, 4}` dizisindeki çalışmasını verir; Şekil 1.1.2 aynı izlemeyi sunumdaki şekille gösterir.

Tur	eleman	eleman > enBuyuk?	enBuyuk (tur sonunda)
başlangıç	—	—	5
1	5	5 > 5 yanlış	5
2	8	8 > 5 doğru	8
3	12	12 > 8 doğru	12
4	4	4 > 12 yanlış	12

Tablo 1.1.4: `enBuyukBul({5, 8, 12, 4})` izlemesi.



Şekil 1.1.2: `enBuyukBul` her öğeye bir kez bakar. Üst satır dizinin öğeleri, alt satır o öğe incelendikten sonraki `enBuyuk` değeridir. 12 bulunduktan sonra değer değişmez.

Metot dört öğeli dizide dört karşılaştırma yapar ve 12 döndürür. Genel olarak n öğeli dizide tam n karşılaştırma yapılır; bu sayı dizinin içeriğine bağlı değildir. Güncelleme sayısı ise içeriğe bağlıdır. Dizi azalan sıradaysa (`{9, 5, 3, 2}`) hiç güncelleme olmaz. Artan sıradaysa (`{2, 3, 5, 9}`) her öğe öncekinden büyük olduğu için $n - 1$ güncelleme yapılır. Her iki durumda da toplam iş n ile orantılıdır: $T(n) = \Theta(n)$.

Metodun kullandığı ek bellek `enBuyuk` ve `eleman` değişkenlerinden ibarettir. Dizi ne kadar büyük olursa olsun bu iki değişken yeter. Bu yüzden alan karmaşıklığı $O(1)$'dir. Girdi dizisinin kendisi alan hesabına katılmaz; alan karmaşıklığı algoritmanın girdiye ek olarak ayırdığı belleği ölçer.

Sık yapılan hata

`enBuyuk` değişkenini `0` ile başlatmak. Dizideki bütün sayılar negatifse, örneğin `{-7, -3, -9}` , hiçbir öğe 0'dan büyük olmadığı için metot yanlış olarak 0 döndürür. Başlangıç değeri dizinin kendi öğesi (`dizi[0]`) olmalıdır. Sunumdaki kod bu yüzden önce boş diziyi ayırır, sonra `dizi[0]` ile başlar.

1.1.3.3 Karekök zaman: asallık testi

Tanım 1.1.6

1'den büyük olup 1 ve kendisinden başka pozitif böleni olmayan tam sayıya **asal sayı** denir. 2, 3, 5, 7, 11, 13 asaldır. 1'den büyük olup asal olmayan sayılara **bileşik sayı** denir: $9 = 3 \cdot 3$, $91 = 7 \cdot 13$.

Bir n sayısının asal olup olmadığını anlamanın en doğrudan yolu, 2'den $n - 1$'e kadar her sayının n'yi bölüp bölmediğine bakmaktır. Bu yöntem $n - 2$ bölme yapar; 97 için 95 bölme gerekir. Ama bu bölmelerin çoğu gereksizdir. Aşağıdaki teorem, aramanın çok daha erken bitirilebileceğini söyler.

Teorem 1.1.1 (Küçük bölen)

n > 1 bileşik bir sayıysa, *n*'nin $\sqrt{n}$'den küçük ya da ona eşit, 1'den büyük bir böleni vardır.

İspat. *Kanıt.* *n* bileşik olduğu için $1 < a, b < n$ olmak üzere $n = a \cdot b$ yazılabilir. İki çarpanın ikisi de $\sqrt{n}$'den büyük olsaydı çarpımları $\sqrt{n} \cdot \sqrt{n} = n$'den büyük olurdu. Bu, $a \cdot b = n$ ile çelişir. Demek ki *a* ile *b*'den en az biri $\sqrt{n}$'den küçük ya da ona eşittir ve bu çarpan aranan bölendir. ■

Teoremin karşıt tersi şunu söyler: $\sqrt{n}$'ye kadar hiçbir bölen yoksa *n* asaldır. Örneğin $91 = 7 \cdot 13$'te küçük çarpan 7'dir ve $7 \le \sqrt{91} \approx 9{,}5$'tir. Büyük çarpan 13'e hiç bakmaya gerek yoktur; 7 bulunduğunda sayının bileşik olduğu anlaşılır.

İkinci bir tasarruf daha yapılabilir. 2'den büyük bir çift sayı 2'ye bölündüğü için asal olamaz. Tek sayılar ise hiçbir çift sayıya bölünemez, çünkü çift bir bölen sayıyı da çift yapardı. Bu yüzden önce sayının çift olup olmadığına bakılır; tek sayılarda ise yalnız tek bölenler denenir. Sunumdaki `asalMi` metodu bu iki fikri birleştirir.

```
boolean asalMi(int n) {
  if (n <= 1)
    return false;
  else if (n == 2)
    return true;
  else if (n % 2 == 0)
    return false;
  int kokN = (int) Math.sqrt(n) + 1;
  for (int i = 3; i < kokN; i += 2)
    if (n % i == 0)
      return false;
  return true;
}
```

Metodun ilk üç dalı özel durumları ayırır. 1 ve daha küçük sayılar tanım gereği asal değildir. 2, çift olan tek asal sayıdır. 2'den büyük çift sayılar asal değildir. Bu üç dal döngüye girilmeden sonuç döndürür.

Tek sayılar için `kokN = (int) Math.sqrt(n) + 1` hesaplanır. `Math.sqrt` karekökü `double` olarak verir; `(int)` dönüşümü kesirli kısmı atar ve $\lfloor \sqrt{n} \rfloor$ değerini bırakır. Döngü koşulu `i < kokN`, yani $i \le \lfloor \sqrt{n} \rfloor$ olduğu için $\sqrt{n}$'nin tam kısmı da denenir. Döngü 3'ten başlar ve `i += 2` ile yalnız tek sayıları dener. Bir bölen bulunursa metot hemen `false` döndürür. Döngü bölen bulamadan biterse sayı asaldır ve metot `true` döndürür.

Kaç bölme yapılır? En kötü durum, sayının asal olduğu durumdur, çünkü o zaman döngü sonuna kadar çalışır. Döngü 3'ten $\sqrt{n}$'ye kadar ikişer ikişer ilerler. Bu aralıkta yaklaşık $\sqrt{n} / 2$ tek sayı vardır. Sabit katsayı $1/2$ atıldığında $T(n) = O(\sqrt{n})$ bulunur. Birkaç somut değer bu sayıyı doğrular:

- *n = 97*: $\lfloor\sqrt{97}\rfloor = 9$; denenen bölenler 3, 5, 7, 9: **4 bölme**. 2'den 96'ya kadar her sayı denenseydi 95 bölme gerekirdi.
- *n = 1.000.000.007* (asal): $\sqrt{n} \approx 31.622{,}8$; denenen tek sayılar 3, 5, …, 31.621: **15.810 bölme**. Basit yöntem ise yaklaşık bir milyar bölme yapardı.

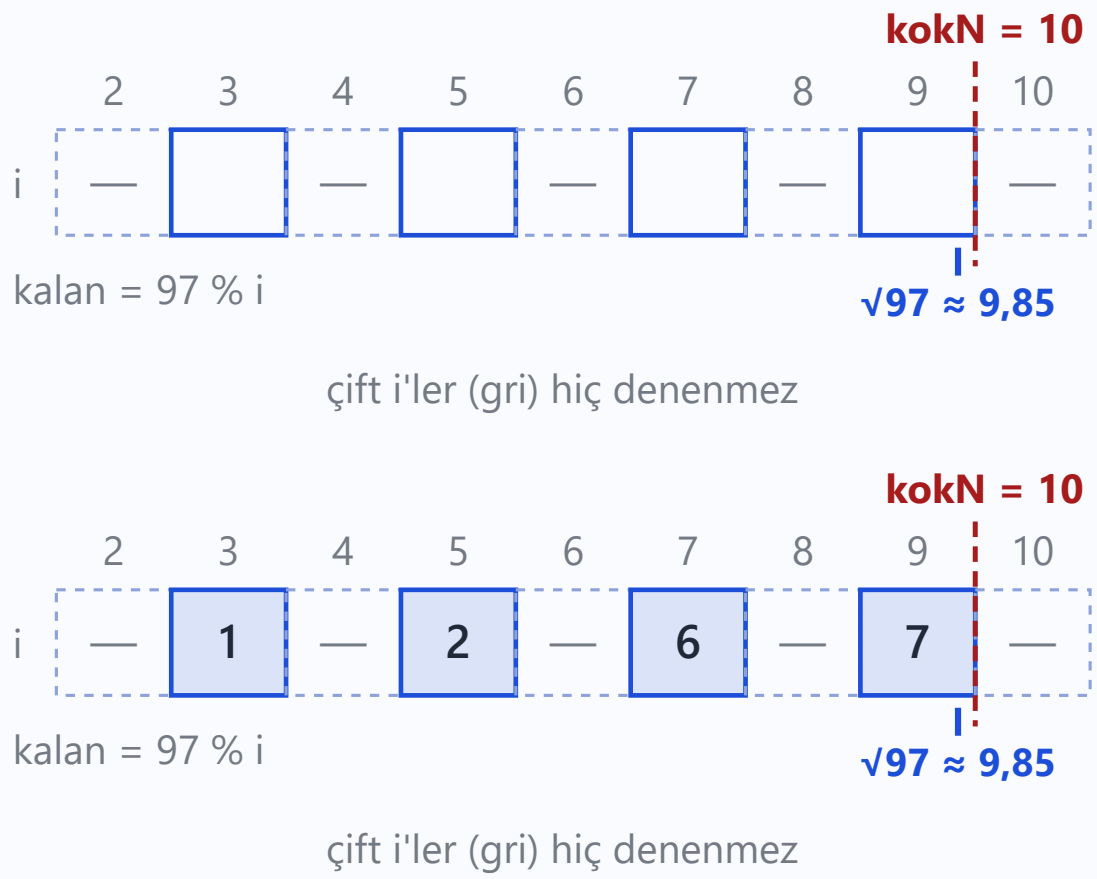
Sayı bileşikse döngü genellikle çok daha erken biter. 221 için $\lfloor\sqrt{221}\rfloor = 14$ ve `kokN = 15` 'tir; 3, 5, 7, 9, 11 bölmez, 13 böler ($221 = 13 \cdot 17$) ve metot 6 bölmeden sonra `false` döndürür.

1.1.3.4 Kod izleme: asalMi(97)

Sunumun izleme slaytı, `asalMi(97)` çağrısını satır satır yürütür. Tablo 1.1.5 bu izlemenin değişken değerlerini, Şekil 1.1.3 ise başlangıçtaki ve sondaki durumu gösterir.

Adım	Satır	kokN	i	97 % i	Açıklama
1	2, 4, 6	—	—	—	97 > 1, 97 ≠ 2, 97 tek: üç kontrol de geçilir
2	8	10	—	—	(int) √97 + 1 = 9 + 1 = 10
3	9, 10	10	3	1	3 bölmez
4	9, 10	10	5	2	5 bölmez
5	9, 10	10	7	6	7 bölmez
6	9, 10	10	9	7	9 bölmez
7	9	10	11	—	11 < 10 yanlış: döngü biter
8	12	10	11	—	`return true` : 97 asaldır

Tablo 1.1.5: `asalMi(97)` izlemesi. Satır numaraları kodun satırlarıdır.



Şekil 1.1.3: `asalMi(97)` izlemesinin iki anı. (a) `kokN = 10` hesaplanmıştır; kesikli dikey çizgi sınırı, $\sqrt{97} \approx 9{,}85$ etiketi gerçek karekökü gösterir. Gri kutular çift sayılardır ve hiç denenmez. (b) Döngü bitmiştir: 3, 5, 7 ve 9'un altında kalanlar (1, 2, 6, 7) yazılıdır; hiçbiri 0 değildir.

İzlemede dikkat çeken nokta, *i = 11* adımıdır. `i += 2` artırması 9'dan sonra `i`'yi 11 yapar. Döngü koşulu `11 < 10` yanlış olduğu için gövde çalışmaz; 11 hiç denenmez. 10'dan büyük bir tek sayıyı denemeye de gerek yoktur. 97'nin $\sqrt{97}$'den büyük bir böleni olsaydı, Teorem 1.1.1 gereği küçük bir böleni de olurdu. O küçük bölen ise zaten denenmişti.

Sık yapılan hata

Döngü sınırını `i < (int) Math.sqrt(n)` diye kurmak. Bu durumda $\lfloor\sqrt{n}\rfloor$'in kendisi denenmez. Kare sayılarda tek küçük bölen tam $\sqrt{n}$ olabilir: 9, 25, 49, 121 asal çıkar. Hata yalnız kare sayılarla da sınırlı değildir. 15 ($\lfloor\sqrt{15}\rfloor = 3$) ve 35 ($\lfloor\sqrt{35}\rfloor = 5$) gibi sayılarda küçük bölen tam $\lfloor\sqrt{n}\rfloor$'e eşittir ve o da atlanır. 100.000'den küçük sayılarda bu hatalı sürüm 84 sayı için yanlış sonuç verir. Sunumdaki `+ 1`, sınırın kendisini döngüye katar.

1.1.3.5 Girdi boyutu: sayının değeri mi, uzunluğu mu?

Asallık testinde girdi boyutu olarak n sayısının *değeri* alındı ve süre $O(\sqrt{n})$ bulundu. Bu ilk bakışta doğrusal süreden bile iyi görünür. Ama bir sayının bilgisayardaki büyüklüğü, değeriyle değil, kaç bitle yazıldığıyla ölçülür. 1.000.000.007 sayısı bir milyar değerindedir, ama yalnız 30 bitle yazılır.

Sayının bit uzunluğu b olsun. O zaman n yaklaşık 2^b'dir ve

> $\sqrt{n} \approx \sqrt{(2^b)} = 2^{b/2}$

bulunur. Yani bölme sayısı, bit uzunluğuna göre *üstel* büyür. Sayıya bir bit eklemek değerini iki katına, bölme sayısını ise yaklaşık $\sqrt{2} \approx 1{,}41$ katına çıkarır. 30 bitlik bir sayı için 15.810 bölme kolaydır. Kriptografide kullanılan 1024 bitlik bir sayı için ise 2^{512} mertebesinde bölme gerekir; bu, büyüme tablosundaki “evrenin yaşından çok” satırına düşer.

Bu gözlem, karmaşıklık hesabında girdi boyutunun dikkatle seçilmesi gerektiğini gösterir. Bir dizi işleyen algoritmada öğe sayısı doğal bir ölçüdür. Tek bir sayıyla çalışan algoritmalarda ise iki ölçü çok farklı sonuç verir. Karmaşıklık kuramı, girdi boyutunu her zaman girdiyi yazmak için gereken bit sayısı olarak alır. Değere göre polinom, ama bit uzunluğuna göre üstel olan bu tür sürelere **sözde-polinom** (*pseudo-polynomial*) süre denir. Büyük sayıların asallığını hızla sınayan yöntemler bu yüzden deneme bölmesinden tamamen farklı fikirlere dayanır.

1.1.3.6 Kübik zaman: matris çarpımı

İki matrisin çarpımı, bilimsel hesaplamadan bilgisayar grafiğine ve yapay sinir ağlarına kadar birçok alanda temel işlemdir. $n \times n$ boyutlu $A = [a_{ij}]$ ve $B = [b_{ij}]$ matrislerinin çarpımı $C = A \cdot B$'nin her öğesi şu formülle tanımlanır:

> $c_{ij} = \Sigma_{k=1..n}\, a_{ik} \cdot b_{kj} \quad (i, j = 1, 2, \ldots, n)$

Yani c_{ij}, A'nın i. satırı ile B'nin j. sütununun karşılıklı öğelerinin çarpımlarının toplamıdır. Sunumdaki örnekte

1	2	·	5	6	=	19	22
3	4		7	8		43	50

bulunur. Örneğin $c_{11} = 1 \cdot 5 + 2 \cdot 7 = 19$, $c_{21} = 3 \cdot 5 + 4 \cdot 7 = 43$'tür. Bu tanım doğrudan üç iç içe döngüye dönüşür:

```
int[][] matrisCarp(int[][] A, int[][] B) {
  int n = A.length, m = B[0].length, p = B.length;
  int[][] sonuc = new int[n][m];
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
      int eleman = 0;
      for (int k = 0; k < p; k++)
        eleman += A[i][k] * B[k][j];
      sonuc[i][j] = eleman;
    }
  return sonuc;
}
```

Kod yalnız kare matrislerle sınırlı değildir. `n`, `A`'nın satır sayısıdır; `m`, `B`'nin sütun sayısıdır; `p` ise `B`'nin satır sayısıdır ve `A`'nın sütun sayısına eşit olmalıdır. Sonuç *n × m* boyutundadır. Java'da kod, matrisin indislerini 0'dan başlatır; formüldeki 1'den başlayan indislerle arada bir kayma vardır, ama hesap aynıdır.

Dış iki döngü sonuç matrisinin her *(i, j)* hücresini bir kez ziyaret eder. Her hücre için `eleman` sıfırlanır, en içteki döngü `p` çarpım toplar ve sonuç `sonuc[i][j]`'ye yazılır. En içteki satır `eleman += A[i][k] * B[k][j]` toplam $n \cdot m \cdot p$ kez çalışır. Kare matrislerde $n = m = p$ olduğu için bu sayı n^3'tür:

> $T(n) = n \cdot n \cdot n = n^3$ çarpma $\Rightarrow T(n) = \Theta(n^3)$

Gerekçe kısaca şudur: sonuçta n^2 öğe vardır ve her öğe n çarpma gerektirir. *n = 100* için bir milyon, *n = 200* için sekiz milyon çarpma yapılır. n'nin iki katına çıkması süreyi sekiz katına çıkarır; kübik büyümenin işareti budur. Sonuç matrisi n^2 hücre tuttuğu için alan karmaşıklığı $\Theta(n^2)$'dir.

Bu düz yöntem matris çarpımının en hızlı yolu değildir. Böl ve fethet fikriyle çalışan Strassen algoritması aynı işi yaklaşık $n^{2{,}81}$ çarpmayla yapar. Ama düz yöntem hem en anlaşılır olanıdır hem de küçük ve orta boy matrislerde pratikte hâlâ çok kullanılır.

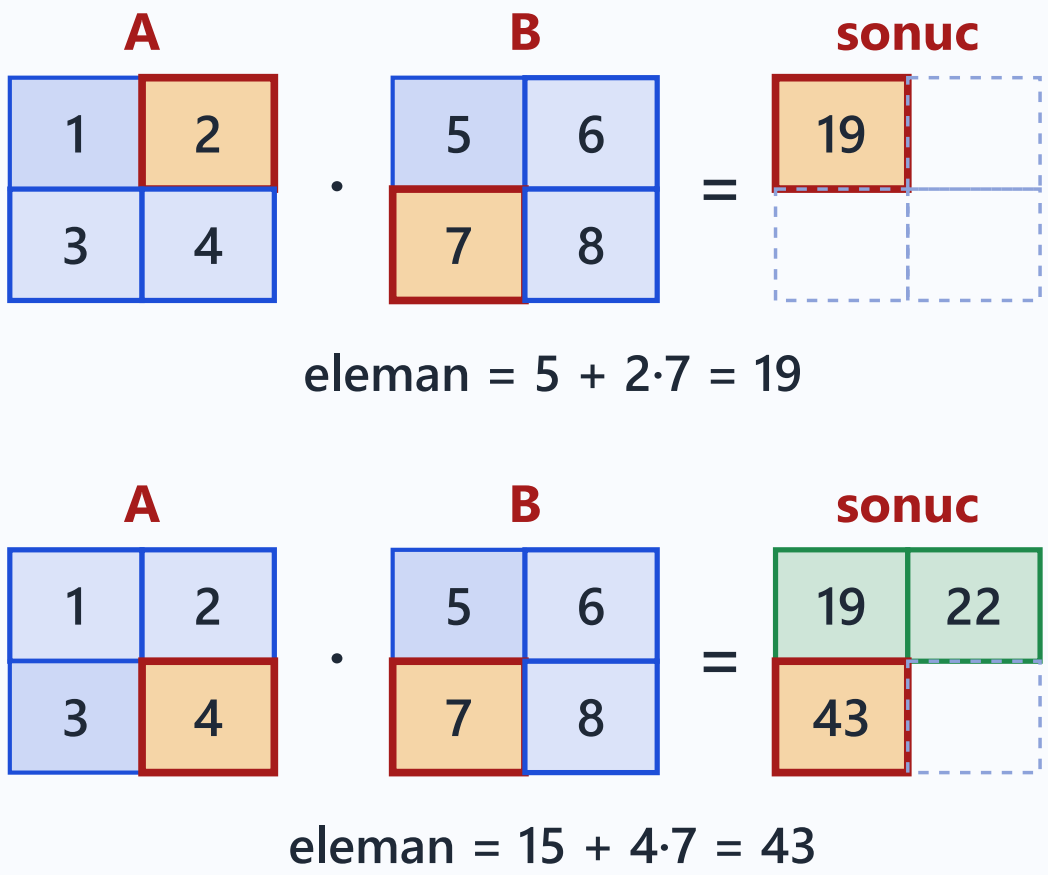
1.1.3.7 Kod izleme: 2 × 2 matris çarpımı

`matrisCarp`, sunumdaki *2 × 2* matrislerle çağrıldığında *n = m = p = 2* olur ve sonuç *2 × 2* sıfır matrisi olarak oluşturulur. Tablo 1.1.6 her çarpmadan sonraki değişken değerlerini verir.

i	j	k	A[i][k] · B[k][j]	eleman	çarpma	Yazılan hücre
0	0	0	1 · 5 = 5	5	1	—
0	0	1	2 · 7 = 14	19	2	sonuc[0][0] = 19
0	1	0	1 · 6 = 6	6	3	—
0	1	1	2 · 8 = 16	22	4	sonuc[0][1] = 22
1	0	0	3 · 5 = 15	15	5	—

i	j	k	A[i][k] · B[k][j]	eleman	çarpma	Yazılan hücre
1	0	1	4 · 7 = 28	43	6	sonuc[1][0] = 43
1	1	0	3 · 6 = 18	18	7	—
1	1	1	4 · 8 = 32	50	8	sonuc[1][1] = 50

Tablo 1.1.6: `matrisCarp` izlemesi, *A = [[1, 2], [3, 4]]*, *B = [[5, 6], [7, 8]]*.



Şekil 1.1.4: İzlemeden iki an. Kırmızı çerçeveli turuncu hücreler o anda çarpılan öğeleri, mavi dolgulu hücreler aynı satır ve sütunun öteki öğelerini gösterir. (a) *i = 0, j = 0, k = 1*: *eleman = 5 + 2 · 7 = 19*. (b) *i = 1, j = 0, k = 1*: *eleman = 15 + 4 · 7 = 43*; yeşil hücreler daha önce tamamlanmıştır.

İzleme sonunda `sonuc = [[19, 22], [43, 50]]` döner ve tam $8 = 2^3$ çarpma yapılmış olur. Tablodaki desen genel durumu da gösterir: `i` ve `j` her sonuç hücresini bir kez seçer, `k` ise her seçimde n değer alır. 3×3 matrislerde aynı sayım $3^3 = 27$ çarpma verir.

Sık yapılan hata

Matris çarpımının n^2 süreli olduğunu düşünmek. Sonuç matrisinde n^2 öğe vardır ve her öğe bir kez yazılır; bu, işin n^2 olduğu izlenimini verir. Oysa her öğenin hesabı ayrı bir n uzunluklu döngüdür. İç içe döngü sayısı, en içteki işlemin kaç kez çalıştığını belirler: burada üç döngü, n^3

işlem.

1.1.3.8 Faktöriyel zaman: bütün permütasyonlar

Tanım 1.1.7

n öğenin her bir sıralanışına bir **permütasyon** denir. *n* farklı öğenin permütasyon sayısı $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$'dir (*n* faktöriyel).

Permütasyon sayısının neden *n!* olduğu, sıralamanın konum konum kurulmasıyla görülür. İlk konuma *n* öğeden herhangi biri konabilir. İkinci konum için geriye $n - 1$ seçenek kalır, üçüncüsü için $n - 2$, ve böyle devam eder. Son konuma kalan tek öğe yerleşir. Seçenekler çarpılınca *n!* elde edilir. Üç öğe için $3 \cdot 2 \cdot 1 = 6$ permütasyon vardır: 123, 132, 213, 231, 321, 312.

Bazı problemlerin en doğrudan çözümü bütün permütasyonları tek tek denemektir. Örneğin bir gezginin *n* şehri en kısa toplam yolla dolaşma sırası, bütün sıralar denenerek bulunabilir. Sunumdaki `permutasyon` metodu, bir dizinin bütün permütasyonlarını yazdırır.

```
void permutasyon(int[] dizi, int k) {
  if (k == dizi.length) {          // bir permütasyon hazır
    System.out.println(Arrays.toString(dizi));
    return;
  }
  for (int i = k; i < dizi.length; i++) {
    takas(dizi, k, i);             // i. elemanı k. konuma al
    permutasyon(dizi, k + 1);      // geri kalanları diz
    takas(dizi, k, i);             // eski hâline getir
  }
}
```

Metot özyinelemelidir ve `k` parametresi, dizinin hangi konumunun doldurulmakta olduğunu söyler. 0'dan `k - 1`'e kadar olan konumlar yerleşmiştir; `k`'den sona kadar olan öğeler ise henüz sıralanmamıştır.

- **Taban durum:** `k == dizi.length` olduğunda bütün konumlar dolmuştur. Dizi o anda bir permütasyondur ve yazdırılır.
- **Özyinelemeli adım:** `k`. konuma sırayla `k`, `k + 1`, ..., son konumdaki öğeler getirilir. Bunun için `takas(dizi, k, i)` ile `i`. öğe `k`. konuma alınır. Ardından `permutasyon(dizi, k + 1)` geri kalan konumları aynı yolla doldurur.
- **Geri alma:** özyinelemeli çağrı döndükten sonra ikinci `takas` diziyi eski hâline getirir. Böylece döngünün bir sonraki turu, `k`. konumdan sonraki öğelerin özgün sırasıyla başlar.

Kodda kullanılan `takas` yardımcı metodu, dizinin iki konumundaki değerleri yer değiştirir: `int t = dizi[a]; dizi[a] = dizi[b]; dizi[b] = t;`. `Arrays.toString` ise diziyi `[1, 2, 3]` biçiminde metne çevirir ve `java.util.Arrays` sınıfındandır.

Örnek 1.1.4

`permutasyon({1, 2, 3}, 0)` çağrısı izlenirse:

1. *k = 0, i = 0*: `takas(0, 0)` diziyi değiştirmez: [1, 2, 3]. Çağrı *k = 1*'e iner.
   - *k = 1, i = 1*: dizi [1, 2, 3] kalır; *k = 2*'de tek seçenek vardır ve *k = 3*'te **[1, 2, 3]** yazılır.
   - *k = 1, i = 2*: `takas(1, 2)` → [1, 3, 2]; **[1, 3, 2]** yazılır. Geri alma takası diziyi [1, 2, 3]'e döndürür.
2. *k = 0, i = 1*: `takas(0, 1)` → [2, 1, 3]. Aynı süreç **[2, 1, 3]** ve **[2, 3, 1]** yazdırır. Geri alma takası diziyi [1, 2, 3] yapar.
3. *k = 0, i = 2*: `takas(0, 2)` → [3, 2, 1]. Bu kez **[3, 2, 1]** ve **[3, 1, 2]** yazılır. Son takas diziyi yine [1, 2, 3]'e döndürür.
4. **Sonuç:** 6 satır yazılır ve metot döndüğünde dizi başlangıçtaki hâlindedir. Metot toplam 16 kez çağrılır: 1 (*k = 0*) + 3 (*k = 1*) + 6 (*k = 2*) + 6 (*k = 3*).

Çıkan sıranın 123, 132, 213, 231, 321, 312 olduğuna dikkat edilmeli. Bu, sözlük sırası değildir: 312 ile 321'in yeri terstir. Takas yöntemi her permütasyonu bir kez üretir, ama sözlük sırasını korumaz.

Kaç işlem yapılır? Yazılan satır sayısı tam *n!*'dir. Her satırda `Arrays.toString` *n* öğeyi metne çevirir. Bu yüzden toplam iş *Θ(n · n!)*'dir. Ara düğümlerdeki çağrılar bu sayıyı değiştirmez: çağrı sayısı *n!*'in sabit katıdır (yaklaşık $e \cdot n! \approx 2{,}72 \cdot n!$). Bu sınıf kısaca "*O(n!)* sınıfı" diye anılır. Tablo 1.1.7, faktöriyelin ne kadar hızlı büyüdüğünü gösterir.

n	n!	Saniyede 10^9 permütasyonla süre
3	6	anında
5	120	anında
10	3.628.800	yaklaşık 3,6 ms
15	$1{,}3 \cdot 10^{12}$	yaklaşık 22 dakika
20	$2{,}4 \cdot 10^{18}$	yaklaşık 77 yıl

Tablo 1.1.7: Permütasyon sayısı *n!*.

On öğenin permütasyonları bir göz açıp kapayıncaya kadar üretilir; yirmi öğeninkiler ise bir insan ömrü sürer. Bütün permütasyonları denemek, bu yüzden yalnız çok küçük girdilerde kullanılabilen bir yöntemdir. Daha büyük girdiler için geri izleme ve dal sınır gibi teknikler aramanın büyük kısmını budar; ya da problemin yapısından yararlanan bambaşka bir algoritma aranır.

Sık yapılan hata

İkinci `takas` çağrısını unutmak. Geri alma yapılmazsa dizi her turda bir öncekinin bıraktığı karışık hâlden devam eder. `{1, 2, 3}` için çıktı [1, 2, 3], [1, 3, 2], [3, 2, 1], [3, 2, 1], [1, 2, 3], [1, 3, 2] olur: 123 ve 132 iki kez yazılır, 213 ve 231 hiç üretilmez. Bir çağrının yaptığı değişikliği çağrı dönünce geri almak, geri izleme algoritmalarının da temel kuralıdır.

1.1.3.9 Üstel zaman: saf özyinelemeli Fibonacci

Fibonacci dizisi 0 ve 1 ile başlar; sonraki her terim önceki iki terimin toplamıdır: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … Tanım şöyle yazılır:

> $F(0) = 0,\ \ F(1) = 1,\ \ F(n) = F(n-1) + F(n-2)\ \ (n \geq 2)$

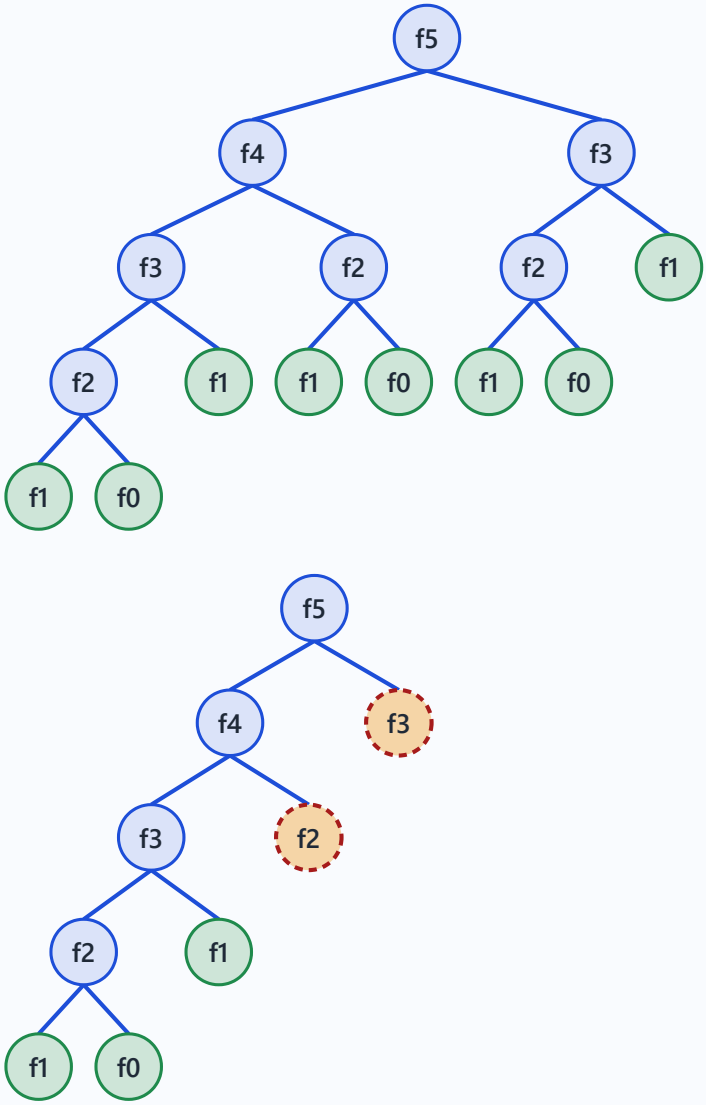
Bu tanım özyinelemelidir ve doğrudan bir Java metoduna dönüştürülebilir. Sunumdaki kodun ilk metodu `fib` bunu yapar; ikinci metot `fibBellek` ise bir sonraki alt kesimin konusudur.

```
long fib(int n) {           // saf özyineleme
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

long[] bellek = new long[N + 1]; // 0: yok
long fibBellek(int n) {     // bellekleme
  if (n <= 1) return n;
  if (bellek[n] == 0)
    bellek[n] = fibBellek(n - 1) + fibBellek(n - 2);
  return bellek[n];
}
```

`fib` metodu tanımın birebir kopyasıdır. `n ≤ 1` taban durumdur ve `n` değerini döndürür: *F(0) = 0*, *F(1) = 1*. Aksi hâlde metot kendini iki kez çağırır ve sonuçları toplar. Dönüş tipi `long`'dur, çünkü Fibonacci sayıları çabuk büyür: *F(47)* artık `int`'e sığmaz.

Kod kısa ve doğrudur, ama çok yavaştır. Nedeni Şekil 1.1.5'teki çağrı ağacında açıkça görülür. `fib(5)`, `fib(4)` ve `fib(3)`'ü çağırır. `fib(4)` de kendi içinde yine `fib(3)`'ü çağırır. Böylece `fib(3)` iki kez, baştan sona ayrı ayrı hesaplanır. Aşağı indikçe tekrar artar: `fib(2)` üç kez, `fib(1)` beş kez çağrılır.



Şekil 1.1.5: (a) `fib(5)` 'in çağrı ağacı: 15 çağrı. Yeşil düğümler taban durumlardır (`f1` , `f0`). Sol alt ağaçla sağ alt ağaç aynı alt problemleri baştan çözer. (b) `fibBellek(5)` 'in çağrı ağacı: 9 çağrı. Turuncu kesikli düğümler (`f2` , `f3`) sonucu bellek dizisinden hazır alır; alt ağaçları hiç açılmaz.

Kaç çağrı yapılır? `fib(n)` 'nin yaptığı toplam çağrı sayısına *Ç(n)* diyelim. Taban durumlarda yalnız çağrının kendisi vardır: *Ç(0) = Ç(1) = 1*. Öteki durumlarda çağrının kendisine iki alt çağrının sayıları eklenir:

> Ç(n) = 1 + Ç(n − 1) + Ç(n − 2)

Bu bağıntı, Fibonacci bağıntısına çok benzer. Değerleri sırayla hesaplanırsa *Ç(0..6)* = 1, 1, 3, 5, 9, 15, 25 bulunur. Tümevarımla *Ç(n) = 2F(n + 1) − 1* olduğu gösterilebilir. *n = 5* için *2 · 8 − 1 = 15*, *n = 30* için *2 · 1.346.269 − 1 = 2.692.537* çağrı. `fib(40)` ise 331.160.281 çağrı yapar; bu, 40'ıncı Fibonacci sayısını bulmak için üç yüz milyondan fazla metot çağrısı demektir.

Büyüme hızı. Her çağrı en fazla iki çağrı doğurur ve ağacın derinliği en fazla n'dir. Derinliği n olan bir ikili ağaçta en fazla 2^n düğüm bulunabilir. Bu yüzden $T(n) = O(2^n)$'dir. Gerçek büyüme biraz daha yavaştır, çünkü sağ dallar sol dallardan kısadır. $Ç(n)$, $F(n)$ ile aynı hızda büyür ve $F(n)$ her adımda yaklaşık altın oran $\varphi = (1 + \sqrt{5}) / 2 \approx 1{,}618$ katına çıkar. Kesin sınıf $\Theta(\varphi^n) = \Theta(1{,}618^n)$'dir. Taban 2 değil 1,618 olsa da büyüme yine üsteldir: n bir artınca süre yaklaşık yüzde 62 artar, n on artınca yüz yirmi kattan fazla artar.

1.1.3.10 Bellekleme: aynı alt problemi bir kez çözmek

Saf Fibonacci kodunun yavaşlığı, aynı alt problemin defalarca çözülmesinden gelir. `fib(30)` hesaplanırken $F(2)$ yüz binlerce kez yeniden bulunur. Oysa $F(2)$ her seferinde aynı değerdir. Bir kez hesaplanıp bir yere yazılsa, sonraki her istekte oradan okunabilir.

Tanım 1.1.8

Bellekleme (*memoization*), özyinelemeli bir algoritmada hesaplanan alt problem sonuçlarını bir tabloda saklayıp aynı alt problem yeniden istendiğinde hesap yapmadan tablodan döndürme tekniğidir.

Sunumdaki `fibBellek` metodu, `bellek` adlı bir `long` dizisi kullanır. Dizinin k. hücresi $F(k)$'yi tutar. Java'da yeni bir `long` dizisinin bütün hücreleri 0 ile başlar. 0 değeri burada "henüz hesaplanmadı" anlamında kullanılır. Bu seçim güvenlidir, çünkü $k \geq 1$ için $F(k)$ hiçbir zaman 0 değildir; $F(0) = 0$ ise taban durumda ayrıca ele alınır. `N` sabiti, hesaplanacak en büyük n'dir ve dizi $N + 1$ hücreyle ayrılır.

Metot önce taban durumu sınar. Sonra `bellek[n] == 0` ise, yani $F(n)$ daha önce hesaplanmamışsa, onu iki özyinelemeli çağrıyla hesaplar ve diziye yazar. Her durumda sonucu diziden döndürür. Böylece her $F(k)$ yalnız bir kez hesaplanır.

Örnek 1.1.5

`fibBellek(5)` çağrısı, `bellek` dizisi sıfırken izlenirse:

1. `fibBellek(5)` → `bellek[5] == 0`; önce `fibBellek(4)` çağrılır. Aynı biçimde 4 → 3 → 2 zinciri iner.
2. `fibBellek(2)`, `fibBellek(1) = 1` ve `fibBellek(0) = 0` çağrılarını yapar; `bellek[2] = 1` yazılır.
3. `fibBellek(3)`'e dönülür; ikinci çağrı `fibBellek(1) = 1`'dir. `bellek[3] = 1 + 1 = 2`.
4. `fibBellek(4)`'e dönülür; ikinci çağrı `fibBellek(2)`'dir. `bellek[2]` artık 1'dir; çağrı hesap yapmadan 1 döndürür. `bellek[4] = 2 + 1 = 3`.
5. `fibBellek(5)`'e dönülür; ikinci çağrı `fibBellek(3)` bellekten 2 döndürür. `bellek[5] = 3 + 2 = 5`.

6. **Sonuç:** *F(5) = 5*. Toplam 9 çağrı yapılmıştır: `f5, f4, f3, f2, f1, f0, f1, f2, f3`. Son ikisi bellekten dönen çağrılardır.

Kaç çağrı? $n \geq 1$ için `fibBellek(n)`, 2'den n'ye kadar her k için tam iki alt çağrı yapar. Birincisi $F(k-1)$'i hesaplar. İkincisi $F(k-2)$'yi ister; bu çağrı ya bellekten döner ya da taban durumdur. İlk çağrı da eklenince toplam $2(n-1)+1 = 2n-1$ çağrı bulunur. $n = 5$ için 9, $n = 30$ için 59 çağrı. Saf yöntemin 2.692.537 çağrısıyla karşılaştırıldığında fark yaklaşık kırk beş bin kattır. Çalışma süresi $O(n)$'ye iner.

Bu kazancın bir bedeli vardır: `bellek` dizisi $n+1$ hücre tutar. Ayrıca özyineleme n derinliğe indiği için çağrı yığını da $O(n)$ yer kaplar. Bellekleme, zamandan kazanmak için bellek harcar. Zaman ile bellek arasındaki bu değiş tokuş, algoritma tasarımında sık görülür. Aynı fikrin aşağıdan yukarı, döngüyle yazılmış biçimi dinamik programlamanın temelidir.

Sık yapılan hata

Özyinelemeli bir tanımı, alt problemlerin tekrar edip etmediğine bakmadan doğrudan koda dökmek. Saf `fib` doğru sonuç verir, ama $O(2^n)$ süreyle. Bir özyinelemenin çağrı ağacında aynı argümanla birden çok çağrı görülüyorsa, sonuçları saklamak süreyi çoğu zaman üstelden polinoma indirir.

1.1.4 Öklid algoritması

Öklid algoritması, bilinen en eski algoritmalardan biridir. Yunan matematikçi Öklid, MÖ 300 dolaylarında yazdığı *Öğeler* adlı eserinde bu yöntemi anlatır. Algoritma yaklaşık iki bin üç yüz yıldır kullanılır ve bugün de kriptografiden kesirlerin sadeleştirilmesine kadar birçok yerde iş görür. Kısa, doğruluğu kolay kanıtlanan ve şaşırtıcı ölçüde hızlı bir algoritma olduğu için analiz öğrenmeye de çok uygundur.

1.1.4.1 EBOB ve Öklid'in fikri

Tanım 1.1.9

İki tam sayının ikisini de kalansız bölen en büyük pozitif tam sayıya bu sayıların **en büyük ortak böleni** (EBOB; *greatest common divisor*, gcd) denir ve $ebob(a, b)$ ile gösterilir. Örneğin $ebob(12, 18) = 6$, $ebob(17, 5) = 1$.

EBOB'u bulmanın en doğrudan yolu, $min(a, b)$'den başlayıp aşağı doğru her sayıyı denemektir. İki sayıyı da bölen ilk sayı EBOB'dur. Bu yöntem en kötü durumda $min(a, b)$ kadar deneme yapar: $O(min(a, b))$. Bir başka yol, iki sayıyı asal çarpanlarına ayırıp ortak çarpanları çarpmaktır; ama büyük sayıları çarpanlarına ayırmak zaten zor bir problemdir.

Öklid'in fikri, büyük sayıyı küçüğe bölüp kalanla devam etmektir. Temel eşitlik şudur:

> ebob(a, b) = ebob(b, a mod b), ebob(a, 0) = a

Burada *a mod b*, *a*'nın *b*'ye bölümünden kalandır. Algoritma üç adımla anlatılır:

1. Büyük sayı küçüğe bölünür ve kalan bulunur.
2. Kalan sıfır olana kadar işlem (bölen, kalan) çiftiyle tekrarlanır.
3. Kalan sıfır olduğunda son bölen EBOB'dur.

Eşitlik doğrudan özyinelemeli bir metoda dönüşür:

```
int ebob(int a, int b) {
  if (b == 0)
    return a;
  return ebob(b, a % b);
}
```

`b == 0` taban durumdur; *ebob(a, 0) = a*, çünkü her sayı 0'ı böler ve *a*'yı bölen en büyük sayı *a*'nın kendisidir. Aksi hâlde metot kendini `(b, a % b)` ile çağırır. Java'da `%` işleci kalanı verir.

Metot, *a < b* olduğunda da doğru çalışır. Örneğin `ebob(650, 1599)` çağrısında *650 % 1599 = 650*'dir ve ilk çağrı `ebob(1599, 650)` olur. Sayılar kendiliğinden yer değiştirir; bedeli yalnız bir fazla bölmedir.

1.1.4.2 Algoritma neden doğru?

Teorem 1.1.2 (Öklid)

a ≥ 0 ve *b > 0* tam sayılar olsun. *a* ile *b*'nin ortak bölenleri, *b* ile *a mod b*'nin ortak bölenleriyle aynıdır. Bu yüzden *ebob(a, b) = ebob(b, a mod b)*'dir.

İspat. *Kanıt.* *a*'yı *b*'ye bölelim: *a = q · b + r*, burada *q* bölüm, *r = a mod b* kalandır ve *0 ≤ r < b*'dir.

(1) *d*, hem *a*'yı hem *b*'yi bölsün. O zaman *d*, *q · b*'yi de böler ve *r = a − q · b* farkını da böler. Demek ki *a* ile *b*'nin her ortak böleni, *b* ile *r*'nin de ortak bölenidir.

(2) Tersine, *d* hem *b*'yi hem *r*'yi bölsün. O zaman *a = q · b + r* toplamını da böler. Demek ki *b* ile *r*'nin her ortak böleni, *a* ile *b*'nin de ortak bölenidir.

İki küme aynı olduğuna göre en büyük öğeleri de aynıdır. ■

Teorem, her özyinelemeli çağrının EBOB'u koruduğunu söyler. Sonluluk da kolay gösterilir. Her çağrıda ikinci argüman b'den $a \bmod b$'ye geçer ve $a \bmod b < b$'dir. Negatif olmayan tam sayılar sonsuza kadar küçülemeyeceği için ikinci argüman sonunda 0 olur ve taban duruma ulaşılır. Taban durumda döndürülen a, doğru EBOB'dur. Böylece algoritma hem durur hem de doğru sonuç verir.

1.1.4.3 Kod izleme: ebob(1599, 650)

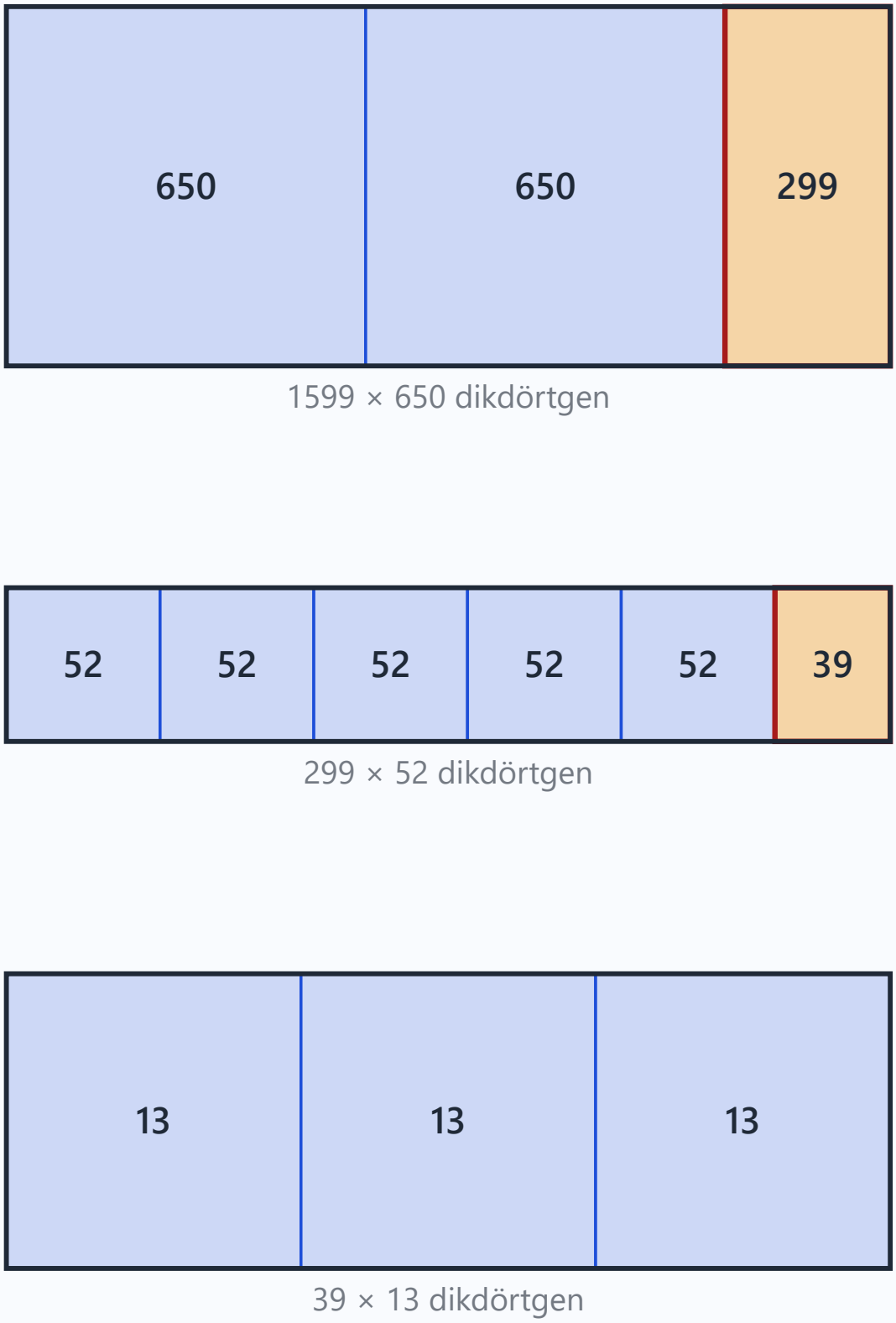
Sunumun izleme örneği `ebob(1599, 650)`'dir. Tablo 1.1.8 her çağrıdaki bölmeyi verir.

Çağrı	a	b	Bölme	a % b
1	1599	650	1599 = 2 · 650 + 299	299
2	650	299	650 = 2 · 299 + 52	52
3	299	52	299 = 5 · 52 + 39	39
4	52	39	52 = 1 · 39 + 13	13
5	39	13	39 = 3 · 13 + 0	0
6	13	0	`b == 0`: `return 13`	—

Tablo 1.1.8: `ebob(1599, 650)` izlemesi.

Beş bölmeden sonra kalan 0 olur ve altıncı çağrı taban durumdur. 13 değeri, hiç değişmeden bütün çağrılardan geri döner; her çağrı yalnız alt çağrısının sonucunu iletir. Sonuç *ebob(1599, 650) = 13*'tür. Çarpanlara ayırma da bunu doğrular: $1599 = 3 \cdot 13 \cdot 41$, $650 = 2 \cdot 5^2 \cdot 13$; tek ortak çarpan 13'tür.

Sunumdaki şekiller algoritmanın geometrik bir yorumunu verir. $a \times b$ boyutlu bir dikdörtgenden kenarı b olan kareler kesilir. Kaç kare sığarsa (q tane) kesilir ve geriye $b \times r$ boyutlu bir şerit kalır. Aynı işlem bu şeride uygulanır. Şerit sonunda karelere tam bölündüğünde son karenin kenarı EBOB'dur. Şekil 1.1.6 bu sürecin üç adımını gösterir.



Şekil 1.1.6: Öklid algoritmasının kare kesme yorumu. (a) *1599 × 650* dikdörtgenden iki *650 × 650* kare kesilir; turuncu şerit *650 × 299* kalır. (b) *299 × 52* dikdörtgenden beş kare kesilir; kalan şerit *52 × 39*'dur. (c) *39 × 13* dikdörtgen üç kareye tam bölünür; artık kalmaz. Kenarı 13 olan kareler, ilk dikdörtgeni de boşluksuz döşer.

Son gözlem EBOB'un anlamını da açıklar. Kenarı 13 olan kareler *39 × 13*'ü döşer. Bu kareler *52 × 39*'u da döşer, çünkü 52 de 39 da 13'ün katıdır. Geriye doğru her adımda aynı akıl yürütme işler ve sonunda *1599 × 650* dikdörtgeni de *123 × 50* tane *13 × 13* kareyle döşenir. Bu dikdörtgeni döşeyebilen en büyük kare budur.

Özyinelemeli çağrılar çağrı yığınında birikir. Taban duruma ulaşıldığında yığında altı çerçeve vardır: `ebob(1599, 650)`'den `ebob(13, 0)`'a kadar. Bu derinlik bölme sayısıyla orantılı olduğu için özyinelemeli sürümün alan karmaşıklığı, bölme sayısıyla aynı mertebededir.

1.1.4.4 Algoritma neden hızlı?

`ebob(1599, 650)` yalnız beş bölmeyle bitti. Deneme yöntemi ise 650'den aşağı doğru 13'e kadar 638 sayı denerdi. Bu fark rastlantı değildir. Öklid algoritmasında sayılar çok hızlı küçülür ve bunu aşağıdaki gözlem güvenceye alır.

> **Teorem 1.1.3 (Yarılama)**
>
> $a \geq b > 0$ ise $a \bmod b < a / 2$'dir.

İspat. *Kanıt.* İki durum vardır. $b \leq a / 2$ ise kalan bölenden küçük olduğu için $a \bmod b < b \leq a / 2$'dir. $b > a / 2$ ise b, a'ya yalnız bir kez sığar; bölüm 1'dir ve kalan $a - b$'dir. $b > a / 2$ olduğu için $a - b < a / 2$ bulunur. İki durumda da kalan a'nın yarısından küçüktür. ∎

Algoritmanın ardışık iki çağrısına bakalım: $(a, b) \to (b, r) \to (r, b \bmod r)$. İki adım sonra ilk sayının yerinde $r = a \bmod b$ vardır ve teoreme göre $r < a / 2$'dir. Yani her iki adımda büyük sayı en az yarıya iner. Bir a sayısı en fazla $\log_2 a$ kez yarıya inebilir; sonra 1'in altına düşer. Bu yüzden bölme sayısı en fazla yaklaşık $2 \log_2 \min(a, b)$ olur. İlk adımdan sonra çiftin büyüğü zaten $\min(a, b)$'dir; bu yüzden sınırda küçük sayı yer alır. Çalışma süresi

```
T(a, b) = O(log min(a, b))
```

bulunur. Deneme yönteminin $O(\min(a, b))$ süresiyle karşılaştırıldığında fark büyüktür. Bir milyar büyüklüğündeki iki sayı için deneme yöntemi bir milyar deneme yapabilirken Öklid algoritması en fazla altmış kadar bölme yapar.

Burada girdi boyutu konusuna bir kez daha dönmek yararlıdır. Sayıların bit uzunluğu ℓ ise $\log_2 \min(a, b) \leq \ell$'dir. Yani Öklid algoritması bit uzunluğuna göre *doğrusal* sayıda bölme yapar. Asallık testindeki deneme bölmesi ise bit uzunluğuna göre üsteldi. Bu yüzden Öklid algoritması yüzlerce basamaklı sayılarda bile anında çalışır ve kriptografide rahatça kullanılır.

1.1.4.5 En kötü durum: ardışık Fibonacci sayıları

Yarılama sınırı bir üst sınırdır; algoritma çoğu girdide daha erken biter. Peki hangi girdiler algoritmayı en uzun çalıştırır? Sayıların en yavaş küçülmesi, her bölmede bölümün 1 çıkmasıyla olur. O zaman kalan, iki sayının farkıdır: $r = a - b$. Bu durumda geriye doğru bakıldığında her sayı, kendinden sonraki iki sayının toplamıdır. Bu, tam olarak Fibonacci dizisinin kuralıdır. En kötü girdi, ardışık iki Fibonacci sayısıdır.

Örnek 1.1.6

`ebob(55, 34)` izlenirse (55 ve 34, ardışık Fibonacci sayılarıdır):

1. $55 = 1 \cdot 34 + 21$
2. $34 = 1 \cdot 21 + 13$
3. $21 = 1 \cdot 13 + 8$
4. $13 = 1 \cdot 8 + 5$
5. $8 = 1 \cdot 5 + 3$
6. $5 = 1 \cdot 3 + 2$
7. $3 = 1 \cdot 2 + 1$
8. $2 = 2 \cdot 1 + 0$
9. **Sonuç:** 8 bölme, $ebob(55, 34) = 1$. Kalanlar Fibonacci dizisini geriye doğru izler: 21, 13, 8, 5, 3, 2, 1, 0. Son bölme dışında her bölüm 1'dir. Sayılar 55 ve 34 gibi küçük olmasına karşın bölme sayısı, çok daha büyük olan `ebob(1599, 650)` 'nin 5 bölmesini geçer.

Bu gözlemin kesin biçimi, 1844'te Gabriel Lamé'nin kanıtladığı teoremdir.

Teorem 1.1.4 (Lamé)

$a > b \geq 1$ için `ebob(a, b)` çağrısı k bölme yapıyorsa $b \geq F(k + 1)$'dir. Buna göre bölme sayısı, b'nin onluk tabandaki basamak sayısının beş katını geçmez.

İspat. *Kanıt taslağı.* Kalanlar $r_0 = a$, $r_1 = b$, r_2, ..., r_k, $r_{k+1} = 0$ olsun; i. bölme $r_{i-1} = q_i \cdot r_i + r_{i+1}$'dir. Son sıfırdan farklı kalan $r_k \geq 1 = F(2)$'dir. Son bölmede kalan 0 ve $r_{k-1} > r_k$ olduğu için $q_k \geq 2$ ve $r_{k-1} \geq 2 = F(3)$'tür. Öteki her bölmede $q_i \geq 1$ olduğu için $r_{i-1} \geq r_i + r_{i+1}$'dir. Bu eşitsizlik geriye doğru uygulandığında her kalan bir Fibonacci sayısından küçük olmaz ve $r_1 = b \geq F(k + 1)$ elde edilir. Fibonacci sayıları yaklaşık $\varphi \approx 1{,}618$ oranında büyüdüğü için k, $\log_\varphi b$ mertebesinde kalır. ■

Teorem, `ebob(55, 34)` örneğinde tam sınırdadır: 8 bölme yapılmıştır ve $b = 34 = F(9)$'dur. Yani 8 bölme yapan en küçük b, 34'tür. Bir sonraki çift `ebob(89, 55)` dokuz bölme yapar. 1000'den küçük bütün sayı çiftleri arasında en çok bölmeyi (14) `ebob(987, 610)` yapar; 987 ve 610 da ardışık

Fibonacci sayılarıdır. Böylece Öklid algoritmasının en kötü durumu bile logaritmiktir: *Θ(log min(a, b))*.

1.1.4.6 Döngüyle Öklid

Özyinelemeli `ebob`, kendini son işlem olarak çağırır ve dönüş değerini değiştirmeden iletir. Böyle bir özyineleme, doğrudan bir döngüye çevrilebilir: her turda *(a, b)* çifti *(b, a mod b)* ile değiştirilir.

```
int ebob(int a, int b) {
  while (b != 0) {
    int r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

Döngü `b` sıfır olana kadar döner. Her turda önce kalan `r` hesaplanır, sonra `a`'ya eski `b`, `b`'ye kalan atanır. Atamaların sırası önemlidir: `a = b` önce yapılmazsa eski `b` değeri kaybolur. Döngü bittiğinde `a`, son sıfırdan farklı kalandır, yani EBOB'dur. Tablo 1.1.9, `ebob(1071, 462)` çağrısının döngü turlarını verir.

Tur	a (başta)	b (başta)	r = a % b	Tur sonunda (a, b)
1	1071	462	147	(462, 147)
2	462	147	21	(147, 21)
3	147	21	0	(21, 0)
—	21	0	—	`b == 0` : `return 21`

Tablo 1.1.9: Döngülü `ebob(1071, 462)` izlemesi.

Bölmeler *1071 = 2 · 462 + 147*, *462 = 3 · 147 + 21* ve *147 = 7 · 21 + 0*'dır. Döngülü sürüm özyinelemeli sürümle aynı sayıda bölme yapar ve aynı süreye sahiptir: *O(log min(a, b))*. Farkı bellektedir. Özyinelemeli sürüm her çağrı için yığında bir çerçeve açar; döngülü sürüm ise yalnız üç değişken kullanır ve *O(1)* ek bellekle çalışır. Java derleyicisi kuyruk özyinelemeyi kendiliğinden döngüye çevirmediği için bu dönüşüm elle yapılır.

1.1.5 Sık yapılan hatalar

Notun farklı yerlerinde ele alınan hatalar burada bir arada toplanmıştır. Hepsi aynı kaynaktan doğar: kodun ya da analizin bir ayrıntısı, bütün girdiler düşünülmeden kurulur.

- **Süreyi saniyeyle ölçmek.** Saniye makineye, dile ve o anki yüke bağlıdır. Analizde n'ye bağlı işlem sayısı ve onun büyüme hızı kullanılır. Bir ölçüm, ancak farklı n değerlerinde tekrarlanıp büyüme biçimi görüldüğünde algoritma hakkında bilgi verir.
- **Belleklemeyi unutmak.** Saf özyinelemeli Fibonacci $O(2^n)$'dir, çünkü aynı alt problemleri defalarca çözer. Sonuçları bir dizide saklamak çağrı sayısını $2n - 1$'e, süreyi $O(n)$'ye indirir.
- **$\sqrt{n}$ sınırını eksik kurmak.** `i < (int) Math.sqrt(n)` yazılırsa $\lfloor\sqrt{n}\rfloor$'in kendisi denenmez; 9, 25, 49 gibi kare sayılar ve 15, 35 gibi küçük çarpanı tam $\lfloor\sqrt{n}\rfloor$'e eşit sayılar asal çıkar. Doğrusu `i <= (int) Math.sqrt(n)` ya da sunumdaki gibi `i < (int) Math.sqrt(n) + 1`'dir.
- **Girdi boyutunu karıştırmak.** Asallık testinde n sayının değeridir. b bitlik bir sayı için $\sqrt{n} = 2^{b/2}$ olur; yani deneme bölmesi bit sayısına göre üsteldir. Tek bir sayıyı girdi alan algoritmalarda "n" sözcüğünün neyi gösterdiği her zaman açıkça söylenmelidir.
- **Matris çarpımını n^2 sanmak.** Sonuçta n^2 öğe vardır, ama her öğe n çarpma ister: toplam n^3 çarpma. İç içe döngülerde en içteki işlemin toplam kaç kez çalıştığı sayılır.
- **Takası geri almamak.** Permütasyon üretiminde ikinci `takas` unutulursa dizi bozulur; `{1, 2, 3}` için bazı permütasyonlar iki kez yazılır, bazıları hiç üretilmez.
- **Başlangıç değerini girdiden bağımsız seçmek.** En büyük öğe aranırken `enBuyuk = 0` ile başlamak, bütün öğeleri negatif dizilerde yanlış sonuç verir. Başlangıç değeri dizinin kendi öğesi olmalıdır.
- **Tam sayı bölmesini gözden kaçırmak.** Java'da iki `int`'in bölümü kesirli kısmı atar. Ortalama gibi kesirli sonuç beklenen hesaplarda en az bir işlenen `double` olmalıdır.

1.1.6 Özet

- **Algoritma**, bir problemi çözmek için izlenen, sonlu sayıda ve açıkça tanımlı adımlardan oluşan yönergeler dizisidir. Girdi alır, çıktı üretir; adımları **açık**, **etkin** (yapılabilir) ve **sonlu**dur; her geçerli girdi için **doğru** sonuç verir. Algoritma dilden bağımsızdır; program onun belirli bir dildeki karşılığıdır.
- Algoritmalar sıralama, arama, çizge, dizgi ve sıkıştırma gibi problem gruplarına ayrılır. Böl ve fethet, dinamik programlama, açgözlü, geri izleme ve dal sınır ise tasarım teknikleridir. **Deterministik** algoritma aynı girdide hep aynı yolu izler; **rastgele** algoritma adımlarında rastgele seçim yapar.
- **Algoritma analizi**, zaman ve bellek kullanımını girdi boyutu n'ye göre inceler. Çalışma süresi $T(n)$, saniyeyle değil temel işlem sayısıyla ölçülür. Dizi toplamı için satır satır sayım $T(n) = 2n + 3$ verir.
- Büyük n'de sabit katsayılar ve küçük terimler önemsizleşir; yalnız büyüme hızı kalır: $2n + 3 = \Theta(n)$, $5n^2 + 100n \log_2 n + 7 = \Theta(n^2)$.
- Büyüme sırası: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$. Daha hızlı bir bilgisayar yalnız sabiti küçültür; daha iyi bir algoritma büyüme hızını değiştirir.

- Örnekler: ilk öğeyi yazdırma $O(1)$; en büyük öğeyi bulma $\Theta(n)$ zaman, $O(1)$ ek bellek; tek bölenlerle $\sqrt{n}$'ye kadar asallık testi $O(\sqrt{n})$; düz matris çarpımı $\Theta(n^3)$; bütün permütasyonlar $\Theta(n \cdot n!)$; saf özyinelemeli Fibonacci $\Theta(1{,}618^n)$, yani üstel.
- **Bellekleme**, hesaplanan alt problem sonuçlarını saklar; Fibonacci'de çağrı sayısını $2F(n+1) - 1$'den $2n - 1$'e, süreyi $O(n)$'ye indirir. Bedeli $O(n)$ bellektir.
- **Öklid algoritması** $ebob(a, b) = ebob(b, a \bmod b)$, $ebob(a, 0) = a$ eşitliğine dayanır. Doğruluğu, iki çiftin ortak bölenlerinin aynı olmasından gelir. Her iki adımda büyük sayı en az yarıya indiği için süre $O(\log \min(a, b))$'dir. En kötü girdi ardışık Fibonacci sayılarıdır (Lamé). Döngülü sürüm aynı sürede $O(1)$ ek bellekle çalışır.

Kod	Girdi boyutu	Sayılan işlem	Zaman	Ek bellek
dizi toplamı	öğe sayısı n	karşılaştırma, toplama	$2n + 3 = \Theta(n)$	$O(1)$
`ilkElemaniYazdir`	öğe sayısı n	karşılaştırma, erişim	$O(1)$	$O(1)$
`enBuyukBul`	öğe sayısı n	karşılaştırma	$n = \Theta(n)$	$O(1)$
`asalMi`	sayının değeri n	bölme (`%`)	$\approx \sqrt{n} / 2 = O(\sqrt{n})$	$O(1)$
`matrisCarp`	matris boyutu n	çarpma	$n^3 = \Theta(n^3)$	$\Theta(n^2)$ (sonuç)
`permutasyon`	öğe sayısı n	yazılan satır	$n!$ satır, $\Theta(n \cdot n!)$	$O(n)$ (yığın)
`fib`	n	çağrı	$2F(n+1) - 1 = \Theta(1{,}618^n)$	$O(n)$ (yığın)
`fibBellek`	n	çağrı	$2n - 1 = O(n)$	$O(n)$
`ebob`	a, b	bölme (`%`)	$O(\log \min(a, b))$	özyinelemeli $O(\log \min(a, b))$, döngülü $O(1)$

Tablo 1.1.10: Notta incelenen kodlar.

1.1.7 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

algoritma *algorithm* · girdi *input* · çıktı *output* · açıklık *definiteness* · sonluluk *finiteness* · etkinlik *effectiveness* · doğruluk *correctness* · sözde kod *pseudocode* · deterministik algoritma *deterministic algorithm* · rastgele algoritma *randomized algorithm* · algoritma analizi *algorithm analysis* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity* · girdi boyutu *input size* · çalışma süresi *running time* · en kötü durum *worst case* · büyüme hızı *growth rate* · büyük O gösterimi *Big O notation* · büyük Teta *Big Theta (Θ)* · sabit zaman *constant time* · logaritmik *logarithmic* · doğrusal *linear* · log-doğrusal *linearithmic* · karesel *quadratic* · kübik *cubic*

üstel *exponential* · faktöriyel *factorial* · asallık testi *primality test* · sözde-polinom *pseudo-polynomial* · takas *swap* · özyineleme *recursion* · taban durum *base case* · bellekleme *memoization* · Öklid algoritması *Euclid's algorithm*

1.1.8 Kendini Dene

1. "*n* pozitif bir tam sayı olsun. *n* 1 olana kadar şunu tekrarla: *n* çiftse $n = n / 2$, tekse $n = 3n + 1$." Bu yöntem $n = 6$ için kaç adımda durur? Yöntemin bir algoritma olduğu kesin olarak söylenebilir mi?

   Cevap

   $n = 6$ için dizi 6, 3, 10, 5, 16, 8, 4, 2, 1'dir: 8 adımda durur. Adımlar açık ve yapılabilirdir. Ama sonluluk her *n* için kanıtlanmış değildir. Bu soru, matematikte Collatz sanısı denen ve hâlâ çözülmemiş bir problemdir. Yöntem çok büyük sayılara kadar bilgisayarla denenmiş ve dizi hep 1'e ulaşmıştır. Ama her pozitif tam sayı için durduğunu gösteren bir kanıt yoktur. Bu yüzden yöntemin her girdi için sonlu olduğu, dolayısıyla bir algoritma olduğu kesin olarak söylenemez.

2. `for (int i = 0; i < n; i++) for (int j = 0; j < i; j++) say++;` kodunda `say++` kaç kez çalışır? $n = 10$ için değer nedir, karmaşıklık sınıfı nedir?

   Cevap

   i. turda iç döngü *i* kez döner. Toplam $0 + 1 + \dots + (n - 1) = n(n - 1) / 2$'dir. $n = 10$ için $10 \cdot 9 / 2 = 45$. Baskın terim $n^2 / 2$ olduğu için sınıf $\Theta(n^2)$'dir.

3. `asalMi(199)` çağrısında `kokN` kaçtır, döngüde hangi `i` değerleri denenir, kaç bölme yapılır ve sonuç nedir?

   Cevap

   $\sqrt{199} \approx 14{,}1$, *(int)* $\sqrt{199} = 14$, `kokN = 15`. Denenen değerler 3, 5, 7, 9, 11, 13'tür; kalanlar sırasıyla 1, 4, 3, 1, 1, 4'tür ve hiçbiri 0 değildir. 6 bölmeden sonra döngü biter ve metot `true` döndürür: 199 asaldır.

4. `matrisCarp` ile 3×2 boyutlu `A` ve 2×3 boyutlu `B` çarpılıyor. `n`, `m`, `p` kaçtır, sonuç kaç boyut-

ludur ve kaç çarpma yapılır? Çarpım ters sırada (`B · A`) yapılsaydı?

Cevap

`A · B` : `n = 3` , `m = 3` , `p = 2` ; sonuç *3 × 3*; *3 · 3 · 2 = 18* çarpma. `B · A` : `n = 2` , `m = 2` , `p = 3` ; sonuç *2 × 2*; *2 · 2 · 3 = 12* çarpma. Matris çarpımında sıra hem sonucu hem de işlem sayısını değiştirir.

5. `permutasyon({1, 2, 3, 4}, 0)` çağrısında 2 ile başlayan ilk permütasyon kaçıncı satırda yazılır ve nedir? Metot toplam kaç kez çağrılır?

Cevap

İlk konumda 1 kaldığı sürece kalan üç öğenin *3! = 6* permütasyonu yazılır. *k = 0, i = 1* turunda `takas(0, 1)` diziyi [2, 1, 3, 4] yapar ve ilk yazılan satır budur: **7. satır, [2, 1, 3, 4]**. Çağrı sayısı her düzeydeki çağrıların toplamıdır: *1 + 4 + 12 + 24 + 24 = 65*.

6. `fib(7)` ve `fibBellek(7)` (bellek dizisi başta sıfır) ayrı ayrı çağrılıyor. Her biri kaç kez çağrılır ve ne döndürür?

Cevap

İkisi de *F(7) = 13* döndürür. `fib(7)` : *Ç(7) = 2F(8) − 1 = 2 · 21 − 1 = 41* çağrı. `fibBellek(7)` : *2 · 7 − 1 = 13* çağrı.

7. `ebob(377, 233)` kaç bölme yapar ve sonuç nedir? Bu sayı Lamé teoreminin verdiği sınırla nasıl karşılaştırılır?

Cevap

377 ve 233 ardışık Fibonacci sayılarıdır (*F(14)* ve *F(13)*). Kalanlar 144, 89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 0 olur: **12 bölme**, sonuç 1. Teoreme göre *k* bölme için *b ≥ F(k + 1)* gerekir; *b = 233 = F(13)* olduğundan *k ≤ 12*'dir. Örnek sınıra tam oturur. Basamak sınırı ise *5 · 3 = 15* verir.

8. Özyinelemeli `ebob(210, 84)` çağrısı ne döndürür? Kaç çağrı yapılır, çağrı yığını en çok kaç çerçeve derinliğe iner?

Cevap

210 = 2 · 84 + 42, *84 = 2 · 42 + 0*. Çağrılar `ebob(210, 84)` , `ebob(84, 42)` , `ebob(42, 0)` 'dır: 3 çağrı, 2 bölme, sonuç 42. Taban duruma ulaşıldığında yığında 3 çerçeve vardır.

9. Bir bilgisayar 100 kat hızlandırılıyor. Aynı sürede çözülebilecek en büyük girdi, $\Theta(n^2)$ bir algorit-

mada kaç katına çıkar? $\Theta(2^n)$ bir algoritmada n kaç artar?

Cevap

Karesel algoritmada $(kn)^2 = 100\,n^2$ koşulundan $k = 10$ bulunur: girdi 10 katına çıkar. Üstel algoritmada $2^{n+d} = 100 \cdot 2^n$ koşulundan $d = \log_2 100 \approx 6{,}6$ bulunur: n yalnız 6 artar (tam sayı olarak).

10. 40 bitlik ve 80 bitlik iki asal sayı `asalMi` ile sınanıyor. Her biri için yaklaşık kaç bölme yapılır? Saniyede 10^9 bölmeyle ikincisi ne kadar sürer?

Cevap

Bölme sayısı yaklaşık $\sqrt{n}\,/\,2$'dir. 40 bitlik sayıda $n \approx 2^{40}$, $\sqrt{n} \approx 2^{20}$; bölme sayısı yaklaşık $2^{19} \approx 524.000$'dir ve anında biter. 80 bitlik sayıda $\sqrt{n} \approx 2^{40}$; bölme sayısı yaklaşık $2^{39} \approx 5{,}5 \cdot 10^{11}$'dir ve yaklaşık 550 saniye, yani dokuz dakikadan fazla sürer. Bit sayısı iki katına çıkınca süre yaklaşık bir milyon kat artmıştır: bit uzunluğuna göre üstel büyüme.

1.1.9 İleri okuma

- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3. baskı, 1997), Kesim 1.1: algoritma kavramını Öklid algoritmasıyla açar; girdi, çıktı, açıklık, sonluluk ve etkinlik özelliklerini bu nottaki adlarıyla tartışır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 1–3: algoritmaların hesaplamadaki yerini, eklemeli sıralamanın satır satır analizini ve asimptotik gösterimi biçimsel tanımlarıyla verir; Bölüm 31 Öklid algoritmasının çalışma süresini Fibonacci sayılarıyla inceler.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.4: çalışma süresini deneylerle ölçmeyi ve ölçümleri matematiksel modelle açıklamayı Java örnekleriyle gösterir.
- J. Kleinberg ve É. Tardos, *Algorithm Design* (2005), Bölüm 2: polinom süreyi verimliliğin ölçütü olarak ele alır ve yaygın çalışma sürelerini örnek algoritmalarla karşılaştırır.

1.1.10 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3. baskı, Addison-Wesley, 1997.
4. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM I · BÖLÜM 1.2

Algoritma Karmaşıklığı

Aynı problemi çözen iki algoritmadan hangisinin daha iyi olduğunu söylemek için bir ölçüye gerek vardır. Saniye cinsinden süre bu işe yaramaz, çünkü makineden makineye değişir. Karmaşıklık analizi bunun yerine algoritmanın yaptığı temel işlemleri sayar ve bu sayının girdi büyüdükçe nasıl büyüdüğüne bakar. Not, zaman ve alan karmaşıklığı kavramlarıyla başlar; en iyi, en kötü ve ortalama durumu ayırır. Ardından büyüme hızını anlatan O, Ω ve Θ gösterimlerini tanımlar ve sabitlerin nasıl bulunduğunu gösterir. Sonraki kesimler döngülü kodun adım sayısını toplam formülleriyle, özyinelemeli kodun süresini ise yineleme bağıntılarıyla hesaplar. Bağıntılar açma yöntemiyle, özyineleme ağacıyla ve Ana Teorem ile çözülür. Bütün bu araçlar, her biri ayrı bir büyüme sınıfını temsil eden altı kısa Java fonksiyonu (f1–f6) üzerinde uygulanır. Not, amortize analize kısa bir bakışla kapanır.

ÖĞRENME ÇIKTILARI

Kavramlar
Zaman ve alan karmaşıklığını tanımlamak; en iyi, en kötü ve ortalama durum analizini birbirinden ayırmak; bir kodun adım sayısını RAM modeline göre saymak.

Gösterim
Büyük O, Ω ve Θ tanımlarını yazmak; bir fonksiyon için c ve n_0 sabitlerini bularak sınır kanıtlamak; yaygın büyüme sınıflarını sıralamak.

Döngü analizi
Tek, iç içe ve çarpımsal döngülerin tur sayısını toplam formülleriyle hesaplamak; iç döngü sınırının sonucu nasıl değiştirdiğini göstermek (f1, f2, f6).

Özyineleme
Özyinelemeli kod için bağıntı kurmak; bağıntıyı açma yöntemiyle, özyineleme ağacıyla ve Ana Teorem ile çözmek; teoremin uygulanamadığı bağıntıları tanımak (f3, f4, f5).

Amortize analiz
Bir işlem dizisinin toplam maliyetinden işlem başına maliyeti bulmak ve bunu ortalama durumdan ayırmak.

1.2.1 Karmaşıklık kavramı

Bir algoritmanın doğru çalışması ilk koşuldur, ama tek koşul değildir. Doğru bir algoritma bin öğelik girdide bir saniyede biterken milyon öğelik girdide günlerce sürebilir. Algoritma seçerken bu yüzden "ne kadar kaynak harcar?" sorusu da sorulur. Bu kesim, sorunun nasıl ölçüleceğini anlatır. Ölçülen şey saniye değil, temel işlem sayısıdır; ilgilenilen şey de bu sayının girdiyle birlikte nasıl büyüdüğüdür.

1.2.1.1 Algoritma karmaşıklığı nedir?

Karmaşıklık kuramı (*complexity theory*), bir algoritmanın çalışırken harcadığı kaynakları inceler. İki kaynak öne çıkar: zaman ve bellek. Her ikisi de girdinin büyüklüğüne bağlıdır. Beş sayıyı sıralamak, beş milyon sayıyı sıralamaktan çok daha kısa sürer. Bu yüzden kaynak kullanımı tek bir sayı olarak değil, girdi boyutunun bir fonksiyonu olarak yazılır.

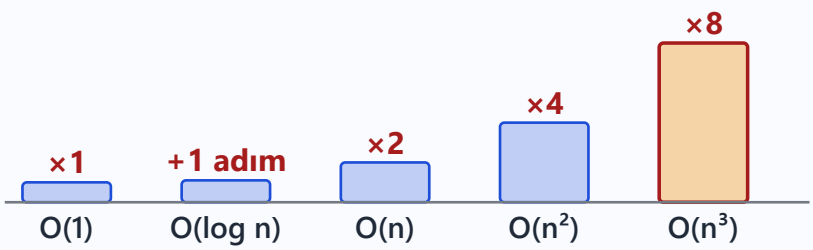
Girdi boyutu genellikle n ile gösterilir. Dizi işleyen bir algoritmada n dizinin öğe sayısıdır. Dizgi işleyen bir algoritmada karakter sayısı, çizge işleyen bir algoritmada düğüm ve kenar sayısıdır. Zaman ihtiyacı $T(n)$, bellek ihtiyacı $S(n)$ ile gösterilir. Örneğin $T(n) = 3n + 3$, algoritmanın n boyutlu bir girdide $3n + 3$ temel işlem yaptığını söyler.

Tanım 1.2.1

Zaman karmaşıklığı (*time complexity*), bir algoritmanın n boyutlu girdide yaptığı temel işlem sayısıdır ve $T(n)$ ile gösterilir. Temel işlemler atama, karşılaştırma, aritmetik işlem ve dizi erişimi gibi sabit sürede biten işlemlerdir.

Alan karmaşıklığı (*space complexity*), algoritmanın girdiye ek olarak kullandığı bellek miktarıdır ve $S(n)$ ile gösterilir. Yerel değişkenler, yardımcı diziler ve özyinelemeli çağrıların çağrı yığınında kapladığı yer bu belleğe girer.

Analizin amacı saniye ölçmek değildir. Amaç, girdi büyüdükçe kaynak ihtiyacının *nasıl* büyüdüğünü görmektir. Temel soru şudur: girdi boyutu iki katına çıkarsa çalışma süresi ne olur? Aynı mı kalır, iki katına mı çıkar, dört katına mı, yoksa karesi mi alınır? Şekil 1.2.1 bu sorunun birkaç büyüme sınıfı için cevabını gösterir.



Şekil 1.2.1: Girdi boyutu iki katına çıkınca adım sayısının değişimi. Sabit süreli bir algoritmada değişiklik olmaz (×1). Logaritmik algoritma yalnız bir adım daha atar. Doğrusal algoritmanın süresi iki, karesel algoritmanınki

dört, kübik algoritmanınki sekiz katına çıkar.

Şekildeki çarpanlar basit bir hesaptan gelir. $T(n) = n^2$ ise $T(2n) = 4n^2 = 4 \cdot T(n)$'dir. $T(n) = n^3$ ise $T(2n) = 8n^3$'tür. Logaritmada ise $log_2(2n) = log_2\ n + 1$ olduğundan süre yalnız bir adım uzar. Bu tek soru bile algoritmalar arasındaki farkı açıkça ortaya koyar. Girdi bin kat büyüdüğünde kübik bir algoritma bir milyar kat yavaşlar; logaritmik bir algoritma ise yalnız on adım kadar fazla çalışır.

> **Sık yapılan hata**
>
> Karmaşıklığı tek bir sayıyla, örneğin "bu algoritma 12 adım sürer" diye vermek. 12 adım ancak belirli bir girdi için doğrudur. Karmaşıklık her zaman n'nin fonksiyonudur: "bu algoritma $3n + 3$ adım sürer, $n = 3$ için 12 adım eder."

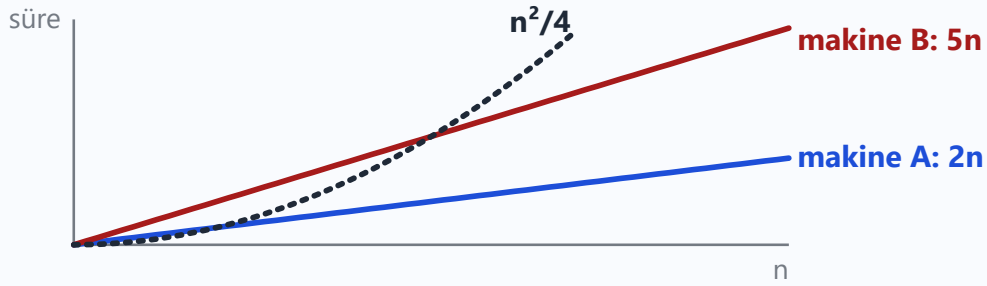
1.2.1.2 Süre ölçümü neden yetmez?

Bir algoritmanın hızını öğrenmenin ilk akla gelen yolu, onu çalıştırıp süresini ölçmektir. Java'da bunun için çalıştırmadan önce ve sonra sistem saati okunur, aradaki fark yazdırılır:

```
long baslangic = System.currentTimeMillis();
dogrusalArama(dizi, 10001);  // 10^8 eleman
long bitis = System.currentTimeMillis();
System.out.println("Geçen süre: "
    + (bitis - baslangic) + " ms");
```

Kod, 10^8 öğeli bir dizide doğrusal aramanın kaç milisaniye sürdüğünü ölçer. `System.currentTimeMillis()` o anki zamanı milisaniye olarak döndürür. İki okuma arasındaki fark, aramanın harcadığı süredir. Bu yönteme **deneysel ölçüm** denir. Gerçek bir sayı verir; ama bu sayının üç ciddi zayıflığı vardır.

- **Sonuç makineye bağlıdır.** Aynı kod hızlı bir işlemcide 40 ms, eski bir dizüstü bilgisayarda 200 ms sürebilir. Java'da durum daha da karışıktır. **JIT derleyicisi** (*just-in-time compiler*), sık çalışan kodu program çalışırken makine koduna çevirir; bu yüzden aynı ölçüm ilk denemede yavaş, sonrakilerde hızlı çıkar. **Önbellek** (*cache*), işletim sisteminin o anki yükü ve çöp toplayıcının ne zaman devreye girdiği de sonucu değiştirir.
- **Tek ölçüm geleceği söylemez.** 10^8 öğede 40 ms süren bir arama, 10^9 öğede ne kadar sürer? Ölçüm bunu söylemez. Algoritma doğrusal büyüyorsa süre on katına çıkar ve 400 ms olur; karesel büyüyorsa yüz katına çıkar ve 4 saniyeyi bulur. Cevap algoritmanın büyüme biçimine bağlıdır ve tek bir ölçüm bu biçimi göstermez.
- **Kodun yazılmış olması gerekir.** İki tasarımı karşılaştırmak için ikisinin de kodlanması, test edilmesi ve aynı koşullarda ölçülmesi gerekir. Tasarım aşamasında, henüz kod yokken, ölçüm yapılamaz.



Şekil 1.2.2: Aynı doğrusal algoritma iki makinede farklı sabitlerle büyür: A makinesinde *2n*, B makinesinde *5n*. Karesel bir algoritma ($n^2/4$) küçük n'de ikisinden de hızlıdır, ama bir noktadan sonra ikisini de geçer.

Şekil 1.2.2 ikinci zayıflığı somutlaştırır. B makinesi A'dan 2,5 kat yavaştır; ama iki çizgi de doğrudur ve aynı biçimde büyür. Kesikli eğri ise farklı bir algoritmaya aittir ve karesel büyür. Küçük girdilerde bu algoritma iki makinedeki doğrusal algoritmadan da hızlı görünür. $n^2/4 = 5n$ eşitliği $n = 20$'de sağlanır; bu noktadan sonra karesel algoritma yavaş makinedeki doğrusal algoritmayı bile geçer ve fark giderek açılır. Küçük girdilerle yapılan bir ölçüm, burada yanlış algoritmayı seçtirirdi.

Bu zayıflıkların çaresi **asimptotik analizdir**. Asimptotik analiz, adım sayısını n'nin bir fonksiyonu olarak verir ve yalnız n büyürken ne olduğuna bakar. Makineden bağımsızdır; kod yazmadan, yalnız algoritmanın tarifinden yapılabilir. Deneysel ölçüm değersiz değildir: iki algoritma aynı büyüme sınıfındaysa hangisinin gerçekte daha hızlı olduğunu ancak ölçüm söyler. İki yöntem birbirini tamamlar. Önce analizle doğru büyüme sınıfı seçilir, sonra ölçümle sabit çarpanlar karşılaştırılır.

	Deneysel ölçüm	**Asimptotik analiz**
Verdiği sonuç	gerçek süre (ms)	adım sayısının büyüme sınıfı
Makineye bağlılık	bağlı	bağımsız
Kod gerekir mi?	evet	hayır, tarif yeter
Büyük girdi için tahmin	vermez	verir
Sabit çarpanları görür mü?	evet	hayır

Tablo 1.2.1: Deneysel ölçüm ile asimptotik analizin karşılaştırması.

1.2.1.3 Adım sayma ve RAM modeli

Makineden bağımsız bir sayım yapmak için hayalî, basit bir makine düşünülür. Bu makinede her temel işlem tam bir birim zaman alır. Bellekteki her hücreye, adresi ne olursa olsun, aynı sürede erişilir. İşlemler birer birer, sırayla yürür. Bu soyut makineye **RAM modeli** (*random-access machine*, rastgele erişimli makine) denir. Gerçek bilgisayarlar böyle çalışmaz: önbellekte bulunan veri, ana bellektekinden kat kat hızlı okunur. Ama model, büyüme sınıfını bulmak için yeterince doğrudur ve hesabı çok kolaylaştırır.

Modeli doğrusal arama üzerinde uygulayalım. Doğrusal arama, dizinin öğelerine baştan sona tek tek bakar. Aranan değer (`hedef`) bulununca indisini döndürür; dizi biterse −1 döndürür:

```
for (int i = 0; i < a.length; i++) {
  if (a[i] == hedef)
    return i;
}
return -1;
```

Sayım için her işlemin kaç kez çalıştığı bulunur. En kötü durumu ele alalım: hedef dizide yoktur, dolayısıyla döngü sonuna kadar döner. Dizinin uzunluğu *n* olsun.

- `i = 0` ataması döngü başlarken bir kez yapılır.
- `i < a.length` koşulu her turun başında denetlenir. *i* 0'dan *n* − *1*'e kadar her değerde koşul doğrudur; bu *n* denetim eder. *i = n* olduğunda koşul bir kez daha denetlenir ve yanlış çıkar. Toplam *n + 1* denetim yapılır.
- `a[i] == hedef` karşılaştırması her turda bir kez yapılır: *n* kez.
- `i++` her turun sonunda bir kez yapılır: *n* kez.
- `return -1` en sonda bir kez çalışır.

İşlem	En kötü durumda kaç kez?
`i = 0`	1
`i < a.length`	n + 1
`a[i] == hedef`	n
`i++`	n
`return -1`	1
Toplam *T(n)*	3n + 3

Tablo 1.2.2: `dogrusalArama`'nın en kötü durumda adım sayımı.

Toplam *T(n) = 1 + (n + 1) + n + n + 1 = 3n + 3*'tür. Bu sayıda iki tür bilgi vardır. 3 katsayısı ve +3 sabiti, neyin bir adım sayıldığına ve makineye bağlıdır. Dizi erişimi ile karşılaştırma ayrı iki işlem sayılsaydı katsayı 4 olurdu; derleyici koşul denetimini artırmayla birleştirseydi 2 olurdu. Değişmeyen şey, adım sayısının *n* ile doğru orantılı büyümesidir. Dizi iki katına çıkınca adım sayısı da yaklaşık iki katına çıkar. Asimptotik analiz bu değişmeyen kısmı saklar, ötekini atar: *T(n) = Θ(n)*. Buradaki *Θ(n)*, "*n* ile aynı hızda büyür" diye okunur; kesin tanımı asimptotik gösterimle birlikte verilir.

> **Sık yapılan hata**
>
> Döngü koşulunun *n* kez denetlendiğini sanmak. Koşul, döngüyü bitiren son denetim dahil *n + 1* kez denetlenir. Bu fark büyüme sınıfını değiştirmez, ama tam adım sayısı istenen sorularda yanlış sonuca götürür.

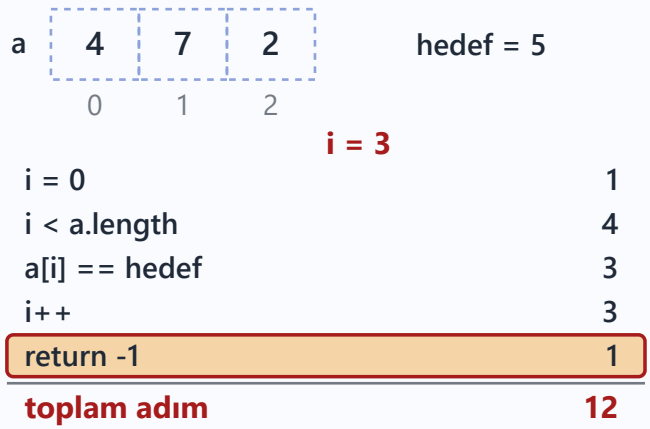
1.2.1.4 Kod izleme: dogrusalArama

Sayımı küçük bir örnekte adım adım doğrulayalım. Dizi `a = [4, 7, 2]`, hedef 5'tir. 5 dizide olmadığı için en kötü durum oluşur; beklenen adım sayısı *3 · 3 + 3 = 12*'dir.

Adım	Yapılan işlem	`i`	Sonuç	Toplam adım
1	`i = 0`, sonra `0 < 3`	0	koşul doğru	2
2	`a[0] == 5`	0	4 ≠ 5	3
3	`i++`, sonra `1 < 3`	1	koşul doğru	5
4	`a[1] == 5`	1	7 ≠ 5	6
5	`i++`, sonra `2 < 3`	2	koşul doğru	8
6	`a[2] == 5`	2	2 ≠ 5	9
7	`i++`, sonra `3 < 3`	3	koşul yanlış, döngü biter	11
8	`return -1`	3	−1 döner	12

Tablo 1.2.3: `dogrusalArama([4, 7, 2], 5)` izlemesi. Son sütun o ana kadarki toplam adım sayısıdır.

Koşul dört kez denetlenmiştir (*n + 1 = 4*), karşılaştırma ve artırma üçer kez yapılmıştır. Toplam 12 adım, *3n + 3* formülünün *n = 3* için verdiği değerdir. Her öğe için sabit sayıda, üç işlem harcanmıştır: bir koşul denetimi, bir karşılaştırma ve bir artırma. Öğe sayısı arttıkça bu üçlü tekrarlanır; süre bu yüzden doğrusal büyür. Şekil 1.2.3 izlemenin son anını gösterir.



Şekil 1.2.3: `dogrusalArama([4, 7, 2], 5)` izlemesinin sonu. Döngü *i = 3*'te bitmiş, `return -1` çalışmıştır. Tablo her işlemin kaç kez yapıldığını gösterir: koşul 4, karşılaştırma 3, artırma 3 kez. Toplam 12 adımdır.

1.2.1.5 En iyi, en kötü ve ortalama durum

Doğrusal aramanın adım sayısı yalnız *n*'ye bağlı değildir. Aynı uzunluktaki iki dizide hedefin yeri farklıysa adım sayısı da farklıdır. Hedef ilk hücredeyse arama hemen biter; hiç yoksa bütün dizi taranır. Bu yüzden "doğrusal arama kaç adım sürer?" sorusunun tek bir cevabı yoktur. Soru, hangi girdiden söz edildiği belirtilerek sorulmalıdır. Bunun için üç ölçü kullanılır.

Tanım 1.2.2

Boyutu n olan bütün girdiler düşünülsün.

- **En iyi durum** (*best case*): bu girdiler arasında en az işlem gerektirenin işlem sayısı.
- **En kötü durum** (*worst case*): en çok işlem gerektirenin işlem sayısı.
- **Ortalama durum** (*average case*): girdilerin bir olasılık dağılımına göre beklenen işlem sayısı.

Üç ölçünün değeri aynı değildir. **En kötü durum** en çok kullanılanıdır, çünkü bir garanti verir: algoritma hiçbir girdide bundan uzun sürmez. Gerçek zamanlı bir sistemde, bir uçağın kontrol yazılımında ya da bir web sunucusunda bu garanti önemlidir. **En iyi durum** bir alt sınır verir, ama tek başına pek bilgi taşımaz. Hemen her algoritma bazı şanslı girdilerde hızlıdır. **Ortalama durum** pratikte en gerçekçi tahmini verebilir; ama hesaplamak için girdilerin hangi olasılıkla geldiğini bilmek gerekir. Bu bilgi çoğu zaman yoktur ve varsayılan dağılım yanlışsa sonuç da yanlıştır.

Örnek 1.2.1

Dizi `[7, 3, 9, 4, 8, 1]`'dir ($n = 6$). Doğrusal aramanın karşılaştırma sayısı üç durum için hesaplanıyor.

1. **En iyi durum.** Hedef 7, yani ilk öğedir. İlk karşılaştırmada bulunur: 1 karşılaştırma. Dizi ne kadar uzun olursa olsun bu sayı değişmez: $\Theta(1)$.
2. **En kötü durum.** Hedef 5 dizide yoktur. Altı öğenin hepsine bakılır: 6 karşılaştırma. Hedef son öğe olsaydı (1) yine 6 karşılaştırma gerekirdi. Genel olarak n karşılaştırma: $\Theta(n)$.
3. **Ortalama durum.** Hedefin dizide olduğu ve altı konumdan her birinde eşit olasılıkla (1/6) bulunduğu varsayılır. k. konumdaki hedef k karşılaştırmayla bulunur. Beklenen değer $(1 + 2 + 3 + 4 + 5 + 6) / 6 = 21 / 6 = 3{,}5$'tir. Genel olarak $(1 + 2 + \ldots + n) / n = (n + 1) / 2$ karşılaştırma yapılır. Bu sayı en kötü durumun yarısıdır, ama yine n ile doğrusal büyür: $\Theta(n)$.





Şekil 1.2.4: Örnek 1.2.1'deki dizide üç arama. Üstte en iyi durum: hedef 7 ilk hücrede, 1 karşılaştırma. Ortada en kötü durum: hedef 5 yok, 6 karşılaştırma. Altta ortalamaya yakın bir arama: hedef 4, dördüncü hücrede bulunur. Turuncu hücreler bakılıp geçilen, yeşil hücre bulunan öğedir.

Ortalama durum hesabındaki $(n + 1)/2$ sonucu, Gauss'un toplam formülünden gelir: $1 + 2 + \ldots + n = n(n + 1)/2$. Aynı formül döngü analizinde de sık geçer. Ortalama ile en kötü durum arasındaki 1/2 çarpanı, asimptotik analizde atılan türden bir sabittir. Bu yüzden doğrusal aramanın ortalama ve en kötü durumu aynı sınıfa düşer. Her algoritmada böyle değildir: hızlı sıralama ortalamada $\Theta(n \log n)$, en kötü durumda ise $\Theta(n^2)$ sürer.

> **Sık yapılan hata**
>
> Ortalama karşılaştırma sayısını $n/2$ diye yuvarlamak. Hedef her konumda eşit olasılıkla bulunuyorsa ortalama tam olarak $(n + 1)/2$'dir; $n = 6$ için 3 değil 3,5. Sınıf aynıdır, ama tam sayı istenen sorularda bu fark yanlış cevap demektir.

1.2.1.6 Alan karmaşıklığı

Zaman kadar önemli ikinci kaynak bellektir. Alan karmaşıklığı genellikle **ek bellek** (*auxiliary space*) için verilir. Girdinin kendisi sayılmaz, çünkü her algoritma girdiyi bir yerde tutmak zorundadır; asıl merak edilen, algoritmanın bunun üstüne ne kadar bellek istediğidir. Bir diziyi tek bir toplam değişkeniyle toplayan algoritma, dizi ne kadar uzun olursa olsun yalnız birkaç değişken kullanır. Ek belleği $\Theta(1)$'dir.

Özyinelemeli kodda gözden kaçan bir bellek kalemi vardır. Bir metot çağrıldığında, parametreleri, yerel değişkenleri ve dönüş adresi için **çağrı yığınında** (*call stack*) bir **çerçeve** (*stack frame*) açılır. Metot bitince çerçeve kaldırılır. Özyinelemeli bir fonksiyon kendini çağırdıkça çerçeveler üst üste biner. Aynı anda yığında bulunan en çok çerçeve sayısı, özyinelemenin **derinliğidir**. Ek bellek en az bu derinlik kadardır.

Kod	Ek bellek nereden gelir?	Bellek	Zaman
f1: tek döngü	yalnız `x` ve `i`	Θ(1)	Θ(n)
`new int[n]` **oluşturmak**	*n* öğeli yardımcı dizi	Θ(n)	Θ(n)
f3: `f3(n-1) + f3(n-1)`	yığın derinliği *n*	Θ(n)	$\Theta(2^n)$
f4: `f4(n/2) + f4(n/2)`	yığın derinliği yaklaşık $\log_2 n$	Θ(log n)	Θ(n)
Birleştirme sıralaması	birleştirme için yardımcı dizi	Θ(n)	Θ(n log n)

Tablo 1.2.4: Bölümdeki örneklerin ek belleği ve zamanı.

Tablo 1.2.4 önemli bir gerçeği gösterir: zaman ve bellek farklı hızlarda büyüyebilir. Kendini iki kez $n - 1$ ile çağıran f3 fonksiyonu üstel sayıda ($2^n - 1$) çağrı yapar. Buna karşın belleği yalnız doğrusaldır. Sebep, çağrıların aynı anda değil sırayla yürümesidir. `f3(n)` önce soldaki `f3(n - 1)`'i çağırır ve bu çağrının bütün alt çağrıları bitip çerçeveleri kalkmadan sağdakine geçmez. Yığında herhangi bir anda yalnız bir zincir bulunur: `f3(n)`, `f3(n - 1)`, …, `f3(1)`. Bu zincirin uzunluğu *n*'dir.

Aynı düşünce f4 için de geçerlidir. f4'te girdi her çağrıda yarıya indiği için zincir *n, n/2, n/4, …, 1* biçimindedir ve uzunluğu yaklaşık $\log_2 n + 1$'dir. Örneğin `f4(16)` çalışırken yığında en çok 5 çerçeve bulunur (16, 8, 4, 2, 1).

> **Sık yapılan hata**
>
> Özyinelemeli bir fonksiyonun belleğini toplam çağrı sayısıyla karıştırmak. f3 için "2^n çağrı var, her biri bir çerçeve açar, öyleyse bellek $\Theta(2^n)$" demek yanlıştır. Biten çağrının çerçevesi kaldırılır ve yeri bir sonraki çağrı için kullanılır. Bellek, aynı anda açık olan çerçevelerin en büyük sayısıyla, yani özyineleme derinliğiyle belirlenir.

1.2.2 Asimptotik gösterim

Adım sayısı *3n + 3* çıktığında katsayıların atılıp yalnız "doğrusal" bilgisinin saklanacağı söylendi. Bu "atma" işleminin kuralları kesin olmalıdır; yoksa iki kişi aynı koda farklı sınıflar verebilir. **Asimptotik gösterim** (*asymptotic notation*) bu kuralları koyar. "Asimptotik" sözcüğü, *n* sonsuza giderken olanla ilgilenildiğini anlatır. Küçük girdilerdeki davranış ve sabit çarpanlar bilerek göz ardı edilir. Üç gösterim kullanılır: üst sınır için O, alt sınır için Ω, sıkı sınır için Θ.

O gösterimi algoritmalardan çok önce, 19. yüzyılın sonunda sayılar kuramında ortaya çıktı. Paul Bachmann 1894'te kullandı, Edmund Landau yaygınlaştırdı. Donald Knuth 1976'da yazdığı kısa bir makalede Ω ve Θ'yı bugünkü anlamlarıyla tanımladı ve üçünün algoritma analizinde birlikte kullanılmasını önerdi. Bugün bütün algoritma kitapları bu tanımları kullanır.

1.2.2.1 Büyük O

Büyük O bir üst sınır bildirir. "*f(n) = O(g(n))*" ifadesi, yeterince büyük her *n* için *f*'nin *g*'nin bir sabit katını aşmadığını söyler. İki ayrıntı önemlidir. Birincisi, sabit kat serbesttir: *g*, 4 ya da 100 ile çarpılabilir. İkincisi, küçük *n*'ler önemsizdir: eşitsizliğin bir eşik değerinden sonra sağlanması yeter.

Tanım 1.2.3

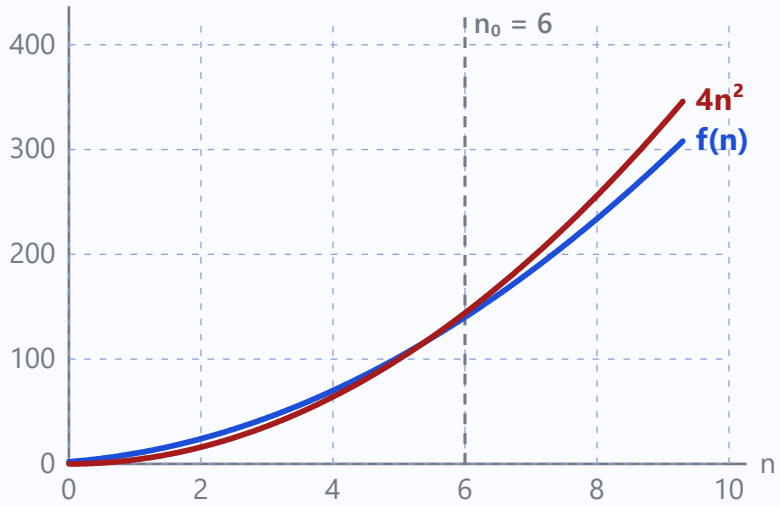
f(n) = O(g(n)) olması için gerek ve yeter koşul, öyle pozitif bir *c* sabiti ve bir n_0 eşiği bulunmasıdır ki her $n \geq n_0$ için

$$0 \leq f(n) \leq c \cdot g(n)$$

sağlansın. Bu durumda *g*, *f* için bir **üst sınırdır** (*upper bound*). Gösterimin adı **büyük O gösterimidir** (*Big O notation*).

Tanımda iki sabit vardır ve ikisinin de görevi ayrıdır. *c*, sabit çarpanları yutar. $3n^2$ ile n^2 arasındaki 3 katlık fark, *c = 3* seçilerek ortadan kalkar. n_0 ise küçük girdileri dışarıda bırakır. $n^2 + 100$ fonksiyonu *n = 1*'de $2n^2$'den büyüktür; ama $n \geq 10$ olunca onu aşmaz. Tanım yalnız n_0'dan sonrasıyla ilgilendiği için baştaki bu farklar sonucu etkilemez.

Bir örnek: $f(n) = 3n^2 + 5n + 2$ fonksiyonu $O(n^2)$'dir. Gerçekten *c = 4* seçilirse her $n \geq 6$ için $3n^2 + 5n + 2 \leq 4n^2$ sağlanır. Bu sabitlerin nasıl bulunduğu biraz aşağıda adım adım gösterilir. Şekil 1.2.5 durumu grafikte gösterir.



Şekil 1.2.5: $f(n) = 3n^2 + 5n + 2$ (mavi) ve $4n^2$ (kırmızı). Küçük *n*'de *f* üsttedir. $n \approx 5{,}37$'de eğriler kesişir. $n_0 = 6$'dan sonra *f* hep $4n^2$'nin altında kalır; bu yüzden $f(n) = O(n^2)$'dir.

Büyük O bir üst sınır olduğu için gevşek de olabilir. $3n^2 + 5n + 2$, $O(n^2)$ olduğu gibi $O(n^3)$ ve $O(2^n)$'dir de; bu ifadelerin hepsi doğrudur. Ama bilgi değerleri aynı değildir. "Bu algoritma $O(2^n)$ sürer" cümlesi, karesel bir algoritma için doğru ama yanıltıcıdır. Analizde genellikle bulunabilen en küçük üst sınır verilir.

Gösterimdeki eşittir işareti de yanlış anlaşılmaya açıktır. $f(n) = O(g(n))$ aslında "f, g'nin sabit katıyla üstten sınırlanan fonksiyonlar kümesine aittir" demektir. Bu yüzden ifade tek yönlüdür. $n = O(n^2)$ doğrudur; ama buradan "$O(n^2) = n$" sonucu çıkmaz. Bazı kitaplar bu nedenle $f(n) \in O(g(n))$ yazar.

Sık yapılan hata

"Büyük O en kötü durumu temsil eder" demek. Büyük O bir *sınır türüdür*; en kötü durum ise bir *girdi türüdür*. İkisi birbirinden bağımsızdır. En kötü durum süresi için de, en iyi durum süresi için de, ortalama süre için de üst sınır yazılabilir. "Doğrusal aramanın en iyi durumu $O(1)$'dir" geçerli bir cümledir. Karışıklığın kaynağı, Büyük O'nun pratikte çoğunlukla en kötü durum süresi için kullanılmasıdır.

1.2.2.2 Büyük Omega ve büyük Teta

Üst sınırın tersi alt sınırdır. Bir algoritmanın "en az şu kadar" süreceğini söylemek için Ω (büyük Omega) kullanılır. İki sınır aynı fonksiyonda buluşursa büyüme hızı kesin olarak bilinir; bunu da Θ (büyük Teta) anlatır.

Tanım 1.2.4

$f(n) = \Omega(g(n))$: öyle $c > 0$ ve n_0 vardır ki her $n \geq n_0$ için $f(n) \geq c \cdot g(n) \geq 0$. Bu durumda g, f için bir **alt sınırdır** (*lower bound*).

$f(n) = \Theta(g(n))$: öyle $c_1, c_2 > 0$ ve n_0 vardır ki her $n \geq n_0$ için $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. Bu durumda g, f için bir **sıkı sınırdır** (*tight bound*).

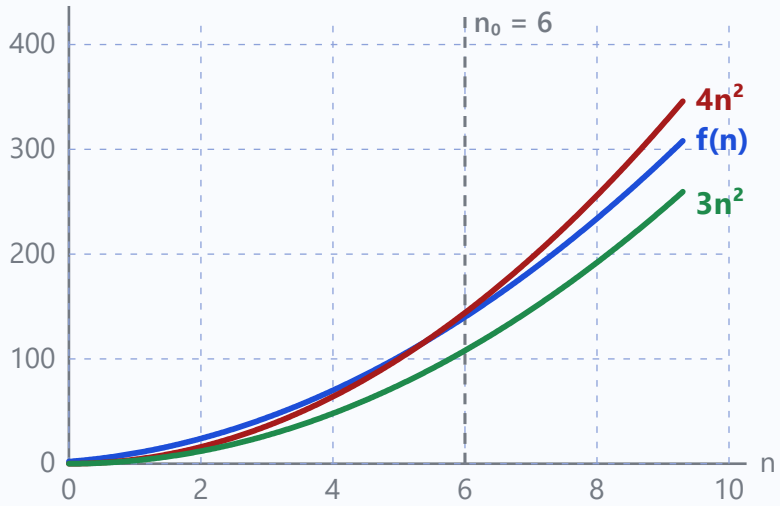
Gösterim	Anlamı	Tanım	Sayılardaki benzeri
f = O(g)	üst sınır	$f(n) \leq c \cdot g(n)$	$f \leq g$
f = Ω(g)	alt sınır	$f(n) \geq c \cdot g(n)$	$f \geq g$
f = Θ(g)	sıkı sınır	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$f = g$

Tablo 1.2.5: Üç asimptotik gösterim. Tanımlarda her $n \geq n_0$ için eşitsizliğin sağlandığı pozitif sabitler aranır.

Tablonun son sütunu kullanışlı bir benzetmedir: O, "küçük ya da eşit"; Ω, "büyük ya da eşit"; Θ, "eşit" gibi davranır. Benzetme şu temel ilişkiyi de hatırlatır: bir sayı hem $\leq g$ hem $\geq g$ ise g'ye eşittir. Aynı biçimde

> $f = \Theta(g) \Leftrightarrow f = O(g)$ ve $f = \Omega(g)$.

Θ, iki sabit kat arasına sıkıştırmak demektir. Şekil 1.2.6 bunu $3n^2 + 5n + 2$ üzerinde gösterir: fonksiyon, $n \geq 6$ için $3n^2$ ile $4n^2$ arasında kalır. Bu yüzden $\Theta(n^2)$'dir. Aynı fonksiyon $O(n^3)$ olduğu hâlde $\Theta(n^3)$ değildir. $c \cdot n^3$ biçimindeki her alt eğri, c ne kadar küçük seçilirse seçilsin, bir noktadan sonra fonksiyonu geçer. Çünkü $f(n)/n^3$ oranı n büyüdükçe sıfıra gider.



Şekil 1.2.6: Θ'nın sıkıştırma anlamı. $n \geq 6$ için $f(n) = 3n^2 + 5n + 2$ (mavi), alttan $3n^2$ (yeşil) ve üstten $4n^2$ (kırmızı) ile sınırlanır.

Büyüme sınıfı ile durum kavramı burada bir kez daha ayrılmalıdır. Doğrusal arama için şu cümlelerin hepsi doğrudur: en kötü durum süresi $\Theta(n)$'dir; en iyi durum süresi $\Theta(1)$'dir; bütün girdiler için süre $O(n)$ ve $\Omega(1)$'dir. Ama "doğrusal arama $\Theta(n)$'dir" cümlesi, hangi durumdan söz edildiği belirtilmezse eksiktir. Eklemeli sıralama da benzerdir: en kötü durumda $\Theta(n^2)$, sıralı girdide $\Theta(n)$ sürer; bütün girdiler için tek bir Θ yazılamaz. Birleştirme sıralaması ise her girdide $\Theta(n \log n)$ sürer; onun için durum belirtmeden Θ yazmak doğrudur.

Sık yapılan hata

Θ ile O'yu aynı anlamda kullanmak. "$3n^2 + 5n + 2 = \Theta(n^3)$" yanlıştır; doğru olan $O(n^3)$'tür. Θ hem alt hem üst sınır ister ve fonksiyonun büyüme hızını tam olarak verir. Gündelik konuşmada "bu algoritma $O(n^2)$" denirken çoğu zaman $\Theta(n^2)$ kastedilir; yazılı bir analizde bu ayrım korunmalıdır.

1.2.2.3 Sabitleri bulmak

Bir sınırı kanıtlamak, tanımdaki sabitleri somut olarak bulmak demektir. Yöntem her seferinde aynıdır: bir c seçilir, eşitsizlik çözülür ve hangi n'den sonra sağlandığı bulunur. Sabitler tek değildir; geçerli herhangi bir çift kanıt için yeter.

Örnek 1.2.2

$f(n) = 3n^2 + 5n + 2 = \Theta(n^2)$ olduğu gösterilecek. Bunun için iki eşitsizlik gerekir: bir üst sınır ve bir alt sınır.

1. **Üst sınır için c_2 seçimi.** Baştaki katsayı 3 olduğu için 3'ten biraz büyük bir sayı denenir: $c_2 = 4$. İstenen eşitsizlik $3n^2 + 5n + 2 \le 4n^2$'dir.
2. **Eşitsizliği çözmek.** Her şey bir tarafa toplanır: $n^2 - 5n - 2 \ge 0$. İkinci derece denklemin pozitif kökü $(5 + \sqrt{33})/2 \approx 5{,}37$'dir. Eşitsizlik bu kökten büyük n'ler için sağlanır. En küçük tam sayı 6 olduğundan $n_0 = 6$.
3. **Sınırı denetlemek.** $n = 5$'te sol taraf $75 + 25 + 2 = 102$, sağ taraf 100'dür; eşitsizlik bozulur. $n = 6$'da sol taraf $108 + 30 + 2 = 140$, sağ taraf 144'tür; eşitsizlik sağlanır.
4. **Bundan sonra da sağlandığını görmek.** n bir artınca sol taraf $6n + 8$, sağ taraf $8n + 4$ artar. $n \ge 6$ için $8n + 4 > 6n + 8$ olduğundan sağ taraf her adımda daha çok büyür. $n = 6$'da açılan fark (4) bu yüzden hiç kapanmaz, hep büyür.
5. **Alt sınır.** $c_1 = 3$ seçilir. $3n^2 + 5n + 2 \ge 3n^2$ eşitsizliği, $5n + 2 > 0$ olduğu için her $n \ge 1$ için doğrudur.
6. **Birleştirmek.** İki eşitsizlik birlikte $n \ge 6$ için geçerlidir: $3n^2 \le f(n) \le 4n^2$. $c_1 = 3$, $c_2 = 4$, $n_0 = 6$ ile tanım sağlanır: $f(n) = \Theta(n^2)$.

n	f(n)	$4n^2$	$f(n) \le 4n^2$?
1	10	4	hayır
3	44	36	hayır
5	102	100	hayır
6	140	144	evet
7	184	196	evet
8	234	256	evet

Tablo 1.2.6: $f(n) = 3n^2 + 5n + 2$ ile $4n^2$'nin karşılaştırması.

Sabitlerin tek olmadığını görmek için başka bir seçim yapılabilir. $c_2 = 10$ seçilirse $n = 1$'de bile $3 + 5 + 2 = 10 \le 10$ olur. Ayrıca her $n \ge 1$ için $5n \le 5n^2$ ve $2 \le 2n^2$ olduğundan $3n^2 + 5n + 2 \le 3n^2 + 5n^2 + 2n^2 = 10n^2$ yazılabilir. Bu kanıt kökü hesaplamayı gerektirmez: her terim en büyük kuvvetle sınırlanır, katsayılar toplanır. Büyük c, küçük n_0 demektir; küçük c ise daha büyük bir eşik gerektirir.

Aynı yöntem doğrusal bir fonksiyona da uygulanır. $2n + 10 = O(n)$ olduğunu göstermek için $c = 3$ seçilirse $2n + 10 \le 3n$ eşitsizliği $n \ge 10$ için sağlanır: $c = 3$, $n_0 = 10$. Terimleri tek tek sınırlama yöntemiyle $2n + 10 \le 2n + 10n = 12n$ yazılır; bu da $c = 12$, $n_0 = 1$ verir.

Bir sınırın *yanlış* olduğunu göstermek de mümkündür. Bunun için hiçbir sabit çiftinin işe yaramadığı kanıtlanır; genellikle çelişkiyle. $n^2 = O(n)$ olduğunu varsayalım. O hâlde bir c ve n_0 için her $n \ge n_0$'da $n^2 \le c \cdot n$, yani $n \le c$ olmalıdır. Oysa n, n_0'dan ve c'den büyük seçilebilir, örneğin $n = max(n_0, \lceil c \rceil + 1)$. Bu n için $n \le c$ bozulur. Varsayım çelişkiye götürdüğü için yanlıştır: $n^2 \ne O(n)$.

> **Sık yapılan hata**
>
> Kanıtta yalnız tek bir n değerini denemek. "$n = 10$'da $f(10) \le 4 \cdot 100$, öyleyse $O(n^2)$" yetersizdir. Tanım, eşitsizliğin n_0'dan sonraki *her* n için sağlanmasını ister. Eşitsizliği cebirle çözmek ya da her terimi en büyük kuvvetle sınırlamak gerekir.

1.2.2.4 Hesap kuralları

Her seferinde sabit aramak zahmetlidir. Tanımlardan çıkan birkaç kural, sınıfı çoğu zaman doğrudan okumayı sağlar.

- **Sabit çarpanlar atılır.** $c \cdot f(n) = \Theta(f(n))$; örneğin $5n = \Theta(n)$ ve $n^2/3 = \Theta(n^2)$. Sabit çarpan tanımdaki c'ye gömülür.
- **Düşük dereceli terimler atılır.** Bir polinomda en büyük kuvvetli terim kalır: $4n^3 + 100n^2 + 7 = \Theta(n^3)$. n büyüdükçe $100n^2$, $4n^3$'ün yanında önemsizleşir; $n = 1000$'de kırkta biri kadardır.
- **Toplam kuralı: ardışık kod.** Arka arkaya çalışan iki kod parçasının toplam süresi, büyük olanın süresidir: $O(f) + O(g) = O(max(f, g))$. Önce $\Theta(n)$ süren bir döngü, sonra $\Theta(n^2)$ süren bir döngü çalışırsa toplam $\Theta(n^2)$'dir.
- **Çarpım kuralı: iç içe kod.** n kez dönen bir döngünün içinde $O(g)$ süren bir iş varsa toplam $O(n \cdot g)$'dir. Kural, iç işin her turda aynı olduğunu varsayar. İç işin süresi dış döngünün sayacına bağlıysa çarpım yerine toplam hesaplanır.
- **Logaritmanın tabanı önemsizdir.** Taban değiştirme formülüne göre $\log_a n = \log_b n \,/\, \log_b a$'dır. İki taban arasındaki fark yalnız sabit bir çarpandır. Bu yüzden $O(\log_2 n) = O(\log_{10} n) = O(\log n)$ yazılır ve taban çoğu zaman belirtilmez.
- **Üstel ifadelerde taban önemlidir.** $3^n = O(2^n)$ yanlıştır. $3^n/2^n = 1{,}5^n$ oranı sonsuza gider; hiçbir sabit çarpan bu farkı kapatamaz. Logaritmanın tersine, üstel tabanlar birbirine sabit çarpanla dönüşmez.

Bu kuralların sonucunda sık geçen fonksiyonlar kesin bir büyüme sırasına dizilir:

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$$

Buradaki "<" işareti, soldakinin sağdakine göre ihmal edilebilir hâle geldiğini anlatır: büyük n'de oran sıfıra gider. Örneğin $\log n / \sqrt{n}$ oranı, başta yavaş da olsa, sıfıra yaklaşır.

> **Sık yapılan hata**
>
> Sabitleri ve küçük terimleri sonuçta bırakmak: $O(3n + 3)$ ya da $O(n^2/3 + n)$ yazmak. Bu ifadeler yanlış değildir, ama gösterimin amacına aykırıdır. Doğru yazımlar $O(n)$ ve $O(n^2)$'dir. Benzer biçimde $O(2^{n+1})$ yerine $O(2^n)$ yazılır, çünkü $2^{n+1} = 2 \cdot 2^n$'dir; ama $2^{2n} = 4^n$ ifadesi $O(2^n)$ değildir.

1.2.2.5 Yaygın büyüme sınıfları

Algoritmaların çok büyük bir bölümü birkaç büyüme sınıfından birine düşer. Bir algoritmanın hangi sınıfta olduğu, girdi büyüdükçe nasıl davranacağını hemen söyler. Tablo 1.2.7 bu sınıfları en hızlıdan en yavaşa doğru sıralar. Son sütun, bölümün altı örnek fonksiyonunun hangi sınıfa düştüğünü gösterir.

Sınıf	Adı	Tipik örnek	n iki katına çıkınca	Bölümde
$O(1)$	sabit	dizi erişimi, ilk öğeyi yazdırma	değişmez	—
$O(\log n)$	logaritmik	ikili arama, hızlı üs alma, Öklid algoritması	+1 adım	f6 (üst sınır)
$O(\sqrt{n})$	karekök	deneme bölmesiyle asallık testi	≈ 1,41 katı	—
$O(n)$	doğrusal	doğrusal arama, en büyük öğe, dizi toplamı	2 katı	f1, f4
$O(n \log n)$	log-doğrusal	birleştirme sıralaması, heap sort, hızlı sıralama (ortalama)	2 katından biraz fazla	f5
$O(n^2)$	karesel	kabarcık, seçmeli ve eklemeli sıralama	4 katı	—
$O(n^3)$	kübik	klasik matris çarpımı	8 katı	f2
$O(2^n)$	üstel	bütün alt kümeler, saf özyinelemeli Fibonacci	karesi alınır	f3
$O(n!)$	faktöriyel	bütün permütasyonlar	çok daha hızlı patlar	—

Tablo 1.2.7: Yaygın büyüme sınıfları.

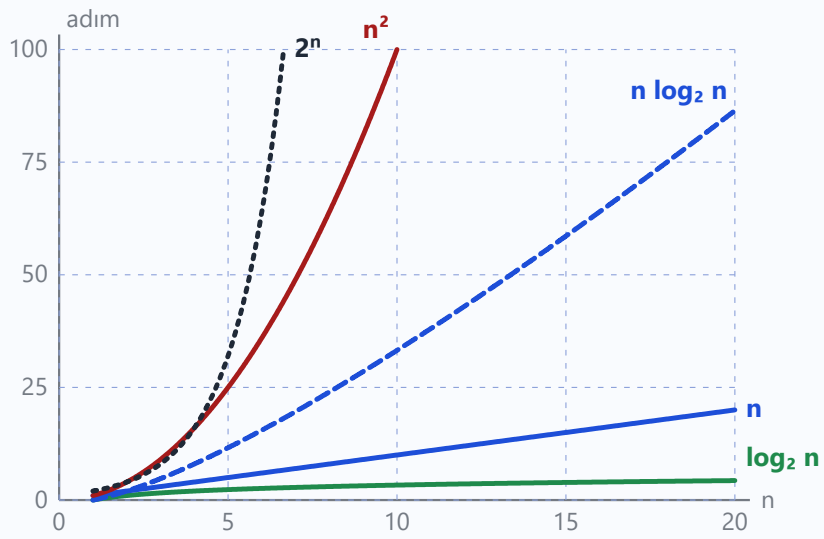
Tablodaki "iki katına çıkınca" sütunu, sınıfların farkını sezdirmenin en kısa yoludur. Logaritmik algoritmada girdi iki katına çıkınca yalnız bir adım eklenir; ikili arama bu yüzden bir milyar öğede bile otuz kadar karşılaştırmayla biter. Karekökte çarpan $\sqrt{2} \approx 1{,}41$'dir. Log-doğrusalda çarpan 2'den biraz fazladır: $2n \log(2n) = 2n \log n + 2n$. Üstel sınıfta durum bambaşkadır: $2^{2n} = (2^n)^2$ olduğu için süre karesine çıkar. Saniyede 10^9 işlem yapan bir bilgisayarda 1 saniye süren 2^n adımlık bir hesap ($n \approx 30$), girdi iki katına çıkınca 36 yıl kadar sürer.

Sınıflar arasında önemli bir çizgi daha vardır. n^k biçimindeki sınıflara **polinom zamanlı** denir; bunlar genelde "verimli" sayılır. Üstel ve faktöriyel sınıflar ise "verimsiz" sayılır; küçük girdilerin ötesinde pratikte çalıştırılamazlar. Bu ayrım kaba bir kuraldır (n^{100} polinomdur ama hiç de verimli değildir), yine de algoritma tasarımında yol gösterir.

Tablonun “Bölümde” sütunundaki f6 için bir not gerekir. f6'nın *O(log n)* olduğu doğrudur, ama bu sınır sıkı değildir. f6'nın gerçek sınıfı, logaritmadan bile çok daha yavaş büyüyen *log* n*'dir. Bu ilginç fonksiyon döngü analizi kesiminde ayrıntısıyla incelenir.

1.2.2.6 Büyüme eğrileri ve sayılar

Sınıfların sırası grafikte de görülür. Şekil 1.2.7 beş fonksiyonu *n = 1* ile 20 arasında çizer. Küçük *n*'de eğriler birbirine yakındır; hatta *n = 4*'te n^2 ile 2^n eşittir (ikisi de 16). *n = 5*'ten sonra 2^n öne geçer ve hızla uzaklaşır. 2^n eğrisi *n ≈ 6,6*'da 100'ü aşar; n^2 ancak *n = 10*'dan sonra aşar. $n \log_2 n$ doğrusala yakın kalır, $\log_2 n$ ise neredeyse yataydır.



Şekil 1.2.7: *n = 1* ile 20 arasında $\log_2 n$, n, $n \log_2 n$, n^2 ve 2^n. Dikey eksen 100 adımda kesilmiştir; 2^n ve n^2 kısa sürede bu sınırı aşar.

Grafikteki fark, sayılarla bakıldığında daha da çarpıcıdır. Tablo 1.2.8, saniyede 10^9 temel işlem yapan bir bilgisayarda her sınıfın ne kadar süreceğini gösterir. Değerler, işlem sayısının 10^9'a bölünmesiyle hesaplanmıştır.

f(n)	n = 10	n = 20	n = 50	n = 100	n = 10^6
$\log_2 n$	3,3 ns	4,3 ns	5,6 ns	6,6 ns	19,9 ns
n	10 ns	20 ns	50 ns	100 ns	1 ms
$n \log_2 n$	33 ns	86 ns	282 ns	664 ns	19,9 ms
n^2	100 ns	400 ns	2,5 µs	10 µs	16,7 dakika
n^3	1 µs	8 µs	125 µs	1 ms	31,7 yıl
2^n	1 µs	1 ms	13 gün	$4 \cdot 10^{13}$ yıl	—
$n!$	3,6 ms	77 yıl	10^{48} yıl	$3 \cdot 10^{141}$ yıl	—

Tablo 1.2.8: Saniyede 10^9 işlem yapan bir bilgisayarda yaklaşık çalışma süreleri.

Tablo iki şey söyler. Birincisi, polinom sınıflar bir milyon öğede bile makul kalır; karesel algoritma 17 dakika, kübik algoritma ise otuz yılı aşar. İkincisi, üstel ve faktöriyel sınıflar çok küçük girdilerde bile çalıştırılamaz hâle gelir. $n = 100$ için 2^n adım, evrenin yaşının (yaklaşık $1{,}4 \cdot 10^{10}$ yıl) üç bin katı kadar sürer.

Sunumdaki etkileşimli hesaplayıcı aynı tabloyu istenen n için üretir; "n × 2" düğmesi girdiyi ikiye katlar. Düğmeye basıldığında n sütunu 2, n^2 sütunu 4, n^3 sütunu 8 katına çıkar; $\log_2 n$ yalnız 1 artar, 2^n ise karesini alır. Buradan çıkan ders şudur: on kat hızlı bir bilgisayar sabit bir çarpan kazandırır, daha iyi bir algoritma ise büyüme sınıfını değiştirir. n^2 yerine $n \log n$ süren bir algoritma, $n = 10^6$'da 17 dakikalık işi 20 milisaniyeye indirir; hiçbir donanım iyileştirmesi bu kadar kazandırmaz.

1.2.3 Döngü analizi

Döngüsüz ve özyinelemesiz bir kod parçası sabit sayıda işlem yapar; süresi $\Theta(1)$'dir. Süreyi girdiye bağlayan şey döngüler ve özyinelemedir. Bu kesim döngüleri ele alır. Yöntem tek cümleyle söylenebilir: her döngünün kaç tur döndüğü bulunur, iç içe döngülerde de bu tur sayıları toplanır. Yöntem üç örnek fonksiyon üzerinde uygulanır: tek döngülü f1, iç döngüsünün sınırı dış sayaca bağlı olan f2 ve sayacı alışılmadık hızla büyüyen f6.

Bölümün altı örnek fonksiyonunda ortak bir hile vardır: fonksiyonlar, yaptıkları iş kadar artan bir `x` sayacını döndürür. Böylece dönüş değeri adım sayısının kendisi olur. Analizle bulunan formül, fonksiyon çalıştırılıp dönüş değerine bakılarak doğrudan denetlenebilir. Bu notta verilen bütün dönüş değerleri ve çağrı sayıları Java ile çalıştırılarak elde edilmiştir.

1.2.3.1 f1: tek döngü

```java
public int f1(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    x++;
  }
  return x;
}
```

Kod üç parçadan oluşur.

- **İlk değer verme:** `int x = 0;` satırı `x`'e 0 atar. Bu tek bir işlemdir, girdiye bağlı değildir: $O(1)$.
- **Döngü:** `for (int i = 0; i < n; i++)` başlığında `i` 0'dan başlar, her turda bir artar ve $n - 1$'e kadar ilerler. $0, 1, \ldots, n - 1$ değerleri n tanedir; döngü n tur döner.
- **Gövde:** `x++;` her turda bir kez çalışır ve sabit süreli bir işlemdir: $O(1)$.

Çarpım kuralına göre *n* kez dönen bir döngü, her turda *O(1)* iş yapıyorsa toplam iş *n · O(1) = O(n)*'dir. Döngünün dışındaki ilk değer verme ve `return` sabit işlerdir; toplam kuralına göre sonucu değiştirmezler:

$$T(n) = O(1) + n \cdot O(1) = \Theta(n)$$

Gövde tam *n* kez çalıştığı için alt sınır da *n*'dir; sonuç bu yüzden Θ ile yazılır. Fonksiyon `x = n` döndürür: `f1(16) = 16`, `f1(32) = 32`. Dönüş değeri adım sayısının kendisidir ve girdi iki katına çıkınca iki katına çıkar. Doğrusal büyümenin işareti budur.

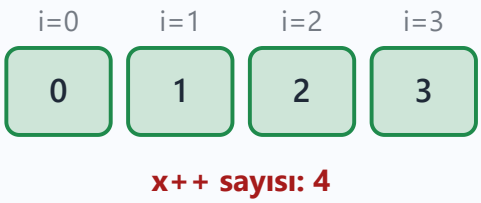
1.2.3.2 Kod izleme: f1(4)

f1'in *n = 4* için çalışmasını izleyelim. Her satırda koşulun değeri ve `x`'in yeni değeri yazılır.

Adım	`i`	Koşul `i < 4`	`x` (gövdeden sonra)
başlangıç	—	—	0
1. tur	0	doğru	1
2. tur	1	doğru	2
3. tur	2	doğru	3
4. tur	3	doğru	4
çıkış	4	yanlış	4 döner

Tablo 1.2.9: `f1(4)` izlemesi.

Koşul beş kez denetlenmiştir (*n + 1*), gövde dört kez çalışmıştır (*n*). Bu sayılar doğrusal aramadaki sayımla aynı biçimdedir. Gövde tam *n* kez çalıştığı için *Θ(n)* sonucu izlemede de görülür. Şekil 1.2.8 izlemenin son durumunu gösterir.



Şekil 1.2.8: `f1(4)` 'ün sonu. Her kutu bir döngü turudur; *i = 0, 1, 2, 3* turlarının dördü de tamamlanmış, `x++` dört kez yapılmıştır.

1.2.3.3 f2: iç içe döngü

```
public int f2(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
      x++;
    }
  }
  return x;
}
```

f2'de iki döngü iç içedir. Dış döngü f1'deki gibidir: `i` 0'dan $n - 1$'e kadar ilerler, n tur döner. İç döngünün sınırı ise sabit değildir. `j` , 0'dan $i \cdot i - 1$'e kadar ilerler; yani iç döngü, dış döngünün o anki turunda i^2 kez döner. $i = 0$'da iç döngü hiç dönmez, $i = 1$'de bir kez, $i = 2$'de dört kez, $i = 3$'te dokuz kez döner.

İç döngünün tur sayısı i'nin güncel değerine bağlı olduğu için çarpım kuralı doğrudan uygulanamaz. "Dış döngü n tur, iç döngü n tur, toplam $n \cdot n$" demek yanlıştır, çünkü iç döngü her turda farklı sayıda döner. Doğru yol, her dış tur için iç döngünün tur sayısını yazıp bunları toplamaktır:

$$T(n) = 0^2 + 1^2 + 2^2 + \ldots + (n-1)^2 = \Sigma_{i=0}^{n-1} i^2$$

Sık yapılan hata

İç içe iki döngü görünce sonucu otomatik olarak $O(n^2)$ saymak. Sonucu iç döngünün sınırı belirler. f2'de sınır i^2 olduğu için sonuç n^3 mertebesindedir. Sınır `j < i` olsaydı karesel, `j < 10` olsaydı doğrusal olurdu. Dış döngü `i *= 2` ile ilerleseydi sonuç yine başka çıkardı.

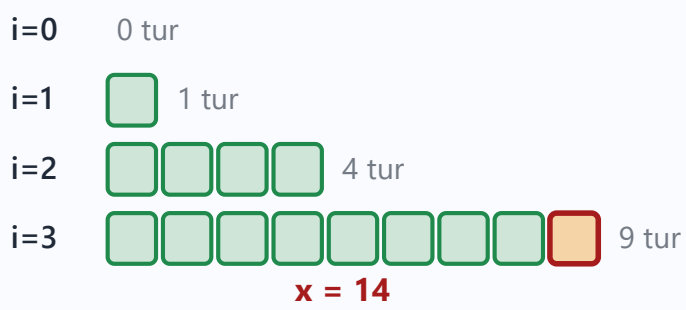
1.2.3.4 Kod izleme: f2(4) ve f2(5)

`f2(4)` çağrısında dış döngü dört tur döner ($i = 0, 1, 2, 3$). Tablo 1.2.10 her dış turdan sonra `x` 'in değerini gösterir. Değerler Java ile çalıştırılarak doğrulanmıştır.

`i`	İç döngü koşulu	İç tur sayısı	`x` (turdan sonra)
0	`j < 0`	0	0
1	`j < 1`	1	1
2	`j < 4`	4	5
3	`j < 9`	9	14
4	dış koşul `4 < 4` yanlış	—	14 döner

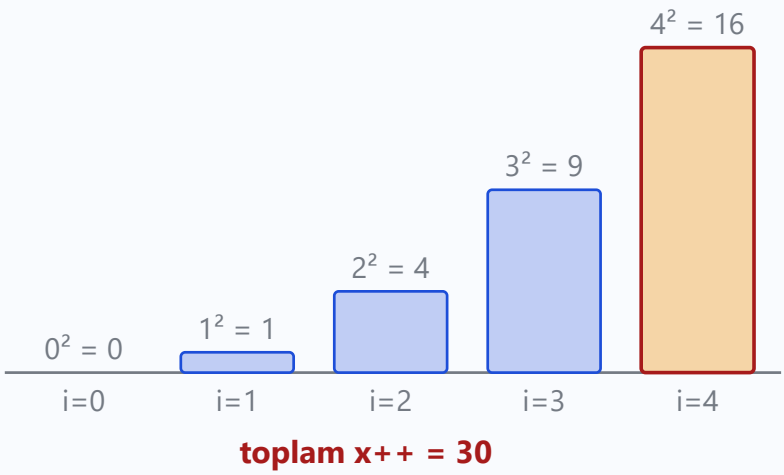
Tablo 1.2.10: `f2(4)` izlemesi.

i = 0 turu ilginçtir: iç döngünün koşulu `j < 0 · 0`, yani `0 < 0`'dır ve ilk denetimde yanlış çıkar. İç döngü hiç dönmez, ama dış tur yine de sayılır. Sonuç *0 + 1 + 4 + 9 = 14*'tür. Şekil 1.2.9 turları satır satır kutularla gösterir: *i*. satırda i^2 kutu vardır.



Şekil 1.2.9: `f2(4)`'te iç döngünün turları. Her satır bir dış tur, her kutu bir `x++`'tır. Satırların uzunluğu 0, 1, 4, 9'dur; toplam 14.

Bir adım daha büyük girdide örüntü daha belirgin olur. `f2(5)`'te dış döngü bir tur daha döner ve *i = 4*'te iç döngü 16 kez çalışır. Toplam *0 + 1 + 4 + 9 + 16 = 30*'dur. Şekil 1.2.10 her dış turu bir çubukla gösterir. Çubukların yüksekliği i^2'dir ve toplam alan fonksiyonun yaptığı `x++` sayısıdır. Son çubuk tek başına toplamın yarısından fazlasını oluşturur: son turlar baskındır.



Şekil 1.2.10: `f2(5)`'te her dış turda iç döngünün tur sayısı. Çubuklar 0, 1, 4, 9 ve 16 yüksekliğindedir; toplamları 30'dur ve `f2(5)` 30 döndürür.

1.2.3.5 Toplamı hesaplamak

Ardışık karelerin toplamı için kapalı bir formül vardır:

$$\Sigma_{i=0}^{n-1} i^2 = (n-1) \cdot n \cdot (2n-1) / 6 = (2n^3 - 3n^2 + n) / 6$$

Formül küçük değerlerle denetlenebilir. *n = 5* için *4 · 5 · 9 / 6 = 30* bulunur; bu, izlemedeki toplamdır. *n = 4* için *3 · 4 · 7 / 6 = 14* bulunur. Açılmış biçimde baskın terim $2n^3/6 = n^3/3$'tür. Sabit *1/3* ve düşük dereceli terimler atılınca sonuç çıkar:

$$T(n) = \Theta(n^3)$$

n	f2(n)	$n^3/3$
4	14	21
5	30	42
16	1.240	1.365
32	10.416	10.923

Tablo 1.2.11: `f2(n)`'nin dönüş değeri ile $n^3/3$'ün karşılaştırması (Java ile ölçülmüştür).

Tablodaki iki sütun *n* büyüdükçe birbirine yaklaşır. *n = 4*'te oran 14/21 ≈ 0,67 iken *n = 32*'de 10.416/10.923 ≈ 0,95'tir. Düşük dereceli $-3n^2/6$ terimi küçük *n*'de etkilidir, büyük *n*'de önemini yitirir. Kübik büyümenin bir işareti de ikiye katlama testidir. *n*, 16'dan 32'ye çıkınca dönüş değeri *10.416 / 1.240 = 8,4* katına çıkar. Bu sayı $2^3 = 8$'e yakındır; *n* büyüdükçe oran 8'e daha da yaklaşır.

Bu kesimdeki sonuç genel bir kuralın örneğidir: $\Sigma_{i<n} i^k = \Theta(n^{k+1})$. İç döngünün tur sayısı dış sayacın *k*. kuvvetiyse toplam iş bir derece yükselir. Kural, toplamın bir integrale benzemesinden gelir: $\int_0^n x^k\,dx = n^{k+1}/(k+1)$. f2 için *k = 2*'dir ve integral $n^3/3$ verir; açık formüldeki baskın terimle aynıdır.

1.2.3.6 Döngü kalıpları ve toplam formülleri

Döngü analizinde birkaç kalıp tekrar tekrar karşımıza çıkar. Her biri bilinen bir seriye indirgenir: aritmetik seri, kareler toplamı ya da geometrik seri. Tablo 1.2.12 bu kalıpları toplar. Tur sayıları *n = 16* ve *n = 20* için Java ile sayılarak denetlenmiştir.

Kalıp	Tur sayısı	n = 16	Sonuç
`for (i = 0; i < n; i++)`	$\Sigma\ 1 = n$	16	$\Theta(n)$
`i < n` içinde `for (j = 0; j < n; j++)`	$n \cdot n$	256	$\Theta(n^2)$
`i < n` içinde `for (j = 0; j < i; j++)`	$\Sigma\ i = n(n-1)/2$	120	$\Theta(n^2)$
`i < n` içinde `for (j = 0; j < i * i; j++)`	$\Sigma\ i^2 = (n-1)n(2n-1)/6$	1.240	$\Theta(n^3)$
`for (i = 1; i < n; i *= 2)`	$\lceil \log_2 n \rceil$	4	$\Theta(\log n)$
`for (i = n; i > 0; i /= 2)` içinde `j < n`	$n \cdot (\lfloor \log_2 n \rfloor + 1)$	80	$\Theta(n \log n)$

Kalıp	Tur sayısı	n = 16	Sonuç
`for (i = 1; i < n; i *= 2)` içinde `j < i`	1 + 2 + 4 + … < 2n	15	Θ(n)
`for (i = 0; i < n; i = 1 << i)` (f6)	log* n + 1	4	Θ(log* n)

Tablo 1.2.12: Sık karşılaşılan döngü kalıpları.

Tablonun üçüncü satırı, f2'nin bir derece düşük kardeşidir. İç döngü i kez döner; toplam $0 + 1 + \ldots + (n - 1) = n(n - 1)/2$'dir. Bu sayı $n \cdot n$'nin yaklaşık yarısıdır; sabit çarpan farkı olduğu için sınıf yine $\Theta(n^2)$'dir. Seçmeli ve eklemeli sıralamanın karesel olmasının sebebi bu kalıptır.

Beşinci satır çarpımsal bir döngüdür. `i` her turda iki katına çıkar: 1, 2, 4, 8, …. k turdan sonra $i = 2^k$ olur. Döngü $2^k < n$ iken sürer; $k \geq \log_2 n$ olunca durur. Tur sayısı bu yüzden $\lceil \log_2 n \rceil$'dir. $n = 16$'da `i` 1, 2, 4, 8 değerlerini alır ve 4 tur döner; $n = 1000$'de 1'den 512'ye kadar 10 tur döner. Sayacı sabit bir oranla büyüyen ya da küçülen her döngü logaritmiktir.

Altıncı satırda dış döngü logaritmik, iç döngü doğrusaldır. İç döngünün sınırı dış sayaca bağlı değildir; her turda tam n kez döner. Bu yüzden çarpım kuralı uygulanır: $(\lfloor \log_2 n \rfloor + 1) \cdot n$. $n = 16$ için dış döngü `i` = 16, 8, 4, 2, 1 değerlerinde 5 tur döner; toplam $5 \cdot 16 = 80$'dir.

Yedinci satır şaşırtıcıdır. Görünüşte yapısı bir öncekine benzer: logaritmik bir dış döngü ve onun içinde bir iç döngü. Ama iç döngünün sınırı i'dir ve i hızla büyür. Toplam, bir geometrik seridir:

$$1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1 < 2n$$

Burada 2^k, n'den küçük en büyük ikinin kuvvetidir; bu yüzden $2^{k+1} < 2n$ olur. Dış döngü $\log n$ kez dönse de toplam iş doğrusaldır. $n = 16$'da $1 + 2 + 4 + 8 = 15$ tur yapılır. Geometrik serinin özelliği budur: toplamı son terimi belirler, son terim de öncekilerin hepsinin toplamından büyüktür.

Örnek 1.2.3

Aşağıdaki iki kod parçasının `x++` sayısı ve büyüme sınıfı bulunacak.

```
// birinci parça
for (int i = 1; i < n; i *= 2)
  for (int j = 0; j < i; j++)
    x++;

// ikinci parça
for (int i = n; i > 0; i /= 2)
  for (int j = 0; j < n; j++)
    x++;
```

1. **Birinci parça.** `i` , 1, 2, 4, …, $2^k < n$ değerlerini alır. Her turda iç döngü `i` kez döner. Toplam $1 + 2 + \ldots + 2^k = 2^{k+1} - 1 < 2n$'dir: $\Theta(n)$. $n = 16$ için 15 kez `x++` yapılır.

2. **İkinci parça.** Dış döngü `i` = *n, n/2, n/4, …, 1* değerlerinde $\lfloor \log_2 n \rfloor + 1$ tur döner. İç döngü her turda *n* kez döner, çünkü sınırı *n*'dir. Toplam $n \cdot (\lfloor \log_2 n \rfloor + 1)$: $\Theta(n \log n)$. *n = 10* için `i` = 10, 5, 2, 1 olur ve *4 · 10 = 40* kez `x++` yapılır.

İki parçanın dış döngüleri benzer sayıda döner; farkı iç döngünün sınırı yaratır. Sınır dış sayaca bağlıysa tur sayıları toplanır, bağlı değilse çarpılır.

Sık yapılan hata

Logaritmik bir dış döngünün içinde bir iç döngü görünce sonucu her zaman $\Theta(n \log n)$ saymak. Yukarıdaki örnekteki birinci parça, dış döngü *log n* kez döndüğü hâlde yalnız $\Theta(n)$'dir. Her dış tur için iç döngünün tur sayısı yazılmadan ve toplanmadan karar verilmez.

1.2.3.7 f6: kaydırmalı döngü

```java
public static int f6(int n) {
  int x = 0;
  // 1 << i, 2^i ile aynıdır (taşma yok sayılır)
  for (int i = 0; i < n; i = 1 << i) {
    x++;
  }
  return x;
}
```

f6'nın döngüsü ilk bakışta çarpımsal döngüye benzer, ama güncellemesi farklıdır.

- **İlk değer:** `int i = 0;` sabit süreli bir atamadır.
- **Koşul:** `i < n`, her turun başında `i`'nin *n*'den küçük olup olmadığını denetler.
- **Güncelleme:** `i = 1 << i`. Sola kaydırma işleci `<<`, 1 sayısının bitlerini `i` basamak sola kaydırır. İkili tabanda 1'i *i* basamak sola kaydırmak, onu 2^i ile çarpmaktır. Sonuç 2^{i}'dir ve tek bir işlemde, sabit sürede hesaplanır.

Kritik nokta şudur: güncelleme `i = 2 * i` değil, `i = 2`$^{\texttt{i}}$'dir. Çarpımsal döngüde `i` her turda iki katına çıkar. f6'da ise `i`'nin kendisi üs olur. Bu, `i`'yi çok daha hızlı büyütür: 4 değeri bir sonraki turda 8 değil, $2^4 = 16$ olur. 16 da 32 değil, $2^{16} = 65.536$ olur.

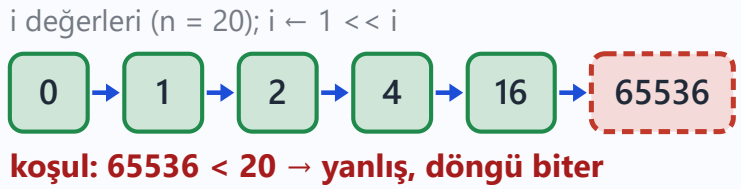
1.2.3.8 Kod izleme: f6(20)

Tur	`i`	Koşul `i < 20`	`x`	Sonraki `i = 1 << i`
1	0	doğru	1	1 << 0 = 1
2	1	doğru	2	1 << 1 = 2

Tur	`i`	Koşul `i < 20`	`x`	Sonraki `i = 1 << i`
3	2	doğru	3	1 << 2 = 4
4	4	doğru	4	1 << 4 = 16
5	16	doğru	5	1 << 16 = 65.536
—	65.536	yanlış	5 döner	—

Tablo 1.2.13: `f6(20)` izlemesi.

Döngü yalnız beş tur döner. `i`'nin aldığı değerler 0, 1, 2, 4, 16'dır; bir sonraki değer 65.536 olduğu için döngü biter. Şekil 1.2.11 bu zinciri gösterir. Aynı döngü *n = 65.536*'da da beş tur döner; *n* üç bin kat büyüdüğü hâlde tur sayısı değişmez.



Şekil 1.2.11: `f6(20)`'de `i`'nin aldığı değerler: 0 → 1 → 2 → 4 → 16 → 65.536. Son değer koşulu bozar (*65.536 < 20* yanlış) ve döngü beş turda biter.

1.2.3.9 f6 gerçekte kaç tur döner?

f6'nın `i` değerleri, ikinin kuvvetlerinden oluşan bir kule gibi büyür:

$$0 \to 2^0 = 1 \to 2^1 = 2 \to 2^2 = 4 \to 2^4 = 16 \to 2^{16} = 65.536 \to 2^{65.536}$$

Son değer, 19.729 basamaklı bir sayıdır. Evrendeki atom sayısının yaklaşık 10^{80} olduğu tahmin edilir; bu sayının yanında 10^{80} çok küçük kalır. Hiçbir gerçek girdi bu kadar büyük olamayacağına göre f6, akla gelebilecek her *n* için en çok altı tur döner.

Bu döngü için sık görülen bir akıl yürütme şöyledir: "*k* turdan sonra $i \geq 2^k$ olur; döngü $2^k \geq n$ olunca durur; öyleyse $k \approx \log_2 n$ ve süre $O(\log n)$." Akıl yürütme, `i = 2 * i` döngüsü için doğrudur. f6'da da sonuç yanlış değildir, çünkü `i` en az iki katına çıktığı için tur sayısı $\log_2 n$'yi aşamaz. Ama bu sınır sıkı değildir. f6'da *k* turdan sonra `i`, $k - 1$ katlı bir kuledir ve tur sayısı çok daha yavaş büyüyen bir fonksiyonla verilir.

Tanım 1.2.5

Yinelenmiş logaritma (*iterated logarithm*) $\log^* n$, sonuç 1'e eşit ya da 1'den küçük olana kadar *n*'ye art arda kaç kez $\log_2$ uygulanması gerektiğini verir. Örneğin $\log_2 16 = 4$, $\log_2 4 = 2$, $\log_2 2 = 1$ olduğundan $\log^* 16 = 3$'tür.

f6'nın tur sayısı tam olarak *log* n + 1*'dir. Bu eşitlik *n = 1*'den 70.000'e kadar her değer için bir Python betiğiyle denetlenmiştir. *log* n*, akla gelebilecek her girdide 5'i geçmez: *log* 65.536 = 4*'tür ve 5'e ulaşmak için *n*'nin 65.536'yı aşması gerekir. Değeri 6 olan en küçük sayı ise $2^{65.536}$'dan büyüktür. Pratikte sabit gibi davranan bu fonksiyon, birleşim-bul (*union-find*) veri yapısının analizinde de karşımıza çıkar.

n	f6(n)	$\lceil \log_2 n \rceil$	log* n
4	3	2	2
16	4	4	3
32	5	5	4
1.000	5	10	4
65.536	5	16	4

Tablo 1.2.14: `f6(n)`'nin dönüş değeri ile $\lceil \log_2 n \rceil$'nin karşılaştırması (Java ile ölçülmüştür).

Küçük *n*'de f6 ile $\lceil \log_2 n \rceil$ birbirine yakındır; *n = 32*'de ikisi de 5'tir. *n* büyüdükçe logaritma artmayı sürdürür, f6 ise 5'te kalır. Sonuç: $O(\log n)$ doğru bir üst sınırdır, ama sıkı sınır $\Theta(\log^* n)$'dir.

f6'nın kodundaki yorum satırı "taşma yok sayılır" der. Bu varsayım Java'da önemlidir. Java'da `int` 32 bitliktir ve kaydırma işleci, kaydırma miktarının yalnız son beş bitini kullanır; yani miktarın 32'ye göre kalanını alır. *65.536 = 32 · 2048* olduğundan `1 << 65536`, `1 << 0` ile aynıdır ve 1 verir. Bu yüzden *n > 65.536* için `i` 0, 1, 2, 4, 16, 65.536, 1, 2, 4, 16, 65.536, … döngüsüne girer; koşul hiç yanlış olmaz ve f6 hiç bitmez. Bu davranış, *n = 100.000* ile çalıştırılan ve tur sayısı sınırlanan bir kopyayla gözlenmiştir. Analizdeki $\Theta(\log^* n)$ sonucu, sınırsız tam sayılarla çalışan soyut makine için geçerlidir.

Sık yapılan hata

Kaydırmayı çarpmayla karıştırmak. `i = 1 << i` güncellemesi `i = 2 * i` değil, `i = 2^i` 'dir. İlki `i <<= 1` ya da `i = i << 1` diye yazılır. Kaydırılan sayı ile kaydırma miktarının yeri karışırsa döngünün tur sayısı *log n* ile *log* n* arasında değişir.

1.2.4 Özyinelemeli analiz

Döngülü kodda tur sayıları toplanır. Özyinelemeli kodda ise süre, aynı fonksiyonun daha küçük girdilerdeki süresine bağlıdır. *T(n)*'yi bulmak için önce *T(n − 1)* ya da *T(n/2)* gibi değerler gerekir. Bu ilişkiyi yazan denkleme yineleme bağıntısı denir. Bu kesim bağıntının nasıl kurulduğunu ve en temel çözüm yöntemi olan açma yöntemini anlatır.

1.2.4.1 Yineleme bağıntısı

Tanım 1.2.6

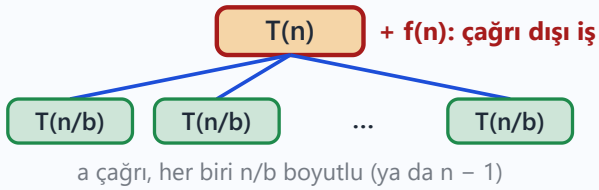
Yineleme bağıntısı (*recurrence relation*), *T(n)*'yi daha küçük girdilerdeki *T* değerleri cinsinden tanımlayan denklemdir. Özyinelemeli bir fonksiyonun süresi için genel biçim şudur:

> T(n) = (alt çağrıların süreleri) + (çağrı dışında yapılan iş), T(taban) = Θ(1)

İkinci denklem, özyinelemenin kendini çağırmadan doğrudan çözdüğü **taban durumun** (*base case*) süresini verir.

Bağıntıyı kurmak için koda üç soru sorulur:

1. **Kaç özyinelemeli çağrı var?** Bu sayı, bağıntıda *T*'nin katsayısı olur.
2. **Her çağrının girdisi ne kadar küçülüyor?** Girdi bir azalıyorsa *T(n − 1)*, yarıya iniyorsa *T(n/2)* yazılır.
3. **Çağrıların dışında ne kadar iş yapılıyor?** Bölme, birleştirme, toplama gibi işlerin süresi bağıntıya ek terim olarak girer.



Şekil 1.2.12: Bir özyinelemeli çağrının yapısı. *T(n)* süren çağrı *a* alt çağrı başlatır; her biri daha küçük bir girdiyle (*n/b* ya da *n − 1*) çalışır. Çağrının kendi yaptığı iş *f(n)*'dir.

Bağıntı kurulduktan sonra çözülmesi gerekir; yani *T(n)*'nin, içinde *T* geçmeyen kapalı bir biçimi bulunur. Üç yöntem vardır:

- **Açma** (yerine koyma): Bağıntı kendi içine tekrar tekrar konur, bir örüntü bulunur, taban duruma ulaşınca terimler toplanır. f3'te kullanılır.
- **Özyineleme ağacı** (*recursion tree*): Her çağrı bir düğüm olarak çizilir, her düzeydeki iş toplanır, sonra düzey toplamları toplanır. f3, f4 ve f5'te kullanılır.
- **Ana Teorem** (*master theorem*): *T(n) = aT(n/b) + f(n)* biçimindeki bağıntılar için hazır bir sonuç verir. f4 ve f5'te kullanılır.

1.2.4.2 f3: iki çağrı, n − 1

```
public long f3(int n) {
  if (n <= 1) {
    return 1;
  }
  return f3(n - 1) + f3(n - 1);
}
```

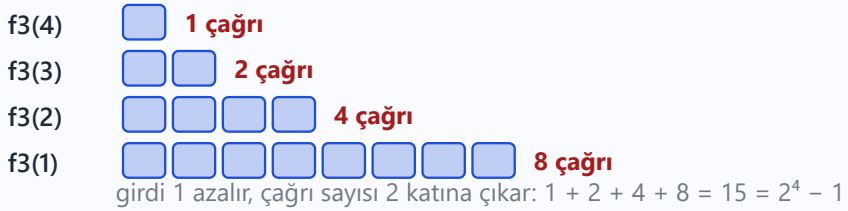
f3 iki parçadan oluşur.

- **Taban durum:** $n \le 1$ iken fonksiyon 1 döndürür. Bu sabit süreli bir iştir: $T(1) = c$.
- **Özyinelemeli durum:** $n > 1$ iken `f3(n - 1)` iki kez çağrılır ve sonuçlar toplanır. Toplama ve karşılaştırma sabit bir iştir; buna da c denir.

Üç soru cevaplanınca bağıntı ortaya çıkar: iki çağrı vardır, her biri $n - 1$ boyutludur, çağrı dışında sabit iş yapılır.

$$T(n) = 2T(n-1) + c, \quad n > 1; \qquad T(1) = c$$

Girdi her çağrıda yalnız bir azalır, ama çağrı sayısı her düzeyde iki katına çıkar. `f3(4)` , iki `f3(3)` 'ü; her `f3(3)` iki `f3(2)` 'yi; her `f3(2)` de iki `f3(1)` 'i çağırır. Düzeylerde 1, 2, 4, 8 çağrı vardır ve toplam $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ eder (Şekil 1.2.13).



Şekil 1.2.13: `f3(4)` 'ün düzeyleri. Girdi her düzeyde bir azalır, çağrı sayısı ise iki katına çıkar: 1, 2, 4, 8. Toplam $15 = 2^4 - 1$ çağrı.

Bu bağıntı $T(n/b)$ biçiminde değildir; alt problem bölünerek değil bir çıkarılarak küçülür. Bu yüzden Ana Teorem uygulanamaz ve bağıntı açma yöntemiyle çözülür.

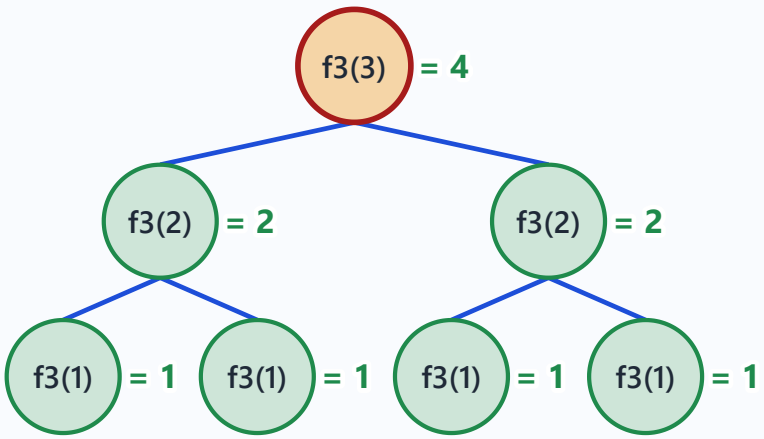
1.2.4.3 Kod izleme: f3(3)

Özyinelemeli kodun izlenmesinde çağrıların hangi sırayla başlayıp bittiği önemlidir. `f3(3)` çağrısının adımları şöyledir; girinti, çağrının derinliğini gösterir. Sıra Java ile çalıştırılan ve her çağrının başını ve sonunu yazdıran bir kopyayla doğrulanmıştır.

Adım	Olay	Yığın (alttan üste)	Çağrı
1	`f3(3)` çağrılır; 3 > 1, taban durum değil	f3(3)	1
2	soldaki `f3(2)` çağrılır	f3(3) f3(2)	2
3	`f3(1)` : taban durum, 1 döner	f3(3) f3(2) f3(1)	3
4	ikinci `f3(1)` : 1 döner	f3(3) f3(2) f3(1)	4
5	`f3(2) = 1 + 1 = 2` döner	f3(3)	4
6	sağdaki `f3(2)` çağrılır; aynı iş baştan yapılır	f3(3) f3(2)	5
7	iki `f3(1)` , her biri 1 döner	f3(3) f3(2) f3(1)	7
8	ikinci `f3(2) = 2` döner	f3(3)	7
9	`f3(3) = 2 + 2 = 4` döner	boş	7

Tablo 1.2.15: `f3(3)` izlemesi. Son sütun o ana kadar başlatılan çağrı sayısıdır.

İzleme üç şey gösterir. Toplam $7 = 2^3 - 1$ çağrı yapılmıştır. Yığında aynı anda en çok üç çerçeve bulunmuştur; bu, n'ye eşit olan derinliktir. Üçüncüsü ve en önemlisi, sağdaki `f3(2)` soldakinin yaptığı işi aynen tekrarlar. f3'ün üstel süresinin kaynağı bu tekrardır. Kod `return 2 * f3(n - 1);` diye yazılsaydı dönüş değeri aynı kalırdı (2^{n-1}), ama her düzeyde tek çağrı yapılırdı. Bağıntı $T(n) = T(n-1) + c$'ye iner ve süre $\Theta(n)$ olur.



Şekil 1.2.14: `f3(3)` 'ün tamamlanmış çağrı ağacı. Her düğümün yanında döndürdüğü değer yazılıdır. Dört yaprak 1, iki `f3(2)` 2, kök 4 döndürür. Ağaçta 7 düğüm vardır.

1.2.4.4 Bağıntıyı açma

Açma yönteminde bağıntının sağ tarafındaki $T(n-1)$, kendi tanımıyla değiştirilir. İşlem birkaç kez tekrarlanınca katsayılarda bir örüntü belirir. Örüntü genel biçimde yazılır ve taban duruma ulaşılan adım bulunur.

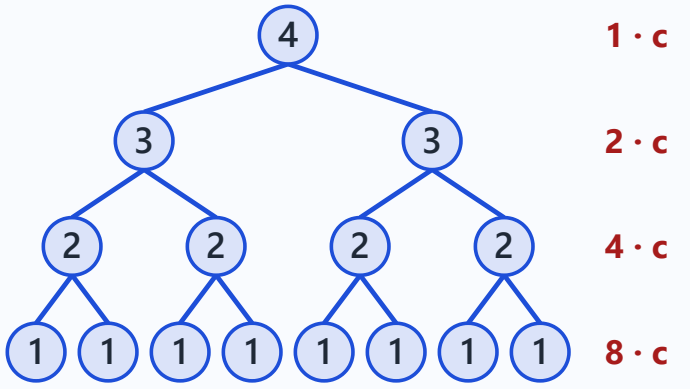
Örnek 1.2.4

$T(n) = 2T(n-1) + c$, $T(1) = c$ bağıntısı açma yöntemiyle çözülecek.

1. **Bağıntının kendisi:** $T(n) = 2T(n-1) + c$. İki alt çağrı, sabit ek iş.
2. **Bir kez açmak:** $T(n-1) = 2T(n-2) + c$ yerine konur: $T(n) = 2[2T(n-2) + c] + c = 4T(n-2) + 3c$.
3. **Bir kez daha:** $T(n-2) = 2T(n-3) + c$ konur: $T(n) = 4[2T(n-3) + c] + 3c = 8T(n-3) + 7c$.
4. **Örüntü:** T'nin katsayıları 2, 4, 8; c'nin katsayıları 1, 3, 7'dir. Birinciler ikinin kuvvetleri, ikinciler ikinin kuvvetlerinin bir eksiğidir. k adım sonra: $T(n) = 2^k\,T(n-k) + (2^k - 1)c$.
5. **Taban duruma ulaşmak:** $n - k = 1$, yani $k = n - 1$ olunca $T(1) = c$ kullanılır: $T(n) = 2^{n-1}c + (2^{n-1} - 1)c = (2^n - 1)c$.
6. **Sonuç:** $T(n) = \Theta(2^n)$.

Sonuç, f3'ün çağrı sayısıyla birebir uyuşur: her çağrı c iş yapar ve $2^n - 1$ çağrı vardır. Formül küçük değerlerle denetlenebilir: $T(1) = c$, $T(2) = 3c$, $T(3) = 7c$. Java ile sayıldığında `f3(16)` 65.535, `f3(32)` ise 4.294.967.295 çağrı yapar; ikisi de $2^n - 1$'dir. n 16'dan 32'ye çıkınca çağrı sayısı 65.537 katına çıkar. Üstel büyümenin işareti budur: girdi iki katına çıkınca süre karesine yakın bir değere ulaşır.

Açma yönteminde ek işi (c) ihmal edip $T(n) = 2^k\,T(n-k)$ yazmak da olur. Bu durumda $T(n) = 2^{n-1}\,T(1)$ çıkar ve büyüme sınıfı değişmez. Ek iş, yalnız sabit çarpanı ikiye katlar.



Şekil 1.2.15: `f3(4)`'ün özyineleme ağacı. Düğümlerdeki sayı girdidir. Düzeylerin işi $c, 2c, 4c, 8c$'dir; toplam $15c = (2^4 - 1)c$.

Açma yöntemi başka çıkarmalı bağıntılara da uygulanır. $T(n) = T(n-1) + n$, $T(1) = 1$ bağıntısı açılınca $T(n) = n + (n-1) + \ldots + 2 + 1 = n(n+1)/2$ bulunur: $\Theta(n^2)$. Bu bağıntı, her adımda n iş yapıp problemi yalnız bir küçülten algoritmalarda çıkar. Hızlı sıralamanın en kötü durumu buna bir örnektir.

Sık yapılan hata

"İki özyinelemeli çağrı varsa süre üsteldir" kuralını ezberlemek. f3'te iki çağrı $\Theta(2^n)$ verir, çünkü girdi yalnız bir azalır ve ağacın derinliği n'dir. Girdi yarıya inseydi derinlik $\log_2 n$ olurdu ve süre doğrusala inerdi. Belirleyici olan çağrı sayısı değil, çağrı sayısı ile küçülme biçiminin birlikte etkisidir.

1.2.5 Ana Teorem

Açma yöntemi her bağıntıda işe yarar, ama zahmetlidir. Algoritmaların büyük bir kısmı, problemi eşit büyüklükte parçalara bölen **böl ve fethet** (*divide and conquer*) yaklaşımını kullanır. Bu algoritmaların bağıntıları hep aynı biçimdedir. Ana Teorem, bu biçimdeki bağıntılar için hazır bir sonuç verir. Teorem, Jon Bentley, Dorothea Haken ve James Saxe'ın 1980'de yayımladığı genel bir yönteme dayanır; "Ana Teorem" adı ise Cormen ve arkadaşlarının ders kitabıyla yaygınlaştı.

1.2.5.1 Böl ve fethet bağıntısı

Böl ve fethet bir algoritma, n boyutlu problemi her biri n/b boyutlu a alt probleme böler, alt problemleri özyinelemeyle çözer ve sonuçları birleştirir. Bağıntısı şöyledir:

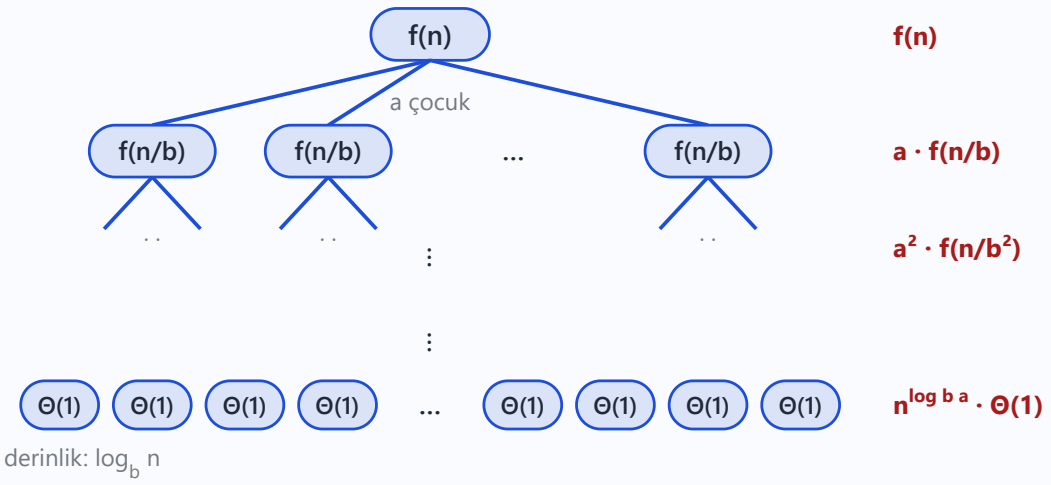
$$T(n) = a \cdot T(n/b) + f(n), \quad a \geq 1, \quad b > 1$$

- a: alt problem (özyinelemeli çağrı) sayısı.
- b: küçülme oranı; her alt problemin boyutu n/b'dir.
- $f(n)$: çağrının kendi yaptığı iş, yani problemi bölme ve sonuçları birleştirme işi.

Bu bağıntının özyineleme ağacı düzenli bir yapıdadır (Şekil 1.2.16). Kökte tek bir düğüm vardır ve $f(n)$ iş yapar. Her düğümün a çocuğu vardır; j. düzeyde a^j düğüm bulunur ve her biri n/b^j boyutlu bir problemle uğraşır. Girdi her düzeyde b'ye bölündüğü için ağacın derinliği $\log_b n$'dir. Son düzeydeki yaprak sayısı

$$a^{\log_b n} = n^{\log_b a}$$

olur. İki ifadenin eşitliği, iki tarafın da $\log_b$'si alınınca görülür: ikisi de $\log_b n \cdot \log_b a$ verir. Her yaprak sabit iş yaptığı için yaprakların toplam işi $\Theta(n^{\log_b a})$'dır. Bu nicelik teoremin merkezindedir: $f(n)$ her zaman onunla karşılaştırılır.



Şekil 1.2.16: *T(n) = aT(n/b) + f(n)* bağıntısının özyineleme ağacı. Kökte *f(n)*, birinci düzeyde toplam $a \cdot f(n/b)$, ikinci düzeyde $a^2 \cdot f(n/b^2)$ iş vardır. Derinlik $\log_b n$'dir; yapraklarda toplam $n^{\log_b a} \cdot \Theta(1)$ iş yapılır.

1.2.5.2 Üç durum

Teorem, *f(n)* ile yaprak işi $n^{\log_b a}$'yı karşılaştırır. Karşılaştırmanın üç sonucu vardır: kök tarafındaki iş baskındır, yapraklardaki iş baskındır ya da düzeyler dengelidir. Aşağıda teoremin, *f(n)*'nin polinom derecesi *c* ile yazılan sık kullanılan biçimi verilir.

Teorem 1.2.1 (Ana Teorem)

$a \geq 1$ ve $b > 1$ sabitler, $T(n) = aT(n/b) + f(n)$ olsun.

1. Bir $c < \log_b a$ için $f(n) = O(n^c)$ ise $T(n) = \Theta(n^{\log_b a})$. (Yapraklar baskın.)
2. $c = \log_b a$ için $f(n) = \Theta(n^c)$ ise $T(n) = \Theta(n^{\log_b a} \cdot \log n)$. (Düzeyler dengeli.)
3. Bir $c > \log_b a$ için $f(n) = \Omega(n^c)$ ise ve bir $k < 1$ sabiti ile yeterince büyük her n için $a \cdot f(n/b) \leq k \cdot f(n)$ sağlanıyorsa $T(n) = \Theta(f(n))$. (Kök baskın.)

Durum	Koşul	Sonuç	Yorum
1	$f(n) = O(n^c)$, $c < \log_b a$	$\Theta(n^{\log_b a})$	yapraklar baskın
2	$f(n) = \Theta(n^c)$, $c = \log_b a$	$\Theta(n^{\log_b a} \cdot \log n)$	düzeyler dengeli
3	$f(n) = \Omega(n^c)$, $c > \log_b a$ ve $a \cdot f(n/b) \leq k \cdot f(n)$, $k < 1$	$\Theta(f(n))$	kök baskın

Tablo 1.2.16: Ana Teoremin üç durumu (*c*: *f(n)*'nin polinom derecesi).

Üç durumda farklı gösterimlerin kullanılması rastlantı değildir. Durum 1'de *f(n)* için bir üst sınır yeter, çünkü yaprak işi zaten *f*'den büyüktür ve onu bastırır. Durum 2 ve 3'te ise yalnız $f(n) = O(n^c)$ yazmak yetmez. Örneğin $f(n) = 1$ de $O(n^2)$'dir; ama $T(n) = T(n/2) + 1$ bağıntısının çözümü $\Theta(n^2)$ olamaz. Bu yüzden Durum 2'de $f = \Theta(n^c)$, Durum 3'te $f = \Omega(n^c)$ ve ayrıca düzenlilik koşulu istenir. Bu koşullar sağlanınca sonuçlar Θ ile, yani sıkı olarak verilir.

Düzenlilik koşulu (*regularity condition*), Durum 3'e özgüdür: $a \cdot f(n/b) \le k \cdot f(n)$. Anlamı şudur: kökün çocuklarının toplam işi, kökün kendi işinin sabit bir kesrinden fazla değildir. Böylece iş, köke doğru giderek büyür. $f(n) = n^c$ gibi polinomlarda koşul kendiliğinden sağlanır: $a(n/b)^c = (a/b^c)\, n^c$'dir ve $c > \log_b a$ olduğundan $a/b^c < 1$'dir; $k = a/b^c$ seçilebilir.

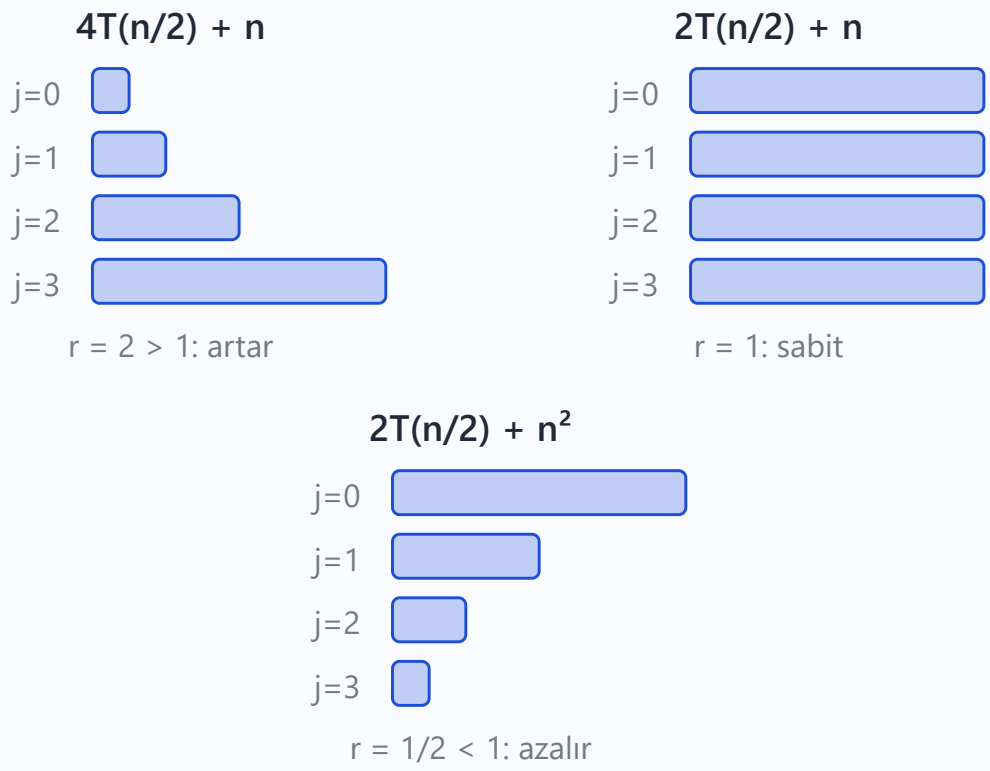
1.2.5.3 Neden üç durum? Düzey toplamları

Teoremin neden üç durumdan oluştuğu, özyineleme ağacının düzey toplamlarına bakınca anlaşılır. $f(n) = n^c$ olsun. j. düzeyde a^j düğüm vardır ve her biri $(n/b^j)^c$ iş yapar. Düzeyin toplam işi

$$a^j \cdot (n/b^j)^c = n^c \cdot (a/b^c)^j$$

olur. Düzey toplamları, oranı $r = a/b^c$ olan bir **geometrik seri** oluşturur. $T(n)$, bu serinin $j = 0$'dan $\log_b n$'ye kadar toplamıdır. Geometrik serinin davranışı ise yalnız orana bağlıdır:

- ***r > 1*: seri artar.** Toplamı son terim, yani yapraklar belirler. Bu Durum 1'dir. Örnek: $4T(n/2) + n$ için $r = 4/2 = 2$; düzey toplamları $n, 2n, 4n, \ldots$ diye artar ve sonuç $\Theta(n^2)$ çıkar.
- ***r = 1*: terimler eşittir.** Her düzey aynı işi yapar; toplam, düzey sayısı ile düzey işinin çarpımıdır. Bu Durum 2'dir. Örnek: $2T(n/2) + n$ için $r = 2/2 = 1$; $\log n$ düzeyin her biri n iş yapar ve sonuç $\Theta(n \log n)$ çıkar.
- ***r < 1*: seri azalır.** Toplamı ilk terim, yani kök belirler. Bu Durum 3'tür. Örnek: $2T(n/2) + n^2$ için $r = 2/4 = 1/2$; düzey toplamları $n^2, n^2/2, n^2/4, \ldots$ diye azalır, toplamları $2n^2$'yi geçmez ve sonuç $\Theta(n^2)$ çıkar.



Şekil 1.2.17: Üç durumun düzey toplamları; her çubuk bir düzeyin toplam işidir. Solda *4T(n/2) + n*: çubuklar uzar (*r = 2*), yapraklar baskın. Ortada *2T(n/2) + n*: çubuklar eşit (*r = 1*). Sağda *2T(n/2) + n²*: çubuklar kısalır (*r = 1/2*), kök baskın.

Oranın 1'den büyük olması ile $c < \log_b a$ aynı şeydir: $r > 1 \Leftrightarrow b^c < a \Leftrightarrow c < \log_b a$. Ana Teoremin üç durumu böylece geometrik serinin artan, sabit ya da azalan olmasına karşılık gelir. Teoremi ezberlemek yerine bu tabloyu akılda tutmak yeter: a ile b^c karşılaştırılır.

Bu sonuçlar küçük bir hesapla da denetlenebilir. Bağıntılar $T(1) = 1$ ile doğrudan hesaplandığında $n = 16$ için $2T(n/2) + n$ 80 verir; bu, $n \log_2 n + n$ değeridir. $2T(n/2) + n^2$ ise 496 verir, yani $2n^2 - n$; kökteki $n^2 = 256$ toplamın yarısından fazlasıdır.

1.2.5.4 Örnek: 2T(n/2) + n

Örnek 1.2.5

$T(n) = 2T(n/2) + \Theta(n)$ bağıntısı Ana Teorem ile çözülecek.

1. **Parametreleri okumak.** İki alt problem ($a = 2$), her biri yarı boyutta ($b = 2$), doğrusal birleştirme işi ($f(n) = \Theta(n)$).
2. **Yaprak işinin üssü.** $\log_b a = \log_2 2 = 1$; yaprak işi $n^1 = n$'dir.
3. ***f(n)*'nin derecesi.** $f(n) = \Theta(n^1)$, yani $c = 1$.
4. **Karşılaştırma.** $c = \log_b a = 1$: üsler eşittir, her düzey aynı (n) işi yapar. Durum 2.

5. **Sonuç.** $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

Bu bağıntı algoritmaların en tanıdıklarından birkaçında karşımıza çıkar.

- **Birleştirme sıralaması** (*merge sort*): dizi ikiye bölünür, iki yarı ayrı ayrı sıralanır, sıralı yarılar $\Theta(n)$ sürede birleştirilir.
- **f5**: bölümün örnek fonksiyonlarından biridir; *f1(n)*'yi çağırır (doğrusal iş) ve kendini iki kez *n/2* ile çağırır.
- **Hızlı sıralama** (*quick sort*): pivot diziyi her seferinde iki eşit parçaya bölerse bağıntı aynıdır.

1.2.5.5 Daha fazla örnek

Tablo 1.2.17 teoremi sık karşılaşılan bağıntılara uygular. Her satırda önce $\log_b a$ hesaplanır, sonra *f(n)*'nin derecesi *c* ile karşılaştırılır. Logaritma değerleri Python ile hesaplanmıştır.

Bağıntı	a, b	$\log_b a$	c	Durum	Sonuç	Algoritma
$T(n/2) + 1$	1, 2	0	0	2	$\Theta(\log n)$	ikili arama
$2T(n/2) + 1$	2, 2	1	0	1	$\Theta(n)$	f4, ağaç gezme
$2T(n/2) + n$	2, 2	1	1	2	$\Theta(n \log n)$	f5, birleştirme sıralaması
$3T(n/2) + n$	3, 2	≈ 1,585	1	1	$\Theta(n^{1,585})$	Karatsuba çarpımı
$4T(n/2) + n$	4, 2	2	1	1	$\Theta(n^2)$	—
$7T(n/2) + n^2$	7, 2	≈ 2,807	2	1	$\Theta(n^{2,807})$	Strassen algoritması
$2T(n/2) + n^2$	2, 2	1	2	3	$\Theta(n^2)$	—
$T(n/2) + n$	1, 2	0	1	3	$\Theta(n)$	hızlı seçme (ortalama)

Tablo 1.2.17: Ana Teoremin örnek bağıntılara uygulanması.

Tablonun ilk satırı ikili aramadır: her adımda tek bir alt problem ($a = 1$) yarı boyutta çözülür ve sabit iş yapılır. $\log_2 1 = 0$ ve $c = 0$ olduğundan Durum 2 uygulanır: $\Theta(n^0 \cdot \log n) = \Theta(\log n)$. Sabit bir *f(n)*'nin Durum 2'ye girmesi şaşırtıcı görünebilir; ama ağaçta her düzeyde tek düğüm vardır ve her düzey aynı sabit işi yapar.

Karatsuba ve Strassen satırları, böl ve fethetin gücünü gösterir. İki *n* basamaklı sayıyı okulda öğretilen yöntemle çarpmak $\Theta(n^2)$ sürer. Sayıları ikiye bölüp dört yarı çarpım yapmak da bunu değiştirmez: $4T(n/2) + n = \Theta(n^2)$. Anatoli Karatsuba 1962'de dört çarpımdan birinin ötekilerden hesaplanabildiğini gösterdi. Üç çarpımla bağıntı $3T(n/2) + n$ olur ve sonuç $\Theta(n^{\log_2 3}) \approx \Theta(n^{1,585})$'e iner. Volker Strassen 1969'da aynı düşünceyi matris çarpımına uyguladı: sekiz yerine yedi alt çarpım. $8T(n/2) + n^2$ bağıntısı $\Theta(n^3)$ verir; $7T(n/2) + n^2$ ise $\Theta(n^{\log_2 7}) \approx \Theta(n^{2,807})$ verir. İki örnekte de tek bir alt çağrıyı azaltmak, üssü küçültmüştür. Durum 1'de sonucu yalnız *a* ve *b* belirlediği için *a*'daki her azalma doğrudan kazanca dönüşür.

Durum 3 satırlarında düzenlilik koşulu da denetlenmelidir. $2T(n/2) + n^2$ için $2 \cdot (n/2)^2 = n^2/2 \le \tfrac{1}{2} \cdot n^2$ olduğundan $k = \tfrac{1}{2}$ ile koşul sağlanır. $T(n/2) + n$ için $1 \cdot (n/2) \le \tfrac{1}{2} \cdot n$, yine $k = \tfrac{1}{2}$.

Örnek 1.2.6

Üç bağıntı çözülecek: (a) $T(n) = 3T(n/3) + n$, (b) $T(n) = 9T(n/3) + n$, (c) $T(n) = T(n/2) + n$.

1. **(a)** $a = 3$, $b = 3$, $\log_3 3 = 1$; $f(n) = n$, $c = 1$. Üsler eşittir: Durum 2, $T(n) = \Theta(n \log n)$. Ağaçta her düzey üç parçaya bölünür ama toplam iş yine n'dir; düzey sayısı $\log_3 n$'dir. $n = 729$ için doğrudan hesap $5103 = 729 \cdot 6 + 729$ verir.
2. **(b)** $a = 9$, $b = 3$, $\log_3 9 = 2$; $c = 1 < 2$. Durum 1, $T(n) = \Theta(n^2)$. Her düzeyin işi bir öncekinin üç katıdır ($r = 9/3 = 3$); yapraklar baskındır.
3. **(c)** $a = 1$, $b = 2$, $\log_2 1 = 0$; $c = 1 > 0$. Düzenlilik: $1 \cdot (n/2) \le \tfrac{1}{2} \cdot n$, $k = \tfrac{1}{2} < 1$. Durum 3, $T(n) = \Theta(n)$. Toplam $n + n/2 + n/4 + \ldots < 2n$'dir.

1.2.5.6 Teoremin uygulanamadığı bağıntılar

Ana Teorem güçlüdür, ama yalnız belirli bir biçimdeki bağıntılara uygulanır. Biçimin dışına çıkan bağıntılarda başka bir yöntem gerekir.

- **Çıkarmalı bağıntılar.** $T(n) = 2T(n - 1) + c$ (f3) ya da $T(n) = T(n - 1) + n$: alt problem n/b değil, $n - 1$'dir. Açma yöntemi kullanılır.
- **Polinom olmayan fark.** $T(n) = 2T(n/2) + n / \log n$: $n^{\log_2 2} = n$ ile $f(n) = n / \log n$ karşılaştırılır. $f(n)$, n'den küçüktür ama yalnız bir log çarpanı kadar; n^{ε} gibi bir polinom çarpanı kadar değil. Bu yüzden ne Durum 1'in ne de Durum 2'nin koşulu sağlanır. (Bağıntının çözümü $\Theta(n \log \log n)$'dir ve özyineleme ağacıyla bulunur.)
- **Eşit olmayan bölme.** $T(n) = T(n/3) + T(2n/3) + n$: alt problemler farklı boyuttadır. Özyineleme ağacında her düzeyin işi en çok n'dir ve en uzun dal $\log_{3/2} n$ derinliktedir; sonuç $\Theta(n \log n)$'dir. Bu tür bağıntıları genel olarak Akra–Bazzi yöntemi çözer.
- **Sabit olmayan a ya da $a < 1$.** $T(n) = 2^n\, T(n/2) + n$ ya da $T(n) = \tfrac{1}{2}\, T(n/2) + n$: teoremin $a \ge 1$ sabit varsayımı bozulur.

Temel biçimin sık kullanılan bir genişlemesi, Durum 2'yi logaritma çarpanlarına uzatır: $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ ve $k \ge 0$ ise $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$. Örneğin $T(n) = 2T(n/2) + n \log n$ için $k = 1$ ve sonuç $\Theta(n \log^2 n)$'dir. Bu genişleme, temel biçimin "uygulanamaz" dediği bazı bağıntıları da çözer.

Sık yapılan hata

Durum 3'te yalnız $c > \log_b a$ koşuluna bakıp düzenliliği unutmak. Polinomlarda koşul kendiliğinden sağlandığı için hata çoğu zaman sonucu değiştirmez; ama $f(n)$'nin salınım yaptığı örneklerde koşul bozulur. $T(n) = T(n/2) + n(2 - \cos n)$ bağıntısında $c = 1 > 0$'dır; ama $\cos n$'nin

1'e, *cos(n/2)*'nin −1'e yakın olduğu *n*'lerde *f(n/2)/f(n)* oranı 1,5'e yaklaşır ve hiçbir $k < 1$ koşulu sağlamaz. Durum 3 kullanılırken $a \cdot f(n/b) \le k \cdot f(n)$ eşitsizliği açıkça yazılmalıdır.

1.2.6 Uygulama: f4 ve f5

Bu kesim, iki özyinelemeli fonksiyonu üç yöntemle birden çözer: izleme, özyineleme ağacı ve Ana Teorem. f4, f3'ten yalnız özyinelemeli çağrıların argümanıyla (`n - 1` yerine `n / 2`) ayrılır; ama bu değişiklik süreyi üstelden doğrusala indirir. f5 ise f4'e her çağrıda doğrusal bir iş ekler ve birleştirme sıralamasının bağıntısını verir.

1.2.6.1 f4: iki çağrı, n/2

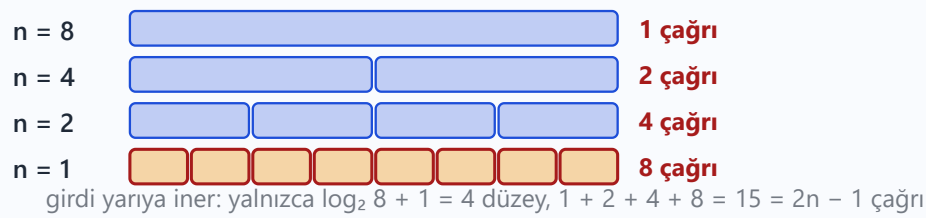
```
public int f4(int n) {
  if (n <= 1) {
    return 1;
  }
  return f4(n / 2) + f4(n / 2);
}
```

- **Taban durum:** $n \le 1$ iken 1 döner; $O(1)$.
- **Özyinelemeli durum:** $n > 1$ iken `f4(n / 2)` iki kez çağrılır ve sonuçlar toplanır. Toplama sabit bir iştir.

$$T(n) = 2T(n/2) + O(1), \quad T(1) = O(1)$$

Java'da `n / 2` tam sayı bölmesidir; kesirli kısım atılır. Analizde n'nin ikinin kuvveti olduğu varsayılır, böylece her bölme tam çıkar. Bu varsayım büyüme sınıfını değiştirmez.

f3 ile tek fark, girdinin $n - 1$ yerine $n/2$ olmasıdır. Çağrı sayısı yine her düzeyde ikiye katlanır; ama girdi her düzeyde yarıya indiği için ağaç yalnız $\log_2 n$ düzey derinliğindedir. $n = 8$ için düzeylerde 1, 2, 4, 8 çağrı vardır ve girdiler 8, 4, 2, 1'dir. f3'te 8'den başlayan bir ağaç 8 düzey inerdi; burada 4 düzeyde biter.



Şekil 1.2.18: `f4(8)`'in düzeyleri. Girdi her düzeyde yarıya iner (8, 4, 2, 1), çağrı sayısı iki katına çıkar (1, 2, 4,

8). Yalnız $log_2\ 8 + 1 = 4$ düzey vardır; toplam $1 + 2 + 4 + 8 = 15 = 2n - 1$ çağrı yapılır.

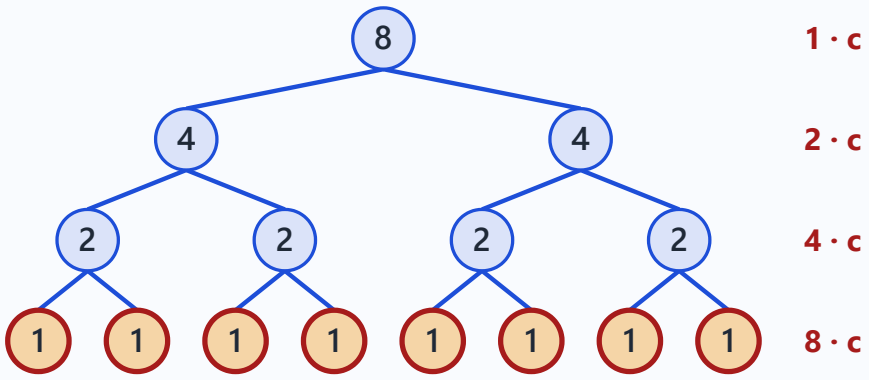
İki çağrı her zaman üstel süre demek değildir. Belirleyici olan, girdinin ne kadar küçüldüğüdür. Girdi bir azalırsa ağaç n derinliğe iner ve 2^n yaprağa ulaşır. Girdi yarıya inerse ağaç $log_2\ n$ derinlikte kalır ve $2^{log_2\ n} = n$ yaprağa ulaşır.

1.2.6.2 f4'ün özyineleme ağacı

Özyineleme ağacı yönteminde her çağrı bir düğüm olarak çizilir ve yanına yaptığı iş yazılır. Sonra her düzeyin işi toplanır, en sonda da düzey toplamları toplanır. `f4(8)` için:

1. **Kök:** `f4(8)` yalnız sabit iş (c) yapar ve iki çağrı başlatır. Düzey 0'ın işi c'dir.
2. **Düzey 1:** iki `f4(4)` çağrısı; toplam $2c$.
3. **Düzey 2:** dört `f4(2)` çağrısı; toplam $4c$. Düzey işi her adımda iki katına çıkar.
4. **Yapraklar:** sekiz `f4(1)` ; toplam $8c = n \cdot c$.

Toplam $c(1 + 2 + 4 + 8) = 15c = (2n - 1)c$'dir; yani $\Theta(n)$. Düzey toplamları geometrik olarak artar ve toplamı son terim, yaprakların işi belirler. Java ile sayıldığında `f4(16)` 31, `f4(32)` 63 çağrı yapar; ikisi de $2n - 1$'dir. Fonksiyonun dönüş değeri de yaprak sayısına eşittir: `f4(n) = n`.



Şekil 1.2.19: `f4(8)` 'in özyineleme ağacı. Düğümlerdeki sayı girdidir. Sağdaki sütun düzey toplamlarını verir: $c, 2c, 4c, 8c$. Turuncu çizilen sekiz yaprak, toplam işin yarısından fazlasını yapar.

n ikinin kuvveti değilse tam sayı bölmesi devreye girer. `f4(10)` önce iki `f4(5)` 'i, her `f4(5)` iki `f4(2)` 'yi, her `f4(2)` de iki `f4(1)` 'i çağırır. Çağrı sayısı 15, dönüş değeri 8'dir. Sonuç, n'den küçük en büyük ikinin kuvvetine (8) göre çalışmış gibidir; büyüme sınıfı yine doğrusaldır.

1.2.6.3 f4'ü Ana Teorem ile çözmek

Örnek 1.2.7

1. **Eşleştirme.** $T(n) = 2T(n/2) + O(1)$ bağıntısı genel biçimle karşılaştırılır: $a = 2$, $b = 2$, $f(n) = O(1)$.
2. **Yaprak işi.** $\log_b a = \log_2 2 = 1$; yaprak işi $n^1 = n$.
3. **Karşılaştırma.** $f(n) = O(1) = O(n^0)$, yani $c = 0$. $0 < 1$: Durum 1.
4. **Sonuç.** $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Ana Teorem, özyineleme ağacının verdiği sonucu toplamı hesaplamadan verir. Ağaçta $2n - 1$ düğüm sayılmıştı; teorem yalnız "yapraklar baskın" der ve yaprak sayısı n olduğu için $\Theta(n)$ sonucuna varır.

f4'ün belleği ise farklı bir sorudur. Çağrılar sırayla yürüdüğü için yığında aynı anda yalnız bir dal bulunur: `f4(n)`, `f4(n/2)`, ..., `f4(1)`. Bu dalın uzunluğu $\log_2 n + 1$'dir; `f4(16)` çalışırken yığında en çok 5 çerçeve bulunur. Zaman $\Theta(n)$, ek bellek $\Theta(\log n)$'dir.

Sık yapılan hata

f4 için "girdi her adımda yarıya iniyor, öyleyse $O(\log n)$" demek. Bu akıl yürütme, tek çağrı yapan ikili arama için doğrudur. f4 iki çağrı yapar ve iki dalın ikisi de sonuna kadar iner. $\log n$ yalnız yığın derinliğidir, yani belleği verir; zaman $\Theta(n)$'dir.

1.2.6.4 f5: doğrusal iş ve iki çağrı

```
public int f5(int n) {
  if (n <= 1) {
    return 1;
  }
  return f1(n) + f5(n / 2) + f5(n / 2);
}
```

- **Taban durum:** $n \le 1$ iken 1 döner; $O(1)$.
- **Özyinelemeli durum:** $n > 1$ iken önce `f1(n)` çağrılır. f1, n tur dönen bir döngüdür ve $\Theta(n)$ iş yapar. Ardından `f5(n / 2)` iki kez çağrılır.

$$T(n) = 2T(n/2) + f1(n) = 2T(n/2) + \Theta(n)$$

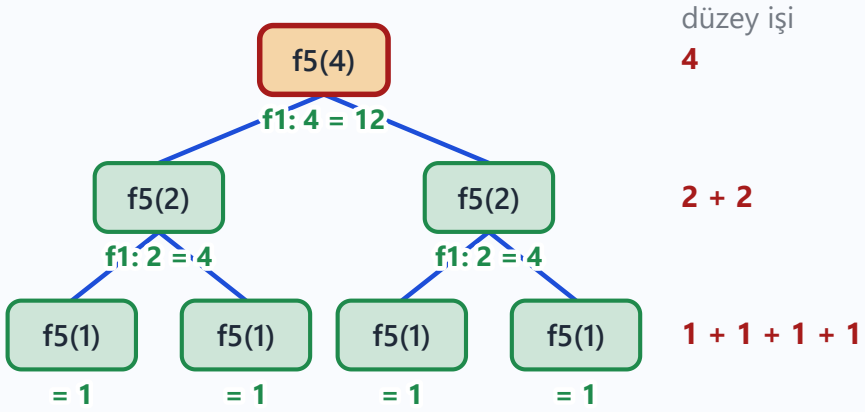
f4'ten tek farkı her çağrıda yapılan doğrusal iştir. Bağıntı birleştirme sıralamasının bağıntısıyla aynıdır. Burada sık yapılan bir hata, iki parçayı ayrı ayrı düşünüp toplamaktır: "f1 doğrusal, özyineleme logaritmik, toplam *O(n + log n) = O(n)*." Bu yanlıştır. f1 yalnız kökte değil, ağacın her düğümünde çağrılır; her düğüm kendi girdisi kadar iş yapar. Bu işlerin toplamı bulunmalıdır.

1.2.6.5 Kod izleme: f5(4)

Adım	Olay	Değer
1	`f5(4)` çağrılır; 4 > 1	—
2	`f1(4)` : 4 turluk döngü, 4 birim iş	4
3	soldaki `f5(2)` çağrılır; `f1(2)` : 2 birim iş	2
4	iki `f5(1)` : taban durum	1, 1
5	`f5(2) = f1(2) + f5(1) + f5(1) = 2 + 1 + 1`	4
6	sağdaki `f5(2)` : aynı iş baştan yapılır	4
7	`f5(4) = f1(4) + f5(2) + f5(2) = 4 + 4 + 4`	12

Tablo 1.2.18: `f5(4)` izlemesi (Java ile çalıştırılan bir kopyanın çıktısından).

Sonuç 12'dir. Bu sayı işin düzeylere nasıl dağıldığını da gösterir. Kökte `f1(4)` 4 birim, birinci düzeyde iki `f1(2)` toplam 4 birim iş yapar. Dört yaprağın her biri 1 döndürür; toplam 4. Her düzeyin işi $n = 4$'tür. $\log_2 4 = 2$ iç düzey ve bir yaprak düzeyi vardır: $2 \cdot 4 + 4 = 12 = n \log_2 n + n$.



Şekil 1.2.20: `f5(4)` 'ün tamamlanmış ağacı. Her düğümün altında `f1` 'in yaptığı iş ve düğümün döndürdüğü toplam yazılıdır: kökte `f1` 4 iş yapar ve `f5(4)` 12 döndürür; `f5(2)` 'lerde `f1` 2 iş yapar ve her biri 4 döndürür. Sağdaki sütun düzey işlerini verir: 4, 2 + 2, 1 + 1 + 1 + 1.

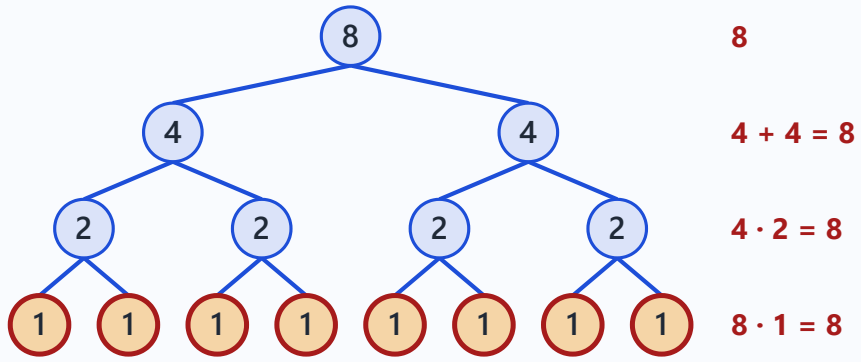
1.2.6.6 f5'in özyineleme ağacı ve Ana Teorem

`f5(8)` 'in ağacında örüntü daha açık görülür (Şekil 1.2.21).

1. **Kök:** `f5(8)` içinde `f1(8)` 8 adım yapar.

2. **Düzey 1:** iki `f5(4)`, her biri `f1(4)` ile 4 adım; toplam 8.
3. **Düzey 2:** dört `f5(2)`, her biri 2 adım; toplam yine 8.
4. **Yapraklar:** sekiz `f5(1)`, her biri 1; toplam 8.

Toplam $3 \cdot 8 + 8 = 32 = n \log_2 n + n$'dir ve Java'da `f5(8)` gerçekten 32 döndürür. Her düzey n iş yapar ve $\log_2 n$ iç düzey vardır; sonuç $\Theta(n \log n)$'dir. Daha büyük girdilerde de formül tutar: `f5(16) = 80 = 16 · 4 + 16`, `f5(32) = 192 = 32 · 5 + 32`.



Şekil 1.2.21: `f5(8)` 'in özyineleme ağacı. f4'ün ağacıyla aynı biçimdedir, ama her düğüm kendi girdisi kadar iş yapar. Düzey toplamları eşittir: $8 = 4 + 4 = 4 \cdot 2 = 8 \cdot 1$.

Ana Teorem ile çözüm, f4'tekiyle aynı adımları izler; yalnız $f(n)$ değişmiştir:

1. $T(n) = 2T(n/2) + \Theta(n)$: $a = 2$, $b = 2$, $f(n) = \Theta(n)$.
2. $\log_b a = 1$, $c = 1$. Üsler eşit: Durum 2.
3. $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

	f4	f5
Çağrılar	2 × n/2	2 × n/2
Ek iş f(n)	O(1)	Θ(n)
c ile $\log_b a$	0 < 1	1 = 1
Durum	1	2
Düzey toplamları	c, 2c, 4c, … (artar)	n, n, n, … (eşit)
Sonuç	Θ(n)	Θ(n log n)

Tablo 1.2.19: f4 ile f5'in karşılaştırması.

Tablo, f3–f4–f5 üçlüsünün öğrettiğini özetler. Üç fonksiyon da kendini iki kez çağırır. f3 ile f4 arasındaki fark küçülme biçimidir ($n - 1$ ile $n/2$) ve sonucu $\Theta(2^n)$'den $\Theta(n)$'ye indirir. f4 ile f5 arasındaki fark ek iştir (sabit ile doğrusal) ve sonucu $\Theta(n)$'den $\Theta(n \log n)$'ye çıkarır.

1.2.7 Karşılaştırma ve amortize analiz

1.2.7.1 f1–f6: ölçülen adım sayıları

Altı fonksiyonun analizle bulunan sınıfları, fonksiyonlar çalıştırılarak denetlenebilir. Tablo 1.2.20 her fonksiyonun *n = 16* ve *n = 32* için ölçülen adım sayısını verir. f1, f2, f5 ve f6 için dönüş değeri, f3 ve f4 için çağrı sayısı kullanılmıştır. Değerler Java ile ölçülmüştür.

Fonksiyon	Bağıntı / toplam	n = 16	n = 32	Oran	Sınıf
f1 (dönüş)	$\Sigma\ 1 = n$	16	32	× 2	$\Theta(n)$
f2 (dönüş)	$\Sigma\ i^2$	1.240	10.416	× 8,4	$\Theta(n^3)$
f3 (çağrı)	$2T(n-1) + c$	65.535	4.294.967.295	× 65.537	$\Theta(2^n)$
f4 (çağrı)	$2T(n/2) + c$	31	63	× 2	$\Theta(n)$
f5 (dönüş)	$2T(n/2) + n$	80	192	× 2,4	$\Theta(n \log n)$
f6 (dönüş)	$i \leftarrow 2^i$	4	5	+ 1	$\Theta(\log^* n)$

Tablo 1.2.20: f1–f6'nın ölçülen adım sayıları. Oran sütunu, *n* iki katına çıkınca adım sayısının nasıl değiştiğini verir.

Oran sütunu, büyüme sınıfını ele veren pratik bir testtir. *n* iki katına çıkınca doğrusal fonksiyonlar (f1, f4) iki katına çıkar. Log-doğrusal f5, 2'den biraz fazla, 2,4 katına çıkar; fazlalık *log n* çarpanının 4'ten 5'e artmasından gelir. Kübik f2 yaklaşık 8 katına çıkar. Üstel f3'ün oranı ise $2^{16} + 1 = 65.537$'dir; girdi iki katına çıkınca adım sayısı yaklaşık karesine çıkmıştır. f6 yalnız bir adım artar.

Bu test, analizi doğrulamanın deneysel bir yoludur ve kodun karmaşıklığı bilinmediğinde de kullanılır. Bir program farklı *n*'lerle çalıştırılır, süre ya da adım sayısı ölçülür ve ardışık ölçümlerin oranına bakılır. Sedgewick ve Wayne bu yönteme “ikiye katlama deneyi” der. Sunumdaki etkileşimli adım sayacı aynı deneyi 1 ile 40 arasındaki her *n* için yapar ve sonuçları logaritmik ölçekli çubuklarla gösterir. Orada f5 için ayrıca çağrı sayısı da sayılır, bu yüzden değerler tablodakinden biraz büyüktür; oranlar ise aynı sınıfı gösterir.

1.2.7.2 Amortize analiz

Bazı veri yapılarında işlemlerin çoğu ucuz, arada bir yapılan bazıları ise pahalıdır. Böyle bir yapıda tek bir işlemin en kötü durumuna bakmak, toplam maliyeti olduğundan çok gösterir. Amortize analiz, işlem dizisinin bütününe bakar.

> **Tanım 1.2.7**
>
> **Amortize analiz** (*amortized analysis*), bir işlem dizisinin en kötü durumdaki toplam maliyetini işlem sayısına bölerek işlem başına maliyeti bulur. Bulunan değere **amortize maliyet** denir.

Klasik örnek, Java'daki `ArrayList` gibi **dinamik dizidir** (*dynamic array*). Dizi dolduğunda kapasitesi iki katına çıkarılır: iki kat büyüklükte yeni bir dizi ayrılır ve bütün öğeler oraya kopyalanır. Bu ekleme pahalıdır. Yer olduğu sürece yapılan eklemeler ise tek bir yazmadır ve 1 birim tutar. Aşağıdaki kod bu mantığı yazar; maliyet, yazma sayısı ile kopyalanan öğe sayısının toplamıdır.

```
int[] veri = new int[1];
int boyut = 0;
long maliyet = 0;

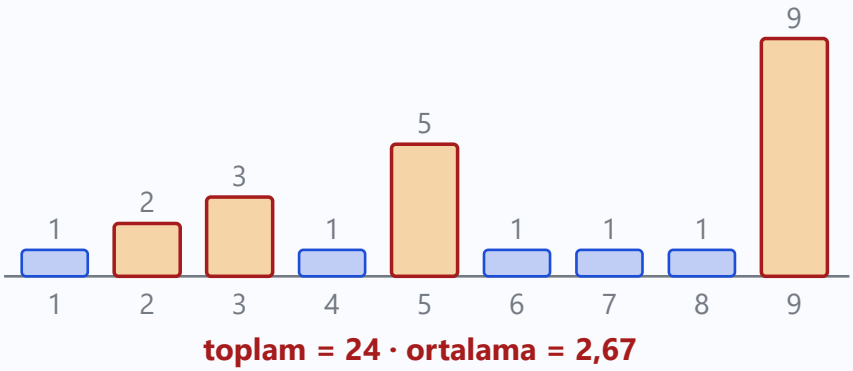
void ekle(int x) {
  if (boyut == veri.length) {              // dizi dolu
    int[] yeni = new int[2 * veri.length]; // kapasiteyi iki katına çıkar
    for (int i = 0; i < boyut; i++) {
      yeni[i] = veri[i];                   // kopya: 1 birim
      maliyet++;
    }
    veri = yeni;
  }
  veri[boyut++] = x;                       // yazma: 1 birim
  maliyet++;
}
```

Örnek 1.2.8

Kapasitesi 1 olan boş bir dinamik diziye 9 öğe ekleniyor. Her eklemenin maliyeti (1 yazma + kopya sayısı):

1. 1. ekleme: yer var, 1 yazma. Maliyet 1.
2. 2. ekleme: dizi dolu; kapasite 1 → 2, 1 kopya + 1 yazma. Maliyet 2.
3. 3. ekleme: dolu; kapasite 2 → 4, 2 kopya + 1 yazma. Maliyet 3.
4. 4. ekleme: yer var. Maliyet 1.
5. 5. ekleme: dolu; kapasite 4 → 8, 4 kopya + 1 yazma. Maliyet 5.
6. 6., 7. ve 8. ekleme: yer var. Her biri 1.
7. 9. ekleme: dolu; kapasite 8 → 16, 8 kopya + 1 yazma. Maliyet 9.

Toplam $1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 = 24$ birimdir; ekleme başına ortalama $24/9 \approx 2{,}67$. Tek bir eklemenin en kötü maliyeti 9'dur, ama dizinin tamamı için $24 < 3 \cdot 9 = 27$ birim harcanmıştır.



Şekil 1.2.22: Kapasitesi 1'den başlayıp iki katına büyüyen dinamik diziye 9 ekleme. Turuncu çubuklar kapasite artırma (kopyalama) içeren eklemelerdir: 2., 3., 5. ve 9. eklemeler. Toplam 24, ortalama 2,67.

Teorem 1.2.2 (İki katına büyütmenin amortize maliyeti)

Kapasitesi 1'den başlayıp dolunca iki katına çıkarılan bir dinamik diziye yapılan n eklemenin toplam maliyeti $3n$'den azdır. Ekleme başına amortize maliyet $\Theta(1)$'dir.

İspat. Kapasite artırmaları dizi 1, 2, 4, …, 2^k öğeyle dolduğunda yapılır; burada 2^k, n'den küçük en büyük ikinin kuvvetidir. Toplam kopya sayısı geometrik bir seridir: $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1 < 2n$. Her ekleme ayrıca bir yazma yapar; bu da n birimdir. Toplam $< 2n + n = 3n$'dir. n'ye bölününce ekleme başına maliyet 3'ten küçük çıkar, yani sabittir. □

Teorem, kopyalamanın pahalı olduğu anların seyrek olmasına dayanır. Bir kapasite artırması m öğeyi kopyalar, ama ondan sonra m ekleme daha ucuza yapılır. Pahalı işlemin maliyeti, onu izleyen ucuz işlemlere "yayılır". Python ile yapılan bir sayımda 1000 eklemenin toplam maliyeti 2023 çıkar; bu, $3n = 3000$'in altındadır.

İki katına büyütmenin neden önemli olduğu, sabit miktarla büyütmeyle karşılaştırınca görülür. Dizi dolduğunda kapasite yalnız 1 artırılsaydı her ekleme bütün diziyi kopyalardı. n eklemenin toplam maliyeti $1 + 2 + \ldots + n = n(n + 1)/2$, yani $\Theta(n^2)$ olurdu; $n = 1000$ için 500.500 birim. Kapasite sabit bir k kadar artırılsa da sonuç değişmez, yalnız sabit çarpan küçülür.

Amortize analiz, ortalama durum analizinden farklıdır. Ortalama durum, girdilerin bir olasılık dağılımına dayanır ve "beklenen" süreyi verir. Amortize analizde olasılık yoktur: her işlem dizisi için, en kötü dizide bile geçerli bir garanti verilir. Dinamik dizide tek bir ekleme $\Theta(n)$ sürebilir; ama hangi sırayla ekleme yapılırsa yapılsın, n eklemenin toplamı $3n$'yi aşmaz. Amortize maliyeti bulmanın üç standart yöntemi vardır: toplama yöntemi (burada kullanılan), hesap (*accounting*) yöntemi ve potansiyel yöntemi.

Sık yapılan hata

Amortize $\Theta(1)$'i "her ekleme sabit sürer" diye okumak. Amortize sınır, tek bir işlemin süresi hakkında bir şey söylemez; 9. ekleme 9 birim sürmüştür. Her işlemin kısa sürmesi gereken gerçek zamanlı sistemlerde amortize sınır yetmez, en kötü durum sınırı gerekir.

1.2.8 Sık yapılan hatalar

Bölüm boyunca kutularda verilen hataların en önemlileri bir arada şöyledir:

- **Büyük O'yu en kötü durumla eş tutmak.** O bir üst sınırdır, en kötü durum ise bir girdi türüdür. "En iyi durum $O(1)$'dir" geçerli bir cümledir.
- **Sabitleri ve küçük terimleri yazmak.** $O(3n + 3)$ ya da $O(n^2/3 + n)$ yazılmaz; $O(n)$ ve $O(n^2)$ yeterlidir.
- **"İç içe döngü = n^2" kuralı.** Sonucu iç döngünün sınırı belirler. f2'de sınır i^2 olduğu için $\Theta(n^3)$ çıkar; `i *= 2` ile ilerleyen bir döngü ise $\log n$ tur döner.
- **"İki çağrı = üstel" kuralı.** f3 $(n - 1)$ $\Theta(2^n)$, f4 $(n/2)$ $\Theta(n)$'dir. Küçülme biçimi sonucu belirler.
- **Ana Teoremi her bağıntıya uygulamak.** $T(n - 1)$ içeren, eşit bölmeyen ya da $f(n) = n / \log n$ gibi polinomdan farklı bir fark taşıyan bağıntılarda teorem geçerli değildir.
- **Durum 3'te düzenliliği unutmak.** $c > \log_b a$ yetmez; $a \cdot f(n/b) \le k \cdot f(n)$, $k < 1$ de sağlanmalıdır.
- **Özyinelemenin belleğini çağrı sayısıyla karıştırmak.** Bellek, aynı anda açık olan çerçeve sayısıyla, yani derinlikle belirlenir: f3'te $\Theta(n)$, f4'te $\Theta(\log n)$.
- **Kaydırmayı çarpmayla karıştırmak.** `i = 1 << i`, 2^i'dir; `2 * i` değildir. f6'nın sıkı sınırı $\Theta(\log^* n)$'dir.

1.2.9 Özet

- **Zaman karmaşıklığı** $T(n)$, temel işlem sayısının girdi boyutu n ile nasıl büyüdüğünü verir. **Alan karmaşıklığı** $S(n)$, girdiye ek olarak kullanılan belleği verir; özyinelemede çağrı yığınının derinliği de bu belleğe girer.
- Süre ölçümü makineye bağlıdır ve büyük girdiler için tahmin vermez. **RAM modelinde** her temel işlem bir birim sayılır; doğrusal arama en kötü durumda $3n + 3$ adım sürer ve $\Theta(n)$'dir.
- **En iyi**, **en kötü** ve **ortalama durum**, aynı boyuttaki farklı girdilerle ilgilidir. Doğrusal aramada sırasıyla 1, n ve $(n + 1)/2$ karşılaştırma yapılır.
- $f = O(g)$ üst sınır, $f = \Omega(g)$ alt sınır, $f = \Theta(g)$ sıkı sınırdır. Sınırlar c ve n_0 sabitleri bulunarak kanıtlanır; $3n^2 + 5n + 2$ için $c_1 = 3$, $c_2 = 4$, $n_0 = 6$. Durum (hangi girdi) ile sınır (O, Ω, Θ) ayrı kavramlardır.

- Hesap kuralları: sabit çarpanlar ve düşük dereceli terimler atılır; ardışık kodda büyük olan kalır, iç içe kodda tur sayıları çarpılır ya da toplanır; logaritmanın tabanı önemsiz, üstel ifadenin tabanı önemlidir. Büyüme sırası: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$.
- **Döngü analizinde** her dış tur için iç döngünün tur sayısı yazılır ve toplanır. f1: $\Sigma\ 1 = n$, $\Theta(n)$. f2: $\Sigma\ i^2 = (n-1)n(2n-1)/6$, $\Theta(n^3)$. Çarpımsal döngü $\Theta(\log n)$; içinde `j < i` olan çarpımsal döngü geometrik seri nedeniyle $\Theta(n)$'dir.
- f6'da `i = 1 << i`, yani $i \leftarrow 2^i$'dir; tur sayısı $\log^* n + 1$'dir. $O(\log n)$ doğru ama sıkı değildir; sıkı sınır $\Theta(\log^* n)$'dir. Java'da $n > 65.536$ için kaydırma taşar ve döngü bitmez.
- **Yineleme bağıntısı**, çağrı sayısı, küçülme biçimi ve ek işten kurulur. f3: $T(n) = 2T(n-1) + c$; açma yöntemiyle $(2^n - 1)c = \Theta(2^n)$. Belleği ise yalnız $\Theta(n)$'dir.
- **Ana Teorem**, $T(n) = aT(n/b) + f(n)$ bağıntısında $f(n)$'yi yaprak işi $n^{\log_b a}$ ile karşılaştırır: yapraklar baskın (Durum 1), düzeyler dengeli (Durum 2, ek $\log n$ çarpanı), kök baskın (Durum 3, düzenlilik koşuluyla). Üç durum, oranı a/b^c olan geometrik serinin artan, sabit ya da azalan olmasına karşılık gelir.
- Teorem çıkarmalı, eşit bölmeyen, $f(n)$ farkı polinom olmayan ya da a'sı sabit olmayan bağıntılara uygulanmaz.
- f4: $2T(n/2) + 1$, Durum 1, $\Theta(n)$; $2n - 1$ çağrı, $\Theta(\log n)$ bellek. f5: $2T(n/2) + n$, Durum 2, $\Theta(n \log n)$; $f5(n) = n \log_2 n + n$.
- **Amortize analiz**, bir işlem dizisinin toplam maliyetini işlem sayısına böler. İki katına büyüyen dinamik dizide n eklemenin toplamı $3n$'den azdır; ekleme amortize $\Theta(1)$'dir. Ortalama durumdan farklı olarak olasılık kullanmaz.

Fonksiyon	Yapı	Analiz	Zaman	Ek bellek
f1	tek döngü	$\Sigma\ 1 = n$	$\Theta(n)$	$\Theta(1)$
f2	iç döngü `j < i * i`	$\Sigma\ i^2$	$\Theta(n^3)$	$\Theta(1)$
f3	iki çağrı, n − 1	$2T(n-1) + c$, açma	$\Theta(2^n)$	$\Theta(n)$
f4	iki çağrı, n/2	$2T(n/2) + 1$, Durum 1	$\Theta(n)$	$\Theta(\log n)$
f5	f1(n) + iki çağrı, n/2	$2T(n/2) + n$, Durum 2	$\Theta(n \log n)$	$\Theta(\log n)$
f6	`i = 1 << i`	$i \leftarrow 2^i$	$\Theta(\log^* n)$	$\Theta(1)$

Tablo 1.2.21: Altı örnek fonksiyonun özeti.

1.2.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity* · karmaşıklık sınıfı *complexity class* · büyüme hızı *growth rate* · RAM modeli *random-access machine* · en iyi durum *best case* · en kötü durum *worst case* · ortalama durum *average case* · asimptotik gösterim *asymptotic notation*

büyük O gösterimi *Big O notation* · büyük Omega *Big Omega (Ω)* · büyük Teta *Big Theta (Θ)* · üst sınır *upper bound* · alt sınır *lower bound* · sıkı sınır *tight bound* · sabit zaman *constant time* · logaritmik *logarithmic* · doğrusal *linear* · log-doğrusal *linearithmic* · karesel *quadratic* · kübik *cubic* · üstel *exponential* · faktöriyel *factorial* · polinom zaman *polynomial time* · yinelenmiş logaritma *iterated logarithm* · özyineleme *recursion* · taban durum *base case* · çağrı yığını *call stack* · yineleme bağıntısı *recurrence relation* · özyineleme ağacı *recursion tree* · böl ve fethet *divide and conquer* · ana teorem *master theorem* · düzenlilik koşulu *regularity condition* · amortize analiz *amortized analysis* · dinamik dizi *dynamic array*

1.2.11 Kendini Dene

1. $5n^3 + 2n + 8 = O(n^3)$ sınırı için $c = 6$ seçiliyor. $5n^3 + 2n + 8 \le 6n^3$ eşitsizliğinin her $n \ge n_0$ için sağlandığı en küçük n_0 kaçtır?

   Cevap

   $5n^3 + 2n + 8 \le 6n^3 \Leftrightarrow n^3 \ge 2n + 8$. $n = 2$'de $8 < 12$, eşitsizlik bozulur; $n = 3$'te $27 \ge 14$, sağlanır. n artınca n^3, $2n + 8$'den hızlı büyüdüğü için eşitsizlik bundan sonra hep sağlanır: $n_0 = 3$. Denetim: $n = 3$'te $149 \le 162$.

2. `for (i = 0; i < n; i++) for (j = i; j < n; j++) x++;` parçası $n = 6$ için kaç kez `x++` yapar? Büyüme sınıfı nedir?

   Cevap

   i. turda iç döngü $n - i$ kez döner: $6 + 5 + 4 + 3 + 2 + 1 = 21$. Genel olarak $n(n + 1)/2$, yani $\Theta(n^2)$. İç döngünün dış sayaca bağlı olması toplamı yarıya indirir, ama sınıfı değiştirmez.

3. `for (int i = 1; i * i <= n; i++) x++;` döngüsü $n = 64$ için kaç tur döner? Büyüme sınıfı nedir?

   Cevap

   Döngü $i^2 \le n$, yani $i \le \sqrt{n}$ iken sürer. $n = 64$ için $i = 1, \ldots, 8$: 8 tur. Sınıf $\Theta(\sqrt{n})$'dir; deneme bölmesiyle asallık testi bu kalıbı kullanır.

4. $2^{n+1} = O(2^n)$ ve $2^{2n} = O(2^n)$ ifadelerinden hangisi doğrudur?

Cevap

Birincisi doğrudur: $2^{n+1} = 2 \cdot 2^n$; $c = 2$, $n_0 = 1$ yeter. İkincisi yanlıştır: $2^{2n} = 4^n$ ve $4^n / 2^n = 2^n$ oranı sonsuza gider; hiçbir sabit c bu farkı kapatamaz. Üsse eklenen sabit yalnız sabit bir çarpan getirir; üssü çarpan sabit ise tabanı değiştirir.

5. $T(n) = 2T(n/4) + \sqrt{n}$ bağıntısının Ana Teoreme göre çözümü nedir?

Cevap

$a = 2$, $b = 4$, $\log_4 2 = 1/2$. $f(n) = \sqrt{n} = n^{1/2}$, $c = 1/2$. Üsler eşit: Durum 2, $T(n) = \Theta(\sqrt{n} \log n)$. Denetim: $T(1) = 1$ ile $T(256) = 80 = 16 \cdot 4 + 16$.

6. $T(n) = 5T(n/2) + n^2$ bağıntısının çözümü nedir? Düzey toplamları artar mı, azalır mı?

Cevap

$\log_2 5 \approx 2{,}32 > c = 2$: Durum 1, $T(n) = \Theta(n^{\log_2 5}) \approx \Theta(n^{2{,}32})$. Oran $r = 5/2^2 = 1{,}25 > 1$; düzey toplamları her düzeyde 1,25 katına çıkar, yapraklar baskındır.

7. $T(n) = T(n-1) + 1$, $T(1) = 1$ bağıntısının açma yöntemiyle bulunan kapalı biçimi nedir? Ana Teorem neden uygulanamaz?

Cevap

$T(n) = T(n-1) + 1 = T(n-2) + 2 = \ldots = T(n-k) + k$. $k = n - 1$'de $T(n) = T(1) + n - 1 = n$: $\Theta(n)$. Alt problem n/b değil $n - 1$ olduğu için bağıntı Ana Teoremin biçimine uymaz. Tek çağrılı bir özyinelemenin girdisi her adımda bir azalırsa süre, her çağrının işinin n katıdır.

8. `f4(10)` kaç çağrı yapar ve ne döndürür? (Java'da `n / 2` tam sayı bölmesidir.)

Cevap

`f4(10)` iki `f4(5)`'i, her `f4(5)` iki `f4(2)`'yi, her `f4(2)` iki `f4(1)`'i çağırır. Çağrılar $1 + 2 + 4 + 8 = 15$'tir. Sekiz yaprak 1 döndürür; sonuç 8'dir. Ağaç, girdisi 8 olan ağaçla aynı biçimdedir.

9. Kapasitesi 1 olan ve dolunca iki katına büyüyen boş bir dinamik diziye 17 öğe ekleniyor. Toplam

kaç kopya yapılır, toplam maliyet nedir, son kapasite kaçtır?

> **Cevap**
>
> Kapasite artırmaları 2., 3., 5., 9. ve 17. eklemelerde yapılır; kopyalar *1 + 2 + 4 + 8 + 16 = 31*'dir. 17 yazma ile toplam 48 birimdir; *3 · 17 = 51*'in altında kalır. Son kapasite 32'dir.

10. `for (i = 0; i < n; i++) for (j = 1; j < n; j *= 2) x++;` parçasının *n = 10* için `x++` sayısı ve büyüme sınıfı nedir?

> **Cevap**
>
> İç döngü dış sayaçtan bağımsızdır; `j` = 1, 2, 4, 8 değerlerinde 4 tur döner. Dış döngü 10 tur döner; toplam *10 · 4 = 40*. Genel olarak $n \cdot \lceil \log_2 n \rceil$, yani $\Theta(n \log n)$.

1.2.12 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 3 ve 4: asimptotik gösterimin biçimsel tanımlarını, yerine koyma ve özyineleme ağacı yöntemlerini ve Ana Teoremin kanıtını verir; Akra–Bazzi yöntemine de değinir.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.4: çalışma süresini deneylerle ölçmeyi, ikiye katlama deneyini ve matematiksel modelleri Java örnekleriyle birleştirir.
- J. Kleinberg ve É. Tardos, *Algorithm Design* (2005), Bölüm 2 ve 5: verimlilik kavramını ve polinom zaman ölçütünü tartışır; böl ve fethet bağıntılarını özyineleme ağacıyla açık bir dille çözer.
- D. E. Knuth, “Big Omicron and Big Omega and Big Theta” (1976): O, Ω ve Θ gösterimlerinin tarihini ve bugünkü tanımlarının neden seçildiğini anlatan kısa bir makaledir.

1.2.13 Kaynaklar

1. J. L. Bentley, D. Haken, J. B. Saxe, “A General Method for Solving Divide-and-Conquer Recurrences”, *SIGACT News* 12(3), 1980.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
3. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005.
4. D. E. Knuth, “Big Omicron and Big Omega and Big Theta”, *SIGACT News* 8(2), 1976.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM I · BÖLÜM 2

Sıralama Algoritmaları

Sıralama, bilgisayar biliminin en eski ve en çok çalışılmış problemlerinden biridir. Problem basittir: bir dizinin öğeleri küçükten büyüğe dizilecektir. Bu basit problemin çözümleri ise algoritma tasarımının temel fikirlerini neredeyse eksiksiz gösterir. Kaba kuvvet, artımlı yaklaşım, böl ve fethet, rastgele seçim, bir veri yapısından yararlanma ve karşılaştırma yapmadan sayma bu fikirlerden bazılarıdır. Not, bir sıralama algoritmasını değerlendirmek için kullanılan ölçütlerle başlar. Ardından kabarcık, eklemeli, seçimli ve Shell sıralamasını; birleştirme sıralamasını, hızlı sıralamayı ve heap sort'u anlatır. Karşılaştırmalı sıralamanın alt sınırı ve bu sınırı aşan saymalı, basamak ve kova sıralaması bunları izler. Son kesimler Timsort'u ve algoritmaların ölçümlerle karşılaştırılmasını ele alır.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum02/src`

ÖĞRENME ÇIKTILARI

Kavramlar
Bir sıralama algoritmasını zaman, ek bellek, kararlılık, yerinde çalışma, uyarlanabilirlik ve karşılaştırmalı olup olmama ölçütleriyle değerlendirmek.

Basit algoritmalar
Kabarcık, eklemeli, seçimli ve Shell sıralamasını bir dizi üzerinde adım adım izlemek, Java ile yazmak ve karşılaştırma ile taşıma sayılarını hesaplamak.

Böl ve fethet, heap
Birleştirme sıralamasının, hızlı sıralamanın ve heap sort'un çalışmasını açıklamak; $O(n \log n)$ sürelerini ve en kötü durum davranışlarını gerekçelendirmek.

Doğrusal zaman ve seçim
Karşılaştırma alt sınırını kanıtlamak; saymalı, basamak ve kova sıralamasının bu sınırı neden aşabildiğini açıklamak ve bir problem için uygun algoritmayı seçmek.

2.1 Sıralama problemi ve değerlendirme ölçütleri

Bir algoritmanın iyi olup olmadığını konuşabilmek için önce problemi kesin biçimde tanımlamak, sonra da "iyi" sözcüğünün neyi ölçtüğünü belirlemek gerekir. Bu kesim önce sıralama problemini tanımlar. Ardından bölümdeki algoritmaları kısaca tanıtır ve onları karşılaştırırken kullanılacak ölçütleri açıklar. Ölçütlerden biri, kararlılık, bölüm boyunca birkaç kez karşımıza çıkacağı için ayrıca ele alınır.

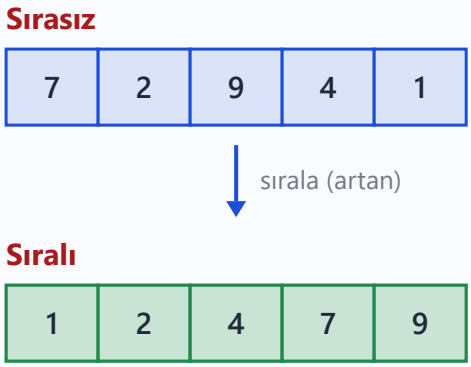
2.1.1 Sıralama nedir?

Günlük hayatta sürekli veriyle karşılaşırız: alışveriş listeleri, banka hareketleri, sosyal medya gönderileri, sınav sonuçları. Bu verinin işe yaraması için çoğu zaman belirli bir düzende tutulması gerekir. Bir e-posta kutusu tarihe göre, bir sözlük alfabeye göre, bir sınav listesi puana göre dizilir. **Sıralama** (*sorting*), bir listedeki öğeleri belirli bir ölçüte göre düzene koymaktır. Ölçüt sayısal değer, alfabetik sıra ya da tarih olabilir; düzen de artan (küçükten büyüğe) ya da azalan olabilir. Bu notta aksi söylenmedikçe tam sayılar artan sırada sıralanır. Azalan sıralama, karşılaştırmanın yönünü çevirmekten ibarettir.

> **Tanım 2.1**
>
> **Girdi:** n sayıdan oluşan bir dizi $\langle a_1, a_2, \ldots, a_n \rangle$.
> **Çıktı:** Aynı sayıların $a'_1 \le a'_2 \le \ldots \le a'_n$ koşulunu sağlayan bir **permütasyonu** (yeniden dizilişi) $\langle a'_1, a'_2, \ldots, a'_n \rangle$.

Tanımdaki iki koşulun ikisi de önemlidir. Çıktı sıralı olmalıdır, ama yalnız bu yetmez: çıktı girdinin bir permütasyonu olmalıdır. Yani hiçbir öğe kaybolmamalı, hiçbir öğe çoğalmamalı, yeni bir değer eklenmemelidir. Her öğeyi 0 yapan bir "algoritma" sıralı bir dizi üretir, ama permütasyon koşulunu çiğnediği için doğru değildir.

Gerçek uygulamalarda sıralanan şey çoğu zaman tek bir sayı değil, bir **kayıttır**: öğrenci numarası, adı ve bölümü olan bir satır gibi. Kaydın sıralamada kullanılan alanına **anahtar** (*key*) denir. Kaydın geri kalan alanları anahtarla birlikte taşınır. Bu bölümdeki örneklerde kayıt yalnız anahtardan oluşur; kararlılık konusunda ise anahtarın yanına bir etiket eklenir.



Şekil 2.1: Sıralama problemi. Sırasız 7, 2, 9, 4, 1 dizisi artan sırada 1, 2, 4, 7, 9 olur. Çıktı, girdideki beş sayının yeniden dizilişidir.

Sıralama neden bu kadar önemli? Çünkü sıralı veri üzerinde pek çok işlem çok kolaylaşır. Sıralı dizide bir değeri **ikili arama** ile *O(log n)* adımda bulmak mümkündür; sırasız dizide ise en kötü durumda bütün öğelere bakmak gerekir. Tekrar eden öğeler sıralı dizide yan yana durur; onları bulmak tek bir geçiş ister. İki sıralı liste tek geçişte birleştirilebilir. Ortanca, en küçük ya da en büyük *k* öğe doğrudan indisle okunur. Bu yüzden sıralama çoğu zaman tek başına bir amaç değil, başka bir algoritmanın ilk adımıdır.

2.1.2 Algoritma çeşitleri

Sıralama problemini çözen çok sayıda algoritma vardır ve her biri farklı bir çalışma ilkesine dayanır. Tablo 2.1 bu bölümde incelenen algoritmaları temel fikirleriyle birlikte listeler. Tablodaki ilk dört algoritma basit ve karesel sürelidir. Sonraki üçü *n log n* düzeyinde çalışır. Saymalı ve basamak sıralaması öğeleri karşılaştırmaz. Timsort ise iki algoritmanın karmasıdır ve bugün Java ile Python kütüphanelerinde nesne sıralamasında kullanılır.

Algoritma	Temel fikir
Kabarcık (*bubble sort*)	Komşu öğeleri karşılaştırır, yanlış sıradaysa yer değiştirir.
Eklemeli (*insertion sort*)	Sıradaki öğeyi sıralı kısımda doğru konuma yerleştirir.
Seçimli (*selection sort*)	En küçük öğeyi bulup başa koyar, sonra kalanları aynı yolla sıralar.
Shell (*shell sort*)	Önce uzak öğeleri, giderek küçülen aralıklarla eklemeli sıralar.
Birleştirme (*merge sort*)	Diziyi yarıya böler, yarıları sıralar, sonra birleştirir.
Hızlı (*quick sort*)	Bir pivot etrafında parçalar, iki parçayı özyinelemeli sıralar.
Heap sort	Max-heap kurar, en büyüğü tekrar tekrar sona taşır.
Saymalı / basamak (*counting / radix sort*)	Karşılaştırma yapmaz; anahtarları sayar ya da basamak basamak dağıtır.
Timsort	Eklemeli ve birleştirme sıralamasının karması; Java ve Python'un nesne sıralaması.

Tablo 2.1: Bölümdeki sıralama algoritmaları ve temel fikirleri.

2.1.3 Değerlendirme ölçütleri

Aynı problemi çözen iki algoritmadan hangisinin daha iyi olduğu, neye önem verildiğine bağlıdır. Sıralama algoritmaları için altı ölçüt öne çıkar.

Zaman. Bir sıralama algoritmasının süresi büyük ölçüde iki temel işlemin sayısıyla belirlenir: iki öğenin **karşılaştırılması** ve öğelerin **taşınması**. Taşıma, iki öğenin yerini değiştiren bir **takas** (*swap*) ya da bir öğeyi bir hücreden diğerine kopyalayan bir **kaydırma** olabilir. Bu sayılar girdi boyutu n'nin yanında girdinin kendisine de bağlıdır. Bu yüzden üç durum ayrı ayrı incelenir: **en iyi durum**, **ortalama durum** ve **en kötü durum**. Sonuçlar asimptotik gösterimle verilir: $\Theta(n^2)$, $\Theta(n \log n)$, $\Theta(n + k)$ gibi.

Ek bellek. Algoritmanın girdi dizisi dışında kullandığı bellek de önemlidir. Yalnız birkaç değişkenle, yani $O(1)$ ek bellekle çalışan algoritmaya **yerinde** (*in-place*) sıralama denir. Yerinde algoritma, öğeleri girdi dizisinin içinde yer değiştirerek düzenler. Birleştirme sıralaması gibi bazı algoritmalar ise yardımcı diziler ister.

Kararlılık. Eşit anahtarlı öğelerin girdideki göreli sırası çıktıda da korunuyorsa algoritma **kararlıdır** (*stable*). Bu ölçüt bir sonraki alt kesimde örnekle açıklanır.

Karşılaştırmalı olup olmama. Bölümdeki algoritmaların çoğu öğeler hakkında yalnız "$a \le b$ mi?" sorusunu sorar. Bu tür algoritmalara **karşılaştırmalı sıralama** denir. Böyle bir algoritma, en kötü durumda $\Omega(n \log n)$ karşılaştırma yapmak zorundadır. Saymalı ve basamak sıralaması anahtarların değerini doğrudan dizi indisi olarak kullanır; bu yüzden bu sınırın dışında kalır.

Uyarlanabilirlik. Neredeyse sıralı bir girdiyi rastgele bir girdiden daha hızlı sıralayan algoritmaya **uyarlanabilir** (*adaptive*) denir. Eklemeli sıralama, bayraklı kabarcık sıralaması ve Timsort uyarlanabilirdir. Seçimli sıralama ise girdi ne olursa olsun aynı sayıda karşılaştırma yapar.

Çevrim içi çalışma. Verinin tamamını baştan görmeden, öğeler geldikçe sıralayabilen algoritmaya **çevrim içi** (*online*) denir. Eklemeli sıralama çevrim içidir: o ana kadar gelen öğeler her an sıralıdır ve yeni gelen öğe doğru yere eklenir. Seçimli sıralama çevrim içi değildir, çünkü en küçüğü bulmak için bütün öğeleri görmesi gerekir.

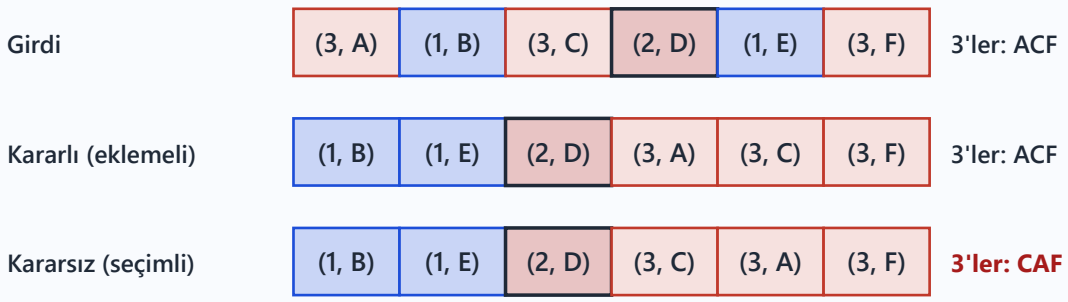
Bu ölçütler birbirinden bağımsızdır ve çoğu zaman biri için diğerinden ödün verilir. Birleştirme sıralaması hem hızlı hem kararlıdır ama ek bellek ister. Heap sort hızlı ve yerindedir ama kararsızdır. Hiçbir algoritma her ölçütte en iyi değildir; uygun algoritma uygulamanın neye ihtiyaç duyduğuna göre seçilir.

2.1.4 Kararlılık

Tanım 2.2

Bir sıralama algoritması, anahtarları eşit olan her iki kayıt için girdide önce gelen kaydı çıktıda da önce yerleştiriyorsa **kararlıdır**. Kısacası eşit anahtarlı kayıtların göreli sırası korunur. Bu koşulu en az bir girdide bozan algoritma **kararsızdır**.

Kararlılık ancak kayıtların anahtar dışında da bir kimliği olduğunda anlam taşır. Yalnız tam sayılardan oluşan bir dizide iki 3'ü birbirinden ayırt etmek mümkün değildir; hangisinin önce geldiğinin bir önemi yoktur. Şekil 2.2 bu yüzden kayıtları (anahtar, etiket) çiftleri olarak gösterir. Sıralama yalnız sayıya göre yapılır; harf, kaydın kimliğidir.



Şekil 2.2: Kararlılık örneği. Girdi (3, A), (1, B), (3, C), (2, D), (1, E), (3, F) kayıtlarından oluşur. Eklemeli sıralama anahtarı 3 olan kayıtları girdideki gibi A, C, F sırasında bırakır. Seçimli sıralamanın uzun mesafeli takasları bu sırayı C, A, F yapar.

Örnek 2.1

Seçimli sıralamanın (2.4) bu girdide neden sırayı bozduğu adım adım şöyle görülür. Her turda sırasız kısmın en küçüğü bulunur ve sırasız kısmın başındaki öğeyle takas edilir.

1. **1. tur.** En küçük anahtar 1'dir; ilk 1, yani (1, B) seçilir ve baştaki (3, A) ile takas edilir. Dizi: (1, B), (3, A), (3, C), (2, D), (1, E), (3, F). Bu takasla (3, A) birinci indise geçti.
2. **2. tur.** Sırasız kısım (3, A)'dan başlar; en küçük (1, E)'dir ve (3, A) ile takas edilir. Dizi: (1, B), (1, E), (3, C), (2, D), (3, A), (3, F). Şimdi (3, A), (3, C)'nin *arkasına* düştü.
3. **3. tur.** En küçük (2, D)'dir ve (3, C) ile takas edilir: (1, B), (1, E), (2, D), (3, C), (3, A), (3, F).
4. **4. ve 5. tur.** Kalan kayıtların anahtarları eşittir ve kod yalnız kesin küçükte en küçüğü değiştirir; takas yapılmaz.

Sonuçta anahtarı 3 olan kayıtlar C, A, F sırasındadır. Dizi anahtara göre doğru sıralanmıştır, ama A ile C'nin göreli sırası bozulmuştur. Bozan, ikinci turdaki uzun mesafeli takastır: (3, A), aradaki (3, C)'nin üzerinden atlayarak geriye gönderilmiştir.

Kararlılık en çok, **çok anahtarlı sıralamada** işe yarar. Bir öğrenci listesi önce ada göre, sonra kararlı bir algoritmayla bölüme göre sıralanırsa, her bölümün içi alfabetik kalır. İkinci sıralama eşit bölümlü öğrencilerin sırasına dokunmadığı için, birinci sıralamanın kurduğu alfabetik düzen korunur. Kararsız bir algoritma ise bölüm içindeki alfabetik sırayı dağıtabilir. Aynı fikir basamak sıralamasının da temelidir: sayılar basamak basamak sıralanırken her ara adımın kararlı olması gerekir (2.9.6).

Sık yapılan hata

Kararlılığı yalnız tam sayılardan oluşan bir dizi üzerinde denemek. Böyle bir dizide eşit öğeler ayırt edilemediği için kararsız bir algoritma da "doğru" görünür. Kararlılık, eşit anahtarlı ama farklı kimlikli kayıtlarla, örneğin (3, A) ve (3, C) gibi çiftlerle sınanır. Sıralama Özellikleri Laboratuvarı simülatörü tam olarak bu tür kayıtları kullanır.

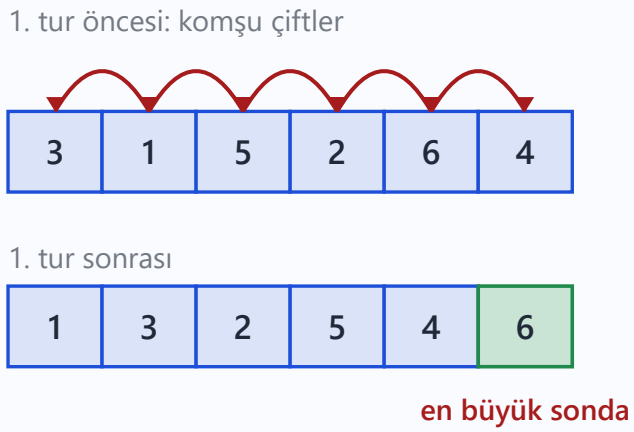
2.2 Kabarcık sıralaması

Kabarcık sıralaması, bir diziyi sıralamanın akla gelen ilk yollarından biridir. Yalnız komşu iki öğeye bakar ve yanlış sıradaysa yerlerini değiştirir. Pratikte yavaştır, ama iki önemli kavramı açıkça gösterir: turdan tura büyüyen sıralı bölge ve **ters çift**. Bu iki kavram eklemeli sıralamanın analizinde de kullanılır.

2.2.1 Fikir

Kabarcık sıralaması (*bubble sort*) diziyi soldan sağa tarar ve her adımda komşu iki öğeyi karşılaştırır. Soldaki öğe sağdakinden büyükse ikisinin yeri değiştirilir. Böylece büyük bir öğe, karşılaştırmalar boyunca sağa doğru taşınır; suyun içinde yükselen bir kabarcık gibi. Algoritma adını bu benzetmeden alır.

Dizinin baştan sona bir kez taranmasına **tur** denir. Bir turun sonunda o ana kadar sıralanmamış kısmın en büyük öğesi, dizinin sonundaki doğru yerine ulaşır. Nedeni basittir: en büyük öğeye gelindiğinde ondan büyük bir komşu yoktur, bu yüzden her karşılaştırmada takas edilip bir sağa geçer ve turun sonuna kadar taşınır. Birinci turdan sonra en büyük öğe, ikinci turdan sonra en büyük iki öğe yerindedir. Genel olarak *i*. tur bittiğinde en büyük *i* öğe son yerlerindedir. Bu yüzden her tur bir öncekinden bir öğe kısa olabilir.



Şekil 2.3: Kabarcık sıralamasının birinci turu. Üstte karşılaştırılan komşu çiftler oklarla gösterilir. Tur sonunda 3, 1, 5, 2, 6, 4 dizisi 1, 3, 2, 5, 4, 6 olur; en büyük öğe 6 son hücreye yerleşmiştir.

Algoritmaya küçük bir iyileştirme eklemek mümkündür. Bir turda hiç takas yapılmadıysa bütün komşu çiftler doğru sıradadır; bu da dizinin sıralı olduğu anlamına gelir. Bu durumda kalan turlara gerek yoktur ve algoritma erken durabilir. Bu bilgi bir **bayrak** değişkeniyle tutulur. Bayraklı sürüm sıralı bir girdiyi tek turda tanır.

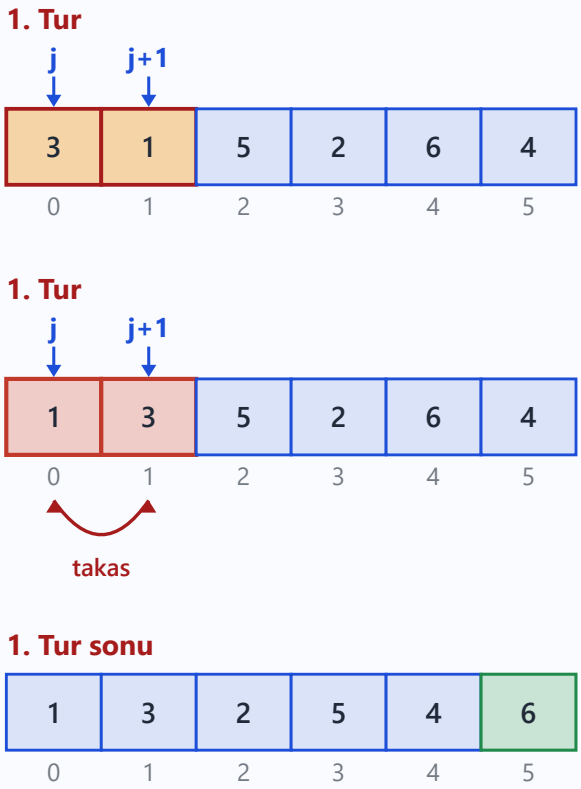
2.2.2 Adım adım: [3, 1, 5, 2, 6, 4]

Algoritmayı altı öğeli [3, 1, 5, 2, 6, 4] dizisi üzerinde izleyelim. Tablo 2.2 her karşılaştırmayı, sonucunu ve takastan sonraki diziyi gösterir. Karşılaştırılan çift `dizi[j]` ile `dizi[j+1]`'dir.

Tur	`j`	Karşılaştırma	Sonuç	Dizi
1	0	3 > 1?	evet, takas	[1, 3, 5, 2, 6, 4]
1	1	3 > 5?	hayır	[1, 3, 5, 2, 6, 4]
1	2	5 > 2?	evet, takas	[1, 3, 2, 5, 6, 4]
1	3	5 > 6?	hayır	[1, 3, 2, 5, 6, 4]
1	4	6 > 4?	evet, takas	[1, 3, 2, 5, 4, **6**]
2	0	1 > 3?	hayır	[1, 3, 2, 5, 4, 6]
2	1	3 > 2?	evet, takas	[1, 2, 3, 5, 4, 6]
2	2	3 > 5?	hayır	[1, 2, 3, 5, 4, 6]
2	3	5 > 4?	evet, takas	[1, 2, 3, 4, **5**, **6**]
3	0	1 > 2?	hayır	[1, 2, 3, 4, 5, 6]
3	1	2 > 3?	hayır	[1, 2, 3, 4, 5, 6]
3	2	3 > 4?	hayır	[1, 2, 3, 4, 5, 6]

Tablo 2.2: [3, 1, 5, 2, 6, 4] dizisinde bayraklı kabarcık sıralamasının bütün karşılaştırmaları.

Birinci turda beş karşılaştırma ve üç takas yapılır; tur sonunda 6 son yerine (indis 5) oturur. İkinci tur bir öğe kısadır: dört karşılaştırma, iki takas; 5 indis 4'e yerleşir. Üçüncü turda üç karşılaştırma yapılır ve hiçbirinde takas gerekmez. Bayrak `false` kaldığı için algoritma burada durur. Dördüncü ve beşinci turlara geçilmez. Toplam 12 karşılaştırma ve 5 takas yapılmıştır.



Şekil 2.4: Birinci turdan üç an. (a) `j = 0` : 3 ile 1 karşılaştırılıyor. (b) 3 > 1 olduğu için takas yapıldı; dizi [1, 3, 5, 2, 6, 4]. (c) Tur sonu: 6 son hücreye yerleşti (yeşil).

2.2.3 Java kodu

Aşağıdaki kod, `n` uzunluklu `dizi` dizisini bayraklı kabarcık sıralamasıyla sıralar.

```
boolean yerDegistiMi;
for(int i = 0; i < n - 1; i++) {
  yerDegistiMi = false;
  for(int j = 0; j < n - 1 - i; j++) {
    if(dizi[j] > dizi[j+1]) {
      int gecici = dizi[j];
      dizi[j] = dizi[j+1];
      dizi[j+1] = gecici;
      yerDegistiMi = true;
    }
  }
  if(yerDegistiMi == false) break;
}
```

Kod satır satır şöyle çalışır:

- **Dış döngü** (`i`) turları sayar. En fazla *n − 1* tur gerekir: *n − 1* öğe yerine oturunca kalan tek öğe de kendiliğinden yerindedir.
- Her turun başında `yerDegistiMi` bayrağı `false` yapılır. Tur boyunca bir takas olursa `true` olur.
- **İç döngü** (`j`) sıralanmamış kısımdaki komşu çiftleri gezer. Sondaki `i` öğe önceki turlarda yerine oturduğu için koşul `j < n - 1 - i`'dir. `j`'nin son değeri `n - 2 - i` olur; bu durumda `dizi[j+1]` hâlâ sıralanmamış kısmın son hücresidir.
- `dizi[j] > dizi[j+1]` ise takas yapılır. Takas üç atamayla olur: soldaki değer `gecici` değişkeninde saklanır, sağdaki sola yazılır, saklanan değer sağa yazılır. `gecici` olmasaydı ilk atama soldaki değeri yok ederdi.
- Tur bittiğinde bayrak hâlâ `false` ise o turda hiç takas olmamıştır; `break` dış döngüden çıkar.

Koşulda `>` kullanıldığına, `>=` kullanılmadığına dikkat etmek gerekir. Eşit iki öğe hiçbir zaman yer değiştirmez; bu yüzden eşit anahtarlı öğelerin sırası korunur ve kabarcık sıralaması **kararlıdır**. Algoritma yalnız `gecici`, `i`, `j` ve bayrak için yer kullanır: ek bellek *O(1)*'dir, sıralama yerindedir.

> **Sık yapılan hata**
>
> İç döngü koşulunu `j < n` yazmak. `j = n - 1` olduğunda `dizi[j+1]`, yani `dizi[n]` okunmaya çalışılır ve Java `ArrayIndexOutOfBoundsException` fırlatır. Komşu çift karşılaştıran her döngüde son indis *n − 2*'dir. `j < n - 1` yazmak doğrudur ama gereksiz iş yapar; `j < n - 1 - i` yerine oturmuş öğelere tekrar bakmaz.

2.2.4 Kod izleme: [5, 1, 9, 2, 10]

Kodu bu kez beş öğeli [5, 1, 9, 2, 10] dizisiyle çalıştıralım. `n = 5` olduğu için dış döngü en çok dört tur döner. Tablo 2.3 değişkenlerin her adımdaki değerini verir.

`i`	`j`	`dizi[j] > dizi[j+1]`	`yerDegistiMi`	`dizi`
0	—	tur başı	false	[5, 1, 9, 2, 10]
0	0	5 > 1: evet, takas	true	[1, 5, 9, 2, 10]
0	1	5 > 9: hayır	true	[1, 5, 9, 2, 10]
0	2	9 > 2: evet, takas	true	[1, 5, 2, 9, 10]
0	3	9 > 10: hayır	true	[1, 5, 2, 9, **10**]
1	—	tur başı	false	[1, 5, 2, 9, 10]
1	0	1 > 5: hayır	false	[1, 5, 2, 9, 10]
1	1	5 > 2: evet, takas	true	[1, 2, 5, 9, 10]
1	2	5 > 9: hayır	true	[1, 2, 5, **9**, **10**]
2	—	tur başı	false	[1, 2, 5, 9, 10]
2	0	1 > 2: hayır	false	[1, 2, 5, 9, 10]
2	1	2 > 5: hayır	false	[1, 2, 5, 9, 10]

`i`	`j`	`dizi[j] > dizi[j+1]`	`yerDegistiMi`	`dizi`
2	—	bayrak false → `break`	false	[1, 2, 5, 9, 10]

Tablo 2.3: `dizi = [5, 1, 9, 2, 10]` için kabarcık sıralaması kodunun izlemesi.

İlk turda (`i = 0`) iç döngü `j = 0`'dan 3'e kadar döner; dört karşılaştırmadan ikisinde takas yapılır ve tur sonunda 10 yerine oturur. İkinci turda (`i = 1`) iç döngü `j < 3` koşuluyla üç kez döner; 5 ile 2 yer değiştirir ve 9 yerine oturur. Üçüncü turda (`i = 2`) iki karşılaştırmanın ikisi de "hayır" çıkar. Bayrak `false` kaldığı için `break` çalışır ve dördüncü tur hiç başlamaz. Sonuç [1, 2, 5, 9, 10]'dur; toplam 9 karşılaştırma ve 3 takas yapılmıştır.

İzlemede ilginç bir ayrıntı vardır: ikinci turun sonunda dizi aslında sıralıdır, ama algoritma bunu bilemez. Sıralı olduğunu ancak takassız bir tur tamamlandığında anlar. Bu yüzden bayraklı kabarcık sıralaması, dizi sıralandıktan sonra her zaman bir tur daha yapar.

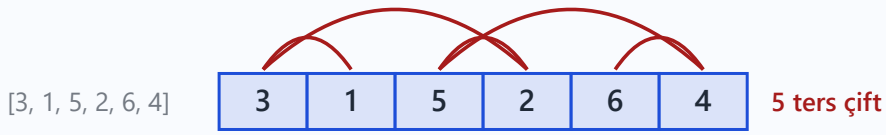
2.2.5 Analiz ve ters çiftler

Kabarcık sıralamasının maliyetini anlamak için önce dizinin ne kadar "sırasız" olduğunu ölçen bir kavram tanımlayalım.

Tanım 2.3

i < j iken *dizi[i] > dizi[j]* olan *(i, j)* indis çiftine **ters çift** (*inversion*) denir. Ters çift, birbirine göre yanlış sırada duran iki öğedir; komşu olmaları gerekmez.

Sıralı bir dizide hiç ters çift yoktur. Ters sıralı *n* öğeli dizide ise her çift ters çifttir; çift sayısı *n(n − 1)/2* olduğu için ters çift sayısı da budur. Diğer diziler bu iki uç arasında yer alır.



Şekil 2.5: [3, 1, 5, 2, 6, 4] dizisinin beş ters çifti: (3, 1), (3, 2), (5, 2), (5, 4) ve (6, 4). Her yay, birbirine göre yanlış sırada duran iki öğeyi birleştirir.

Teorem 2.1 (Komşu takas ve ters çift)

Yanlış sıradaki iki komşu öğenin takası, dizideki ters çift sayısını tam olarak bir azaltır. Bu yüzden kabarcık sıralamasının yaptığı takas sayısı, girdideki ters çift sayısına eşittir.

İspat. *dizi[j] > dizi[j+1]* olsun ve bu iki öğe takas edilsin. Bu iki öğe kendi aralarında bir ters çiftti; takastan sonra doğru sıradadırlar, yani bu ters çift ortadan kalkar. Başka bir öğe *x*'e bakalım. *x* ya iki öğenin solunda ya da sağındadır; takas iki öğenin *x*'e göre konumunu değiştirmez. Örneğin *x* soldaysa hem önce hem sonra iki öğenin de solundadır. Dolayısıyla *x*'i içeren ters çiftler aynen kalır. Toplam ters çift sayısı tam olarak bir azalır. Kabarcık sıralaması yalnız böyle takaslar yapar ve sıralı dizide ters çift kalmaz; demek ki takas sayısı başlangıçtaki ters çift sayısına eşittir. □

Teoremden en kötü durum doğrudan çıkar. Ters sıralı dizide $n(n-1)/2$ ters çift vardır; bu kadar takas gerekir. Karşılaştırma sayısı da aynıdır. Hiçbir tur erken bitmez ve turlar $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ karşılaştırma yapar. En iyi durumda girdi sıralıdır: bayraklı sürüm ilk turda $n-1$ karşılaştırma yapar, hiç takas yapmaz ve durur. Rastgele bir permütasyonda her çiftin ters olma olasılığı 1/2'dir; beklenen ters çift sayısı, dolayısıyla beklenen takas sayısı $n(n-1)/4 \approx n^2/4$'tür. Karşılaştırma sayısı ise ortalamada $n^2/2$'ye yakındır, çünkü dizi çoğu zaman son turlara kadar sıralanmaz.

Durum	Girdi	Karşılaştırma	Takas	Zaman
En iyi	sıralı	$n-1$ (tek tur)	0	$\Theta(n)$
Ortalama	rastgele	$\approx n^2/2$	$\approx n^2/4$	$\Theta(n^2)$
En kötü	ters sıralı	$n(n-1)/2$	$n(n-1)/2$	$\Theta(n^2)$

Tablo 2.4: Bayraklı kabarcık sıralamasının maliyeti.

Kabarcık sıralamasının artıları azdır ama gerçektir: anlaşılması ve yazılması çok kolaydır, yerinde ve kararlıdır, bayrakla sıralı girdiyi $\Theta(n)$'de tanır. Eksileri ise ağır basar. Karesel süre büyük dizilerde kabul edilemez. Her ters çift için ayrı bir takas yapar ve her takas üç atama ister. Ters çift başına bir taşıma yapan eklemeli sıralama ise takas yerine kaydırma kullanır. Bu yüzden kabarcık sıralaması pratikte eklemeli sıralamadan hep yavaştır ve daha çok öğretim amacıyla kullanılır.

Örnek 2.2

[5, 2, 4, 6, 1, 3] dizisine bayraklı kabarcık sıralaması uygulansın. Her turun sonundaki dizi ve sayılar şöyledir:

1. **Tur 1** (5 karşılaştırma): 5 ↔ 2, 5 ↔ 4, 6 ↔ 1, 6 ↔ 3 takas edilir; 4 takas. Dizi [2, 4, 5, 1, 3, 6].
2. **Tur 2** (4 karşılaştırma): 5 ↔ 1 ve 5 ↔ 3; 2 takas. Dizi [2, 4, 1, 3, 5, 6].
3. **Tur 3** (3 karşılaştırma): 4 ↔ 1 ve 4 ↔ 3; 2 takas. Dizi [2, 1, 3, 4, 5, 6].
4. **Tur 4** (2 karşılaştırma): 2 ↔ 1; 1 takas. Dizi [1, 2, 3, 4, 5, 6].
5. **Tur 5** (1 karşılaştırma): takas yok, bayrak `false`, `break`.

Toplam $5 + 4 + 3 + 2 + 1 = 15$ karşılaştırma ve $4 + 2 + 2 + 1 = 9$ takas yapılır. Dizideki ters çiftler (5,2), (5,4), (5,1), (5,3), (2,1), (4,1), (4,3), (6,1), (6,3)'tür; sayıları 9'dur ve takas sayısına eşittir. Dizi dördüncü turda sıralanmasına rağmen beşinci tur, sıralı olduğunu anlamak için yapılır.

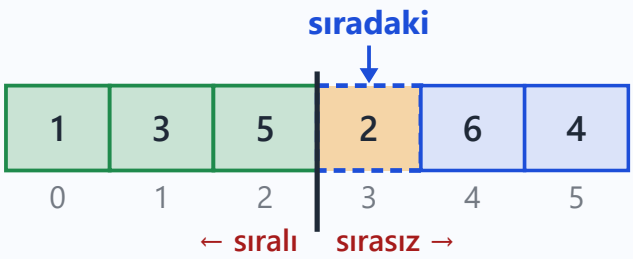
2.3 Eklemeli sıralama

Eklemeli sıralama, küçük dizilerde ve neredeyse sıralı dizilerde bilinen en hızlı algoritmalardan biridir. Bu yüzden yalnız bir öğretim örneği değildir; Timsort gibi gerçek kütüphane algoritmalarının da bir parçasıdır.

2.3.1 Fikir ve değişmez

Eklemeli sıralama (*insertion sort*), eldeki oyun kartlarını sıralamaya benzer. Kartlar tek tek alınır ve her yeni kart, eldeki sıralı kartların arasında doğru yere yerleştirilir. Eldeki kartlar her an sıralıdır; sıralı kısım her adımda bir kart büyür.

Dizide bu fikir şöyle uygulanır. Dizi iki parçaya ayrılır: solda **sıralı kısım**, sağda **sırasız kısım**. Başta sıralı kısım yalnız ilk öğeden oluşur; tek öğeli bir dizi zaten sıralıdır. Her adımda sırasız kısmın ilk öğesi alınır ve sıralı kısımdaki doğru konumuna yerleştirilir. Yer açmak için ondan büyük olan öğeler birer hücre sağa kaydırılır.



Şekil 2.6: Eklemeli sıralamada bir an. Sıralı kısım 1, 3, 5'tir; sırasız kısmın ilk öğesi 2 sıralı kısma yerleştirilecektir. Bunun için 5 ve 3 birer hücre sağa kayar, 2 indis 1'e yazılır.

Algoritmanın doğruluğu bir **döngü değişmezi** (*loop invariant*) ile gösterilir. Değişmez, döngünün her turunun başında doğru olan bir önermedir. Eklemeli sıralamanın değişmezi şudur: dış döngünün her adımının başında `dizi[0..i-1]`, başlangıçta bu indislerde bulunan öğelerin sıralı hâlidir. Başlangıçta *i = 1*'dir ve tek öğeli `dizi[0..0]` sıralıdır. Her adım `dizi[i]`'yi sıralı kısmın doğru yerine ekler; böylece `dizi[0..i]` sıralı olur ve değişmez bir sonraki adım için de doğru kalır. Döngü *i = n* olunca biter; değişmez bu anda `dizi[0..n-1]`'in, yani bütün dizinin sıralı olduğunu söyler.

2.3.2 Adım adım: [3, 1, 5, 2, 6, 4]

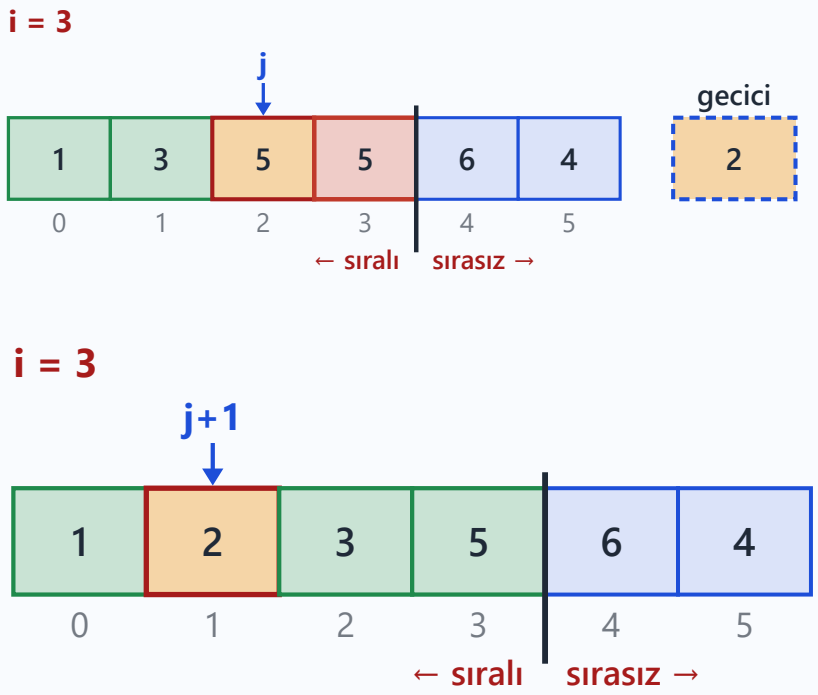
Kabarcık sıralamasında kullanılan [3, 1, 5, 2, 6, 4] dizisini bu kez eklemeli sıralamayla sıralayalım. Yerleştirilecek öğe `gecici` adlı bir değişkende tutulur.

`i`	`gecici`	Sağa kayan öğeler	Yerleştiği indis	Kaydırma	Dizi
1	1	3	0	1	[**1, 3**, 5, 2, 6, 4]

`i`	`gecici`	Sağa kayan öğeler	Yerleştiği indis	Kaydırma	Dizi
2	5	—	2	0	[**1, 3, 5**, 2, 6, 4]
3	2	5, 3	1	2	[**1, 2, 3, 5**, 6, 4]
4	6	—	4	0	[**1, 2, 3, 5, 6**, 4]
5	4	6, 5	3	2	[**1, 2, 3, 4, 5, 6**]

Tablo 2.5: [3, 1, 5, 2, 6, 4] dizisinde eklemeli sıralama. Her satır bir öğenin sıralı kısma eklenmesidir.

İlk adımda `gecici = 1` alınır. Solundaki 3, 1'den büyüktür; 3 bir sağa kayar ve dizinin başına gelinir. 1, indis 0'a yazılır. İkinci adımda `gecici = 5` 'tir; solundaki 3 ondan küçük olduğu için hiçbir öğe kaymaz ve 5 yerinde kalır. Üçüncü adım daha uzundur: `gecici = 2` için önce 5, sonra 3 sağa kayar; 1 ≤ 2 olduğunda durulur ve 2, indis 1'e yazılır. Dördüncü adımda 6 yerinde kalır. Son adımda 4 için 6 ile 5 kayar ve 4, indis 3'e yerleşir. Toplam 5 kaydırma yapılmıştır; bu sayı dizinin ters çift sayısına eşittir.



Şekil 2.7: Üçüncü adımdan iki an (`i = 3`). (a) 5 > 2 olduğu için 5 bir sağa kaydırıldı; 5 şimdi hem indis 2'de hem indis 3'te görünür, çünkü indis 2'nin üzerine henüz yazılmadı. `gecici` kutusu 2'yi saklar. (b) 3 de kaydı, 1 ≤ 2 olduğu için durulur ve 2 indis 1'e yerleştirilir.

Şekil 2.7'deki (a) anı önemli bir noktayı gösterir. Kaydırma bir kopyalamadır: `dizi[j+1] = dizi[j]` atamasından sonra aynı değer iki hücrede birden bulunur. Bu bir hata değildir; soldaki kopyanın üzerine bir sonraki adımda ya başka bir kaydırma ya da `gecici` yazılır. Yerleştirilecek öğenin asıl hücresi ise ilk kaydırmada ezilir. Bu yüzden değer, döngü başlamadan `gecici` 'ye alınmalıdır.

2.3.3 Java kodu

```
for(int i = 1; i < n; i++) {
  int gecici = dizi[i];
  int j = i - 1;
  while(j >= 0 && dizi[j] > gecici) {
    dizi[j+1] = dizi[j];
    j = j - 1;
  }
  dizi[j+1] = gecici;
}
```

- Dış döngü `i = 1`'den başlar, çünkü tek öğeli `dizi[0..0]` zaten sıralıdır.
- `gecici`, yerleştirilecek öğedir; bu değere **anahtar** da denir. Kaydırmalar onun hücresinin üzerine yazacağı için önce saklanır.
- `j`, sıralı kısmın son indisinden (`i - 1`) başlar ve sola doğru ilerler.
- `while` döngüsü, soldaki öğe `gecici`'den büyük olduğu sürece o öğeyi bir sağa kaydırır ve `j`'yi bir azaltır. Koşuldaki `j >= 0` denetimi dizinin başını aşmayı önler. Java'da `&&` kısa devre yapar: `j = -1` olunca ikinci koşul hiç değerlendirilmez ve `dizi[-1]` okunmaz.
- Döngü bittiğinde ya `j = -1`'dir ya da `dizi[j] <= gecici`'dir. Her iki durumda da boşalan yer `j + 1`'dir ve `gecici` oraya yazılır.

Eklemeli sıralama takas değil kaydırma yapar. Bir takas üç atamadır; bir kaydırma ise tek atamadır. Bir öğeyi *k* hücre sola taşımak kabarcık sıralamasında *k* takas, yani *3k* atama ister. Eklemeli sıralamada ise *k* kaydırma ve iki atama (`gecici`'ye alma ve yerine yazma), toplam *k + 2* atama yeter. Eklemeli sıralamanın kabarcıktan belirgin biçimde hızlı olmasının nedeni budur.

Koşulda `dizi[j] > gecici` kullanılır, `>=` kullanılmaz. Eşit bir öğeye gelindiğinde durulur ve yeni öğe eşitin sağına yerleşir. Böylece eşit öğelerin girdideki sırası korunur ve eklemeli sıralama **kararlıdır**. Ek bellek yalnız `gecici`, `i` ve `j`'dir; algoritma **yerindedir**.

> **Sık yapılan hata**
>
> Anahtarı saklamadan kaydırmaya başlamak. `gecici` yerine döngüde doğrudan `dizi[i]` ile karşılaştırma yapılırsa ilk kaydırma (`dizi[i] = dizi[i-1]`) anahtarı ezer. Bundan sonra karşılaştırmalar yanlış değerle yapılır ve dizideki bir değer kaybolup başka bir değer çoğalır. Bu, permütasyon koşulunu bozan bir hatadır.

2.3.4 Kod izleme: [5, 1, 9, 2, 10]

Kodu [5, 1, 9, 2, 10] dizisiyle çalıştıralım (`n = 5`). Tablo 2.6 her `while` denetimini ayrı bir satırda gösterir.

`i`	`gecici`	`j`	`while` **koşulu**	**İşlem**	`dizi`
1	1	0	0 ≥ 0 ve 5 > 1: doğru	`dizi[1] = 5`, `j = -1`	[5, 5, 9, 2, 10]
1	1	−1	−1 ≥ 0: yanlış	`dizi[0] = 1`	[1, 5, 9, 2, 10]
2	9	1	5 > 9: yanlış	`dizi[2] = 9`	[1, 5, 9, 2, 10]
3	2	2	9 > 2: doğru	`dizi[3] = 9`, `j = 1`	[1, 5, 9, 9, 10]
3	2	1	5 > 2: doğru	`dizi[2] = 5`, `j = 0`	[1, 5, 5, 9, 10]
3	2	0	1 > 2: yanlış	`dizi[1] = 2`	[1, 2, 5, 9, 10]
4	10	3	9 > 10: yanlış	`dizi[4] = 10`	[1, 2, 5, 9, 10]

Tablo 2.6: `dizi = [5, 1, 9, 2, 10]` için eklemeli sıralama kodunun izlemesi.

`i = 1`'de 5 sağa kayar ve `j` −1'e iner. Bu noktada `j >= 0` yanlıştır ve döngü biter; 1, `dizi[0]`'a yazılır. `i = 2`'de solundaki 5 ≤ 9 olduğundan hiç kaydırma olmaz; 9 kendi hücresine yeniden yazılır. `i = 3`'te 9 ve 5 kayar, 1 ≤ 2'de durulur ve 2, `dizi[1]`'e girer. `i = 4`'te 10 yerinde kalır. `i = 5` olunca dış döngü koşulu (`5 < 5`) yanlıştır ve algoritma biter. Toplam 6 karşılaştırma ve 3 kaydırma yapılmıştır. Dizideki ters çiftler (5,1), (5,2), (9,2)'dir; sayıları kaydırma sayısına eşittir.

2.3.5 Analiz

Eklemeli sıralamada her kaydırma, yerleştirilen öğe ile ondan büyük bir öğenin oluşturduğu tek bir ters çifti düzeltir. Bu yüzden toplam kaydırma sayısı, kabarcık sıralamasındaki takas sayısı gibi, ters çift sayısı I'ya eşittir. Karşılaştırma sayısı ise en çok $I + (n - 1)$'dir. Her başarılı karşılaştırma bir kaydırma yaptırır. Ayrıca her öğe için döngüyü bitiren en fazla bir başarısız karşılaştırma vardır. Öğe dizinin başına kadar kayarsa bu son karşılaştırma yapılmaz, çünkü `j >= 0` denetimi önce yanlış olur.

Teorem 2.2 (Eklemeli sıralamanın süresi)

Ters çift sayısı I olan n öğeli bir dizide eklemeli sıralama tam I kaydırma ve en çok $I + n - 1$ karşılaştırma yapar. Çalışma süresi $\Theta(n + I)$'dır.

Bu teorem eklemeli sıralamanın **uyarlanabilir** olduğunu açıkça gösterir. Sıralı dizide $I = 0$'dır: her öğe için tek bir karşılaştırma yapılır, toplam $n - 1$, süre $\Theta(n)$'dir. Ters sıralı dizide $I = n(n - 1)/2$'dir: her öğe dizinin başına kadar kayar ve süre $\Theta(n^2)$ olur. Rastgele dizide beklenen I değeri $n(n - 1)/4$'tür; karşılaştırma ve kaydırma sayısı yaklaşık $n^2/4$'tür. Bu, kabarcık sıralamasının ortalama karşılaştırma sayısının yaklaşık yarısıdır.

Durum	**Girdi**	**Karşılaştırma**	**Kaydırma**	**Zaman**
En iyi	sıralı	$n - 1$	0	$\Theta(n)$
Ortalama	rastgele	$\approx n^2/4$	$\approx n^2/4$	$\Theta(n^2)$
En kötü	ters sıralı	$n(n - 1)/2$	$n(n - 1)/2$	$\Theta(n^2)$

Tablo 2.7: Eklemeli sıralamanın maliyeti.

En iyi ve en kötü durumun büyüme hızları farklı olduğu için eklemeli sıralamanın süresi bütün girdiler için tek bir Θ ile yazılamaz. Doğru ifade şudur: süre bütün girdiler için $O(n^2)$ ve $\Omega(n)$'dir; en kötü durum $\Theta(n^2)$, en iyi durum $\Theta(n)$'dir. "Üst sınır n^2, alt sınır n, o hâlde süre $\Theta(n \log n)$" gibi bir ara değer çıkarımı yanlıştır; sınırlar ayrı girdi aileleri için sıkıdır.

Eklemeli sıralamanın artıları uzun bir listedir: kararlıdır, yerindedir, çevrim içidir, uyarlanabilirdir ve döngüsü çok sadedir. Küçük dizilerde (kabaca $n \lesssim 20$–50) ek yükü az olduğu için $n \log n$'lik algoritmalardan bile hızlıdır. Bu yüzden Timsort ve birçok kütüphane, hızlı sıralamada ve birleştirme sıralamasında küçük alt dizileri eklemeli sıralamaya bırakır. Eksiği ise rastgele büyük dizilerdeki karesel süredir. Bunun kökü, bir öğenin tek adımda yalnız bir hücre ilerleyebilmesidir: dizinin sonundaki küçük bir öğe başa ulaşmak için yaklaşık n kaydırma ister. Shell sıralaması tam bu sorunu çözmek için geliştirilmiştir (2.5).

Örnek 2.3

Eklemeli sıralama [5, 2, 4, 6, 1, 3] dizisini sıralarken `dizi[j] > gecici` karşılaştırması kaç kez yapılır, kaç kaydırma yapılır?

1. `i = 1`, `gecici = 2`: 5 > 2, kaydır; `j = -1`, koşul burada biter. 1 karşılaştırma, 1 kaydırma. Dizi [2, 5, 4, 6, 1, 3].
2. `i = 2`, `gecici = 4`: 5 > 4 kaydır; 2 > 4 yanlış. 2 karşılaştırma, 1 kaydırma. Dizi [2, 4, 5, 6, 1, 3].
3. `i = 3`, `gecici = 6`: 5 > 6 yanlış. 1 karşılaştırma, 0 kaydırma.
4. `i = 4`, `gecici = 1`: 6, 5, 4, 2 kayar; `j = -1`. 4 karşılaştırma, 4 kaydırma. Dizi [1, 2, 4, 5, 6, 3].
5. `i = 5`, `gecici = 3`: 6, 5, 4 kayar; 2 > 3 yanlış. 4 karşılaştırma, 3 kaydırma. Dizi [1, 2, 3, 4, 5, 6].

Toplam 12 karşılaştırma ve 9 kaydırma yapılır. Kaydırma sayısı dizinin ters çift sayısına (9) eşittir. Karşılaştırma sayısı $I + n - 1 = 9 + 5 = 14$ sınırının 2 altındadır. Nedeni şudur: `i = 1` ve `i = 4`'te öğe dizinin başına kadar kaymıştır. Döngü bu iki adımda `j >= 0` denetimiyle, ek karşılaştırma yapılmadan bitmiştir.

2.4 Seçimli sıralama

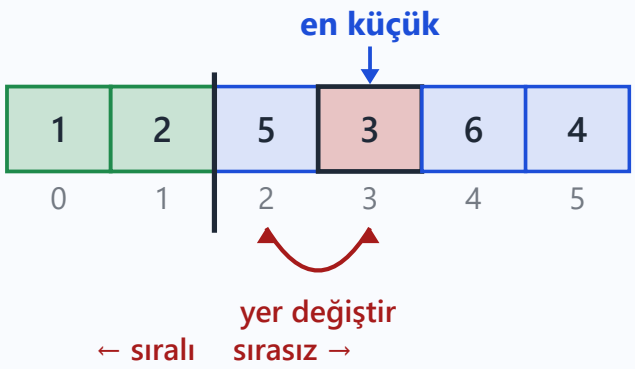
Seçimli sıralama, kabarcık ve eklemeli sıralamadan farklı bir ilkeye dayanır. Öğeleri azar azar kaydırmaz; her turda doğru öğeyi seçer ve onu tek bir takasla kesin yerine koyar. Bu yaklaşım takas sayısını en aza indirir, ama karşılaştırma sayısını girdiden bağımsız olarak hep yüksek tutar.

2.4.1 Fikir

Seçimli sıralama (*selection sort*) da diziyi sıralı ve sırasız iki bölüme ayırır. Başta sıralı bölüm boştur. Her turda üç şey yapılır:

1. Sırasız bölümün en küçük öğesi bulunur.
2. Bu öğe, sırasız bölümün başındaki öğeyle yer değiştirir.
3. Takastan sonra o hücre sıralı bölüme katılır; sıralı bölüm bir öğe büyür.

Sıralı bölüme giren öğe artık kesin yerindedir. Ondan küçük bütün öğeler daha önceki turlarda solundaki hücrelere yerleşmiştir. Sağındakilerin hepsi de ondan büyük ya da ona eşittir. Bu yönüyle seçimli sıralama, eklemeli sıralamadan ayrılır. Eklemeli sıralamada sıralı kısım kendi içinde sıralıdır ama öğeler henüz son yerlerinde değildir; sonra gelen küçük bir öğe hepsini kaydırabilir. *n* öğeli dizide *n − 1* tur yeter, çünkü son kalan öğe kendiliğinden en büyüktür. Her tur en fazla bir takas yapar.



Şekil 2.8: Seçimli sıralamada bir tur. Sıralı bölüm 1, 2'dir. Sırasız bölümün en küçüğü 3 bulunur ve sırasız bölümün başındaki 5 ile yer değiştirir.

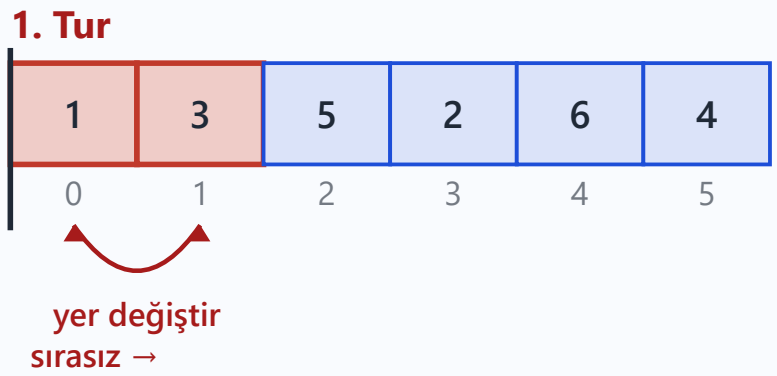
2.4.2 Adım adım: [3, 1, 5, 2, 6, 4]

Her turda sırasız bölümün ilk öğesi geçici en küçük kabul edilir; sonra kalan öğeler tek tek onunla karşılaştırılır. Daha küçük bir öğe bulunursa en küçük güncellenir.

Tur	Başlangıç en küçük	Taranan	Bulunan en küçük	Takas	Dizi
1	3	1, 5, 2, 6, 4	1	3 ↔ 1	[**1**, 3, 5, 2, 6, 4]
2	3	5, 2, 6, 4	2	3 ↔ 2	[**1, 2**, 5, 3, 6, 4]
3	5	3, 6, 4	3	5 ↔ 3	[**1, 2, 3**, 5, 6, 4]
4	5	6, 4	4	5 ↔ 4	[**1, 2, 3, 4**, 6, 5]
5	6	5	5	6 ↔ 5	[**1, 2, 3, 4, 5**, 6]

Tablo 2.8: [3, 1, 5, 2, 6, 4] dizisinde seçimli sıralama. “Taranan” sütunu, geçici en küçükle karşılaştırılan öğeleri sırasıyla verir.

Birinci turda geçici en küçük 3'tür. 1 < 3 olduğu için en küçük 1 olur; sonraki 5, 2, 6, 4'ün hiçbiri 1'den küçük değildir. Beş karşılaştırmadan sonra 1, baştaki 3 ile yer değiştirir. İkinci turda tarama 3'ten başlar ve 2'yi bulur; 2, sırasız bölümün başındaki 3 ile yer değiştirir. Dikkat edilirse 3 bu takasla indis 3'e, yani 5'in sağına gönderilmiştir; üçüncü turda yeniden seçilip indis 2'ye getirilir. Dördüncü turda 4, 5 ile; beşinci turda 5, 6 ile yer değiştirir. Beş tur sonunda son öğe 6 kendiliğinden yerindedir. Toplam *5 + 4 + 3 + 2 + 1 = 15* karşılaştırma ve 5 takas yapılmıştır.



Şekil 2.9: Birinci turun sonu. Bulunan en küçük 1, sırasız bölümün başındaki 3 ile yer değiştirdi; kalın çizgi sıralı bölümün sınırıdır ve bir sonraki turda bir hücre sağa kayar.

2.4.3 Java kodu

```
public void sirala(int[] dizi) {
  int n = dizi.length;
  for(int i = 0; i < n - 1; i++) {
    int enKucuk = i;
    for(int j = i + 1; j < n; j++) {
      if(dizi[j] < dizi[enKucuk]) {
        enKucuk = j;
      }
    }
    int gecici = dizi[enKucuk];
    dizi[enKucuk] = dizi[i];
    dizi[i] = gecici;
  }
}
```

- Dış döngü `i = 0`'dan `n - 2`'ye kadar döner; `i`, sırasız bölümün başıdır ve bu turda doldurulacak hücredir.
- `enKucuk`, o ana kadar bulunan en küçük öğenin *indisini* tutar, değerini değil. Takas için hangi hücreyle yer değiştirileceğini bilmek gerekir; değeri `dizi[enKucuk]` ile her an okunabilir.

- İç döngü `j = i + 1`'den dizinin sonuna kadar sırasız bölümün tamamını tarar. Girdi sıralı olsa bile bu tarama kısalmaz; en küçüğün gerçekten en küçük olduğunu doğrulamanın tek yolu bütün öğelere bakmaktır.
- En küçük yalnız `dizi[j] < dizi[enKucuk]` olduğunda, yani kesin küçüklükte güncellenir. Eşitlikte ilk bulunan korunur.
- İç döngüden sonra `dizi[enKucuk]` ile `dizi[i]` üç atamayla yer değiştirir. `enKucuk == i` ise öğe zaten yerindedir; kod yine de "kendisiyle takas" yapar, bu zararsızdır ama gereksiz üç atamadır.

Seçimli sıralama tur başına bir takas yapar, toplam en fazla *n − 1*. Bu sayı diğer basit algoritmaların takas sayısından çok daha azdır. Yazma işleminin pahalı olduğu ortamlarda, örneğin her yazmanın hücreyi yıprattığı flash bellekte, az sayıda takas seçimli sıralamanın tek gerçek avantajıdır.

> **Sık yapılan hata**
>
> En küçüğü indis yerine değer olarak tutmak. `int enKucukDeger = dizi[i]` yazılıp iç döngüde yalnız değer güncellenirse tur sonunda en küçüğün *hangi hücrede* olduğu bilinmez ve takas yapılamaz. Değer, indisten her an okunabildiği için indisi tutmak yeterlidir.

2.4.4 Kod izleme: [5, 1, 10, 2, 9]

`i`	`j`	`dizi[j] < dizi[enKucuk]`	`enKucuk`	**Tur sonu** `dizi`
0	1	1 < 5: evet	1	
0	2	10 < 1: hayır	1	
0	3	2 < 1: hayır	1	
0	4	9 < 1: hayır	1	`dizi[1] ↔ dizi[0]` : [1, 5, 10, 2, 9]
1	2	10 < 5: hayır	1	
1	3	2 < 5: evet	3	
1	4	9 < 2: hayır	3	`dizi[3] ↔ dizi[1]` : [1, 2, 10, 5, 9]
2	3	5 < 10: evet	3	
2	4	9 < 5: hayır	3	`dizi[3] ↔ dizi[2]` : [1, 2, 5, 10, 9]
3	4	9 < 10: evet	4	`dizi[4] ↔ dizi[3]` : [1, 2, 5, 9, 10]

Tablo 2.9: `dizi = [5, 1, 10, 2, 9]` için seçimli sıralama kodunun izlemesi. Her satır bir karşılaştırmadır.

Her turun başında `enKucuk = i` atanır. `i = 0`'da en küçük hemen `j = 1`'de bulunur ve sonraki üç karşılaştırma onu değiştirmez. `j = 5` olunca iç döngü biter ve 1 ile 5 yer değiştirir. `i = 1`'de en küçük 2'dir (indis 3); `i = 2`'de 5'tir (yine indis 3); `i = 3`'te 9'dur (indis 4). `i = 4` için dış döngü koşulu `4 < n − 1` yanlıştır ve algoritma biter. Toplam *4 + 3 + 2 + 1 = 10* karşılaştırma ve 4 takas yapılmıştır.

2.4.5 Analiz ve kararsızlık

Seçimli sıralamanın karşılaştırma sayısı girdiye hiç bağlı değildir. *i*. turda iç döngü $n - 1 - i$ kez döner; toplam

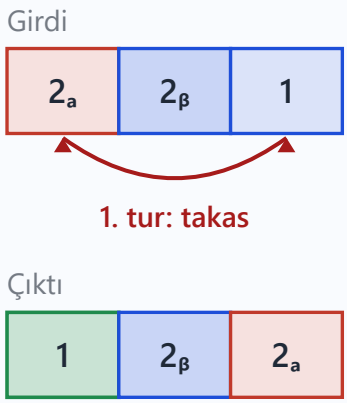
> $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$

karşılaştırmadır. Sıralı, ters sıralı ya da rastgele, her girdide bu kadar karşılaştırma yapılır. Bu yüzden en iyi, ortalama ve en kötü durum aynıdır: $\Theta(n^2)$. Seçimli sıralama **uyarlanabilir değildir**; sıralı bir diziye de $n(n - 1)/2$ karşılaştırma harcar.

Ölçü	Değer
Karşılaştırma	$n(n - 1)/2$, her girdide
Takas	en fazla $n - 1$
Zaman	en iyi = ortalama = en kötü = $\Theta(n^2)$
Ek bellek	$O(1)$, yerinde
Kararlı mı?	Hayır (uzun mesafeli takas)

Tablo 2.10: Seçimli sıralamanın özellikleri.

Seçimli sıralama kararlı değildir. Nedeni, takasın yalnız iki öğenin yerini değiştirmekle kalmayıp birini aradaki bütün öğelerin üzerinden atlatmasıdır. Sırasız bölümün başındaki öğe, en küçüğün bulunduğu uzak bir hücreye gönderilir ve bu sırada kendisine eşit bir öğenin üzerinden atlayabilir. En küçük örnek üç öğelidir.



Şekil 2.10: Seçimli sıralamanın kararsızlığı. Girdi [2_a, 2_b, 1]'dir. Birinci turda en küçük 1, baştaki 2_a ile yer değiştirir ve 2_a, 2_b'nin arkasına geçer. Çıktı [1, 2_b, 2_a]'dır.

İkinci turda sırasız bölüm [2_b, 2_a]'dır; kod kesin küçükte güncelleme yaptığı için 2_b yerinde kalır ve takas olmaz. Sonuç sayısal olarak doğrudur, ama 2_a ile 2_b'nin göreli sırası tersine dönmüştür. Seçimli sıralamayı kararlı yapmak mümkündür: en küçük öğe takasla değil, aradaki öğeleri bir sağa kaydıra-

rak başa getirilir. Ancak bu, takas sayısının az olması avantajını ortadan kaldırır ve maliyeti eklemeli sıralamaya yaklaştırır.

2.4.6 Üç basit algoritma aynı girdide

Kabarcık, eklemeli ve seçimli sıralama aynı karesel sınıftadır, ama sabit çarpanları ve girdiye tepkileri farklıdır. Tablo 2.11 üç algoritmanın üç girdide yaptığı karşılaştırma ve taşıma (takas ya da kaydırma) sayılarını verir. Değerler bu bölümdeki kodlar çalıştırılarak sayılmıştır; $n = 6$'dır.

Algoritma	Taşıma türü	[3, 1, 5, 2, 6, 4]	sıralı 1…6	ters 6…1
Kabarcık (bayraklı)	takas	12 / 5	5 / 0	15 / 15
Eklemeli	kaydırma	9 / 5	5 / 0	15 / 15
Seçimli	takas	15 / 5	15 / 0	15 / 3

Tablo 2.11: Hücreler "karşılaştırma / taşıma" biçimindedir. Kabarcık ve seçimlide taşıma takas, eklemelide kaydırmadır.

Tablodan üç gözlem çıkar. Birincisi, kabarcık ve eklemeli sıralamada taşıma sayısı ters çift sayısına eşittir: [3, 1, 5, 2, 6, 4] için 5, ters sıralı dizi için $6 \cdot 5/2 = 15$. Ama bir takas üç atama, bir kaydırma tek atamadır; aynı sayıdaki taşıma eklemelide çok daha ucuzdur. İkincisi, seçimli sıralama girdiden bağımsız olarak hep $n(n-1)/2 = 15$ karşılaştırma yapar; buna karşılık en az takası o yapar. Ters sıralı dizide yalnız 3 takas yeter: 6 ↔ 1, 5 ↔ 2 ve 4 ↔ 3. İkişer öğe birbirinin yerine geçer ve ortadaki takaslar gereksiz kalır. Üçüncüsü, sıralı girdide bayraklı kabarcık ve eklemeli sıralama yalnız $n - 1 = 5$ karşılaştırma yapar; ikisi de uyarlanabilirdir.

Seçimli sıralamada sıralı girdi için takas sütununda 0 görünmesinin nedeni, sayımın yalnız `enKucuk ≠ i` olan gerçek takasları içermesidir. Kod yazıldığı biçimiyle her turda kendisiyle takas yapar; bunlar atama olarak sayılırsa sıralı girdide de $n - 1$ "takas" vardır.

2.5 Shell sıralaması

Eklemeli sıralamanın zayıf noktası, bir öğenin her adımda yalnız bir hücre ilerleyebilmesidir. Shell sıralaması bu kısıtı kaldırır: öğeleri önce büyük adımlarla, sonra giderek küçülen adımlarla hareket ettirir. Son adımda yine sıradan eklemeli sıralama çalışır, ama bu kez dizi neredeyse sıralıdır.

2.5.1 Fikir ve kod

Shell sıralaması (*shell sort*), 1959'da D. L. Shell'in önerdiği, eklemeli sıralamanın bir genellemesidir. Algoritma bir **aralık** (*gap*) değeriyle çalışır. Birbirinden aralık kadar uzaktaki öğeler aynı grubu oluşturur: aralık 4 ise indis 0, 4, 8, … bir grup; 1, 5, 9, … başka bir grup olur. Her grup kendi içinde eklemeli sıralamayla sıralanır. Bu sırada küçük bir öğe tek hamlede aralık kadar hücre atlar. Ardından aralık küçültülür ve aynı işlem tekrarlanır. Son turda aralık 1'dir; bu tur sıradan eklemeli sıralamadır.

Son tur zaten eklemeli sıralama olduğuna göre önceki turlar neden işe yarar? Çünkü önceki turlar, uzaktaki ters çiftlerin çoğunu ucuza giderir. Aralık 4 olan turda bir kaydırma bir öğeyi dört hücre taşır; aynı taşıma eklemeli sıralamada dört kaydırma isterdi. Aralık 1'e gelindiğinde dizide çok az ters çift kalır ve 2.3.5'teki teoreme göre eklemeli sıralama böyle bir diziyi hızla bitirir.

```
void kabukSirala(int[] dizi) {
  int n = dizi.length;
  for (int aralik = n / 2; aralik > 0; aralik /= 2)
    for (int i = aralik; i < n; i++) {
      int gecici = dizi[i], j;
      for (j = i; j >= aralik && dizi[j - aralik] > gecici; j -= aralik)
        dizi[j] = dizi[j - aralik];
      dizi[j] = gecici;
    }
}
```

- Dış döngü aralığı `n / 2`'den başlatır ve her turda yarıya indirir (`aralik /= 2`); aralık 0 olunca durur. Tam sayı bölmesi olduğu için aralık dizisi *n = 8* için 4, 2, 1; *n = 6* için 3, 1'dir.
- Orta döngü `i = aralik`'tan dizinin sonuna kadar gider. Her `i`, kendi grubunda eklenecek öğedir. Gruplar ayrı ayrı değil, iç içe işlenir: `i` bir artınca sıradaki öğe başka bir gruba aittir.
- İç döngü eklemeli sıralamanın `while` döngüsünün aynısıdır, yalnız adım 1 yerine `aralik`'tır. `j`, `aralik`'tan küçük olunca ya da soldaki grup öğesi `gecici`'den büyük olmayınca durur. Boşalan `dizi[j]` hücresine `gecici` yazılır.

Shell sıralaması yerindedir: yalnız birkaç değişken kullanır, ek bellek *O(1)*'dir. Ama kararlı değildir. Eşit iki öğe farklı gruplarda bulunabilir; büyük aralıklı bir turda biri diğerinin üzerinden atlayarak önüne geçebilir.

2.5.2 Adım adım: [6, 5, 3, 1, 8, 7, 2, 4]

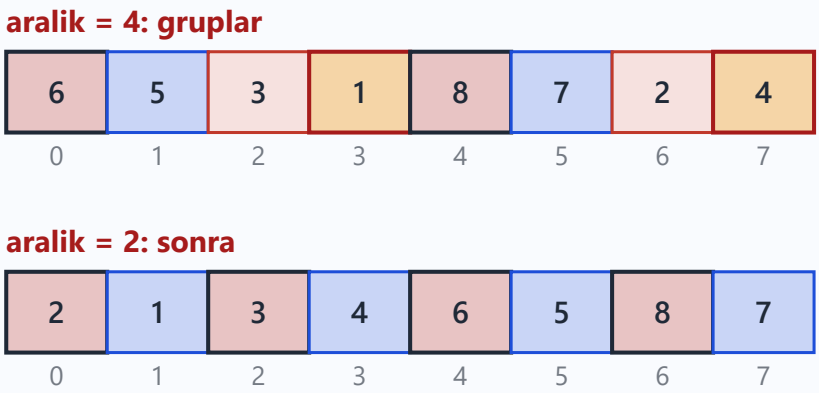
Sekiz öğeli dizide aralık dizisi 4, 2, 1'dir. Tablo 2.12 her aralıktan sonraki durumu verir.

`aralik`	Gruplar (önce)	Gruplar (sonra)	Kaydırma	Dizi
4	[6, 8], [5, 7], [3, 2], [1, 4]	[6, 8], [5, 7], [2, 3], [1, 4]	1	[6, 5, 2, 1, 8, 7, 3, 4]
2	[6, 2, 8, 3], [5, 1, 7, 4]	[2, 3, 6, 8], [1, 4, 5, 7]	6	[2, 1, 3, 4, 6, 5, 8, 7]
1	[2, 1, 3, 4, 6, 5, 8, 7]	[1, 2, 3, 4, 5, 6, 7, 8]	3	[1, 2, 3, 4, 5, 6, 7, 8]

Tablo 2.12: [6, 5, 3, 1, 8, 7, 2, 4] dizisinde Shell sıralaması.

Aralık 4'te dört grup vardır ve her biri iki öğelidir. Yalnız [3, 2] grubu sırasızdır; tek kaydırmayla [2, 3] olur. Aralık 2'de iki grup vardır: çift indisler [6, 2, 8, 3] ve tek indisler [5, 1, 7, 4]. Birinci grup eklemeli sıralamayla [2, 3, 6, 8] olur (2 için 6 kayar; 3 için 8 ve 6 kayar: 3 kaydırma). İkinci grup da 3 kaydırmayla [1, 4, 5, 7] olur. Gruplar dizideki yerlerine geri yazıldığında [2, 1, 3, 4, 6, 5, 8, 7] çıkar. Bu dizide

yalnız üç ters çift kalmıştır: (2, 1), (6, 5), (8, 7). Aralık 1 turu, yani sıradan eklemeli sıralama, bu yüzden yalnız 3 kaydırma yapar. Toplam 10 kaydırma gerekmiştir; aynı dizi doğrudan eklemeli sıralamayla sıralansaydı ters çift sayısı kadar, yani 16 kaydırma gerekirdi.



Şekil 2.11: (a) Aralık 4'te gruplar: aynı renkteki hücreler aynı gruptadır. (b) Aralık 2 turundan sonra dizi [2, 1, 3, 4, 6, 5, 8, 7]; çift ve tek indisli iki grup kendi içinde sıralıdır.

2.5.3 Kod izleme: [5, 2, 6, 1, 4, 3]

Kodun altı öğeli bir dizideki çalışması Tablo 2.13'te görülür. *n = 6* olduğu için ilk aralık 3'tür; gruplar (0, 3), (1, 4) ve (2, 5) indis çiftleridir. Sonraki aralık *3/2 = 1*'dir.

`aralik`	`i`	`gecici`	**İç döngü**	`dizi`
3	3	1	`dizi[0] = 5 > 1` : 5 indis 3'e kayar; `j = 0` 'da durulur	[1, 2, 6, 5, 4, 3]
3	4	4	`dizi[1] = 2 ≤ 4` : kaydırma yok	[1, 2, 6, 5, 4, 3]
3	5	3	`dizi[2] = 6 > 3` : 6 indis 5'e kayar; `j = 2` 'de durulur	[1, 2, 3, 5, 4, 6]
1	1	2	`dizi[0] = 1 ≤ 2` : kaydırma yok	[1, 2, 3, 5, 4, 6]
1	2	3	`dizi[1] = 2 ≤ 3` : kaydırma yok	[1, 2, 3, 5, 4, 6]
1	3	5	`dizi[2] = 3 ≤ 5` : kaydırma yok	[1, 2, 3, 5, 4, 6]
1	4	4	`dizi[3] = 5 > 4` : 5 indis 4'e kayar; `dizi[2] = 3 ≤ 4` 'te durulur	[1, 2, 3, 4, 5, 6]
1	5	6	`dizi[4] = 5 ≤ 6` : kaydırma yok	[1, 2, 3, 4, 5, 6]

Tablo 2.13: `dizi = [5, 2, 6, 1, 4, 3]` için Shell sıralaması kodunun izlemesi.

Aralık 3 turunda 1, tek hamlede indis 3'ten indis 0'a atlar; eklemeli sıralama bunun için üç kaydırma yapardı. Aynı biçimde 3, indis 5'ten indis 2'ye geçer. Bu turdan sonra dizi [1, 2, 3, 5, 4, 6] olur ve yalnız bir ters çift, (5, 4), kalır. Aralık 1 turu bunu tek kaydırmayla düzeltir. Aralık *1/2 = 0* olunca dış döngü biter. Toplam yalnız 3 kaydırma yapılmıştır. Aynı dizinin ters çift sayısı 9'dur; doğrudan eklemeli sıralama 9 kaydırma yapardı.

2.5.4 Aralık dizisi ve karmaşıklık

Shell sıralamasının süresi, kullanılan aralık dizisine bağlıdır ve tam analizi zordur. Shell'in özgün dizisi *n/2, n/4, ..., 1*'dir; bu notun kodu da onu kullanır. Bu dizinin en kötü durumu $\Theta(n^2)$'dir. Kötü durumun nedeni, *n* ikinin kuvveti olduğunda son tura kadar bütün aralıkların çift olmasıdır. Çift indisler ile tek indisler son tura kadar hiç karşılaştırılmaz. Örneğin büyük öğelerin hepsi çift, küçüklerin hepsi tek indislerdeyse ilk turlar onları karıştıramaz ve son tur karesel iş yapar.

Daha iyi aralık dizileri de vardır. T. N. Hibbard'ın $2^k - 1$ dizisi (1, 3, 7, 15, ...) ardışık aralıkları ortak çarpansız tutar ve en kötü durumu $O(n^{1,5})$'e indirir. Başka diziler bunu daha da iyileştirir; pratikte orta büyüklükteki dizilerde Shell sıralaması oldukça hızlıdır. En iyi durumda, yani sıralı girdide, her tur yaklaşık *n* karşılaştırma yapar ve *log n* tur vardır: süre $\Theta(n \log n)$'dir. Shell sıralamasının ortalama süresi için genel bir kapalı formül bilinmez; bu nedenle karşılaştırma tablolarında “aralığa bağlı” diye yazılır.

> **Sık yapılan hata**
>
> Shell sıralamasını, her grup eklemeli sıralamayla sıralandığı için kararlı sanmak. Eklemeli sıralama tek bir grubun içinde kararlıdır, ama eşit iki öğe farklı gruplarda olabilir. Örneğin [5, 2_a, 2_b, 1] dizisinde aralık 2'de çift indis grubu [5, 2_b] sıralanır ve 2_b başa geçer; 2_a tek indis grubunda kalır. Sonuçta 2_b, 2_a'nın önüne düşer.

2.6 Birleştirme sıralaması

Buraya kadar incelenen algoritmaların hepsi en kötü durumda karesel süreliydi. Birleştirme sıralaması bu sınırı aşan ilk algoritmadır: her girdide $\Theta(n \log n)$ sürede çalışır. Bunu, problemi küçük parçalara bölen ve parçaların çözümlerini birleştiren bir tasarım yöntemiyle başarır.

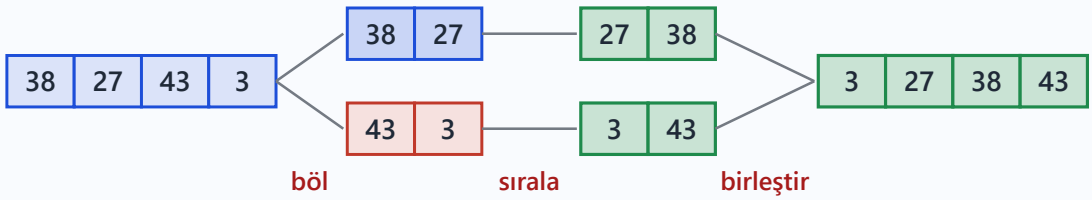
2.6.1 Böl ve fethet

Böl ve fethet (*divide and conquer*), bir problemi aynı türden daha küçük alt problemlere bölen, alt problemleri özyinelemeli olarak çözen ve sonuçları birleştiren bir algoritma tasarım yöntemidir. Her böl ve fethet algoritmasında üç adım vardır.

1. **Böl** (*divide*): Problem daha küçük alt problemlere ayrılır.
2. **Fethet** (*conquer*): Alt problemler aynı yöntemle, özyinelemeli olarak çözülür. Alt problem yeterince küçükse doğrudan çözülür; bu, özyinelemenin **taban durumudur**.
3. **Birleştir** (*combine*): Alt problemlerin çözümleri, asıl problemin çözümüne dönüştürülür.

Birleştirme sıralaması (*merge sort*) bu yöntemi sıralamaya uygular. Dizi ortadan iki yarıya bölünür. Her yarı aynı yöntemle sıralanır. Sonra sıralı iki yarı, sıralı tek bir dizide birleştirilir. Taban durumu tek öğeli dizidir; tek öğeli bir dizi tanım gereği sıralıdır. Algoritma 1945'te John von Neumann'a atfe-

dilir.

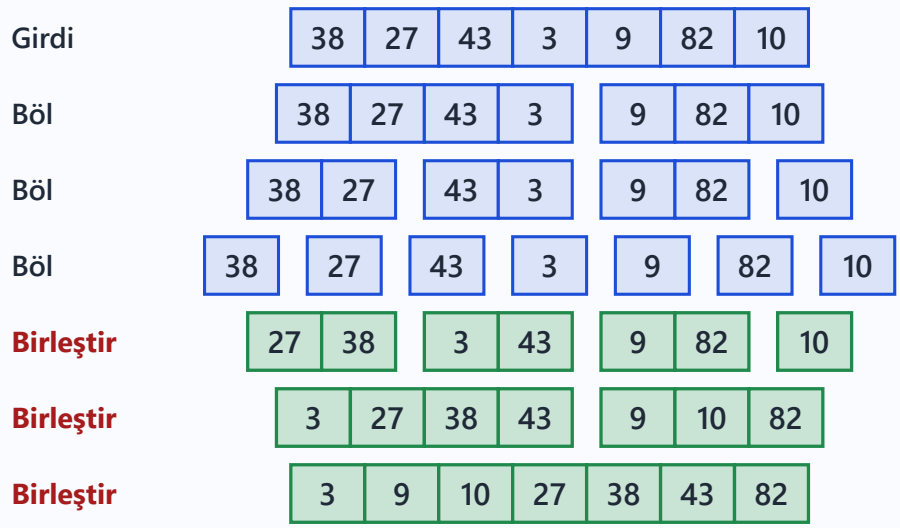


Şekil 2.12: Böl ve fethetin üç adımı [38, 27, 43, 3] üzerinde. Dizi [38, 27] ve [43, 3] yarılarına bölünür; yarılar ayrı ayrı sıralanarak [27, 38] ve [3, 43] olur; birleştirme [3, 27, 38, 43] dizisini verir.

Birleştirme sıralamasında bölme adımı neredeyse bedavadır: yalnız orta indis hesaplanır. Asıl iş birleştirme adımında yapılır. İki sıralı diziyi birleştirmenin fikri basittir. İki dizinin de en küçük öğesi kendi başındadır. Bu iki baş karşılaştırılır, küçük olan çıktıya yazılır ve o dizide bir sonraki öğeye geçilir. Dizilerden biri bitince öbüründe kalanlar sırayla çıktıya kopyalanır. Her karşılaştırma çıktıya bir öğe yazdırdığı için toplam m öğenin birleştirilmesi en fazla $m - 1$ karşılaştırma ve m yazma ister: süre $\Theta(m)$'dir.

2.6.2 Örnek: [38, 27, 43, 3, 9, 82, 10]

Yedi öğeli bir diziyi ele alalım. Orta indis $\lfloor (sol + sağ)/2 \rfloor$ ile hesaplandığı için öğe sayısı tek olan parçalarda sol yarı bir öğe fazla alır: [38, 27, 43, 3] ve [9, 82, 10]. Şekil 2.13 bölmeleri ve birleştirmeleri birlikte gösterir.



Şekil 2.13: Birleştirme sıralaması örneği. Üst yarıda dizi tek öğeli parçalara kadar bölünür. Alt yarıda parçalar ikişer ikişer birleştirilerek önce [27, 38], [3, 43], [9, 82], [10]; sonra [3, 27, 38, 43] ve [9, 10, 82]; en sonda [3, 9, 10, 27, 38, 43, 82] elde edilir.

Kod derinlik öncelikli çalıştığı için birleştirmeler şekildeki düzey sırasıyla değil, aşağıdaki sırayla yapılır. Sol yarı tamamen sıralanmadan sağ yarıya geçilmez.

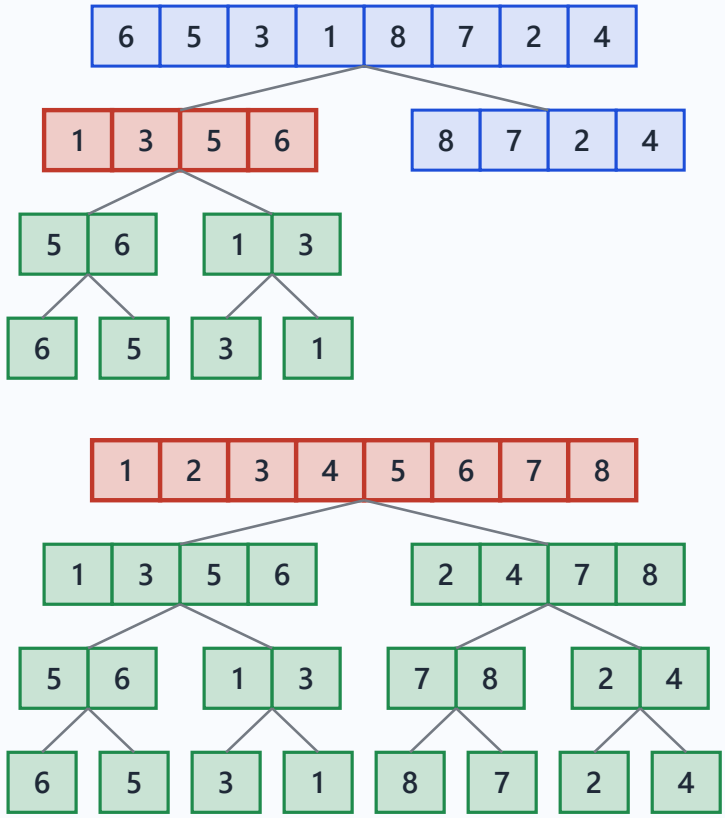
Sıra	Aralık	Sol + sağ	Sonuç	Karşılaştırma
1	[0..1]	[38] + [27]	[27, 38]	1
2	[2..3]	[43] + [3]	[3, 43]	1
3	[0..3]	[27, 38] + [3, 43]	[3, 27, 38, 43]	3
4	[4..5]	[9] + [82]	[9, 82]	1
5	[4..6]	[9, 82] + [10]	[9, 10, 82]	2
6	[0..6]	[3, 27, 38, 43] + [9, 10, 82]	[3, 9, 10, 27, 38, 43, 82]	6

Tablo 2.14: `birlestirmeliSirala(dizi, 0, 6)` çağrısının `birlestir` çağrıları, gerçekleşme sırasıyla.

`birlestir` toplam 6 kez çağrılır. Bu sayı her zaman $n - 1$'dir. Her birleştirme iki parçayı bire indirir; bu yüzden n tek öğeli parçadan tek bir diziye ulaşmak için $n - 1$ birleştirme gerekir. Karşılaştırma toplamı $1 + 1 + 3 + 1 + 2 + 6 = 14$'tür.

2.6.3 Özyinelemenin sırası

Özyinelemeli çağrıların hangi sırayla yürüdüğünü görmek için sekiz öğeli [6, 5, 3, 1, 8, 7, 2, 4] dizisini izleyelim. Dizi önce [6, 5, 3, 1] ve [8, 7, 2, 4] olarak bölünür. Sol yarı [6, 5] ve [3, 1]'e, [6, 5] de [6] ve [5]'e bölünür. İlk birleştirme burada olur: [6] + [5] → [5, 6]. Sonra [3, 1] bölünür ve [1, 3] olarak birleştirilir. [5, 6] ile [1, 3] birleşince sol yarı [1, 3, 5, 6] olarak sıralanmış olur. Ancak bundan sonra sağ yarıya geçilir: [8, 7] → [7, 8], [2, 4] → [2, 4], ikisi birleşince [2, 4, 7, 8]. Son birleştirme [1, 3, 5, 6] + [2, 4, 7, 8] → [1, 2, 3, 4, 5, 6, 7, 8]'dir.



Şekil 2.14: Özyineleme ağacı. (a) Sol yarı tamamlandı: [5, 6] ve [1, 3] birleşip [1, 3, 5, 6] oldu; sağ yarıya ([8, 7, 2, 4]) henüz dokunulmadı. (b) Bütün birleştirmeler bitti; kök [1, 2, 3, 4, 5, 6, 7, 8]'dir. Yeşil kutular sıralanmış parçalardır.

Bu sıra, özyinelemeli çağrıların bir yığında (çağrı yığınında) tutulmasının doğal sonucudur. Herhangi bir anda yığında en çok $\lfloor \log_2 n \rfloor + 1$ çağrı bulunur, çünkü her çağrı parça boyunu yarıya indirir. Sekiz öğe için yığın derinliği en çok 4'tür: boyu 8, 4, 2 ve 1 olan dört parça.

2.6.4 Java kodu

Kod iki metottan oluşur. `birlestirmeliSirala`, `dizi[sol..sag]` aralığını sıralar; bütün dizi için ilk çağrı `birlestirmeliSirala(dizi, 0, n - 1)`'dir.

```
void birlestirmeliSirala(int[] dizi, int sol, int sag) {
  if (sol < sag) {
    int orta = sol + (sag - sol) / 2;
    birlestirmeliSirala(dizi, sol, orta);
    birlestirmeliSirala(dizi, orta + 1, sag);
    birlestir(dizi, sol, orta, sag);
  }
}
```

- `sol < sag` değilse aralıkta en çok bir öğe vardır ve metot hiçbir şey yapmadan döner; bu taban durumudur.
- Orta indis `sol + (sag - sol) / 2` ile hesaplanır. Matematikte bu `(sol + sag) / 2` ile aynıdır. Ama Java'da `sol + sag` toplamı çok büyük dizilerde `int` sınırını ($2^{31} - 1$) aşıp negatif bir sayıya dönüşebilir. İlk yazım bu taşmayı önler.
- Sol yarı `[sol..orta]`, sağ yarı `[orta+1..sag]` özyinelemeli olarak sıralanır; sonra `birlestir` çağrılır.

`birlestir`, sıralı `A[p..q]` ve `A[q+1..r]` parçalarını `A[p..r]`'de birleştirir.

```
void birlestir(int[] A, int p, int q, int r) {
  int[] L = Arrays.copyOfRange(A, p, q + 1);
  int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
  int i = 0, j = 0, k = p;
  while (i < L.length && j < R.length) {
    if (L[i] <= R[j]) A[k++] = L[i++];
    else              A[k++] = R[j++];
  }
  while (i < L.length) A[k++] = L[i++];
  while (j < R.length) A[k++] = R[j++];
}
```

- İki yarı önce `Arrays.copyOfRange` ile `L` ve `R` yardımcı dizilerine kopyalanır. Kopyalama gereklidir, çünkü sonuç doğrudan `A`'nın aynı hücrelerine yazılacaktır ve kopya olmasa henüz okunmamış öğelerin üzerine yazılırdı. `copyOfRange`'in üst sınırı dışlayıcıdır; bu yüzden `q + 1` ve `r + 1` yazılır.
- `i` ve `j` sırasıyla `L` ile `R`'nin sıradaki öğesini, `k` ise `A`'da yazılacak hücreyi gösterir.
- Birinci `while`, iki dizide de öğe kaldıkça başları karşılaştırır; küçük ya da eşit olanı `A[k]`'ya yazar ve ilgili indisi ilerletir. `A[k++] = L[i++]` ifadesi önce yazar, sonra iki indisi birer artırır.
- Dizilerden biri bitince kalan iki `while`'dan yalnız biri çalışır ve öbür dizide kalanları sırayla kopyalar. Kalan öğeler zaten sıralıdır ve yazılanların hepsinden büyük ya da onlara eşittir; karşılaştırmaya gerek yoktur.

Koşul `L[i] <= R[j]`'dir. Eşitlikte soldaki öğe önce alınır. Sol yarıdaki öğeler girdide sağ yarıdakilerden önce geldiği için eşit anahtarların göreli sırası korunur: birleştirme sıralaması **kararlıdır**.

> **Sık yapılan hata**
>
> Birleştirmede `<=` yerine `<` yazmak. Dizi yine doğru sıralanır, ama eşitlikte sağdaki öğe önce alınır. Eşit anahtarlı iki kaydın sırası tersine döner ve algoritma kararlılığını kaybeder. Bu hata sayı dizileriyle yapılan testlerde görünmez; ancak kayıtlar sıralanırken ortaya çıkar.

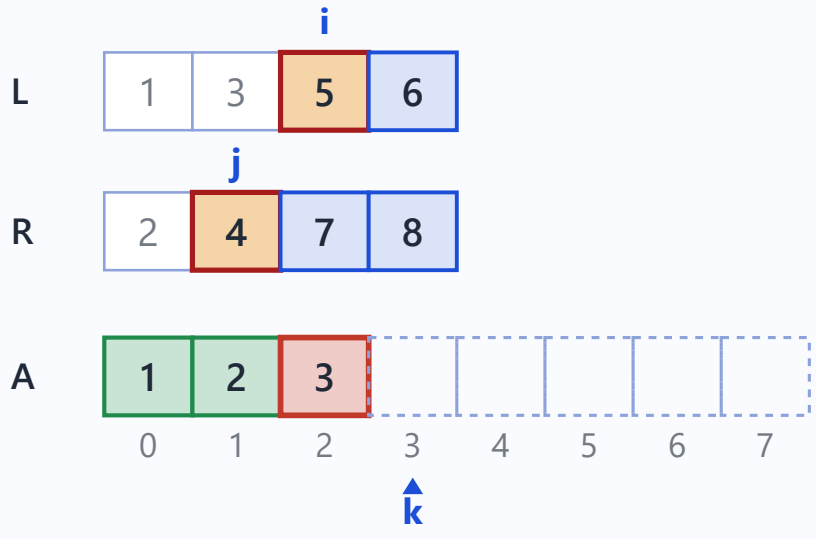
2.6.5 Kod izleme: birlestir()

[6, 5, 3, 1, 8, 7, 2, 4] örneğinin son adımında `birlestir(A, 0, 3, 7)` çağrılır. Bu anda `A[0..3] = [1, 3, 5, 6]` ve `A[4..7] = [2, 4, 7, 8]` 'dir.

Adım	`i, j, k`	Karşılaştırma	Yazılan	`A[0..k]`
1	0, 0, 0	`L[0] = 1 ≤ R[0] = 2`	`A[0] = 1`	[1]
2	1, 0, 1	`L[1] = 3 > R[0] = 2`	`A[1] = 2`	[1, 2]
3	1, 1, 2	`L[1] = 3 ≤ R[1] = 4`	`A[2] = 3`	[1, 2, 3]
4	2, 1, 3	`L[2] = 5 > R[1] = 4`	`A[3] = 4`	[1, 2, 3, 4]
5	2, 2, 4	`L[2] = 5 ≤ R[2] = 7`	`A[4] = 5`	[1, …, 5]
6	3, 2, 5	`L[3] = 6 ≤ R[2] = 7`	`A[5] = 6`	[1, …, 6]
7	4, 2, 6	`L` bitti; kalan `R` kopyalanır	`A[6] = 7`	[1, …, 7]
8	4, 3, 7	—	`A[7] = 8`	[1, …, 8]

Tablo 2.15: `L = [1, 3, 5, 6]`, `R = [2, 4, 7, 8]` için `birlestir` izlemesi.

İlk altı adımda birer karşılaştırma yapılır ve sonuç dönüşümlü olarak iki diziden gelir. Altıncı adımdan sonra `i = 4` 'tür; `L` bitmiştir ve birinci `while` sona erer. Kalan 7 ve 8 karşılaştırma yapılmadan kopyalanır. Toplam 6 karşılaştırma ve 8 yazma yapılmıştır. Sekiz öğeli bir birleştirme en fazla 7 karşılaştırma ister; bu sınır, iki dizi son öğelerine kadar birlikte tükendiğinde dolar. Örneğin `L = [1, 3, 5, 7]`, `R = [2, 4, 6, 8]` için 7 karşılaştırma gerekir.



Şekil 2.15: Üçüncü adımdan sonraki durum. `L` 'den 1 ve 3, `R` 'den 2 alındı (soluk hücreler). Sıradaki karşılaştırma `L[2] = 5` ile `R[1] = 4` arasındadır; `k = 3`.

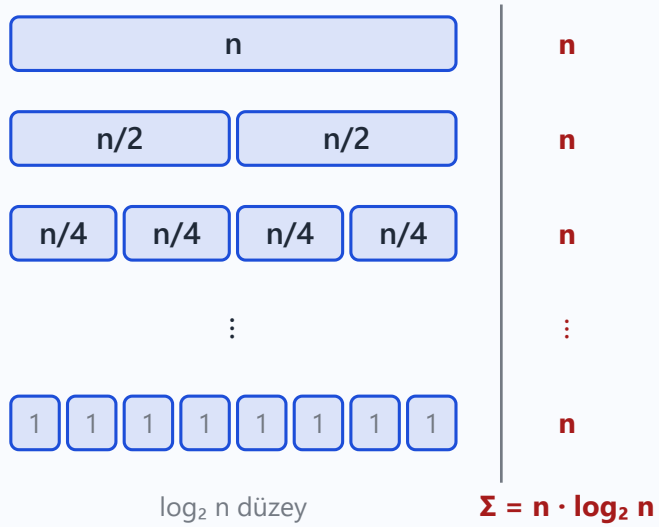
2.6.6 Analiz

Birleştirme sıralamasının süresi bir **yineleme bağıntısıyla** (*recurrence*) ifade edilir. n öğeli diziyi sıralamanın süresi $T(n)$ olsun. Algoritma boyu $n/2$ olan iki alt problemi çözer ve sonuçları $\Theta(n)$ sürede birleştirir. Bölme sabit süredir. Buna göre

> $T(n) = 2\,T(n/2) + \Theta(n),\ T(1) = \Theta(1)$

bağıntısı yazılır.

Bu bağıntıyı çözmenin en sezgisel yolu **özyineleme ağacıdır**. Kökte boyu n olan bir problem vardır ve birleştirme işi n'dir. Bir alt düzeyde boyu $n/2$ olan iki problem vardır; her biri $n/2$ iş yapar, düzeyin toplamı yine n'dir. Bir alt düzeyde dört tane $n/4$ vardır; toplam yine n. Her düzeyde parça sayısı ikiye katlanır, parça boyu yarıya iner, toplam iş değişmez. Parça boyu 1'e inene kadar $\log_2 n$ düzey vardır. Toplam iş düzey başına n çarpı düzey sayısı, yani $n \log_2 n$'dir.



Şekil 2.16: Birleştirme sıralamasının özyineleme ağacı. Her düzeyde toplam birleştirme işi n'dir; $\log_2 n$ düzey vardır ve toplam $n \cdot \log_2 n$ olur.

Aynı sonuç **ana teoremle** (*master theorem*) de bulunur. Bağıntı $T(n) = aT(n/b) + f(n)$ biçimindedir; burada $a = 2$, $b = 2$, $f(n) = \Theta(n)$'dir. $n^{\log_b a} = n^{\log_2 2} = n$ olur ve $f(n) = \Theta(n^{\log_b a})$ eşitliği teoremin ikinci durumudur. Bu durumda $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$'dir.

Birleştirme sıralaması girdinin düzenine bakmaz; her durumda diziyi aynı biçimde böler ve aynı birleştirmeleri yapar. Bu yüzden en iyi, ortalama ve en kötü durum aynıdır: $\Theta(n \log n)$. Karşılaştırma sayısı girdiye göre biraz değişir, çünkü bir dizi erken biterse kalanlar karşılaştırmasız kopyalanır; ama bu değişim büyüme hızını etkilemez.

Ek bellek açısından birleştirme sıralaması yerinde değildir. Birleştirme anında `L` ve `R` toplam n öğe tutar; ek bellek $\Theta(n)$'dir. Buna özyineleme yığınının $\Theta(\log n)$ çerçevesi eklenir. Büyük veri setlerinde bu ek bellek önemli bir maliyettir ve hızlı sıralamanın pratikte tercih edilmesinin nedenlerinden biridir.

Örnek 2.4

Birleştirme sıralaması 8 öğeli bir diziyi en kötü durumda en fazla kaç karşılaştırmayla sıralar?

1. **Düzeyler.** 8 öğe için özyineleme ağacında üç birleştirme düzeyi vardır: dört tane 2 öğeli, iki tane 4 öğeli ve bir tane 8 öğeli birleştirme.
2. **Tek birleştirme.** m öğeli birleştirme en fazla $m - 1$ karşılaştırma ister: her karşılaştırma bir öğe yazar ve son öğe karşılaştırmasız yazılır.
3. **Toplam.** $4 \cdot 1 + 2 \cdot 3 + 1 \cdot 7 = 4 + 6 + 7 = 17$.

Sonuç 17'dir. Bütün $8! = 40.320$ permütasyon üzerinde kod çalıştırıldığında en çok karşılaştırma sayısı gerçekten 17 çıkar. Kaba üst tahmin $n \log_2 n = 24$'tür; basit algoritmaların $n(n-1)/2 = 28$ karşılaştırmasıyla karşılaştırıldığında fark küçük n'de bile görülür.

Birleştirme sıralamasının bir yan ürünü de **ters çift sayımıdır**. Birleştirme sırasında `R`'den bir öğe alındığında, `L`'de kalan bütün öğeler ondan büyüktür ve her biri onunla bir ters çift oluşturur. Sayaca `L.length - i` eklenir. Sol ve sağ yarının kendi içindeki ters çiftler özyinelemeyle sayılır. Toplam iş birleştirme sıralamasıyla aynıdır: $\Theta(n \log n)$. Bütün çiftleri tek tek denetleyen kaba kuvvet yöntemi ise $\Theta(n^2)$ sürer. Örneğin [3, 1, 2, 5, 4] dizisinde ters çiftler (3, 1), (3, 2) ve (5, 4)'tür; sayıları 3'tür.

2.7 Hızlı sıralama

Hızlı sıralama da böl ve fethet yöntemini kullanır, ama işin yerini değiştirir. Birleştirme sıralamasında bölme kolay, birleştirme zordur. Hızlı sıralamada ise bölme zordur ve birleştirmeye hiç gerek kalmaz. Ortalama durumda $\Theta(n \log n)$'dir, yerinde çalışır ve önbellek dostudur; bu yüzden pratikte en hızlı karşılaştırmalı sıralama genellikle odur.

2.7.1 Fikir

Hızlı sıralama (*quick sort*) C. A. R. Hoare'un 1959'da geliştirdiği ve 1961–1962'de yayımladığı bir algoritmadır. Algoritma üç adımdan oluşur:

1. Diziden bir **pivot** öğe seçilir. Pivot x olsun.
2. Dizi pivot etrafında iki parçaya ayrılır: solda x'ten küçük ya da eşit öğeler, sağda x'ten büyük ya da eşit öğeler. Pivot ikisinin arasına yerleşir. Bu işleme **bölümleme** (*partitioning*) denir.
3. İki parça özyinelemeli olarak sıralanır.



Şekil 2.17: Bölümlemeden sonra dizi: solda pivottan küçük ya da eşit öğeler, ortada pivot *x*, sağda büyük ya da eşit öğeler. Sol ve sağ parça ayrı ayrı, özyinelemeli olarak sıralanır.

Bölümlemenin önemli bir sonucu vardır: bölümlemeden sonra pivot kesin yerindedir. Solundaki bütün öğeler ondan küçük ya da eşit, sağındakiler büyük ya da eşittir; sıralı dizide de tam o indiste duracaktır. İki parça sıralandığında öğeler parçaların sınırını hiç geçmez. Bu yüzden iki sıralı parçayı birleştirmek için ek bir iş gerekmez: dizi yerinde sıralanmış olur.

2.7.2 Bölümleme prosedürü

Bölümlemenin birkaç yolu vardır. Bu notta N. Lomuto'nun adıyla anılan yöntem kullanılır. Pivot olarak aralığın son öğesi seçilir ve dizi tek geçişte taranır.

```
parcala(A, p, q):
  pivot = A[q]        // pivot seç
  i = p - 1           // ≤ pivot bölgesinin sonu
  for j = p to q - 1:
    if A[j] <= pivot:
      i = i + 1
      takas(A[i], A[j])
  takas(A[i + 1], A[q])  // pivotu yerleştir
  return i + 1          // pivotun indisi
```

Prosedür iki indisle çalışır. `i`, "pivottan küçük ya da eşit" bölgenin son hücresidir; başta bu bölge boş olduğu için `i = p - 1`'dir. `j` ise sıradaki incelenecek öğedir. Her adımda `A[j]` pivotla karşılaştırılır. Büyükse yalnız `j` ilerler; öğe "büyükler" bölgesine katılmış olur. Küçük ya da eşitse `i` bir artırılır ve `A[i]` ile `A[j]` takas edilir. Böylece öğe küçükler bölgesinin sonuna alınır; büyükler bölgesinin ilk öğesi de `j`'nin yerine geçer. Tarama bitince pivot, büyükler bölgesinin ilk hücresi olan `A[i + 1]` ile takas edilir ve onun indisi döndürülür.

Prosedürün doğruluğu şu döngü değişmezine dayanır: her adımın başında

- `A[p..i]`'deki öğeler pivottan küçük ya da eşittir,
- `A[i+1..j-1]`'deki öğeler pivottan büyüktür,
- `A[j..q-1]` henüz incelenmemiştir ve `A[q]` pivottur.

Başlangıçta ilk iki bölge boştur ve değişmez kendiliğinden doğrudur. Her adım, incelenen öğeyi doğru bölgeye ekleyerek değişmezi korur. Döngü bittiğinde incelenmemiş bölge boştur ve pivotun `A[i+1]`'e alınması diziyi istenen biçime getirir. Tarama tek geçiştir: *n* öğelik aralık için *n − 1* karşılaştırma yapı-

lır, süre $\Theta(n)$, ek bellek $O(1)$'dir.

Aynı fikir pivot ilk öğe seçilerek de uygulanabilir. Bu sürümde pivot `A[p]`'de kalır, tarama `p + 1`'den başlar ve sonda pivot küçükler bölgesinin son öğesiyle yer değiştirir.

```
PARCALA(A, p, q):
  x = A[p]            // pivot
  i = p
  for j = p + 1 to q:
    if A[j] <= x:
      i = i + 1
      takas(A[i], A[j])
  takas(A[p], A[i])
  return i
```

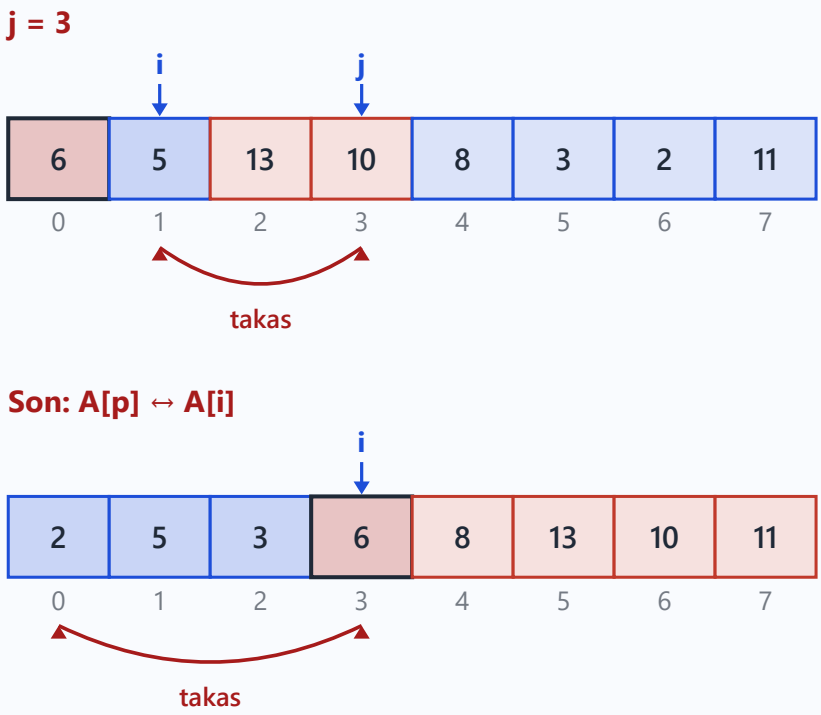
2.7.3 Bölümleme örneği: pivot ilk öğe

İlk öğeyi pivot alan sürümü [6, 10, 13, 5, 8, 3, 2, 11] dizisinde izleyelim. Pivot $x = 6$'dır; `i = p = 0`'dır ve `j` 1'den başlar.

`j`	`A[j]`	`A[j] ≤ 6` ?	`i`	**Dizi**
1	10	hayır	0	[6, 10, 13, 5, 8, 3, 2, 11]
2	13	hayır	0	[6, 10, 13, 5, 8, 3, 2, 11]
3	5	evet: `A[1] ↔ A[3]`	1	[6, 5, 13, 10, 8, 3, 2, 11]
4	8	hayır	1	[6, 5, 13, 10, 8, 3, 2, 11]
5	3	evet: `A[2] ↔ A[5]`	2	[6, 5, 3, 10, 8, 13, 2, 11]
6	2	evet: `A[3] ↔ A[6]`	3	[6, 5, 3, 2, 8, 13, 10, 11]
7	11	hayır	3	[6, 5, 3, 2, 8, 13, 10, 11]
son	—	`A[0] ↔ A[3]`	3	[2, 5, 3, **6**, 8, 13, 10, 11]

Tablo 2.16: `PARCALA` (pivot ilk öğe) ile [6, 10, 13, 5, 8, 3, 2, 11] dizisinin bölümlenmesi.

Tarama boyunca dizinin başı üç bölgeye ayrılır: pivot, küçükler bölgesi ve büyükler bölgesi. İlk iki öğe (10 ve 13) büyüktür ve büyükler bölgesine katılır. 5 geldiğinde küçükler bölgesi bir büyür ve 5, büyükler bölgesinin ilk öğesi olan 10 ile yer değiştirir. 3 ve 2 için aynı şey olur. Tarama bittiğinde `i = 3`'tür. Pivot, küçükler bölgesinin son öğesi olan 2 ile yer değiştirir. Sonuçta pivot 6 indis 3'e oturur; solundaki 2, 5, 3 ondan küçük, sağındaki 8, 13, 10, 11 ondan büyüktür. Parçaların kendi içinde sıralı olmadığına dikkat edilmelidir; bölümleme yalnız pivotu yerine koyar.



Şekil 2.18: (a) `j = 3` : 5 ≤ 6 olduğu için `i = 1` olur ve `A[1]` ile `A[3]` takas edilir. Mavi: ≤ pivot, kırmızı: > pivot, beyaz: henüz bakılmadı. (b) Son takas: pivot 6, küçükler bölgesinin son öğesiyle yer değiştirip indis 3'e oturur.

2.7.4 Kod izleme: parcala()

Lomuto bölümlemesinin Java karşılığı aşağıdadır. `takas(dizi, a, b)`, iki hücrenin değerini üç atamayla değiştiren yardımcı metottur.

```
int parcala(int[] dizi, int alt, int ust) {
  int pivot = dizi[ust];
  int i = alt - 1;
  for (int j = alt; j < ust; j++) {
    if (dizi[j] <= pivot) {
      i++;
      takas(dizi, i, j);
    }
  }
  takas(dizi, i + 1, ust);
  return i + 1;
}
```

Metodu `parcala(dizi, 0, 7)` çağrısıyla [2, 8, 7, 1, 3, 5, 6, 4] dizisinde izleyelim. Pivot `dizi[7] = 4`, başlangıçta `i = -1`'dir.

`j`	`dizi[j] ≤ 4` ?	`i`	İşlem	`dizi`
0	2: evet	0	`takas(0, 0)` (kendisiyle)	[2, 8, 7, 1, 3, 5, 6, 4]
1	8: hayır	0	—	[2, 8, 7, 1, 3, 5, 6, 4]
2	7: hayır	0	—	[2, 8, 7, 1, 3, 5, 6, 4]
3	1: evet	1	`takas(1, 3)`	[2, 1, 7, 8, 3, 5, 6, 4]
4	3: evet	2	`takas(2, 4)`	[2, 1, 3, 8, 7, 5, 6, 4]
5	5: hayır	2	—	[2, 1, 3, 8, 7, 5, 6, 4]
6	6: hayır	2	—	[2, 1, 3, 8, 7, 5, 6, 4]
son	—	2	`takas(3, 7)`, `return 3`	[2, 1, 3, **4**, 7, 5, 6, 8]

Tablo 2.17: `parcala(dizi, 0, 7)` izlemesi, `dizi = [2, 8, 7, 1, 3, 5, 6, 4]`.

İlk öğe 2, pivottan küçüktür; `i` 0 olur ve öğe kendisiyle takas edilir. Bu işlem zararsızdır ama gereksiz bir takastır. 8 ve 7 büyüktür ve yalnız `j` ilerler. 1 geldiğinde `i = 1` olur ve 1, büyükler bölgesinin ilk öğesi 8 ile yer değiştirir. 3 için de aynısı olur: 3 ile 7 yer değiştirir. 5 ve 6 büyüktür. Döngü `j = ust`'ta biter; pivot `i + 1 = 3` indisine alınır ve 3 döndürülür. Sol parça `[0..2]`'deki öğeler 4'ten küçük, sağ parça `[4..7]`'deki öğeler 4'ten büyüktür. Toplam 7 karşılaştırma yapılmıştır; 8 öğelik aralık için $n - 1 = 7$.

2.7.5 Özyinelemeli yapı

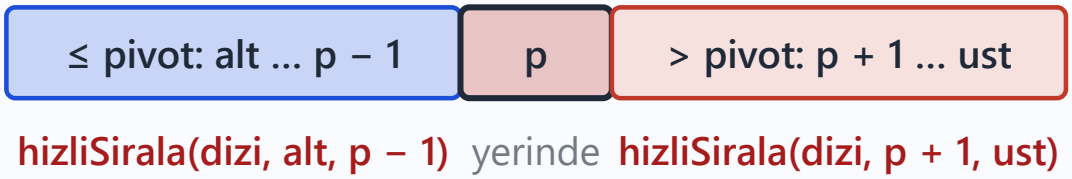
Bölümleme hazır olduğunda hızlı sıralamanın kendisi birkaç satırdır. Sözde kod 1 tabanlı indislerle yazılmıştır:

```
HIZLI-SIRALA(A, p, r)
  if p < r
    q = PARCALA(A, p, r)
    HIZLI-SIRALA(A, p, q - 1)
    HIZLI-SIRALA(A, q + 1, r)

İlk çağrı: HIZLI-SIRALA(A, 1, n)
```

Java sürümü 0 tabanlıdır ve bütün dizi için ilk çağrı `hizliSirala(dizi, 0, n - 1)`'dir.

```
void hizliSirala(int[] dizi, int alt, int ust) {
  if (alt < ust) {
    int p = parcala(dizi, alt, ust);
    hizliSirala(dizi, alt, p - 1);
    hizliSirala(dizi, p + 1, ust);
  }
}
```



Şekil 2.19: `parcala` çağrısından sonra pivot `p` indisinde, yerindedir. Sol parça `alt … p − 1` ve sağ parça `p + 1 … ust` iki ayrı özyinelemeli çağrıyla sıralanır.

Kodda iki noktaya dikkat etmek gerekir. Birincisi, pivot iki özyinelemeli çağrının hiçbirine dahil edilmez; çağrılar `p − 1`'de biter ve `p + 1`'den başlar. Pivot yerine oturmuştur ve ona bir daha dokunulmaz. İkincisi, taban durumu `alt < ust` koşulunun yanlış olmasıdır: aralıkta sıfır ya da bir öğe vardır ve böyle bir aralık zaten sıralıdır. Pivot parçanın en küçüğü ya da en büyüğü olduğunda boş bir parça oluşur; örneğin `hizliSirala(dizi, 0, -1)`. Bu çağrı hemen döner.

Sık yapılan hata

Pivotu özyinelemeye katmak, yani `hizliSirala(dizi, alt, p)` yazmak. Pivot parçanın en büyüğüyse sol parça yine bütün aralık olur ve hiç küçülmez. Örneğin iki öğeli sıralı bir aralıkta pivot en büyüktür, sol parça yine iki öğelidir ve aynı çağrı sonsuza kadar tekrarlanır. Sonuç `StackOverflowError`'dır.

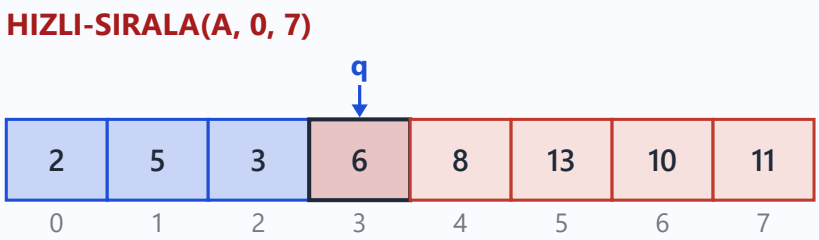
2.7.6 Tüm çalışma: [6, 10, 13, 5, 8, 3, 2, 11]

Tablo 2.16 yalnız ilk bölümlemeyi gösteriyordu. Bütün sıralamayı, yine pivot ilk öğe olacak biçimde, özyinelemenin derinliğiyle birlikte izleyelim.

Derinlik	Aralık	Parça	Pivot → indis	Sonraki çağrılar
0	[0..7]	[6, 10, 13, 5, 8, 3, 2, 11]	6 → 3	[0..2], [4..7]
1	[0..2]	[2, 5, 3]	2 → 0	[0..−1], [1..2]
2	[1..2]	[5, 3]	5 → 2	[1..1], [3..2]
3	[1..1]	[3]	taban	—
1	[4..7]	[8, 13, 10, 11]	8 → 4	[4..3], [5..7]
2	[5..7]	[13, 10, 11]	13 → 7	[5..6], [8..7]
3	[5..6]	[11, 10]	11 → 6	[5..5], [7..6]
4	[5..5]	[10]	taban	—

Tablo 2.18: [6, 10, 13, 5, 8, 3, 2, 11] dizisinin hızlı sıralaması (pivot ilk öğe). Satırlar çağrıların gerçekleşme sırasıdır.

Tabloda boş aralıklar (`[0..-1]` , `[3..2]` gibi) ayrı satır olarak yazılmamıştır; bu çağrılar hemen döner. Bütün çağrılar bittiğinde dizi [2, 3, 5, 6, 8, 10, 11, 13] olur. Birleştirme adımı yoktur: her pivot kendi bölümlemesinde yerine oturmuş, parçalar da yerinde sıralanmıştır. Sağ tarafta özyineleme dört düzey derinliğe iner, çünkü 8 ve 13 pivotları kendi parçalarının en küçüğü ve en büyüğüdür; her biri parçadan yalnız kendisini ayırır.



Şekil 2.20: Derinlik 0'daki bölümlemeden sonra dizi. Pivot 6 indis 3'e oturdu (`q`); mavi sol parça `[0..2]` ve kırmızı sağ parça `[4..7]` ayrı ayrı sıralanacak.

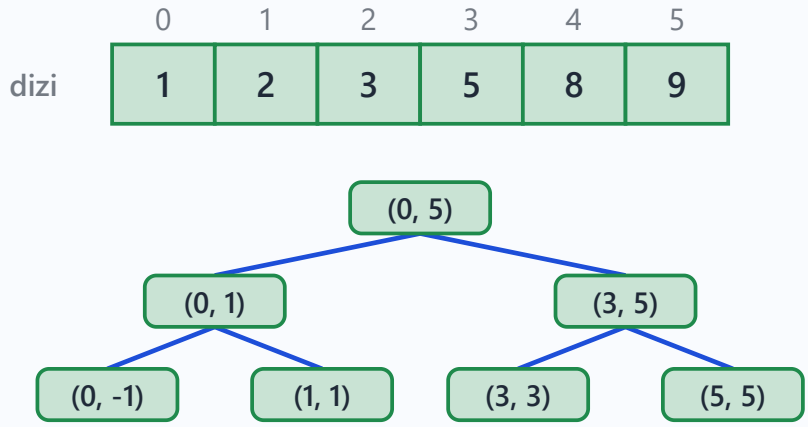
2.7.7 Çağrı izleme: [5, 2, 8, 1, 9, 3]

Java kodunun (pivot son öğe) altı öğeli bir dizideki çağrıları aşağıdaki tabloda sırayla verilmiştir. Her satır bir `hizliSirala` çağrısıdır.

Çağrı	Durum	Pivot → `p`	`dizi` (çağrıdan sonra)
`(0, 5)`	`alt < ust`	3 → 2	[2, 1, 3, 5, 9, 8]
`(0, 1)`	`alt < ust`	1 → 0	[1, 2, 3, 5, 9, 8]
`(0, -1)`	boş parça, taban	—	değişmez
`(1, 1)`	tek öğe, taban	—	değişmez
`(3, 5)`	`alt < ust`	8 → 4	[1, 2, 3, 5, 8, 9]
`(3, 3)`	tek öğe, taban	—	değişmez
`(5, 5)`	tek öğe, taban	—	değişmez

Tablo 2.19: `hizliSirala(dizi, 0, 5)` çağrılarının izlemesi, `dizi = [5, 2, 8, 1, 9, 3]` .

İlk çağrıda pivot 3'tür; 3'ten küçük 2 ve 1 sola toplanır, 5, 9, 8 sağda kalır ve pivot indis 2'ye oturur. Sonra sol parça `(0, 1)` sıralanır: pivot 1, solu boş, sağı [2]. Boş ve tek öğeli parçalar hemen döner. Sol taraf bitince ilk çağrının ikinci özyinelemeli satırı çalışır ve `(3, 5)` çağrılır: pivot 8, solunda [5], sağında [9]. Toplam 3 `parcala` çağrısı ve 4 taban çağrısı yapılmıştır; sonuç [1, 2, 3, 5, 8, 9]'dur.



Şekil 2.21: `hizliSirala` çağrı ağacı. Kök `(0, 5)`, iki çocuğu `(0, 1)` ve `(3, 5)`'tir; yapraklar boş ya da tek öğeli taban çağrılarıdır. Üstte sıralanmış dizi görülür.

2.7.8 Analiz ve pivot seçimi

Hızlı sıralamanın süresi, pivotun diziyi ne kadar dengeli böldüğüne bağlıdır. Her bölümleme, aralık boyuna göre doğrusal iş yapar. Bölmelerin dengesi ise özyineleme ağacının derinliğini belirler.

En iyi durum. Pivot her seferinde parçanın ortancasıysa dizi iki eşit yarıya bölünür. Bağıntı birleştirme sıralamasınınkiyle aynıdır: $T(n) = 2T(n/2) + \Theta(n)$, çözümü $\Theta(n \log n)$.

En kötü durum. Pivot her seferinde parçanın en küçüğü ya da en büyüğüyse bir parça boş, öbürü $n - 1$ öğelidir. Bağıntı $T(n) = T(n - 1) + \Theta(n)$ olur. Açılınca $n + (n - 1) + \ldots + 1$, yani $\Theta(n^2)$ çıkar. Bu durum şaşırtıcı biçimde en "kolay" girdide ortaya çıkar: zaten sıralı bir dizide son öğeyi pivot alan sürüm her seferinde en büyüğü seçer. 1000 öğeli sıralı dizide karşılaştırma sayısı $999 + 998 + \ldots + 1 = 499.500$'dür ve özyineleme 1000 düzey derinliğe iner. Çok büyük dizilerde bu derinlik çağrı yığınını taşırır ve `StackOverflowError` verir.

Ortalama durum. Rastgele bir girdide pivot çoğu zaman "yeterince ortada" düşer. Pivotun hep 1'e 9 oranında böldüğü kötü görünen bir durumda bile özyineleme ağacının derinliği $\log_{10/9} n$'dir, yani yine logaritmiktir; toplam iş $O(n \log n)$ kalır. Ayrıntılı analiz, beklenen karşılaştırma sayısının büyük n için yaklaşık $2n \ln n \approx 1{,}39\, n \log_2 n$ olduğunu gösterir. Küçük n'de alt dereceli terimler de etkilidir. Lomuto bölümlemesinin kesin beklenen değeri $2(n + 1)H_n - 4n$'dir (H_n, harmonik sayı) ve $n = 1000$ için yaklaşık 10.986 eder. Bu sayı birleştirme sıralamasının karşılaştırma sayısından yaklaşık %26 fazladır. Buna rağmen hızlı sıralama pratikte genellikle daha hızlıdır: iç döngüsü çok sadedir, yardımcı dizi kopyalamaz ve diziyi sırayla taradığı için işlemcinin önbelleğini iyi kullanır.

Durum	Ne zaman?	Bağıntı	Zaman
En iyi	pivot her seferinde ortanca	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$
Ortalama	rastgele girdi ya da rastgele pivot	beklenen $\approx 1{,}39\, n \log_2 n$ karşılaştırma	$\Theta(n \log n)$

Durum	Ne zaman?	Bağıntı	Zaman
En kötü	pivot hep en küçük ya da en büyük (ör. sıralı dizi ve son öğe pivot)	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$

Tablo 2.20: Hızlı sıralamanın durumları.

En kötü durumdan kaçınmak için pivot seçimi iyileştirilir.

- **Rastgele pivot.** Pivot aralıktan rastgele seçilir ve bölümlemeden önce son hücreyle takas edilir. Böylece hiçbir sabit girdi algoritmayı kötü duruma düşüremez. Beklenen süre her girdide $\Theta(n \log n)$'dir. Kötü durum hâlâ mümkündür, ama yalnız şanssız rastgele seçimlerle ve olasılığı çok küçüktür.
- **Üçün ortancası** (*median-of-three*). İlk, orta ve son öğenin ortancası pivot olur. Sıralı ve ters sıralı girdide bu seçim tam ortayı verir ve en kötü durumu önler.
- **Küçük parçalar.** Parça boyu küçülünce (kabaca $n \lesssim 16$) eklemeli sıralamaya geçmek pratikte hızlandırır. Ayrıca önce küçük parçaya inilip büyük parça döngüyle işlenirse özyineleme derinliği en kötü durumda bile $O(\log n)$'de kalır.

Hızlı sıralama yerindedir: yalnız özyineleme yığını için ek bellek kullanır, ortalamada $O(\log n)$. Kararlı değildir; bölümlemedeki uzun mesafeli takaslar eşit öğelerin sırasını bozabilir. Java'nın `Arrays.sort(int[])` metodu ilkel tür dizilerinde hızlı sıralamanın iki pivot kullanan bir sürümünü (*dual-pivot quicksort*) kullanır. İlkel türlerde kararlılığın önemi yoktur, çünkü eşit iki `int` ayırt edilemez.

> **Sık yapılan hata**
>
> "Hızlı sıralama her zaman $n \log n$'dir" demek. Sabit pivot kuralıyla (ilk ya da son öğe) sıralı ve ters sıralı girdiler $\Theta(n^2)$ süre ve n derinlikli özyineleme verir. Gerçek uygulamalarda veri çoğu zaman kısmen sıralı geldiği için bu durum kuramsal değildir; rastgele pivot ya da üçün ortancası kullanılmalıdır.

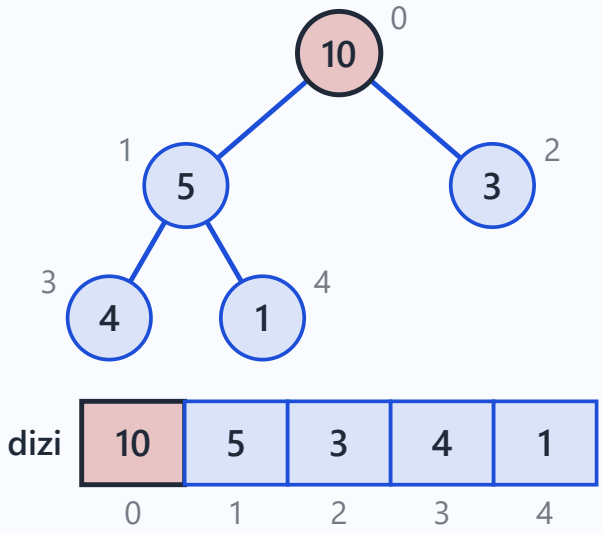
2.8 Heap sort

Birleştirme sıralaması en kötü durumda da $\Theta(n \log n)$'dir ama ek bellek ister. Hızlı sıralama yerindedir ama en kötü durumu karesedir. Heap sort ikisinin iyi yanını birleştirir: en kötü durumda $O(n \log n)$ sürer ve $O(1)$ ek bellekle yerinde çalışır. Bunu, dizinin kendisini bir veri yapısı olarak kullanarak başarır.

2.8.1 Heap ve dizi gösterimi

Heap, son düzeyi dışında her düzeyi dolu, son düzeyi soldan sağa doldurulmuş bir tam ikili ağaçtır ve bir sıralama koşulunu sağlar. **Max-heap**'te her düğüm çocuklarından büyük ya da onlara eşittir. Bu koşul ağacın her yerinde sağlandığı için en büyük öğe köktedir. Heap yapısı veri yapıları konusunda öncelik kuyruğu olarak tanınır; burada yalnız sıralamada nasıl kullanıldığı ele alınır.

Heap'in en kullanışlı özelliği, ağacın ayrı bir yapı olarak tutulmasına gerek olmamasıdır. Tam ikili ağaç bir dizide düzey düzey, soldan sağa saklanır. 0 tabanlı indislerle `dizi[i]`'nin sol çocuğu `2i + 1`, sağ çocuğu `2i + 2`, ebeveyni `(i - 1) / 2` (tam sayı bölmesi) indisindedir. Böylece işaretçi kullanmadan, yalnız indis aritmetiğiyle ağaçta gezinmek mümkündür. Heap sort tam bu yüzden yerindedir: sıralanacak dizi aynı zamanda heap'in kendisidir.



Şekil 2.22: Max-heap [10, 5, 3, 4, 1] ağaç ve dizi olarak. Kök 10'dur (indis 0); çocukları 5 (indis 1) ve 3 (indis 2), 5'in çocukları 4 (indis 3) ve 1 (indis 4)'dir. Düğümlerin yanındaki küçük sayılar dizi indisleridir.

Heap sort iki aşamada çalışır:

1. Dizi bir max-heap'e dönüştürülür.
2. Kök, yani en büyük öğe, heap'in son öğesiyle yer değiştirir. Bu öğe artık dizinin sonundaki sıralı bölgeye geçmiştir ve heap bir öğe küçülür. Köke gelen öğe heap koşulunu bozabileceği için aşağı doğru süzülür ve heap yeniden kurulur. Bu adım heap'te tek öğe kalana kadar tekrarlanır.

Her iki aşamanın temeli aynı işlemdir: bir düğümü **aşağı süzmek**. Düğüm çocuklarından birinden küçükse büyük çocuğuyla yer değiştirir ve aynı denetim yeni konumunda tekrarlanır. Düğüm her iki çocuğundan büyük ya da eşit olduğunda ya da bir yaprağa ulaştığında süzme biter. Bu işlemin ön koşulu, düğümün iki alt ağacının zaten birer heap olmasıdır.

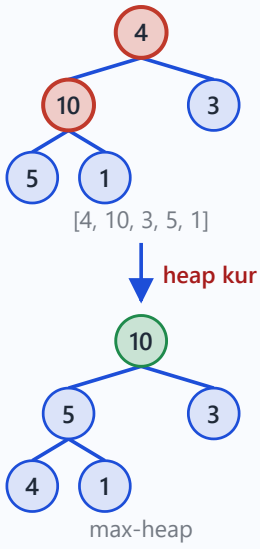
2.8.2 Örnek: [4, 10, 3, 5, 1]

Algoritmanın bütününü küçük bir dizide görelim. Tablo 2.21 her adımda dizinin heap kısmını ve sıralı kısmını ayrı ayrı gösterir.

Adım	Dizi (heap | sıralı)	Açıklama
Girdi	[4, 10, 3, 5, 1]	henüz heap değil
Max-heap kur	[10, 5, 3, 4, 1]	kök en büyük öğedir
Kök ↔ `dizi[4]`	[1, 5, 3, 4 | 10]	10 sıralı bölgeye geçti
Yeniden max-heap	[5, 4, 3, 1 | 10]	1 aşağı süzüldü (1 ↔ 5, 1 ↔ 4)
Kök ↔ `dizi[3]`	[1, 4, 3 | 5, 10]	5 sıralı bölgeye geçti
Yeniden max-heap	[4, 1, 3 | 5, 10]	1 aşağı süzüldü (1 ↔ 4)
Kök ↔ `dizi[2]`	[3, 1 | 4, 5, 10]	4 sıralı bölgeye geçti
Yeniden max-heap	[3, 1 | 4, 5, 10]	3 zaten büyük; değişiklik yok
Kök ↔ `dizi[1]`	[1 | 3, 4, 5, 10]	3 sıralı bölgeye geçti
Sonuç	[1, 3, 4, 5, 10]	tek öğeli heap de yerinde

Tablo 2.21: [4, 10, 3, 5, 1] dizisinin heap sort ile sıralanması. Dizi sütununda dikey çizgi heap ile sıralı bölgeyi ayırır.

Heap kurma adımında yaprak olmayan düğümler aşağıdan yukarıya işlenir. İndis 1'deki 10 çocuklarından (5 ve 1) büyüktür; değişiklik olmaz. Kökteki 4, büyük çocuğu 10 ile yer değiştirir. Yeni konumunda (indis 1) çocukları 5 ve 1'dir; 4, 5 ile yer değiştirir ve yaprağa iner. Sonuç [10, 5, 3, 4, 1]'dir. Bundan sonraki her turda heap'in en büyüğü sona taşınır; sıralı bölge sağdan sola, büyükten küçüğe dolar.



Şekil 2.23: [4, 10, 3, 5, 1] ağacının max-heap [10, 5, 3, 4, 1]'e dönüşmesi. Üstteki ağaçta kök 4 çocuğu 10'dan küçüktür; alttaki ağaçta her ebeveyn çocuklarından büyüktür.

2.8.3 Birinci aşama: max-heap kurma

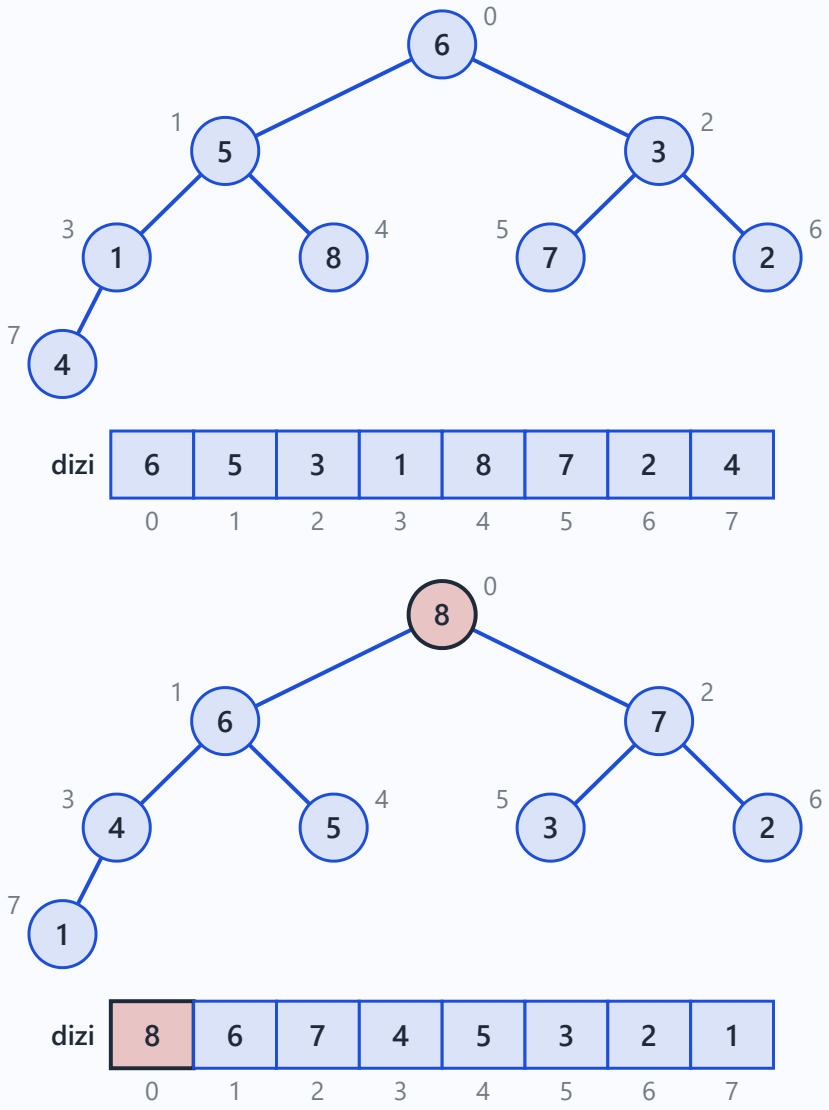
Sekiz öğeli [6, 5, 3, 1, 8, 7, 2, 4] dizisini ele alalım. Dizi tam ikili ağaç olarak düşünüldüğünde henüz heap değildir: örneğin indis 1'deki 5, çocuğu 8'den küçüktür.

Heap'i kurmanın verimli yolu, aşağıdan yukarı çalışmaktır. Yapraklar (indis $\lfloor n/2 \rfloor$ ve sonrası, burada 4–7) tek öğeli birer heap'tir; onlarla bir şey yapmak gerekmez. İşleme son iç düğümden, yani indis $n/2 - 1 = 3$'ten başlanır ve köke doğru ilerlenir. Her düğüm aşağı süzülür. Bir düğüm işlendiğinde onun iki alt ağacı daha önce işlenmiş, yani zaten heap'tir; süzmenin ön koşulu bu yüzden her zaman sağlanır.

Adım	Düğüm	Takaslar	Dizi
`yiginlastir(3)`	1 (çocuğu 4)	1 ↔ 4	[6, 5, 3, 4, 8, 7, 2, 1]
`yiginlastir(2)`	3 (çocukları 7, 2)	3 ↔ 7	[6, 5, 7, 4, 8, 3, 2, 1]
`yiginlastir(1)`	5 (çocukları 4, 8)	5 ↔ 8	[6, 8, 7, 4, 5, 3, 2, 1]
`yiginlastir(0)`	6 (çocukları 8, 7)	6 ↔ 8	[8, 6, 7, 4, 5, 3, 2, 1]

Tablo 2.22: [6, 5, 3, 1, 8, 7, 2, 4] dizisinde aşağıdan yukarı max-heap kurma.

Kökteki 6, büyük çocuğu 8 ile yer değiştirdikten sonra indis 1'e iner. Orada çocukları 4 ve 5'tir; 6 ikisinden de büyük olduğu için süzme durur. Bütün iç düğümler işlendiğinde dizi [8, 6, 7, 4, 5, 3, 2, 1] olur. Bu bir max-heap'tir ve en büyük öğe 8 köktedir. Dikkat edilirse heap sıralı bir dizi değildir; yalnız her ebeveynin çocuklarından büyük olduğunu garanti eder.



Şekil 2.24: (a) Başlangıç: [6, 5, 3, 1, 8, 7, 2, 4] tam ikili ağaç olarak; henüz heap değil. (b) Yaprak olmayan düğümler (indis 3, 2, 1, 0) işlendikten sonra max-heap [8, 6, 7, 4, 5, 3, 2, 1]; en büyük öğe kökte.

2.8.4 İkinci aşama: kökü sona taşıma

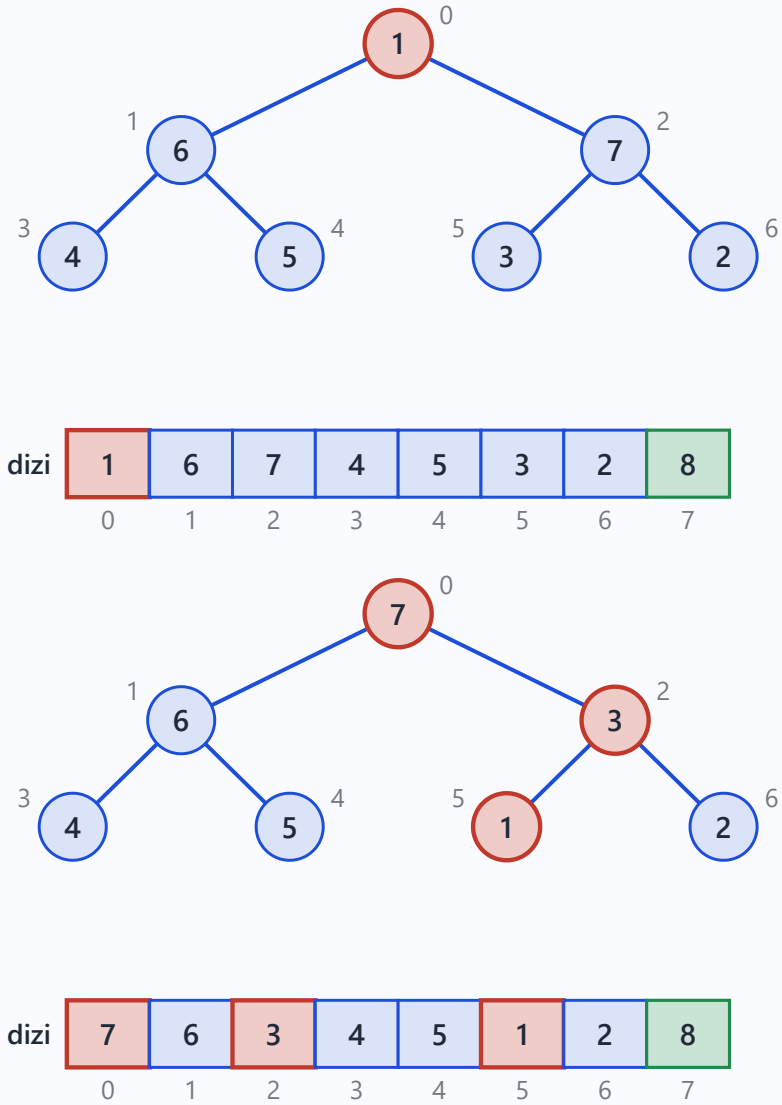
Max-heap [8, 6, 7, 4, 5, 3, 2, 1] hazırdır. Her turda kök, heap'in son öğesiyle yer değiştirir; heap bir küçülür ve köke gelen öğe aşağı süzülür.

Tur	Kök ↔ son	Süzme	Heap	Sıralı bölge
1	8 ↔ 1	1 ↔ 7, 1 ↔ 3	[7, 6, 3, 4, 5, 1, 2]	[8]
2	7 ↔ 2	2 ↔ 6, 2 ↔ 5	[6, 5, 3, 4, 2, 1]	[7, 8]
3	6 ↔ 1	1 ↔ 5, 1 ↔ 4	[5, 4, 3, 1, 2]	[6, 7, 8]
4	5 ↔ 2	2 ↔ 4	[4, 2, 3, 1]	[5, 6, 7, 8]
5	4 ↔ 1	1 ↔ 3	[3, 2, 1]	[4, 5, 6, 7, 8]
6	3 ↔ 1	1 ↔ 2	[2, 1]	[3, 4, 5, 6, 7, 8]

Tur	Kök ↔ son	Süzme	Heap	Sıralı bölge
7	2 ↔ 1	—	[1]	[2, 3, 4, 5, 6, 7, 8]

Tablo 2.23: Heap sort'un ikinci aşaması. “Süzme” sütunu köke gelen öğenin yaptığı takasları verir.

Birinci turda 8 son hücreye (indis 7) geçer ve köke 1 gelir. 1'in çocukları 6 ve 7'dir; büyük olan 7 ile yer değiştirir ve indis 2'ye iner. Orada çocukları 3 ve 2'dir; 3 ile yer değiştirir ve yaprağa ulaşır. Heap [7, 6, 3, 4, 5, 1, 2] olur. Sonraki turlar aynı biçimde ilerler. Yedi tur sonunda heap'te tek öğe kalır; o da en küçük öğedir ve indis 0'da yerindedir. Sıralı dizi [1, 2, 3, 4, 5, 6, 7, 8]'dir.



Şekil 2.25: İkinci aşamanın birinci turu. (a) Kök 8, `dizi[7] = 1` ile yer değiştirdi; 8 sıralı bölgeye (yeşil) geçti, heap boyutu 7. (b) Kökteki 1 aşağı süzüldü (1 ↔ 7, 1 ↔ 3); heap [7, 6, 3, 4, 5, 1, 2].

2.8.5 Java kodu ve analiz

```
void yiginSirala(int[] dizi) {
  int n = dizi.length;
  for (int i = n / 2 - 1; i >= 0; i--)   // 1) yığın kur
    yiginlastir(dizi, n, i);
  for (int i = n - 1; i > 0; i--) {      // 2) sırala
    takas(dizi, 0, i);                   // en büyüğü sona
    yiginlastir(dizi, i, 0);             // kökü süz
  }
}
void yiginlastir(int[] dizi, int n, int i) {
  int enBuyuk = i, sol = 2 * i + 1, sag = 2 * i + 2;
  if (sol < n && dizi[sol] > dizi[enBuyuk]) enBuyuk = sol;
  if (sag < n && dizi[sag] > dizi[enBuyuk]) enBuyuk = sag;
  if (enBuyuk != i) {
    takas(dizi, i, enBuyuk);
    yiginlastir(dizi, n, enBuyuk);
  }
}
```

- `yiginSirala` iki döngüden oluşur. Birinci döngü heap'i kurar: `i`, son iç düğüm `n / 2 - 1`'den köke kadar azalır ve her düğüm için `yiginlastir(dizi, n, i)` çağrılır.
- İkinci döngü sıralar: `i`, son indisten 1'e kadar azalır. Her turda kök `dizi[i]` ile takas edilir. Ardından `yiginlastir(dizi, i, 0)` çağrılır; ikinci argüman `i`, heap'in artık yalnız ilk `i` öğeden oluştuğunu söyler. Böylece sona taşınan öğeler süzmede yeniden kullanılmaz.
- `yiginlastir`, `i` düğümünü `n` boyutlu heap'te aşağı süzer. Önce `i`, sol ve sağ çocuk arasında en büyüğün indisi `enBuyuk`'ta bulunur. `sol < n` ve `sag < n` denetimleri çocuğun heap içinde olup olmadığını sorar.
- `enBuyuk != i` ise düğüm çocuklarından birinden küçüktür; ikisi takas edilir ve süzme çocuğun konumunda özyinelemeli olarak sürer. Eşitse düğüm yerindedir ve özyineleme biter.

Heap kurmanın maliyeti. Kurma döngüsü yaklaşık $n/2$ düğüm için süzme yapar ve tek bir süzme $O(\log n)$ sürer. Bu, kaba bir $O(n \log n)$ sınırı verir. Ama sınır gevşektir, çünkü düğümlerin çoğu alçaktadır. Yüksekliği h olan bir düğümün süzmesi en çok h adımdır. Yüksekliği h olan düğüm sayısı en çok $\lceil n/2^{h+1} \rceil$'dir: yaprakların sayısı yaklaşık $n/2$, bir üst düzeyinki $n/4$'tür ve böyle devam eder. Toplam iş

> $\Sigma_{h \geq 0} (n/2^{h+1}) \cdot h = (n/2) \cdot \Sigma_{h \geq 0} h/2^h = (n/2) \cdot 2 = n$

mertebesindedir. Yani aşağıdan yukarı heap kurma $O(n)$'dir. Bu yönteme R. W. Floyd'un adıyla Floyd yöntemi de denir. Heap, öğeleri tek tek ekleyip her birini yukarı süzerek de kurulabilir; bu da geçerli bir max-heap verir, ama en kötü durumda $O(n \log n)$ sürer.

Sıralama aşamasının maliyeti. İkinci döngü $n - 1$ kez döner ve her turda kökü en çok heap yüksekliği kadar, yani $O(\log n)$ adım süzer. Bu aşama $O(n \log n)$'dir ve toplam süreyi belirler. En kötü durumda bile süre $O(n \log n)$'dir.

Bellek ve kararlılık. Heap sort yalnız birkaç değişken kullanır; $O(1)$ ek bellekle yerinde çalışır. `yiginlastir` burada özyinelemeli yazılmıştır, ama özyineleme son satırdadır ve kolayca döngüye çevrilir; döngülü sürümde yığın da gerekmez. Heap sort kararlı değildir: kök ile son öğenin takası, eşit anahtarlı bir öğeyi diğerlerinin üzerinden atlatabilir. Uyarlanabilir de değildir; sıralı bir girdi bile önce heap'e dönüştürülür ve yine $\Theta(n \log n)$ sürer.

> **Sık yapılan hata**
>
> İkinci döngüde `yiginlastir(dizi, n, 0)` yazmak. Heap boyutu küçültülmezse süzme, sıralı bölgeye taşınmış büyük öğeleri yeniden çocuk sayar ve onları köke geri getirir. Sıralama bozulur. Doğru çağrı `yiginlastir(dizi, i, 0)` 'dır; heap yalnız `dizi[0..i-1]` 'dir.

2.8.6 Kod izleme: heap kurma

Heap kurma döngüsünün [3, 9, 2, 1, 4, 5] dizisindeki çalışması satır satır şöyledir. $n = 6$'dır; yapraklar indis 3, 4 ve 5'tir. Döngü $i = n/2 - 1 = 2$'den başlar.

```
for (int i = n / 2 - 1; i >= 0; i--)
  yiginlastir(dizi, n, i);

void yiginlastir(int[] dizi, int n, int i) {
  int enBuyuk = i, sol = 2 * i + 1, sag = 2 * i + 2;
  if (sol < n && dizi[sol] > dizi[enBuyuk]) enBuyuk = sol;
  if (sag < n && dizi[sag] > dizi[enBuyuk]) enBuyuk = sag;
  if (enBuyuk != i) {
    takas(dizi, i, enBuyuk);
    yiginlastir(dizi, n, enBuyuk);
  }
}
```

Çağrı	Denetimler	`enBuyuk`	İşlem	`dizi`
`yiginlastir(6, 2)`	sol = 5 (5) > 2: evet; sağ = 6 ≥ n	5	`takas(2, 5)`	[3, 9, 5, 1, 4, 2]
`yiginlastir(6, 5)`	sol = 11 ≥ n	5	süzme biter	[3, 9, 5, 1, 4, 2]
`yiginlastir(6, 1)`	sol = 3 (1) > 9: hayır; sağ = 4 (4) > 9: hayır	1	değişiklik yok	[3, 9, 5, 1, 4, 2]
`yiginlastir(6, 0)`	sol = 1 (9) > 3: evet; sağ = 2 (5) > 9: hayır	1	`takas(0, 1)`	[9, 3, 5, 1, 4, 2]
`yiginlastir(6, 1)`	sol = 3 (1) > 3: hayır; sağ = 4 (4) > 3: evet	4	`takas(1, 4)`	[9, 4, 5, 1, 3, 2]
`yiginlastir(6, 4)`	sol = 9 ≥ n	4	süzme biter	[9, 4, 5, 1, 3, 2]

Tablo 2.24: `dizi = [3, 9, 2, 1, 4, 5]` için heap kurma izlemesi. Parantez içindeki sayılar o indisteki değerdir.

`i = 2`'de 2'nin tek çocuğu 5'tir (indis 5); 5 daha büyük olduğu için yer değiştirirler ve süzme indis 5'te sürer. İndis 5 yapraktır; süzme biter. `i = 1`'de 9 çocuklarından (1 ve 4) büyüktür; değişiklik olmaz. `i = 0`'da 3, büyük çocuğu 9 ile yer değiştirir ve indis 1'e iner. Orada çocukları 1 ve 4'tür; 4 ile yer değiştirir ve yaprağa iner. Döngü `i = -1`'de biter. Sonuç [9, 4, 5, 1, 3, 2]'dir: her ebeveyn çocuklarından büyüktür ve en büyük öğe 9 köktedir.

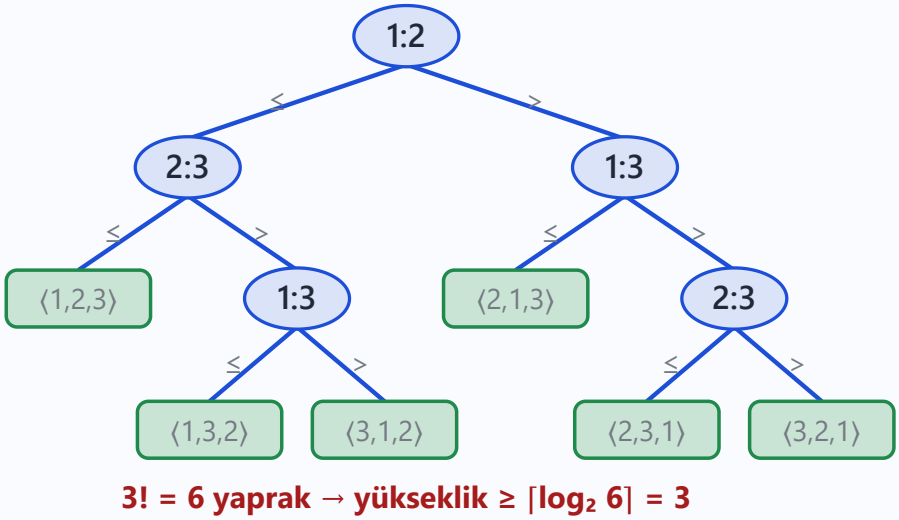
`sag` denetiminin `dizi[enBuyuk]` ile yapıldığına dikkat etmek gerekir; `dizi[i]` ile değil. `i = 0` adımında sağ çocuk 5, düğümün kendi değeri 3'ten büyüktür ama sol çocuk 9'dan küçüktür. Doğru seçim büyük çocuk 9'dur. Sağ çocuk `dizi[i]` ile karşılaştırılsaydı 5 seçilirdi; 5 köke çıkar, 9 onun altında kalırdı ve heap koşulu bozulurdu.

2.9 Doğrusal zamanlı sıralama

Birleştirme sıralaması ve heap sort en kötü durumda $\Theta(n \log n)$'dir. Daha hızlı bir karşılaştırmalı algoritma bulmak mümkün müdür? Bu kesim cevabın "hayır" olduğunu kanıtlar. Ardından, öğeleri karşılaştırmak yerine anahtar değerlerini doğrudan kullanarak doğrusal sürede sıralayan üç algoritmayı anlatır.

2.9.1 Karşılaştırmalı sıralamanın alt sınırı

Karşılaştırmalı bir algoritma, girdi hakkında bilgiyi yalnız "$a_i \le a_j$ mi?" sorularının cevaplarından edinir. Böyle bir algoritmanın davranışı bir **karar ağacı** (*decision tree*) ile gösterilebilir. Ağacın her iç düğümü bir karşılaştırmadır; *i:j* etiketi a_i ile a_j'nin karşılaştırıldığını söyler. Sol dal "≤", sağ dal ">" cevabını izler. Her yaprak, algoritmanın o cevap dizisi sonunda ürettiği sıralamadır. Girdi verildiğinde algoritma kökten bir yaprağa giden tek bir yolu izler. Yolun uzunluğu, o girdide yapılan karşılaştırma sayısıdır.



Şekil 2.26: Üç öğeli eklemeli sıralamanın karar ağacı. İç düğümler *i:j* karşılaştırmaları, yapraklar altı permütas-

yondur. $3! = 6$ yaprak için yükseklik en az $\lceil \log_2 6 \rceil = 3$ olmalıdır.

Teorem 2.3 (Karşılaştırma alt sınırı)

Birbirinden farklı n öğeyi sıralayan her karşılaştırmalı algoritma, en kötü durumda $\Omega(n \log n)$ karşılaştırma yapar.

İspat. Algoritmanın karar ağacını ele alalım. Farklı öğelerden oluşan girdinin $n!$ olası düzeni vardır ve her biri farklı bir permütasyonla sıralanır. Algoritma doğruysa her permütasyon en az bir yaprakta görünmelidir; ağacın en az $n!$ yaprağı vardır. Yüksekliği h olan bir ikili ağacın en çok 2^h yaprağı olabilir. Buradan $2^h \geq n!$, yani $h \geq \log_2(n!)$ çıkar. $n!$'in son $n/2$ çarpanının her biri en az $n/2$ olduğu için $n! \geq (n/2)^{n/2}$'dir. Logaritma alınınca $\log_2(n!) \geq (n/2)\log_2(n/2) = \Omega(n \log n)$ bulunur. Ağacın yüksekliği en kötü durumdaki karşılaştırma sayısıdır. □

Teoremin iki önemli sonucu vardır. Birincisi, birleştirme sıralaması ve heap sort asimptotik olarak en iyi karşılaştırmalı algoritmalardır; en kötü durumları alt sınıra eşittir. İkincisi, daha hızlı sıralamak için karşılaştırma dışında bir bilgiye ihtiyaç vardır. Anahtarlar küçük tam sayılarsa değerlerin kendisi dizi indisi olarak kullanılabilir; saymalı ve basamak sıralaması bunu yapar.

Örnek 2.5

Üç farklı öğeyi yalnız karşılaştırmayla sıralayan her algoritma en kötü durumda en az kaç karşılaştırma yapmak zorundadır?

Karar ağacının $3! = 6$ yaprağı olmalıdır. İki karşılaştırma en çok $2^2 = 4$ farklı sonucu ayırt edebilir; bu altı permütasyona yetmez. Üç karşılaştırma $2^3 = 8 \geq 6$ sonucu ayırt edebilir. Cevap $\lceil \log_2 6 \rceil = 3$'tür. Şekil 2.26'daki ağaçta bazı yapraklara iki karşılaştırmayla ulaşıldığına dikkat edilmelidir; alt sınır yalnız en uzun yol için geçerlidir. Sekiz öğe için sınır $\lceil \log_2 40.320 \rceil = 16$'dır; birleştirme sıralamasının en kötü 17 karşılaştırması buna çok yakındır.

2.9.2 Saymalı sıralama

Saymalı sıralama (*counting sort*), anahtarları küçük bir aralıktaki tam sayılar olan öğeleri hiç karşılaştırmadan sıralar. Girdi `A[1..n]` dizisidir ve her anahtar `{1, 2, …, k}` kümesindedir. Çıktı `B[1..n]` dizisine yazılır. Geçici olarak `C[1..k]` sayaç dizisi kullanılır.

Algoritmanın fikri şudur: anahtarı i'den küçük ya da eşit olan öğelerin sayısı biliniyorsa, anahtarı i olan son öğenin çıktıdaki konumu da bilinir. Örneğin anahtarı 3'ten küçük ya da eşit beş öğe varsa, son 3 çıktının beşinci hücresine yazılır. Algoritma dört döngüden oluşur:

```
for i = 1 to k
  C[i] = 0
for j = 1 to n
  C[A[j]] = C[A[j]] + 1   // C[i] = |{anahtar = i}|
for i = 2 to k
  C[i] = C[i] + C[i-1]    // C[i] = |{anahtar ≤ i}|
for j = n downto 1
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

1. Birinci döngü bütün sayaçları sıfırlar.
2. İkinci döngü girdiyi tarar ve her anahtarın kaç kez geçtiğini sayar. Döngü sonunda `C[i]`, anahtarı *i* olan öğe sayısıdır.
3. Üçüncü döngü **önek toplamını** hesaplar: her sayaca solundakini ekler. Döngü sonunda `C[i]`, anahtarı *i*'den küçük ya da eşit olan öğe sayısıdır.
4. Dördüncü döngü girdiyi *sondan başa* tarar. Her öğe `B[C[A[j]]]` hücresine yazılır ve sayaç bir azaltılır; aynı anahtarlı bir sonraki öğe bir sol hücreye düşer.

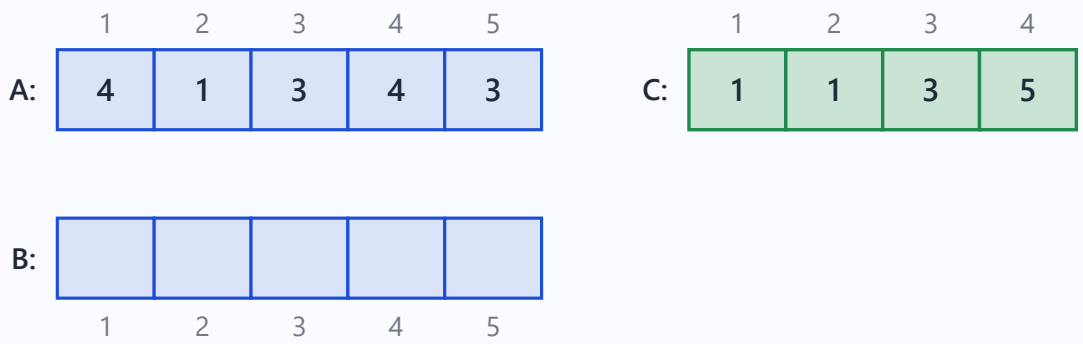
2.9.3 Adım adım: A = [4, 1, 3, 4, 3]

Girdi *A = [4, 1, 3, 4, 3]*, *n = 5*, anahtarlar *{1, …, 4}*'tür (*k = 4*). Diziler 1 tabanlıdır.

Döngü	Adım	İşlem	`C`	`B`
1	—	sayaçlar sıfırlanır	[0, 0, 0, 0]	[–, –, –, –, –]
2	`j = 1`	`A[1] = 4`, `C[4]++`	[0, 0, 0, 1]	
2	`j = 2`	`A[2] = 1`, `C[1]++`	[1, 0, 0, 1]	
2	`j = 3`	`A[3] = 3`, `C[3]++`	[1, 0, 1, 1]	
2	`j = 4`	`A[4] = 4`, `C[4]++`	[1, 0, 1, 2]	
2	`j = 5`	`A[5] = 3`, `C[3]++`	[1, 0, 2, 2]	
3	`i = 2`	`C[2] = 0 + 1`	[1, 1, 2, 2]	
3	`i = 3`	`C[3] = 2 + 1`	[1, 1, 3, 2]	
3	`i = 4`	`C[4] = 2 + 3`	[1, 1, 3, 5]	
4	`j = 5`	`A[5] = 3` → `B[3]`	[1, 1, 2, 5]	[–, –, 3, –, –]
4	`j = 4`	`A[4] = 4` → `B[5]`	[1, 1, 2, 4]	[–, –, 3, –, 4]
4	`j = 3`	`A[3] = 3` → `B[2]`	[1, 1, 1, 4]	[–, 3, 3, –, 4]
4	`j = 2`	`A[2] = 1` → `B[1]`	[0, 1, 1, 4]	[1, 3, 3, –, 4]
4	`j = 1`	`A[1] = 4` → `B[4]`	[0, 1, 1, 3]	[1, 3, 3, 4, 4]

Tablo 2.25: Saymalı sıralamanın izlemesi. `C` dizisi `[C[1], C[2], C[3], C[4]]` biçiminde yazılmıştır.

Sayma döngüsünden sonra `C = [1, 0, 2, 2]` 'dir: bir tane 1, iki tane 3, iki tane 4 vardır; 2 hiç yoktur. Önek toplamından sonra `C = [1, 1, 3, 5]` olur. Bu değerler, her anahtarın çıktıdaki son konumudur: 1'in son konumu 1, 3'ün son konumu 3, 4'ün son konumu 5'tir. Son döngü girdiyi sağdan sola gezer. `A[5] = 3`, `B[3]` 'e yazılır ve `C[3]` 2'ye iner; bir sonraki 3 (`A[3]`) bu yüzden `B[2]` 'ye düşer. Sonuç *B = [1, 3, 3, 4, 4]*'tür. Hiç karşılaştırma yapılmamıştır.



Şekil 2.27: Önek toplamı hazır: `C = [1, 1, 3, 5]`. `C[i]`, anahtarı *i* olan son öğenin `B` 'deki konumudur. `B` henüz boştur.

Son döngünün sondan başa gezmesi kararlılığı sağlar. Eşit anahtarlı iki öğeden girdide sonra gelen önce işlenir ve daha sağdaki hücreye yazılır; önce gelen ise onun soluna düşer. Böylece eşitlerin girdideki sırası çıktıda korunur. Döngü baştan sona gezseydi eşitlerin sırası tersine dönerdi; sonuç yine sıralı olurdu ama algoritma kararsız olurdu.

2.9.4 Karmaşıklık ve Java kodu

Döngü	İş	Süre
1	`C[1..k] = 0`	$\Theta(k)$
2	sayma	$\Theta(n)$
3	önek toplamı	$\Theta(k)$
4	`B` 'ye yerleştirme	$\Theta(n)$
Toplam		$\Theta(n + k)$

Tablo 2.26: Saymalı sıralamanın döngü maliyetleri.

Toplam süre $\Theta(n + k)$'dir. Anahtar aralığı $k = O(n)$ ise süre $\Theta(n)$'dir. Bu, karşılaştırma alt sınırıyla çelişmez; saymalı sıralama karşılaştırma yapmadığı için teoremin varsayımı ona uymaz. Ek bellek $\Theta(n + k)$'dir: çıktı dizisi ve sayaç dizisi. k çok büyükse algoritma pratik değildir. Örneğin 32 bitlik tam sayılar için yaklaşık 4,3 milyar sayaç gerekir.

Java sürümü 0 tabanlıdır ve `0 … maks` aralığını kullanır; `maks`, dizinin en büyük öğesidir.

```
int[] saymaliSirala(int[] dizi) {
  int maks = enBuyuk(dizi);
  int[] sayac = new int[maks + 1];
  for (int x : dizi) sayac[x]++;
  for (int i = 1; i <= maks; i++) sayac[i] += sayac[i - 1];
  int[] cikti = new int[dizi.length];
  for (int i = dizi.length - 1; i >= 0; i--) {
    cikti[sayac[dizi[i]] - 1] = dizi[i];
    sayac[dizi[i]]--;
  }
  return cikti;
}
```

Sözde koddan farklı olarak sayaçlar `new int[maks + 1]` ile otomatik olarak sıfırlanır. Önek toplamından sonra `sayac[x]`, anahtarı x'ten küçük ya da eşit öğe sayısıdır; çıktı 0 tabanlı olduğu için x'in son konumu `sayac[x] - 1`'dir. Örneğin $A = [2, 5, 3, 0, 2, 3, 0, 3]$ için adetler `[2, 0, 2, 3, 0, 1]`, önek toplamları `[2, 2, 4, 7, 7, 8]`'dir; çıktı [0, 0, 2, 2, 3, 3, 3, 5] olur.

> **Sık yapılan hata**
>
> Son döngüyü baştan sona gezmek. Çıktı yine sıralı olur, ama eşit anahtarlı öğeler ters sırada yazılır. Saymalı sıralama tek başına kullanıldığında bu fark görünmeyebilir; basamak sıralamasının ara adımı olarak kullanıldığında ise önceki basamakların kurduğu düzeni bozar ve sonucu yanlış yapar.

2.9.5 Basamak sıralaması

Saymalı sıralama anahtar aralığı küçükken çok hızlıdır, ama aralık büyüyünce bellek ihtiyacı patlar. **Basamak sıralaması** (*radix sort*) bu sorunu, büyük sayıları basamak basamak sıralayarak çözer. Her basamak yalnız 0–9 arasında bir değer alır; bu yüzden her geçişte $k = 10$ olan bir saymalı sıralama yeter.

Burada anlatılan sürüm **LSD** (*least significant digit*) basamak sıralamasıdır: sıralama en anlamsız basamaktan, yani birler basamağından başlar; onlar, yüzler diye devam eder. Her geçişte o basamağa göre *kararlı* bir sıralama yapılır. Basamak sayısı d, taban b olsun. Her geçiş $\Theta(n + b)$ sürer ve d geçiş vardır; toplam süre $\Theta(d \cdot (n + b))$'dir. d ve b sabitse süre $\Theta(n)$'dir. Ek bellek $\Theta(n + b)$'dir; algoritma yerinde değildir.

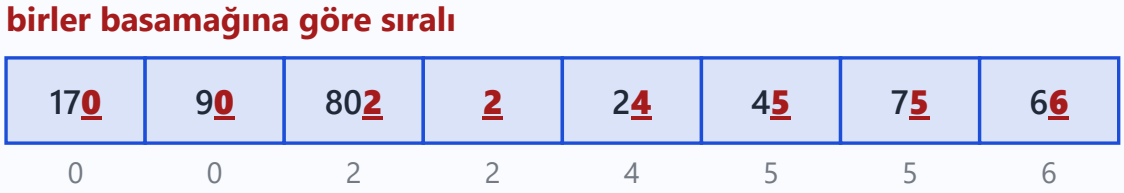
```
int[] basamakSirala(int[] dizi) {
  int maks = enBuyuk(dizi);
  for (int us = 1; maks / us > 0; us *= 10)
    dizi = basamagaGoreSay(dizi, us);
  return dizi;
}
int[] basamagaGoreSay(int[] dizi, int us) {
  int n = dizi.length;
  int[] cikti = new int[n], sayac = new int[10];
  for (int x : dizi) sayac[(x / us) % 10]++;
  for (int i = 1; i < 10; i++) sayac[i] += sayac[i - 1];
  for (int i = n - 1; i >= 0; i--) {
    int b = (dizi[i] / us) % 10;
    cikti[--sayac[b]] = dizi[i];
  }
  return cikti;
}
```

- `basamakSirala`, `us` değişkenini 1, 10, 100, ... diye büyütür. `maks / us > 0` koşulu, en büyük sayının hâlâ işlenmemiş bir basamağı olduğu sürece doğrudur. 802 için `us` 1, 10 ve 100 değerlerini alır; 1000'de *802/1000 = 0* olur ve döngü biter.
- `basamagaGoreSay`, saymalı sıralamanın bir basamağa uyarlanmış hâlidir. Bir sayının o basamağı `(x / us) % 10` ile bulunur: 802 için `us = 10`'da *802/10 = 80*, *80 % 10 = 0*.
- Son döngü yine sondan başa gezer ve kararlılığı sağlar. `cikti[--sayac[b]]` ifadesi önce sayacı bir azaltır, sonra azalmış değeri indis olarak kullanır; saymalı sıralamanın Java sürümündeki `sayac[x] - 1` ile aynı işi tek ifadede yapar.

Örnek 2.6

Girdi [170, 45, 75, 90, 802, 24, 2, 66] olsun. En büyük sayı 802 üç basamaklıdır; üç geçiş yapılır. Eksik basamaklar 0 sayılır: 45, üç basamaklı olarak 045'tir.

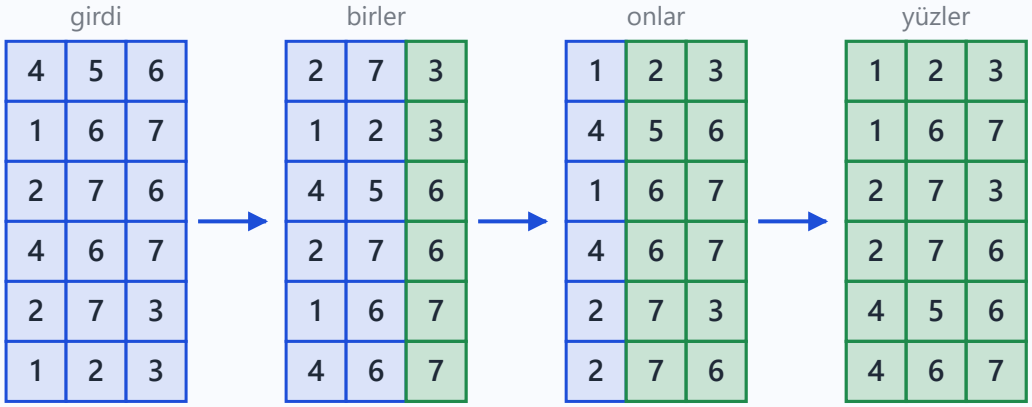
1. **Birler basamağı** (0, 5, 5, 0, 2, 4, 2, 6): [170, 90, 802, 2, 24, 45, 75, 66]. Birler basamağı 0 olan 170 ve 90 girdideki sıralarını korudu; 802 ile 2 ve 45 ile 75 de öyle.
2. **Onlar basamağı**: [802, 2, 24, 45, 66, 170, 75, 90]. Onlar basamağı 0 olan 802 ile 2 bir önceki geçişteki sırasıyla (802 önce) yerleşti.
3. **Yüzler basamağı**: [2, 24, 45, 66, 75, 90, 170, 802]. Bütün basamaklar işlendi ve dizi sıralandı.



Şekil 2.28: Birinci geçişten sonra dizi birler basamağına göre sıralıdır. Altı çizili rakamlar, bu geçişte anahtar olarak kullanılan birler basamağıdır.

2.9.6 Kararlılık neden gerekli?

LSD basamak sıralamasının doğruluğu şu değişmeze dayanır: *k*. geçişten sonra dizi, sayıların son *k* basamağına göre sıralıdır. Bir sonraki geçiş *(k + 1)*. basamağa göre sıralar. Bu basamağı farklı olan iki sayı doğru sıraya girer. Bu basamağı aynı olan iki sayının sırasını ise önceki geçişler belirlemiştir ve o sıra son *k* basamağa göre doğrudur. Bu sıranın korunması, yani alt sıralamanın kararlı olması şarttır.



Şekil 2.29: 456, 167, 276, 467, 273, 123 sayılarının üç geçişte sıralanması. Yeşil sütunlar o ana kadar sıralı hâle gelen son basamaklardır: *k* geçişten sonra dizi son *k* basamağa göre sıralıdır.

Şekildeki örnekte birinci geçişten sonra dizi [273, 123, 456, 276, 167, 467]'dir. İkinci geçişte 167 ile 467'nin onlar basamağı eşittir (6). Kararlı sıralama birinci geçişten gelen sırayı korur: 167 önce gelir. Bu sıra doğrudur, çünkü iki sayının birler basamağı da eşittir (7) ve girdideki sıra korunmuştur. Son geçişte 273 ile 276'nın yüzler basamağı eşittir (2). Alt sıralama kararsız olsaydı bu ikisi yer değiştirebilirdi ve sonuç [123, 167, 276, 273, ...] gibi yanlış bir dizi olurdu. Sonuç [123, 167, 273, 276, 456, 467]'dir.

Basamak sıralaması en anlamlı basamaktan da başlayabilir; bu sürüme **MSD** (*most significant digit*) denir. MSD sürümü sayıları önce ilk basamağa göre kovalara ayırır ve her kovayı ayrı ayrı, özyinelemeli olarak sıralar. Dizgi sıralamasında yaygın kullanılır. LSD sürümü ise daha sadedir ama bütün anahtarların aynı uzunlukta olmasını ister.

2.9.7 Kod izleme: basamagaGoreSay()

```
int[] basamagaGoreSay(int[] dizi, int us) {
  int n = dizi.length;
  int[] cikti = new int[n], sayac = new int[10];
  for (int x : dizi) sayac[(x / us) % 10]++;
  for (int i = 1; i < 10; i++) sayac[i] += sayac[i - 1];
  for (int i = n - 1; i >= 0; i--) {
    int b = (dizi[i] / us) % 10;
    cikti[--sayac[b]] = dizi[i];
  }
  return cikti;
}
```

`basamagaGoreSay(dizi, 1)` çağrısını [32, 15, 41, 23, 11] dizisinde izleyelim. Birler basamakları sırasıyla 2, 5, 1, 3, 1'dir.

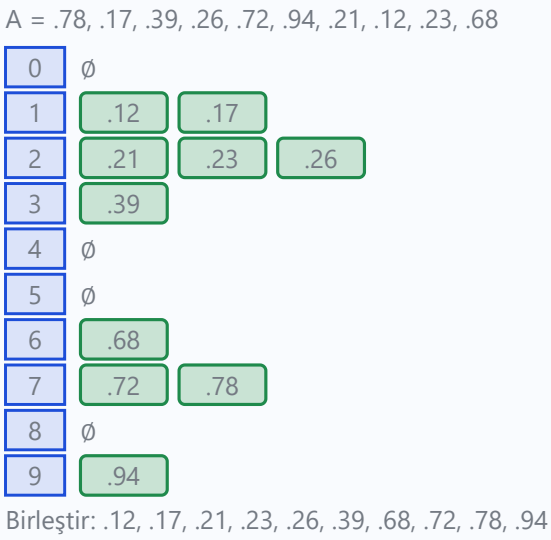
Aşama	İşlem	`sayac[0..5]`	`cikti`
sayma	basamaklar 2, 5, 1, 3, 1	[0, 2, 1, 1, 0, 1]	[–, –, –, –, –]
önek	önek toplamı	[0, 2, 3, 4, 4, 5]	[–, –, –, –, –]
`i = 4`	11, `b = 1`, `--sayac[1] = 1`	[0, 1, 3, 4, 4, 5]	[–, 11, –, –, –]
`i = 3`	23, `b = 3`, `--sayac[3] = 3`	[0, 1, 3, 3, 4, 5]	[–, 11, –, 23, –]
`i = 2`	41, `b = 1`, `--sayac[1] = 0`	[0, 0, 3, 3, 4, 5]	[41, 11, –, 23, –]
`i = 1`	15, `b = 5`, `--sayac[5] = 4`	[0, 0, 3, 3, 4, 4]	[41, 11, –, 23, 15]
`i = 0`	32, `b = 2`, `--sayac[2] = 2`	[0, 0, 2, 3, 4, 4]	[41, 11, 32, 23, 15]

Tablo 2.27: `basamagaGoreSay([32, 15, 41, 23, 11], 1)` izlemesi. `sayac` yalnız 0–5 hücreleriyle gösterilmiştir; 6–9 hücreleri hep 5'tir.

Önek toplamından sonra `sayac[b]`, basamağı *b*'den küçük ya da eşit olan öğe sayısıdır; örneğin `sayac[3] = 4`'tür. Bu sayı, o basamaklı son öğenin `cikti`'daki konumunun bir fazlasıdır. Sağdan sola gezerken 11 önce işlenir ve `cikti[1]`'e yazılır. 41 ile 11'in birler basamağı aynıdır; 41 sonra işlenir ve `cikti[0]`'a, yani 11'in soluna düşer. Girdide de 41, 11'den önceydi: sıra korunmuştur. Sonuç [41, 11, 32, 23, 15]'tir ve birler basamağına göre sıralıdır. Sonraki geçiş (`us = 10`) onlar basamağına bakar ve [11, 15, 23, 32, 41] dizisini üretir. 11 ile 15'in onlar basamağı eşittir; birinci geçişin kurduğu sıra (11 önce) korunur.

2.9.8 Kova sıralaması

Kova sıralaması (*bucket sort*), girdinin dağılımı hakkında bir varsayım yaparak doğrusal beklenen süreye ulaşır. Girdinin *[0, 1)* aralığına düzgün dağılmış gerçel sayılardan oluştuğunu varsayalım. Algoritma *n* kova açar ve *x* değerini $\lfloor n \cdot x \rfloor$ numaralı kovaya atar. Her kova, örneğin eklemeli sıralamayla, kendi içinde sıralanır. Sonra kovalar sırayla birleştirilir.



Şekil 2.30: Kova sıralaması. On sayı (.78, .17, .39, .26, .72, .94, .21, .12, .23, .68) on kovaya dağıtılır: örneğin .78, $\lfloor 10 \cdot 0{,}78 \rfloor = 7$ numaralı kovaya düşer. Kovalar sıralanıp sırayla birleştirilir.

Şekildeki örnekte 2 numaralı kovaya üç sayı (.21, .23, .26), 1 ve 7 numaralı kovalara ikişer sayı düşer; 0, 4, 5 ve 8 numaralı kovalar boştur. Dağılım düzgünse her kovaya ortalama bir öğe düşer ve kovaları sıralamak toplamda $\Theta(n)$ beklenen sürer. Dağıtma ve birleştirme de doğrusaldır; beklenen toplam süre $\Theta(n)$'dir. Ancak varsayım bozulursa, örneğin bütün öğeler tek bir kovaya düşerse, o kova eklemeli sıralamayla $\Theta(n^2)$'de sıralanır. Kova sıralaması, kovalar kararlı bir algoritmayla sıralanırsa kararlıdır ve $\Theta(n)$ ek bellek ister. Sıralama Algoritmaları simülatöründeki onuncu algoritma kova sıralamasıdır.

2.10 Timsort

Kütüphanelerin kullandığı sıralama algoritmaları ders kitaplarındaki saf algoritmalardan biraz farklıdır. Gerçek veride sık görülen düzenlerden yararlanır ve birkaç algoritmayı bir arada kullanır. Timsort bunun en bilinen örneğidir.

2.10.1 Karma bir algoritma

Timsort, Tim Peters'ın 2002'de Python için tasarladığı karma (*hybrid*) bir sıralama algoritmasıdır. Bugün Java'da nesne dizileri (`Arrays.sort(Object[])`, `List.sort`) ve Python'da `sorted` Timsort kullanır. Algoritmanın çıkış noktası şu gözlemdir: gerçek veri çoğu zaman tamamen rastgele değildir. İçinde zaten sıralı uzun parçalar bulunur. Bir tabloya sonradan eklenen birkaç kayıt, kısmen sıralı bir günlük dosyası ya da iki sıralı listenin art arda eklenmesi buna örnektir.

Timsort bu sıralı parçaları, yani **run**'ları kullanır. Önce dizi soldan sağa taranır ve doğal run'lar bulunur; azalan bir run ters çevrilip artana dönüştürülür. Çok kısa run'lar, belirli bir en küçük uzunluğa (*minrun*, pratikte 32 ile 64 arası) kadar eklemeli sıralamayla uzatılır. Eklemeli sıralama küçük ve neredeyse sıralı parçalarda çok hızlı olduğu için bu seçim yerindedir. Ardından run'lar birleştirme sıra-

lamasının birleştirme adımıyla ikişer ikişer birleştirilir. Birleştirme sırasında bir run'dan art arda çok öğe alınıyorsa algoritma "dörtnala" (*galloping*) kipine geçer. Bu kipte sıradaki öğenin yeri üstel aramayla bulunur ve bir öbek öğe tek seferde kopyalanır.

Timsort kararlıdır, çünkü hem eklemeli sıralama hem birleştirme kararlıdır. En kötü durumda $O(n \log n)$'dir. Girdi zaten sıralıysa tek bir run bulunur ve algoritma $n - 1$ karşılaştırmayla, yani $\Theta(n)$'de biter. Birleştirme için en çok $O(n)$ ek bellek kullanır. Nesne sıralamasında kararlılık önemli olduğu için Java ilkel türlerde hızlı sıralamayı, nesnelerde Timsort'u kullanır.

Aşağıdaki kod Timsort'un basitleştirilmiş bir sürümüdür: doğal run'ları aramaz, diziyi sabit uzunluklu parçalara böler. Parçaları `eklemeliSirala(dizi, sol, sag)` ile sıralar, sonra 2.6.4'teki `birlestir` metoduyla birleştirir.

```
static final int MIN_PARCA = 4;
void timSirala(int[] dizi) {
  int n = dizi.length;
  for (int i = 0; i < n; i += MIN_PARCA)            // 1) run'lar
    eklemeliSirala(dizi, i, Math.min(i + MIN_PARCA - 1, n - 1));
  for (int boyut = MIN_PARCA; boyut < n; boyut *= 2)  // 2) birleştir
    for (int sol = 0; sol < n; sol += 2 * boyut) {
      int orta = sol + boyut - 1;
      int sag = Math.min(sol + 2 * boyut - 1, n - 1);
      if (orta < sag) birlestir(dizi, sol, orta, sag);
    }
}
```

- Birinci döngü diziyi `MIN_PARCA` uzunluklu parçalara ayırır ve her birini eklemeli sıralamayla sıralar. Son parça daha kısa olabilir; `Math.min(i + MIN_PARCA - 1, n - 1)` dizinin dışına çıkılmasını önler.
- İkinci döngü birleştirmeleri aşağıdan yukarı yapar. `boyut`, birleştirilecek sıralı parçaların uzunluğudur ve her turda iki katına çıkar. Her turda yan yana iki parça `[sol..orta]` ve `[orta+1..sag]` birleştirilir.
- `orta < sag` değilse sağda birleşecek bir parça yoktur; tek kalan parça bu turda aynen bırakılır.

Bu döngülü yapı, birleştirme sıralamasının özyinelemesiz, "aşağıdan yukarı" (*bottom-up*) bir sürümüdür. Özyineleme yığını kullanmaz.

2.10.2 Adım adım

Basitleştirilmiş Timsort'u `MIN_PARCA = 4` ile [5, 21, 7, 23, 19, 10, 3, 12, 8, 1] dizisinde izleyelim ($n = 10$).

Adım	İşlem	Dizi
Run [0..3]	eklemeli: [5, 21, 7, 23] → [5, 7, 21, 23]	[5, 7, 21, 23, 19, 10, 3, 12, 8, 1]

Adım	İşlem	Dizi
Run [4..7]	eklemeli: [19, 10, 3, 12] → [3, 10, 12, 19]	[5, 7, 21, 23, 3, 10, 12, 19, 8, 1]
Run [8..9]	eklemeli: [8, 1] → [1, 8]	[5, 7, 21, 23, 3, 10, 12, 19, 1, 8]
`boyut = 4`	[0..3] + [4..7] birleştirilir	[3, 5, 7, 10, 12, 19, 21, 23, 1, 8]
`boyut = 4`	`sol = 8` : `orta = 11 ≥ sag = 9` , eşi yok	değişmez
`boyut = 8`	[0..7] + [8..9] birleştirilir	[1, 3, 5, 7, 8, 10, 12, 19, 21, 23]

Tablo 2.28: Basitleştirilmiş Timsort'un adımları.

Dizi önce üç run'a ayrılır; son run iki öğelidir. Her run eklemeli sıralamayla sıralanır. `boyut = 4` turunda ilk iki run birleşir. Üçüncü run'ın birleşeceği bir eşi yoktur, çünkü `orta = 8 + 4 − 1 = 11` dizinin sonunu (`sag = 9`) aşar. `boyut = 8` turunda sekiz öğeli parça ile iki öğeli parça birleşir ve dizi sıralanır. `boyut = 16 ≥ n` olduğunda döngü biter.



Şekil 2.31: `boyut = 4` turunda `birlestir(0, 3, 7)` sonrası. İlk sekiz hücre sıralı (yeşil); son iki hücredeki [1, 8] run'ı bir sonraki turu bekliyor.

2.11 Algoritmaların karşılaştırılması

Bölümdeki algoritmaları artık yan yana koymak mümkündür. Bu kesim önce asimptotik özellikleri bir tabloda toplar, sonra gerçek karşılaştırma sayılarını verir ve hangi durumda hangi algoritmanın seçileceğini tartışır.

2.11.1 Karmaşıklık tablosu

Algoritma	En iyi	Ortalama	En kötü	Ek bellek	Kararlı	Yerinde
Kabarcık (bayraklı)	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Evet	Evet
Eklemeli	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Evet	Evet
Seçimli	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Hayır	Evet
Shell (n/2 aralık)	$\Theta(n \log n)$	aralığa bağlı	$\Theta(n^2)$	$O(1)$	Hayır	Evet
Birleştirme	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	Evet	Hayır
Hızlı	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$O(\log n)$*	Hayır	Evet
Heap sort	$O(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$	Hayır	Evet

Algoritma	En iyi	Ortalama	En kötü	Ek bellek	Kararlı	Yerinde
Saymalı	Θ(n + k)	Θ(n + k)	Θ(n + k)	Θ(n + k)	Evet	Hayır
Basamak (LSD)	Θ(d(n + b))	Θ(d(n + b))	Θ(d(n + b))	Θ(n + b)	Evet	Hayır
Kova	Θ(n)	Θ(n)†	Θ(n^2)	Θ(n)	Evet	Hayır
Timsort	Θ(n)	Θ(n log n)	Θ(n log n)	O(n)	Evet	Hayır

Tablo 2.29: Sıralama algoritmalarının karşılaştırılması. (*) özyineleme yığını, küçük parçaya önce inilirse. (†) düzgün dağılımda beklenen. *k*: anahtar aralığı, *d*: basamak sayısı, *b*: taban. Timsort'un *Θ(n)* en iyi durumu doğal run arayan gerçek sürüm içindir.

Tablodaki iki ayrıntı açıklama ister. Heap sort'un en iyi durumu *Θ* değil *O* ile yazılmıştır: bütün anahtarlar eşitse hiçbir süzme ilerlemez ve algoritma doğrusal sürede biter. Farklı anahtarlarda ise en iyi durum da *Θ(n log n)*'dir. Hızlı sıralamanın ek belleği yalnız özyineleme yığınıdır; büyük parça döngüyle, küçük parça özyinelemeyle işlenirse yığın derinliği *O(log n)*'yi aşmaz.

2.11.2 Ölçüm: n = 1000

Asimptotik gösterim sabit çarpanları gizler. Gerçek davranışı görmek için bu bölümdeki kodlar *n = 1000* öğeli üç tür girdiyle çalıştırılmış ve yapılan öğe karşılaştırmaları sayılmıştır. Rastgele dizi sabit bir tohumla üretilmiştir.

Algoritma	Rastgele	Sıralı	Ters sıralı
Kabarcık (bayraklı)	497.960	999	499.500
Eklemeli	252.205	999	499.500
Seçimli	499.500	499.500	499.500
Shell (n/2)	14.723	8.006	11.716
Birleştirme	8.693	5.044	4.932
Hızlı (son öğe pivot)	11.384	499.500	499.500
Heap sort	16.791	17.583	15.965

Tablo 2.30: *n = 1000* için öğe karşılaştırma sayıları. Karşılaştırma için $n^2/2 \approx 500.000$, $n \log_2 n \approx 9.966$.

Tablo kuramın söylediklerini sayılarla doğrular. Seçimli sıralama üç girdide de tam *n(n − 1)/2 = 499.500* karşılaştırma yapar. Kabarcık ve eklemeli sıralama sıralı girdide yalnız *n − 1 = 999* karşılaştırmayla biter. Rastgele girdide eklemeli sıralamanın sayısı yaklaşık $n^2/4 = 250.000$, kabarcığınki yaklaşık $n^2/2$'dir. Birleştirme sıralaması her girdide $n \log_2 n$'nin altında kalır. Sıralı ve ters sıralı girdilerde birleştirilen dizilerden biri hep erken bittiği için yaklaşık yarısı kadar karşılaştırma yapar. Hızlı sıralama rastgele girdide birleştirmenin biraz üstündedir ve 2.7.8'deki kesin beklenen değere (yaklaşık 10.986) yakındır. Ama sıralı ve ters sıralı girdilerde son öğeyi pivot alan sürüm en kötü durumuna düşer ve seçimli sıralama kadar karşılaştırma yapar. Heap sort girdiden neredeyse bağımsızdır; uyarlanabilir olmadığı için sıralı girdide bile 17.583 karşılaştırma harcar.

Karşılaştırma sayısı tek ölçüt değildir. Hızlı sıralama, karşılaştırma sayısı birleştirmeden fazla olmasına rağmen pratikte genellikle daha hızlıdır. Nedeni, yardımcı dizilere kopyalama yapmaması ve belleği sırayla gezmesidir. Heap sort ise karşılaştırma sayısının yanında belleği atlayarak gezdiği için (indis *i*'den *2i + 1*'e) önbelleği kötü kullanır ve çoğu zaman hızlı sıralamadan yavaştır.

2.11.3 Hangi durumda hangi algoritma?

Küçük ya da neredeyse sıralı dizi
: Eklemeli sıralama: ek yükü azdır, uyarlanabilir ve kararlıdır. Süresi $\Theta(n + I)$ olduğu için ters çift sayısı az olan büyük dizilerde de hızlıdır.

Genel amaçlı, hız öncelikli
: Hızlı sıralama, rastgele pivot ya da üçün ortancasıyla. Önbellek dostudur ve yerindedir.

Kararlılık gerekli
: Birleştirme sıralaması ya da Timsort. Kütüphanelerin nesne sıralaması bu yüzden Timsort'tur.

En kötü durum garantisi ve az bellek
: Heap sort: $O(n \log n)$ ve $O(1)$ ek bellek. Introsort adlı karma algoritma hızlı sıralamayla başlar; özyineleme çok derinleşirse heap sort'a geçer ve böylece en kötü durumu $O(n \log n)$'de tutar.

Küçük tam sayı anahtarlar
: Saymalı ya da basamak sıralaması: $\Theta(n + k)$ ve $\Theta(d(n + b))$.

Bağlı liste ya da dış bellek
: Birleştirme sıralaması. Rastgele erişim gerektirmez; öğeleri sırayla okuyup sırayla yazar. Belleğe sığmayan büyük bir dosya parça parça sıralanır ve parçalar birleştirilir. Bağlı liste de ek dizi olmadan, yalnız bağlar değiştirilerek birleştirilebilir.

Kabarcık ve seçimli sıralama çoğunlukla öğretim amaçlıdır. Seçimli sıralama yalnız yazmanın çok pahalı olduğu durumlarda anlam taşır, çünkü en fazla $n - 1$ takas yapar.

2.12 Sık yapılan hatalar

Sıralama kodlarındaki hataların çoğu sınır indisleri, geçici değişkenler ve eşitlik koşulları çevresinde toplanır. Aşağıdaki liste bölüm boyunca anlatılan hataları bir arada verir.

Hata	Sonucu	Doğrusu
Dizi sınırını aşmak	Kabarcıkta `j < n` yazılınca `dizi[j+1]` dizinin dışına çıkar.	`j < n - 1 - i`
Anahtarı saklamamak	Eklemelide ilk kaydırma yerleştirilecek öğeyi ezer.	Döngüden önce `gecici = dizi[i]`
En küçüğü değer olarak tutmak	Seçimlide takas edilecek hücre bilinmez.	`enKucuk` indisi tutulur
Kararlılığı bozmak	Birleştirmede `<`, saymalıda baştan sona gezmek eşitlerin sırasını tersine çevirir.	`L[i] <= R[j]`; sondan başa gezmek
Pivotu özyinelemeye katmak	`hizliSirala(dizi, alt, p)` parçayı küçültmeyebilir; sonsuz özyineleme.	`alt..p - 1` ve `p + 1..ust`

Hata	Sonucu	Doğrusu
"Hızlı sıralama hep *n log n*"	Sıralı girdide sabit pivot $\Theta(n^2)$ ve n derinlikli özyineleme verir.	Rastgele pivot ya da üçün ortancası
Heap boyutunu küçültmemek	Sona taşınan öğeler yeniden heap'e karışır.	`yiginlastir(dizi, i, 0)`

Tablo 2.31: Sıralama kodlarında sık yapılan hatalar.

Bu hataların ortak noktası, çoğunun küçük bir örnekle elle izlenince hemen görülmesidir. İyi deneme girdileri şunlardır: sınır hatası için en kısa diziler (bir ve iki öğeli), kararlılık hatası için eşit anahtarlı kayıtlar, hızlı sıralamanın kötü durumu için sıralı bir dizi.

2.13 Özet

- **Sıralama**, bir dizinin öğelerini anahtara göre artan (ya da azalan) düzene koyan bir permütasyon bulmaktır. Algoritmalar **zaman** (en iyi, ortalama, en kötü), **ek bellek**, **kararlılık**, **yerinde** çalışma, **uyarlanabilirlik** ve çevrim içi olma ölçütleriyle karşılaştırılır.
- **Kararlı** algoritma eşit anahtarlı kayıtların göreli sırasını korur; çok anahtarlı sıralama ve basamak sıralaması buna dayanır. Kabarcık, eklemeli, birleştirme, saymalı, basamak ve Timsort kararlıdır; seçimli, Shell, hızlı ve heap sort kararsızdır.
- **Ters çift**, birbirine göre yanlış sıradaki iki öğedir. Bir komşu takas tam olarak bir ters çifti düzeltir; kabarcıkta takas sayısı ve eklemelide kaydırma sayısı ters çift sayısına eşittir. Ters sıralı dizide $n(n-1)/2$ ters çift vardır.
- **Kabarcık sıralaması** komşuları takas eder; her turda kalanların en büyüğü sona gider. Bayraklı sürüm en iyi durumda $\Theta(n)$, aksi hâlde $\Theta(n^2)$'dir.
- **Eklemeli sıralama** anahtarı saklar, büyükleri kaydırır ve anahtarı yerleştirir. Süresi $\Theta(n + I)$'dır; küçük ve neredeyse sıralı dizilerde en iyi seçenektir.
- **Seçimli sıralama** her turda en küçüğü başa alır: her girdide $n(n-1)/2$ karşılaştırma, en çok $n-1$ takas.
- **Shell sıralaması** azalan aralıklarla eklemeli sıralama yapar; süresi aralık dizisine bağlıdır ($n/2$ aralıkla en kötü $\Theta(n^2)$, Hibbard dizisiyle $O(n^{1,5})$).
- **Birleştirme sıralaması** böl, sırala, birleştir adımlarını izler: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$, her girdide. $\Theta(n)$ ek bellek ister ve kararlıdır.
- **Hızlı sıralama** pivot etrafında parçalar; pivot kesin yerine oturur ve birleştirme gerekmez. Ortalama $\Theta(n \log n)$, en kötü $\Theta(n^2)$; rastgele pivot ya da üçün ortancası kötü durumu önler.
- **Heap sort** diziyi $O(n)$'de max-heap'e çevirir, sonra $n-1$ kez kökü sona alıp süzer: en kötü durumda $O(n \log n)$, $O(1)$ ek bellek.
- **Karşılaştırmalı** her sıralama en kötü durumda $\Omega(n \log n)$ karşılaştırma yapar (karar ağacının $n!$ yaprağı). **Saymalı** sıralama $\Theta(n + k)$, **basamak** sıralaması $\Theta(d(n + b))$, **kova** sıralaması düzgün dağılımda beklenen $\Theta(n)$ sürer; karşılaştırma yapmadıkları için bu sınıra bağlı değildirler.

- **Timsort**, sıralı run'ları eklemeli sıralamayla uzatıp birleştirme ile birleştiren kararlı bir karma algoritmadır; Java ve Python'un nesne sıralamasıdır.

2.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

sıralama *sorting* · takas *swap* · yerinde *in-place* · kararlı *stable (sort)* · uyarlanabilir *adaptive (sort)* · ters çift *inversion* · kabarcık sıralaması *bubble sort* · eklemeli sıralama *insertion sort* · seçimli sıralama *selection sort* · Shell sıralaması *shell sort* · böl ve fethet *divide and conquer* · birleştirme *merge* · birleştirme sıralaması *merge sort* · yineleme bağıntısı *recurrence (relation)* · özyineleme ağacı *recursion tree* · ana teorem *master theorem* · hızlı sıralama *quick sort* · pivot *pivot* · bölümleme *partition* · üçün ortancası *median-of-three* · heap *heap* · heap sort *heap sort* · karşılaştırma alt sınırı *comparison sort lower bound* · alt sınır *lower bound* · saymalı sıralama *counting sort* · basamak sıralaması *radix sort* · LSD / MSD basamak sıralaması *LSD / MSD radix sort* · kova sıralaması *bucket sort* · Timsort *Timsort* · karesel *quadratic* · log-doğrusal *linearithmic* · en iyi / ortalama / en kötü durum *best / average / worst case*

2.15 Kendini Dene

1. Eklemeli sıralama [2, 4, 1, 3] dizisini sıralarken kaç karşılaştırma ve kaç kaydırma yapar?

   Cevap

   `i = 1` : 2 > 4 değil; 1 karşılaştırma, kaydırma yok. `i = 2` (1): 4 ve 2 kayar, `j = −1` 'de durulur; 2 karşılaştırma, 2 kaydırma. `i = 3` (3): 4 kayar, 2 > 3 yanlış; 2 karşılaştırma, 1 kaydırma. Toplam 5 karşılaştırma ve 3 kaydırma. Ters çiftler (2, 1), (4, 1), (4, 3)'tür; sayıları kaydırma sayısına eşittir.

2. Bayraklı kabarcık sıralaması [2, 3, 4, 5, 1] dizisinde kaç tur, kaç karşılaştırma ve kaç takas yapar?

Dizi neredeyse sıralı olmasına rağmen sonuç neden kötüdür?

Cevap

Her turda 1 yalnız bir hücre sola ilerler: [2, 3, 4, 1, 5], [2, 3, 1, 4, 5], [2, 1, 3, 4, 5], [1, 2, 3, 4, 5]. Dört turun hepsinde takas olduğu için bayrak hiç erken çıkış sağlamaz. *4 + 3 + 2 + 1 = 10* karşılaştırma ve 4 takas yapılır. Sondaki küçük öğe komşu takaslarla tur başına bir hücre ilerleyebildiği için kabarcık sıralaması bu girdide en kötü karşılaştırma sayısına ulaşır. Eklemeli sıralama aynı diziyi 3 + 4 = 7 karşılaştırmayla sıralar.

3. Kayıtlar (4, a), (2, b), (4, c), (1, d) olsun ve sıralama yalnız sayıya göre yapılsın. Seçimli sıralama bu girdide kararlılığı bozar mı?

Cevap

Bozar. Birinci turda en küçük (1, d), baştaki (4, a) ile takas edilir: (1, d), (2, b), (4, c), (4, a). İkinci turda (2, b) yerindedir. Üçüncü turda (4, a) kesin küçük olmadığı için takas olmaz. Sonuçta (4, c), (4, a)'nın önündedir; girdide ise a önceydi.

4. `parcala(dizi, 0, 4)` (Lomuto, pivot son öğe) [3, 7, 1, 6, 4] dizisinde çalışıyor. Metot ne döndürür ve dizi ne olur?

Cevap

Pivot 4, `i = -1`. `j = 0`: 3 ≤ 4, `i = 0`, kendisiyle takas. `j = 1`: 7 büyük. `j = 2`: 1 ≤ 4, `i = 1`, 7 ile 1 yer değiştirir: [3, 1, 7, 6, 4]. `j = 3`: 6 büyük. Son takas `dizi[2] ↔ dizi[4]`: [3, 1, 4, 6, 7]. Metot 2 döndürür; pivot 4 indis 2'de kesin yerindedir.

5. [1, 2, 3, 4, 5, 6, 7] dizisinden aşağıdan yukarı kurulan max-heap nedir?

Cevap

`i = 2`: 3, büyük çocuğu 7 ile yer değiştirir: [1, 2, 7, 4, 5, 6, 3]. `i = 1`: 2, büyük çocuğu 5 ile: [1, 5, 7, 4, 2, 6, 3]. `i = 0`: 1, büyük çocuğu 7 ile yer değiştirir ve indis 2'ye iner; orada çocukları 6 ve 3'tür, 6 ile yer değiştirir. Sonuç [7, 5, 6, 4, 2, 1, 3]'tür. Sıralı dizi bile heap kurmada epey takas ister; heap sort uyarlanabilir değildir.

6. Birleştirme sıralaması 16 öğeli bir diziyi en kötü durumda en fazla kaç karşılaştırmayla sıralar? Karşılaştırma alt sınırı kaçtır?

Cevap

Dört birleştirme düzeyi vardır: sekiz tane 2 öğeli, dört tane 4 öğeli, iki tane 8 öğeli ve bir tane 16 öğeli birleştirme. *8 · 1 + 4 · 3 + 2 · 7 + 1 · 15 = 49*. Alt sınır *⌈log₂ 16!⌉ = ⌈44,25⌉ = 45*'tir; birleştirme sıralaması bu sınıra yakındır.

7. Java'daki `saymaliSirala` metodu [1, 0, 2, 1, 0] dizisiyle çağrılıyor. Sayma döngüsünden ve önek toplamından sonra `sayac` dizisi nedir?

   Cevap

   `maks = 2`, sayaç üç hücrelidir. Adetler [2, 2, 1]'dir; önek toplamları [2, 4, 5] olur. Son döngü sağdan sola gezer ve çıktı [0, 0, 1, 1, 2] olur. Karşılaştırma yapılmaz; süre $\Theta(n + k)$'dir.

8. [53, 21, 33, 12] dizisi LSD basamak sıralamasıyla sıralanıyor. Birinci geçişten sonra dizi nedir? 53 ile 33'ün sırası neden korunur?

   Cevap

   Birler basamakları 3, 1, 3, 2'dir; kararlı sıralama [21, 12, 53, 33] verir. 53 ile 33'ün birler basamağı eşittir ve girdide 53 önce geldiği için önce kalır. İkinci geçiş onlar basamağına göre [12, 21, 33, 53] dizisini üretir.

9. Bir milyon öğeli bir dizide yalnız on öğe yanlış yerdedir; her biri doğru yerinden en çok birkaç hücre uzaktadır. Kararlı ve hızlı bir sıralama için hangi algoritma seçilmelidir?

   Cevap

   Eklemeli sıralama ya da Timsort. Ters çift sayısı I çok küçüktür; eklemeli sıralamanın süresi $\Theta(n + I)$, yani doğrusal olur. Timsort da uzun sıralı run'ları bulup neredeyse doğrusal sürede biter. Birleştirme sıralaması yine $\Theta(n \log n)$ harcar; seçimli sıralama $\Theta(n^2)$'dir.

2.16 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 2, 6, 7 ve 8: eklemeli ve birleştirme sıralamasını döngü değişmeziyle, heap sort'u, hızlı sıralamanın olasılıksal analizini ve doğrusal zamanlı sıralamayı kanıtlarıyla anlatır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Bölüm 2: aynı algoritmaları Java koduyla verir; karşılaştırma sayılarını deneylerle ölçen bölümleri bu nottaki ölçüm tablosunun genişletilmiş bir hâlidir.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2. baskı, 1998), Bölüm 5.2: Shell sıralamasının aralık dizilerini ve sıralama algoritmalarının tam karşılaştırma sayılarını en ayrıntılı biçimde inceler.
- T. Peters, “listsort.txt” (CPython kaynak kodu): Timsort'un run tespiti, minrun seçimi ve galloping kipini tasarımcısının kendi anlatımıyla açıklar.

2.17 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. E. D. Demaine, C. E. Leiserson, MIT 6.046J *Introduction to Algorithms* ders notları, 2005.
3. C. A. R. Hoare, "Quicksort", *The Computer Journal* 5(1), 1962.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. baskı, Addison-Wesley, 1998.
5. T. Peters, "listsort.txt" (Timsort tasarım notu), CPython, 2002.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
7. D. L. Shell, "A High-Speed Sorting Procedure", *Communications of the ACM* 2(7), 1959.

KISIM I · BÖLÜM 3

Arama Algoritmaları

Bir veri kümesinde istenen değeri bulmak, bilgisayarın en sık yaptığı işlerden biridir. Telefon rehberinde bir isim, veri tabanında bir öğrenci numarası, derleyicinin sembol tablosunda bir değişken adı aranır. Soru hep aynıdır: bu değer kümede var mı, varsa nerede? Bu not, aynı soruyu yanıtlayan yedi yöntemi tanıtır. Önce hiçbir önkoşul istemeyen doğrusal arama ve onun bekçili sürümü ele alınır. Ardından sıralı dizide aralığı her adımda yarıya indiren ikili arama gelir. Hash tablosu ise yeri karşılaştırarak aramak yerine hesaplar. Ara değer, atlayarak ve üstel arama da sıralı dizinin farklı özelliklerinden yararlanır. Her yöntem için kod, adım adım izleme ve maliyet hesabı verilir; not, yöntemlerin aynı dizi üzerinde karşılaştırılması ve doğru yöntemin seçilmesiyle biter.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum03/src`

ÖĞRENME ÇIKTILARI

Temel aramalar
Doğrusal ve ikili aramayı (ikiliyi yinelemeli ve özyinelemeli olarak) Java ile yazmak, bir dizi üzerinde adım adım izlemek; en iyi, ortalama ve en kötü durumlarını hesaplamak.

Hash ile arama
Hash fonksiyonu, kova, çakışma, doğrusal yoklama ve zincirleme kavramlarını açıklamak; yük faktörünün maliyete etkisini yorumlamak.

Sıralı dizide gelişmiş aramalar
Ara değer, atlayarak ve üstel aramanın fikrini, kodunu ve $O(\log \log n)$, $O(\sqrt{n})$, $O(\log i)$ karmaşıklıklarını gerekçelendirmek.

Seçim
Verinin sıralı olup olmamasına, dağılımına ve boyutuna göre uygun arama algoritmasını seçmek; yaygın uygulama hatalarından kaçınmak.

3.1 Arama problemi

Arama, bilgisayar biliminin en eski ve en yaygın problemlerinden biridir. Kullanıcı adını denetleyen giriş ekranı, ürün fiyatını bulan kasa programı ve yazım denetimi yapan metin düzenleyici aynı temel işi yapar. Bu işin ne kadar hızlı yapıldığı, programın bütün başarımını belirleyebilir. Arama bir döngünün içinde milyonlarca kez çalışıyorsa, her aramadan kazanılan birkaç mikrosaniye toplamda saatlere dönüşür.

3.1.1 Problemin tanımı

Arama algoritmaları bir veri kümesinde **istenen değeri** bulmak için kullanılır. Algoritma verileri bir **arama ölçütüne** göre tarar ve ölçüte uyan öğeyi bulmaya çalışır. Ölçüt çoğunlukla bir **anahtardır** (*key*). Öğrenci kayıtlarında anahtar öğrenci numarası, sözlükte kelimenin kendisi, telefon rehberinde kişinin adıdır. Bu notta problemi en yalın hâliyle ele alıyoruz: öğeler bir tam sayı dizisindedir ve anahtar öğenin kendi değeridir. Gerçek uygulamalarda dizinin her hücresi bir kayıt tutar ve karşılaştırma kaydın anahtar alanıyla yapılır; algoritmaların mantığı değişmez.

> **Tanım 3.1**
>
> **Arama problemi** (*search problem*): Girdi, n öğeli bir `dizi` ve bir `aranan` değerdir. Çıktı, `dizi[i] == aranan` koşulunu sağlayan bir **indis** i'dir. Böyle bir indis yoksa çıktı -1'dir; bu duruma **başarısız arama** denir.

Tanımın iki ayrıntısı önemlidir. Birincisi, çıktı değerin kendisi değil, *konumudur*. Aranan değer zaten bilinir; asıl soru onun dizide nerede durduğudur. Konum bilinince o hücredeki kaydın diğer alanlarına da ulaşılabilir. İkincisi, başarısız arama da geçerli bir sonuçtur. Java'da dizi indisleri 0'dan başladığı için -1 hiçbir geçerli indisle karışmaz ve "bulunamadı" anlamını güvenle taşır.

Dizide aranan değer birden çok kez geçiyorsa tanım, bunlardan *herhangi birinin* indisini kabul eder. Hangi eşleşmenin döneceği algoritmaya bağlıdır. Doğrusal arama baştan taradığı için ilk eşleşmeyi bulur; ikili arama ise ortadan başladığı için rastgele görünen bir eşleşmeyi verebilir. İlk ya da son eşleşme özellikle isteniyorsa algoritmanın buna göre uyarlanması gerekir.

3.1.2 Maliyet ölçüsü: yoklama sayısı

Bir arama algoritmasının ne kadar hızlı olduğunu saniyeyle ölçmek yanıltıcıdır, çünkü süre işlemciye, dile ve derleyiciye göre değişir. Bunun yerine, algoritmanın doğasını yansıtan bir işlem sayılır. Arama algoritmalarında süreyi belirleyen işlem, aranan değerin dizideki bir öğeyle karşılaştırılmasıdır.

Tanım 3.2

Bir dizi hücresine bakıp aranan değerle karşılaştırma işlemine **yoklama** (*probe*) denir. Bir aramanın maliyeti, yaptığı yoklama sayısıyla ölçülür. Bu notta verilen bütün karmaşıklıklar bu sayının n'ye göre büyümesini anlatır.

Yoklama sayısını seçmenin gerekçesi basittir. Döngü sayaçlarını artırmak, sınırları güncellemek gibi diğer işler her yoklamaya sabit sayıda eşlik eder. Dolayısıyla toplam süre, yoklama sayısıyla bir sabitin çarpımı kadardır ve büyüme mertebesi aynı kalır. Bir hash tablosunda yoklama, bir kovaya bakıp oradaki anahtarı aranan anahtarla karşılaştırmaktır; tanım oraya da aynen uyar.

Maliyet aynı n için bile girdiden girdiye değişir. Aranan değer dizinin başındaysa doğrusal arama hemen biter, sonundaysa bütün dizide gezinir. Bu yüzden her algoritma için üç durum ayrı ayrı incelenir. **En iyi durum** (*best case*) en az yoklama gerektiren girdidir. **En kötü durum** (*worst case*) en çok yoklama gerektiren girdidir ve algoritmanın verebileceği bir güvenceyi gösterir. **Ortalama durum** (*average case*) ise girdilerin belli bir olasılık dağılımına göre geldiği varsayılarak hesaplanan beklenen yoklama sayısıdır. En kötü durum çoğu zaman en önemli ölçüdür, çünkü "hiçbir girdide bundan yavaş olmaz" demeyi sağlar.

3.1.3 Arama algoritmalarının çeşitleri

Farklı arama algoritmaları farklı ilkelerle çalışır ve verinin farklı özelliklerini önkoşul olarak ister. Bir algoritma ne kadar çok önkoşul isterse, o önkoşul sağlandığında o kadar hızlı çalışma eğilimindedir. Tablo 3.1 bu notta incelenen altı yöntemi önkoşulları ve ilkeleriyle özetler.

Algoritma	Önkoşul	Çalışma ilkesi
Doğrusal (*linear search*)	yok	Verileri tek tek karşılaştırarak arar.
İkili (*binary search*)	sıralı dizi	Veri kümesini ikiye bölerek arama alanını daraltır.
Hash (*hash search*)	hash tablosu	Veriler bir hash tablosuna yerleştirilir; arama, anahtarın kovasında yapılır.
Ara değer (*interpolation search*)	sıralı, düzgün dağılımlı	Konumu, değerlerin oranından tahmin eder.
Atlayarak (*jump search*)	sıralı dizi	$\sqrt{n}$ büyüklüğünde bloklar atlar, sonra blok içinde doğrusal arar.
Üstel (*exponential search*)	sıralı dizi	Sınırı 1, 2, 4, 8, … diye büyütür, sonra ikili arama yapar.

Tablo 3.1: Arama algoritmalarının önkoşulları ve çalışma ilkeleri.

Tablodaki önkoşullar üç gruba ayrılır. Doğrusal arama hiçbir şey istemez; dizi sırasız, hatta bir bağlı liste bile olabilir. Hash ile arama verinin önceden özel bir yapıya, hash tablosuna yerleştirilmesini ister. Geriye kalan dört yöntem sıralı dizi ister. Sıralı dizinin sağladığı bilgi şudur: bir hücredeki değer aranandan küçükse, aranan o hücrenin solunda olamaz. Bu tek karşılaştırma, dizinin büyük bir kısmını bir kerede elemeyi sağlar. İkili, ara değer, atlayarak ve üstel arama bu bilgiyi farklı biçimlerde kullanır.

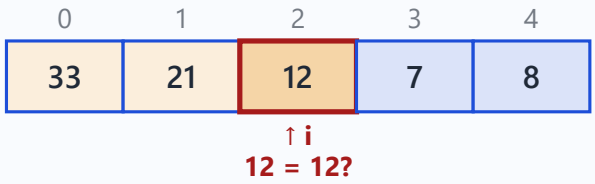
Sıralı olma önkoşulunun bir bedeli vardır. Diziyi sıralamak en iyi karşılaştırmalı algoritmalarla *O(n log n)* zaman alır. Tek bir arama için diziyi sıralamak, doğrudan doğrusal arama yapmaktan pahalıdır. Sıralama ancak aynı dizide çok sayıda arama yapılacaksa kendini amorti eder.

3.2 Doğrusal arama

Doğrusal arama, akla gelen ilk ve en sade yöntemdir. Önkoşul istemediği için her durumda kullanılabilir. Bu genellik, hızdan ödün vermenin karşılığıdır.

3.2.1 Fikir

Doğrusal arama (*linear search*) listedeki her öğeyi sırayla kontrol eder. Bir alışveriş listesindeki ürünleri tek tek gözden geçirmek ya da kayıp bir eşyayı bulmak için odayı köşe köşe taramak aynı fikre dayanır. Aranan değer listenin başından başlanarak her öğeyle karşılaştırılır. Eşleşme bulunursa o öğenin indisi döndürülür. Liste bitmiş ve eşleşme çıkmamışsa sonuç *−1*'dir.



Şekil 3.1: Doğrusal aramada 12 aranıyor. 33 ve 21 kontrol edilmiş, eşleşme yok; şimdi `i = 2` konumundaki 12 karşılaştırılıyor.

Bazen aranan bilgiyi bulmanın en iyi yolu gerçekten tek tek bakmaktır. Dizi sırasızsa başka seçenek de yoktur. Sırasız bir dizide herhangi bir hücreye bakmak, diğer hücreler hakkında hiçbir bilgi vermez. 33'ün aranan olmadığını öğrenmek, 21'in aranan olup olmadığı konusunda hiçbir şey söylemez. Dolayısıyla başarısız bir aramanın “yok” diyebilmesi için her hücreye en az bir kez bakmak zorunludur. Bu basit gözlem, sırasız veride hiçbir algoritmanın en kötü durumda *n*'den az yoklama yapamayacağını gösterir.

3.2.2 Kod: dogrusalArama()

Doğrusal aramanın Java kodu şöyledir:

```
int dogrusalArama(int[] dizi, int n,
                  int aranan) {
  int i = 0;
  while (i < n && dizi[i] != aranan)
    i++;
  if (i < n)
    return i;      // bulundu: indis
  else
    return -1;     // bulunamadı
}
```

Metot üç parametre alır: `dizi` aranacak öğeleri, `n` dizinin kullanılan boyutunu, `aranan` ise aranacak değeri tutar. Dönüş değeri aranan öğenin indisidir; öğe bulunamazsa *−1*'dir. Boyutun ayrı bir parametre olarak verilmesi, dizinin yalnız ilk `n` hücresinin dolu olduğu durumları da kapsar.

`int i = 0;` satırı dizinin başından başlamayı sağlar. `while` döngüsünün koşulu iki parçalıdır: `i < n` dizinin sonuna gelinmediğini, `dizi[i] != aranan` ise o hücrede aranan değerin bulunmadığını denetler. İki koşul da doğru olduğu sürece döngü gövdesi çalışır ve `i` bir artar. Döngü iki nedenden biriyle durur: ya `i`, `n`'ye ulaşmıştır, ya da `dizi[i]` aranana eşittir. Döngüden sonraki `if` bu iki durumu ayırır. `i < n` doğruysa döngü bir eşleşme yüzünden durmuştur ve `i` eşleşmenin indisidir. Yanlışsa dizi baştan sona taranmış, eşleşme çıkmamıştır; metot *−1* döndürür.

Koşulların sırası önemlidir. Java'nın `&&` işleci **kısa devre** çalışır: sol taraf yanlışsa sağ taraf hiç hesaplanmaz. `i == n` olduğunda `i < n` yanlıştır ve `dizi[n]` ifadesine hiç erişilmez. Koşullar ters yazılsaydı, yani `dizi[i] != aranan && i < n` denseydi, başarısız bir aramanın son adımında `dizi[n]` okunmaya çalışılır ve `ArrayIndexOutOfBoundsException` fırlatılırdı.

> **Sık yapılan hata**
>
> Döngüden sonraki denetimi `if (dizi[i] == aranan)` biçiminde yazmak. Başarısız aramada döngü `i = n` ile biter ve bu denetim dizi dışını okur. Döngünün neden durduğunu anlamanın güvenli yolu, önce sınırı (`i < n`) sınamaktır.

3.2.3 En iyi ve en kötü durum

Algoritmanın davranışı, on öğeli sırasız bir örnek dizi üzerinde şöyle görülür:

> `dizi = {33, 21, 12, 7, 8, 1, 3, 44, 56, 18}` , *n = 10*

Örnek 3.1

En iyi durum: 33 aranıyor. `i = 0` iken `dizi[0] = 33` aranana eşittir. Döngü koşulunun ikinci parçası ilk sınamada yanlış çıkar ve döngü gövdesi hiç çalışmaz. `i = 0 < n` olduğu için metot 0 döndürür. Yalnız **1 karşılaştırma** yapılmıştır.

En kötü durum: 17 aranıyor. 17 dizide yoktur. Algoritma bunu ancak her öğeye bakarak anlayabilir:

1. `i = 0`: `dizi[0] = 33 ≠ 17` → `i = 1`.
2. `i = 1`: `dizi[1] = 21 ≠ 17` → `i = 2`.
3. `i = 2`: `dizi[2] = 12 ≠ 17` → `i = 3`.
4. `i = 3`: `dizi[3] = 7 ≠ 17` → `i = 4`.
5. `i = 4`: `dizi[4] = 8 ≠ 17` → `i = 5`.
6. `i = 5`: `dizi[5] = 1 ≠ 17` → `i = 6`.
7. `i = 6`: `dizi[6] = 3 ≠ 17` → `i = 7`.
8. `i = 7`: `dizi[7] = 44 ≠ 17` → `i = 8`.
9. `i = 8`: `dizi[8] = 56 ≠ 17` → `i = 9`.
10. `i = 9`: `dizi[9] = 18 ≠ 17` → `i = 10`.
11. `i = 10 = n`: `i < n` yanlış, döngü biter. `if` denetimi de yanlış çıkar ve metot *−1* döndürür.

Değer karşılaştırması *n = 10* kez yapılmıştır. En kötü durum *O(n)*'dir.



Şekil 3.2: Doğrusal aramanın iki ucu. Üstte en iyi durum: 33 ilk hücrede bulunur. Altta en kötü durum: 17 aranırken bütün hücreler kontrol edilmiş, eşleşme çıkmamıştır.

En kötü durum iki farklı girdiyle ortaya çıkar: aranan dizide yoksa ya da yalnız son hücrede bulunuyorsa. İkisinde de *n* değer karşılaştırması yapılır. Aradaki tek fark, döngünün son sınamasıdır. Aranan son hücredeyse döngü `i = n − 1` konumunda eşleşmeyle durur. Aranan yoksa bir tur daha döner ve `i = n` olunca sınır koşuluyla durur.

3.2.4 Kod izleme: ortalama bir arama

En iyi ve en kötü durum iki uçtur. Tipik bir arama için aynı dizide 1 aranıyor. 1, dizinin ortasına yakın bir yerde, 5. indistedir. Tablo 3.2, `dogrusalArama(dizi, 10, 1)` çağrısında döngü koşulunun her sınanışını gösterir.

Sınama	`i`	`i < n`	`dizi[i]`	`dizi[i] != 1`	Sonuç
1	0	doğru	33	doğru	`i++`
2	1	doğru	21	doğru	`i++`
3	2	doğru	12	doğru	`i++`
4	3	doğru	7	doğru	`i++`
5	4	doğru	8	doğru	`i++`
6	5	doğru	1	yanlış	döngüden çık

Tablo 3.2: `dogrusalArama(dizi, 10, 1)` izlemesi.

Altıncı sınamada `dizi[5] = 1` aranana eşittir; `while` koşulu yanlış olur ve döngüden çıkılır. Ardından `i = 5 < n` doğrudur ve metot 5 döndürür. Toplam 6 değer karşılaştırması yapılmıştır. Genel kural açıktır: k indisindeki bir öğe $k + 1$ karşılaştırmayla bulunur.

3.2.5 Karmaşıklık ve değerlendirme

En iyi ve en kötü durumun maliyeti kolayca okunur: biri 1, öteki n karşılaştırmadır. Ortalama durum için bir varsayım gerekir. En yaygın varsayım, aramanın başarılı olduğu ve aranan değerin her konumda bulunma olasılığının eşit olduğudur.

> **Teorem 3.1 (Başarılı doğrusal aramanın ortalama maliyeti)**
>
> Aranan değer dizide bulunuyor ve n konumdan her birinde olma olasılığı $1/n$ ise, doğrusal aramanın ortalama karşılaştırma sayısı $(n + 1)/2$'dir.

İspat. i. konumdaki öğe (1'den sayarak) i karşılaştırmayla bulunur. Beklenen değer, her konumun maliyetiyle olasılığının çarpımlarının toplamıdır: $\Sigma_{i=1}^{n}\, i \cdot (1/n) = (1/n) \cdot n(n + 1)/2 = (n + 1)/2$. Burada $1 + 2 + \ldots + n = n(n + 1)/2$ özdeşliği kullanılmıştır. □

$n = 10$ için ortalama $11/2 = 5{,}5$ karşılaştırmadır. Sonuç sezgiyle de uyumludur: aranan öğe ortalamada dizinin ortasındadır, dolayısıyla yaklaşık yarısına bakılır. Başarısız aramada ise her zaman n karşılaştırma yapılır. Tablo 3.3 üç durumu bir araya getirir.

Durum	Karşılaştırma	Mertebe	Örnek ($n = 10$)
En iyi (ilk öğe)	1	$O(1)$	33 → 1 karşılaştırma
Ortalama (başarılı, eşit olasılık)	$(n + 1)/2$	$O(n)$	ortalama 5,5

Durum	Karşılaştırma	Mertebe	Örnek ($n = 10$)
En kötü / başarısız	n	$O(n)$	17 → 10 karşılaştırma

Tablo 3.3: Doğrusal aramanın maliyeti.

Ortalama durumun da $O(n)$ olması önemli bir sonuçtur. Ortalama maliyet en kötü durumun yalnız yarısıdır; yarıya inmek mertebeyi değiştirmez. Dizi on kat büyürse ortalama arama da on kat uzar. Ek bellek açısından durum iyidir: algoritma yalnız `i` değişkenini kullanır, ek bellek $O(1)$'dir.

Doğrusal aramanın avantajları şöyle sıralanabilir:

- Kodlaması ve anlaşılması kolaydır; karmaşık bir veri yapısı gerektirmez.
- Ön hazırlık istemez: dizinin sıralı olması gerekmez.
- Küçük veri kümelerinde idealdir. Kişisel bir rehber listesinde bir numara aramak için daha karmaşık bir yönteme gerek yoktur.
- Bağlı liste gibi yalnız sırayla erişilebilen yapılarda da çalışır. İkili arama bir bağlı listede verimli çalışamaz, çünkü ortadaki öğeye ulaşmak bile $O(n)$ adım ister.

Dezavantajları da açıktır:

- Büyük veride yavaştır. 10^6 öğeli bir dizide başarılı bir arama ortalama 500.000 karşılaştırma yapar.
- Dizi sıralı olsa bile bu bilgiden yararlanmaz. Sıralı bir dizide aranandan büyük bir değere rastlandığında arama durdurulabilirdi; temel algoritma bunu yapmaz.
- Aynı dizide çok sayıda sorgu yapılacaksa diziyi bir kez sıralayıp ikili arama kullanmak çok daha ucuzdur.

3.2.6 Bekçili doğrusal arama

Doğrusal aramanın döngüsüne dikkatle bakınca her turda *iki* karşılaştırma yapıldığı görülür: `i < n` ve `dizi[i] != aranan`. Bunlardan yalnız ikincisi aramanın asıl işidir. Birincisi, dizinin dışına taşmamak için konmuş bir güvenlik denetimidir. Bu denetimi döngüden çıkarmanın zekice bir yolu vardır.

> **Tanım 3.3**
>
> **Bekçi** (*sentinel*), döngünün mutlaka durmasını sağlamak için dizinin sonuna yerleştirilen özel değerdir. Bekçili doğrusal aramada aranan değerin kendisi son hücreye yazılır. Böylece döngü, en geç son hücrede eşleşme bulur ve `i < n` sınamasına gerek kalmaz.

Bekçiye "gözcü" de denir; metodun adı `gozcuIleAra` buradan gelir:

```
int gozcuIleAra(int[] dizi, int n,
                int aranan) {
  int sonEleman = dizi[n - 1];
  dizi[n - 1] = aranan;     // bekçi
  int i = 0;
  while (dizi[i] != aranan)
    i++;
  dizi[n - 1] = sonEleman;  // geri koy
  if (i < n - 1 || sonEleman == aranan)
    return i;
  return -1;
}
```

Metot önce son öğeyi `sonEleman` değişkenine kaydeder ve son hücreye aranan değeri yazar. Bu andan sonra dizide aranan değerin en az bir kopyası vardır. Döngü artık yalnız `dizi[i] != aranan` koşulunu sınar; `i` sınırı aşamaz, çünkü en geç `n - 1` konumunda eşleşme bulunur. Döngü bitince son hücreye eski değer geri yazılır, dizi ilk hâline döner.

Son adım dikkat ister. Döngünün durduğu yer iki anlama gelebilir. `i < n - 1` ise eşleşme bekçiden önce, gerçek bir öğede bulunmuştur. `i == n - 1` ise döngü bekçide durmuştur. Bu durumda da aranan değer bulunmuş olabilir: dizinin gerçek son öğesi aranana eşitse. Bu yüzden `if` koşulunun ikinci parçası `sonEleman == aranan`'dır. İki koşuldan biri doğruysa `i` döner, ikisi de yanlışsa *−1*.

Kazanç sayılarla şöyledir. Klasik döngü, başarısız bir aramada *n* değer karşılaştırması ve *n + 1* sınır sınaması yapar; toplam yaklaşık *2n* karşılaştırma eder. Bekçili döngü *n* değer karşılaştırması yapar. Döngüden sonra iki karşılaştırma daha gelir: `i < n - 1` ve `sonEleman == aranan`. Toplam yaklaşık *n + 2*'dir. Karşılaştırma sayısı neredeyse yarıya iner, ama mertebe yine *O(n)*'dir. Bu, sabit çarpanı küçülten bir iyileştirmedir.

> **Sık yapılan hata**
>
> Bekçili aramayı salt okunur bir dizide ya da aynı anda başka iş parçacıklarının okuduğu bir dizide kullanmak. Arama süresince son hücre geçici olarak değişir. Başka bir iş parçacığı o sırada diziyi okursa yanlış bir değer görür. Son öğenin geri yazılmasının unutulması da diziyi kalıcı olarak bozar.

3.2.7 Kod izleme: gozcuIleAra()

Bekçili aramanın başarısız bir aramadaki izlemesi: `dizi = [7, 3, 9, 4, 6]`, `n = 5`, aranan 5. 5 dizide yoktur.

Örnek 3.2

1. **Hazırlık.** Son öğe 6, `sonEleman` 'da saklanır. Son hücreye aranan 5 bekçi olarak yazılır; dizi geçici olarak `[7, 3, 9, 4, 5]` olur. `i = 0`.
2. `dizi[0] = 7 ≠ 5` → `i = 1`. Bu turda yalnız bir karşılaştırma vardır; `i < n` sınaması yapılmaz.
3. `dizi[1] = 3 ≠ 5` → `i = 2`.
4. `dizi[2] = 9 ≠ 5` → `i = 3`.
5. `dizi[3] = 4 ≠ 5` → `i = 4`.
6. `dizi[4] = 5` aranana eşittir, döngü durur. Ancak bulunan gerçek bir öğe değil, bekçidir.
7. Son öğe geri konur: dizi yine `[7, 3, 9, 4, 6]` 'dır.
8. `i < n - 1` sınaması *4 < 4* olduğu için yanlıştır. `sonEleman == aranan` sınaması da *6 ≠ 5* olduğu için yanlıştır. Metot *−1* döndürür.

Toplam karşılaştırma sayısı: döngüde 5, sonda 2, yani *7 = n + 2*. Klasik döngü aynı aramada 5 değer karşılaştırması ve 6 sınır sınaması yapardı; yaklaşık *2n* sayılırsa 10.



Şekil 3.3: Bekçili arama. (a) Son hücreye bekçi olarak aranan 5 yazılmıştır; asıl son öğe 6, `sonEleman` 'da bekler. (b) Döngü bekçide durmuş, 6 geri konmuştur; iki son sınama da yanlış çıktığı için sonuç *−1*'dir ve toplam 7 karşılaştırma yapılmıştır.

İkinci `if` koşulunun neden gerektiğini farklı bir örnek gösterir: `dizi = [12, 5, 9, 20, 7]` ve aranan 7. Burada aranan değer gerçekten son hücrededir. Bekçi olarak yine 7 yazılır, döngü `i = 4` 'te durur. `i < n - 1` yanlıştır, ama `sonEleman == aranan` doğrudur ve metot doğru olarak 4 döndürür.

Bu koşul olmasaydı dizinin gerçek son öğesi hiçbir zaman bulunamazdı.

3.3 İkili arama

Doğrusal arama her yoklamada dizinin yalnız bir hücresini eler. Dizi sıralıysa tek bir yoklama çok daha fazlasını söyler. Ortadaki öğe aranandan küçükse, sıralılık gereği onun solundaki bütün öğeler de aranandan küçüktür ve hiçbiri aranan olamaz. Tek yoklama dizinin yarısını eler. İkili arama bu gözlem üzerine kurulur.

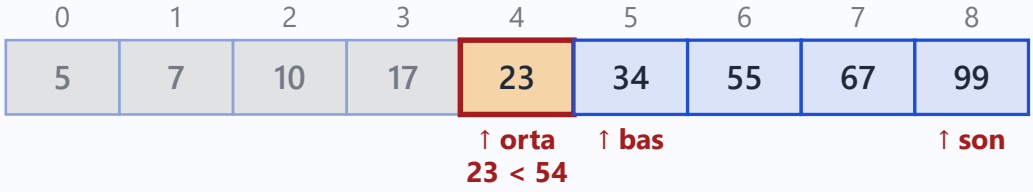
3.3.1 Fikir

Kütüphanede kitap aramak iyi bir benzetmedir. Kitaplar yazarın soyadına göre alfabetik sıralanmıştır. Bütün rafları baştan tek tek gezmek yerine önce orta bölüme gidilir. Aranan kitap ortadaki rafın solunda kalıyorsa sol taraftaki raflarda, sağında kalıyorsa sağ taraftaki raflarda aramaya devam edilir. Kalan bölümde de yine ortaya gidilir. Her karşılaştırma, aranması gereken kısmı yarıya indirir.

Tanım 3.4

İkili arama (*binary search*), sıralı bir dizide arama aralığının ortasındaki öğeyi aranan değerle karşılaştırır. Değerler eşitse arama biter. Aranan daha küçükse aralığın sol yarısıyla, daha büyükse sağ yarısıyla devam edilir. Aralık boşalırsa aranan dizide yoktur.

İkili aramanın iki önkoşulu vardır. Birincisi, dizi sıralı olmalıdır; bu not boyunca sıralı derken küçükten büyüğe sıralı kastediliyor. İkincisi, öğelere indisle *O(1)* sürede erişilebilmelidir. Bu ikinci koşul dizide sağlanır, bağlı listede sağlanmaz. Bağlı listede ortadaki öğeye ulaşmak için baştan itibaren yarı liste yürümek gerekir. Bu durumda tek bir "orta" hesabı bile doğrusal arama kadar pahalı olur.



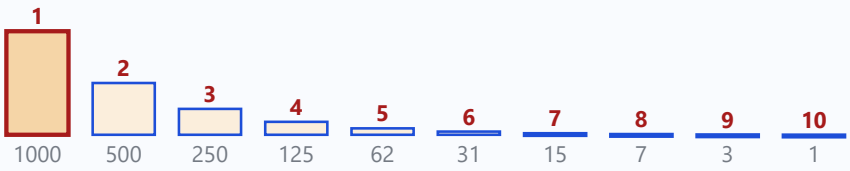
Şekil 3.4: Sıralı bir dizide 54 aranıyor. Orta öğe 23, 54'ten küçüktür; bu yüzden 23 ve solundaki bütün öğeler (gri) tek karşılaştırmada elenir. Arama `bas = 5` ile `son = 8` arasında sürer.

3.3.2 İkili arama neden hızlı?

İkili aramanın gücü, aralığın her adımda yarıya inmesinden gelir. 1000 öğeli bir listede doğrusal arama ortalama 500 kontrol yapar. İkili arama ise en fazla 10 adımda aramayı tamamlar. Aralık boyları her karşılaştırmadan sonra şöyle küçülür:

> 1000 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1

Bu sırada on aralık boyu vardır ve her aralık boyu bir karşılaştırmaya karşılık gelir: toplam 10 karşılaştırma. Yarıya inerken kesirli kısım atılır, çünkü ortadaki öğe karşılaştırıldıktan sonra artık aralıkta değildir. Örneğin 125 öğeli bir aralıkta ortadaki öğe çıkınca iki yanda 62'şer öğe kalır.



Şekil 3.5: Aralık boyu her karşılaştırmada yarıya iner. Çubukların üstündeki sayı adım numarasıdır; 10 adımda 1000 öğelik aralık tek öğeye iner.

Bu küçülmenin ne kadar etkili olduğu büyük *n* değerlerinde ortaya çıkar. Tablo 3.4 iki yöntemin en kötü durumdaki karşılaştırma sayılarını yan yana koyar.

n	Doğrusal (en kötü)	İkili (en kötü)
10	10	4
1.000	1.000	10
10^6	10^6	20
10^9	10^9	30

Tablo 3.4: En kötü durumda karşılaştırma sayıları.

İkili arama en fazla $\lfloor \log_2 n \rfloor + 1$ karşılaştırma yapar. Bir milyar öğeli bir dizide 30 karşılaştırma yeterlidir. Veri 1000 kat büyüdüğünde ikili aramaya yalnız yaklaşık 10 adım eklenir, çünkü $2^{10} = 1024 \approx 1000$'dir. Doğrusal arama ise 1000 kat yavaşlar. Bu tür büyümeye **logaritmik** (*logarithmic*) büyüme denir.

3.3.3 Kod: yinelemeli ikiliArama()

İkili aramanın döngüyle yazılmış (yinelemeli) Java kodu şöyledir:

```
int ikiliArama(int[] dizi, int n, int aranan) {
  int bas = 0, son = n - 1;
  while (bas <= son) {
    int orta = (bas + son) / 2;
    if (aranan < dizi[orta])
      son = orta - 1;      // sol yarı
    else if (aranan > dizi[orta])
      bas = orta + 1;      // sağ yarı
    else
      return orta;         // bulundu
  }
  return -1;
}
```

`bas` ve `son`, arama yapılacak aralığın alt ve üst sınırlarıdır; ikisi de aralığa dahildir. Başlangıçta aralık bütün dizidir: `bas = 0`, `son = n - 1`. `while (bas <= son)` koşulu aralık boş olmadığı sürece döngüyü sürdürür. `bas == son` olduğunda aralıkta tek öğe vardır ve bu öğe de sınanmalıdır; koşulun `<=` olması bu yüzdendir.

Her turda `orta`, aralığın ortasındaki öğenin indisi olarak hesaplanır. Tam sayı bölmesi kesirli kısmı attığı için aralıkta çift sayıda öğe varsa iki ortadan soldaki seçilir. Ardından üç yol vardır. Aranan `dizi[orta]`'dan küçükse aranan, varsa, ortanın solundadır; `son = orta - 1` ile sol yarıya geçilir. Aranan büyükse `bas = orta + 1` ile sağ yarıya geçilir. Eşitse `orta` döndürülür. Döngü bir eşleşme bulunmadan biterse aralık boşalmıştır (`bas > son`) ve metot *−1* döndürür.

Sınırların `orta - 1` ve `orta + 1` ile güncellenmesi kritiktir. Orta öğe az önce sınanmış ve aranan olmadığı anlaşılmıştır; bu yüzden yeni aralığa alınmaz. Bu kural aynı zamanda aralığın her turda en az bir öğe küçülmesini, dolayısıyla döngünün mutlaka bitmesini güvenceye alır.

3.3.4 Özyinelemeli sürüm

İkili arama doğal olarak özyinelemeli bir yapıdadır: aralığın bir yarısında arama, aynı problemin daha küçük bir örneğidir. Özyinelemeli Java kodu şöyledir:

```
int ara(int[] dizi, int bas,
        int son, int aranan) {
  if (bas > son) return -1;  // boş aralık
  int orta = (bas + son) / 2;
  if (dizi[orta] == aranan) return orta;
  if (dizi[orta] < aranan)           // sağ yarı
    return ara(dizi, orta + 1, son, aranan);
  return ara(dizi, bas, orta - 1, aranan);
}
```

İlk çağrı `ara(dizi, 0, n - 1, aranan)` biçimindedir. Metot önce taban durumunu denetler: `bas > son` ise aralık boştur ve *−1* döner. Değilse orta öğe hesaplanır ve aranana eşitse indisi döner. Aksi hâlde aranan büyükse sağ yarıda, küçükse sol yarıda aynı metot yeniden çağrılır ve onun sonucu olduğu gibi döndürülür. Yinelemeli sürümdeki `bas` ve `son` güncellemelerinin yerini burada yeni çağrının parametreleri alır.

Algoritma tasarımı açısından ikili arama, **böl ve fethet** (*divide and conquer*) yaklaşımının en sade örneğidir. Böl adımı ortadaki öğeyi seçmektir. Fethet adımı yalnız bir yarıda devam etmektir; öteki yarıya hiç dokunulmaz. Birleştir adımı yoktur, çünkü alt problemin cevabı doğrudan problemin cevabıdır. Tek dallı olduğu için bu yaklaşıma **azalt ve fethet** (*decrease and conquer*) de denir. Maliyet şu yineleme bağıntısıyla anlatılır:

> $T(n) = T(n/2) + 1$

Bağıntı, *n* öğeli bir aramanın bir karşılaştırma ile *n/2* öğeli bir arama yapmasını ifade eder. Çözümü $T(n) = O(\log n)$'dir.

İki sürüm aynı karşılaştırmaları yapar, ama bellek kullanımları farklıdır. Yinelemeli sürüm yalnız birkaç değişken kullanır: ek bellek $O(1)$'dir. Özyinelemeli sürümde her çağrı çağrı yığınında bir çerçeve açar ve çağrılar iç içe en fazla $\lfloor \log_2 n \rfloor + 1$ derinliğe iner. Ek bellek $O(\log n)$'dir. Bu değer küçüktür; bir milyar öğede bile 30 çerçeve yeter. Yine de yinelemeli sürüm hem bellekte hem de çağrı maliyetinde daha hafiftir ve pratikte daha çok tercih edilir.

3.3.5 İkili arama ve dengeli ikili arama ağacı

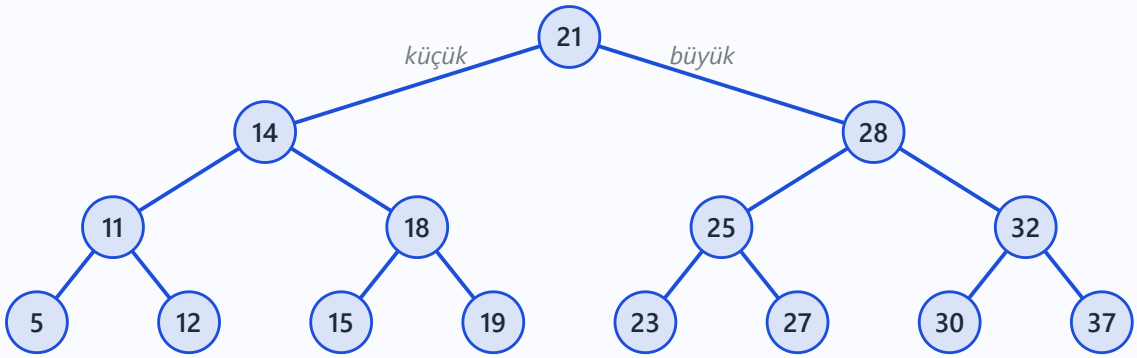
İkili aramanın yaptığı karşılaştırmalar bir ağaç biçiminde çizilebilir. Bu çizim, algoritmanın neden en fazla $\lfloor \log_2 n \rfloor + 1$ adım attığını gözle görmeyi sağlar. Örnek olarak 15 öğeli bir sıralı dizi alınsın:

> `5, 11, 12, 14, 15, 18, 19, 21, 23, 25, 27, 28, 30, 32, 37`

Örnek 3.3

1. **Kök.** Bütün dizinin ortası 7. indisteki 21'dir. İkili aramanın ilk karşılaştırması her zaman 21 ile yapılır; 21 ağacın köküdür.
2. **İkinci düzey.** Aranan 21'den küçükse arama 0–6 indisleri arasında sürer; bu aralığın ortası 3. indisteki 14'tür. Büyükse 8–14 aralığında sürer; ortası 11. indisteki 28'dir. 14 kökün sol, 28 sağ çocuğu olur.
3. **Üçüncü düzey.** Her alt aralığın ortası bir sonraki düzeyi oluşturur: 0–2 aralığından 11, 4–6'dan 18, 8–10'dan 25, 12–14'ten 32.
4. **Yapraklar.** Kalan sekiz öğe (5, 12, 15, 19, 23, 27, 30, 37) tek öğeli aralıklardır ve yaprak olur.

Ağaç 4 düzeylidir, yüksekliği 3'tür. $\lfloor \log_2 15 \rfloor + 1 = 3 + 1 = 4$ olduğundan her arama en fazla 4 karşılaştırma yapar. Başarılı aramalarda ortalama karşılaştırma sayısı $(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 8 \cdot 4)/15 = 49/15 \approx 3{,}27$'dir.



Şekil 3.6: 15 öğeli sıralı diziden elde edilen dengeli ikili arama ağacı. Kökün solunda 21'den küçükler, sağında büyükler vardır. Dört düzey, en fazla dört karşılaştırma demektir.

Bu ağaç bir **ikili arama ağacıdır**: her düğümün sol alt ağacındaki değerler düğümden küçük, sağ alt ağacındakiler büyüktür. Üstelik mümkün olan en dengeli ağaçtır; her düzey tamamen doludur. İkili arama, sıralı dizi üzerinde bu en iyi ağaçta kökten aşağı yürümekle aynı şeydir. Bir ağaçta arama maliyeti, aranan düğümün derinliğine bir eklenerek bulunur. En kötü durum ağacın yüksekliğine bağlıdır. n düğümlü dengeli bir ağacın yüksekliği $\lfloor \log_2 n \rfloor$'dir. Ağaç dengesiz kurulsaydı, örneğin her düğümün yalnız sağ çocuğu olsaydı, yükseklik $n - 1$ olurdu ve arama doğrusal aramaya dönerdi.

3.3.6 Örtük ağaçta arama

Ağacın bellekte kurulması gerekmez. Sıralı dizi ağacın kendisidir; `orta` hesabı ağacı **örtük** olarak gezer. Bir düğümün sol çocuğu, sol yarı aralığın ortasıdır; sağ çocuğu sağ yarının ortasıdır. İkinci bir sıralı dizi bunu gösterir. Dizi yine 15 öğelidir:

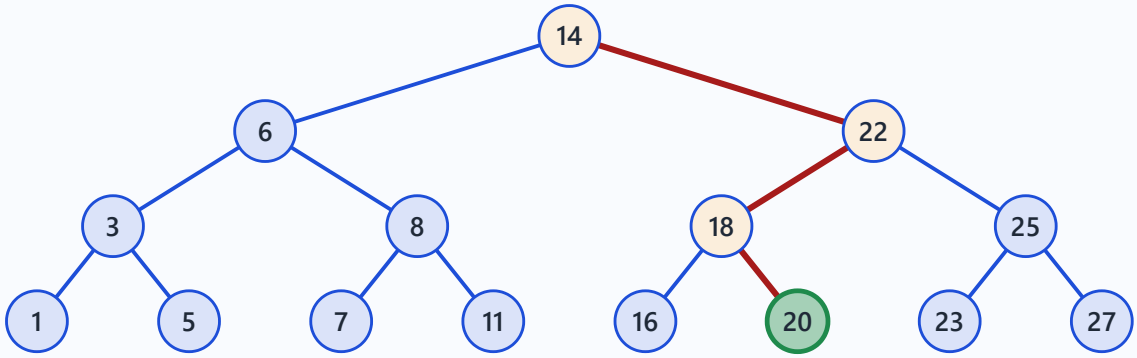
```
1, 3, 5, 6, 7, 8, 11, 14, 16, 18, 20, 22, 23, 25, 27
```

Örnek 3.4

20 aranıyor.

1. `bas = 0`, `son = 14` → `orta = 7`, `dizi[7] = 14`. $20 > 14$ olduğu için sağa gidilir: `bas = 8`.
2. `orta = (8 + 14)/2 = 11`, `dizi[11] = 22`. $20 < 22$ olduğu için sola gidilir: `son = 10`.
3. `orta = (8 + 10)/2 = 9`, `dizi[9] = 18`. $20 > 18$ olduğu için sağa gidilir: `bas = 10`.
4. `orta = (10 + 10)/2 = 10`, `dizi[10] = 20`: bulundu, 10 döner.

Kökten yaprağa 4 karşılaştırma yapılmıştır. Ağaçtaki yol 14 → 22 → 18 → 20'dir.



Şekil 3.7: İkinci dizinin örtük ağacında 20'nin aranması. Kırmızı kenarlar izlenen yolu, yeşil düğüm bulunan öğeyi gösterir. 20 bir yapraktır; bu yüzden en kötü durumdaki 4 karşılaştırma yapılmıştır.

Ağaç bakışı başarısız aramayı da açıklar. Aranan değer dizide yoksa arama bir yaprağa kadar iner ve yaprağın boş bir çocuğuna gitmeye çalışır. Bu boş çocuk, aranan değerin sırayı bozmadan ekleneceği aralığa karşılık gelir. Örneğin bu dizide 19 aranırsa yol 14 → 22 → 18 → 20 olur. 19, 20'den küçük olduğu için 20'nin sol çocuğuna gidilmek istenir, ama o çocuk boştur. 19, 18 ile 20 arasına girecekti.

3.3.7 En iyi ve en kötü durum

Kodun iki ucu dokuz öğeli bir dizide sayılabilir:

> `dizi = {5, 7, 10, 17, 23, 34, 55, 67, 99}` , $n = 9$

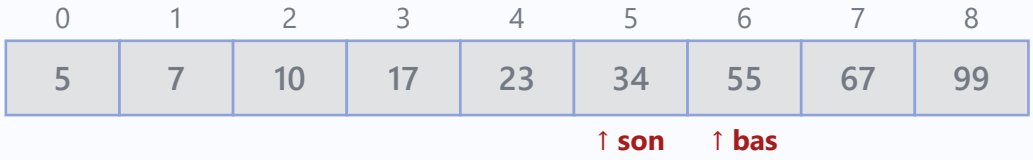
Örnek 3.5

En iyi durum: 23 aranıyor. `bas = 0` , `son = 8` ; `orta = (0 + 8)/2 = 4` ve `dizi[4] = 23` aranana eşittir. Bir adımda bulunur, 4 döner. En iyi durum *O(1)*'dir; aranan değer tam ortadadır.

En kötü durum: 54 aranıyor. 54 dizide yoktur.

1. `orta = 4` : $23 < 54 \rightarrow$ `bas = 5` . Sol yarı (0–4) elenir.
2. `orta = (5 + 8)/2 = 6` : $55 > 54 \rightarrow$ `son = 5` .
3. `orta = (5 + 5)/2 = 5` : $34 < 54 \rightarrow$ `bas = 6` .
4. `bas = 6 > son = 5` : aralık boştur, döngü biter ve -1 döner.

3 karşılaştırmada aramanın başarısız olduğu anlaşılmıştır. 54 dizide olsaydı 34 ile 55 arasında, yani 6. indiste olurdu. Döngü bittiğinde `bas` tam olarak bu ekleme konumunu gösterir.



Şekil 3.8: 54 aramasının sonu. Üç karşılaştırmadan sonra `bas = 6`, `son = 5` olmuştur; aralık boştur ve bütün hücreler elenmiştir.

Bu dizide en kötü durum aslında 4 karşılaştırmadır: $\lfloor \log_2 9 \rfloor + 1 = 4$. 54 aramasının 3 adımda bitmesi, aranan değerin ağacın sığ bir kolunda başarısız olmasındandır. Örneğin 99 aranırsa yoklanan indisler 4, 6, 7, 8 olur ve 4 karşılaştırma yapılır. Sınır en kötü durumu verir; her başarısız arama bu sınıra ulaşmaz.

3.3.8 Kod izleme: ikiliArama(dizi, 9, 47)

Başarılı bir aramanın değişken değerleriyle izlemesi aşağıdadır. Dizi `{0, 4, 7, 10, 14, 23, 45, 47, 53}`, aranan 47'dir.

Tur	bas	son	orta	dizi[orta]	Karşılaştırma	İşlem
1	0	8	4	14	$47 > 14$	`bas = 5`
2	5	8	6	45	$47 > 45$	`bas = 7`
3	7	8	7	47	$47 = 47$	`return 7`

Tablo 3.5: `ikiliArama(dizi, 9, 47)` izlemesi.

Birinci turda `bas ≤ son` doğrudur; orta $(0 + 8)/2 = 4$ olarak hesaplanır ve `dizi[4] = 14`'tür. 47 daha büyük olduğu için sol kısım elenir: `bas = 5`. İkinci turda orta $(5 + 8)/2 = 6$'dır, `dizi[6] = 45` yine küçüktür: `bas = 7`. Üçüncü turda orta $(7 + 8)/2 = 7$'dir ve `dizi[7] = 47` aranana eşittir; metot 7 döndürür. Dokuz öğede 3 karşılaştırma yapılmıştır. Bu dizi için sınır $\lfloor \log_2 9 \rfloor + 1 = 4$'tür.



Şekil 3.9: Üçüncü turun sonu: `bas = orta = 7`, `son = 8`. 47 bulunmuştur (yeşil); ilk iki turda elenen hücreler gridir.

Karşılaştırma sayımında bir ayrıntı vardır. Koddaki her turda `aranan < dizi[orta]` ve gerekirse `aranan > dizi[orta]` olmak üzere iki karşılaştırma yazılıdır. Ama ikisi de aynı hücreye bakar; yoklama sayısı bir artar. Notta "karşılaştırma" sayılırken bir orta öğeye bakış bir karşılaştırma kabul edilir. Java'da tam sayılar için iki karşılaştırma yerine tek bir `Integer.compare` çağrısı da kullanılabilir.

3.3.9 Karmaşıklık

İkili aramanın en kötü durumu kesin olarak şöyle ifade edilir.

> **Teorem 3.2 (İkili aramanın karşılaştırma sınırı)**
>
> $n \geq 1$ öğeli sıralı bir dizide ikili arama en fazla $\lfloor \log_2 n \rfloor + 1$ tur döner. Dolayısıyla en kötü durum maliyeti $O(\log n)$'dir.

İspat. Bir turun başında aralıkta s öğe olsun. Orta öğe çıkarıldıktan sonra solda $\lfloor (s-1)/2 \rfloor$, sağda $\lfloor (s-1)/2 \rfloor = \lfloor s/2 \rfloor$ öğe kalır. Hangi yarıya gidilirse gidilsin yeni aralık en fazla $\lfloor s/2 \rfloor$ öğelidir. Başlangıçta $s = n$ olduğundan, k turdan sonra aralıkta en fazla $\lfloor n/2^k \rfloor$ öğe kalır. Bir turun yapılabilmesi için aralık boş olmamalıdır. $k+1$. turun yapılması için $n/2^k \geq 1$, yani $k \leq \log_2 n$ gerekir. Buradan tur numarası en fazla $\lfloor \log_2 n \rfloor + 1$ olabilir. □

Teorem, $T(n) = T(n/2) + 1$ bağıntısının çözümüyle de uyumludur. Bağıntı açılırsa $T(n) = T(n/2^k) + k$ elde edilir. $n/2^k = 1$ olduğunda, yani $k = \log_2 n$ için alt problem sabit maliyetlidir. Toplam $\log_2 n$ mertebesindedir.

Durum	Mertebe
En iyi (ilk orta eşleşir)	$O(1)$
Ortalama	$O(\log n)$
En kötü / başarısız	$O(\log n)$
Ek bellek	yinelemeli $O(1)$, özyinelemeli $O(\log n)$
Ön hazırlık (sıralama)	$O(n \log n)$, bir kez

Tablo 3.6: İkili aramanın maliyeti.

Ortalama durumun da $O(\log n)$ olması ağaç bakışından anlaşılır. Dengeli bir ağaçta düğümlerin yaklaşık yarısı yapraktır ve yaprakların derinliği en büyük derinliktir. Rastgele seçilen bir öğe büyük olasılıkla alt düzeylerdedir. 15 öğeli örnekte ortalama 3,27, en kötü durum 4 karşılaştırmaydı; ikisi arasındaki fark sabittir.

Tablonun son satırı ikili aramanın gizli bedelidir. Dizi sırasızsa önce sıralanmalıdır ve bu $O(n \log n)$ sürer. Sıralama bir kez yapılır, aramalar defalarca. k sorgu için iki yolun maliyeti şöyledir:

- Doğrusal arama: toplam $O(k \cdot n)$.
- Bir kez sırala, sonra ikili arama: toplam $O(n \log n + k \log n)$.

k küçükse, örneğin tek sorgu varsa, doğrusal arama daha ucuzdur. k, $\log n$ mertebesini aştığında sıralamanın maliyeti kendini çıkarır ve k büyüdükçe fark hızla açılır.

Örnek 3.6

$n = 10^6$ öğeli sırasız bir dizide $k = 1000$ arama yapılacak. Doğrusal aramayla en kötü durumda $1000 \cdot 10^6 = 10^9$ karşılaştırma gerekir. Önce sıralamak yaklaşık $n \log_2 n \approx 10^6 \cdot 20 = 2 \cdot 10^7$ karşılaştırma alır. Ardından 1000 ikili arama en fazla $1000 \cdot 20 = 2 \cdot 10^4$ karşılaştırma yapar. Toplam yaklaşık $2 \cdot 10^7$'dir; doğrusal yoldan yaklaşık 50 kat azdır.

3.3.10 Uygulama ayrıntıları

İkili arama fikir olarak basittir, ama doğru yazmak şaşırtıcı ölçüde zordur. Knuth, ilk ikili arama yayınının 1946'da çıktığını, hatasız bir sürümün ise ancak 1962'de yayımlandığını aktarır. Hataların çoğu dört ayrıntıda toplanır.

Orta hesabında taşma. `(bas + son) / 2` ifadesi matematik olarak doğrudur, ama Java'da `int` en fazla $2^{31} - 1 = 2.147.483.647$ değerini tutar. `bas + son` bu sınırı aşarsa toplam negatif bir sayıya döner ve `orta` negatif olur. İki indisin toplamının bu sınırı aşması için dizinin yaklaşık bir milyar (2^{30}) öğeyi geçmesi gerekir. Bu sınır eskiden uzak görünüyordu; bugün büyük dizilerde ve dosya konumlarında gerçek bir risktir. Java kütüphanesinin `Arrays.binarySearch` metodunda bu hata yıllarca fark edilmeden kalmış ve 2006'da düzeltilmiştir. Güvenli yazımlar şunlardır:

> `bas + (son - bas) / 2` ya da `(bas + son) >>> 1`

Birincisi farkı böler ve hiçbir zaman taşmaz. İkincisi, taşmış toplamı işaretsiz sağa kaydırma ile yeniden doğru değere çevirir. Örneğin `bas = 2.000.000.000` ve `son = 2.100.000.000` için `(bas + son) / 2` ifadesi $-97.483.648$ verir; iki güvenli yazım da doğru sonuç olan 2.050.000.000'i verir.

Sınırların güncellenmesi. `son = orta` ya da `bas = orta` yazmak aralığı küçültmeyebilir. `bas == son` iken aranan `dizi[orta]`'dan küçükse `son = orta` ataması aralığı değiştirmez ve döngü sonsuza dek döner. Orta öğe zaten denetlendiği için sınırlar ± 1 ile güncellenmelidir.

Tekrarlı öğeler. Eşitlikte hemen dönmek, eşleşmelerden herhangi birini verir. `{2, 4, 4, 4, 4, 7, 9, 12}` dizisinde 4 aranırsa yinelemeli ikili arama ilk turda 3. indise bakar ve 3 döndürür; oysa ilk 4, 1. indistedir. İlk konum isteniyorsa eşitlikte sonuç kaydedilir ve `son = orta - 1` ile sola devam edilir.

Java kütüphanesi. `Arrays.binarySearch(dizi, x)` sıralı bir dizide ikili arama yapar. Değer bulunursa indisini döndürür; tekrarlı öğelerde hangisinin döneceği belirsizdir. Bulunamazsa $-(\text{ekleme noktası}) - 1$ döndürür. Ekleme noktası, x'in sırayı bozmadan ekleneceği indistir. Dönüş değerinin her zaman negatif olması bulunamadı bilgisini, mutlak değeri ise konumu taşır. Örneğin `{2, 4, 4, 4, 4, 7, 9, 12}` dizisinde 5 aranırsa ekleme noktası 5'tir ve metot $-5 - 1 = -6$ döndürür. "−1" yerine yalnız "−(ekleme noktası)" döndürülseydi, ekleme noktası 0 olan bir değer için dönüş 0 olur ve bulunmuş bir öğeyle karışırdı.

Sık yapılan hata

Döngü koşulunu `while (bas < son)` diye yazmak. Bu koşul tek öğeli aralığı atlar: aranan değer son kalan hücredeyse sınanmadan döngüden çıkılır ve *−1* döner. `{5, 7, 10}` dizisinde 10 aranırsa ilk turda `bas = 2`, `son = 2` olur ve döngü 10'a hiç bakmadan biter.

3.3.11 Tekrarlı öğelerde ilk konum

Tekrarlı öğe içeren sıralı bir dizide aranan değerin *ilk* konumunu bulan ikili arama şöyle yazılır:

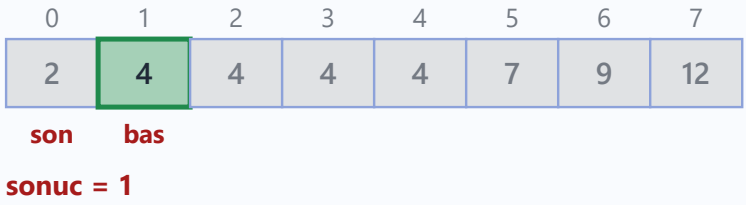
```
int ilkKonum(int[] dizi, int aranan) {
  int bas = 0, son = dizi.length - 1, sonuc = -1;
  while (bas <= son) {
    int orta = bas + (son - bas) / 2;
    if (dizi[orta] == aranan) {
      sonuc = orta;      // kaydet,
      son = orta - 1;    // solda ara
    } else if (dizi[orta] < aranan)
      bas = orta + 1;
    else
      son = orta - 1;
  }
  return sonuc;
}
```

Kod, yinelemeli ikili aramadan iki noktada ayrılır. Birincisi, `sonuc` değişkeni bugüne kadar bulunan en soldaki eşleşmeyi tutar; başlangıç değeri *−1*'dir. İkincisi, eşitlik durumunda metot dönmez. Eşleşmenin indisini `sonuc`'a kaydeder ve `son = orta − 1` ile solda daha önce bir eşleşme olup olmadığına bakar. Döngü, aralık boşalana kadar sürer ve sonunda `sonuc` döner. Orta hesabı da taşmaya karşı güvenli biçimde yazılmıştır.

Tur	bas	son	orta	dizi[orta]	İşlem	sonuc
1	0	7	*0 + (7 − 0)/2 = 3*	4	eşit: kaydet, `son = 2`	3
2	0	2	*0 + (2 − 0)/2 = 1*	4	eşit: kaydet, `son = 0`	1
3	0	0	0	2	*2 < 4*: `bas = 1`	1
—	1	0	`bas > son` : döngü biter			`return 1`

Tablo 3.7: `ilkKonum({2, 4, 4, 4, 4, 7, 9, 12}, 4)` izlemesi.

Dizide dört tane 4 vardır (1–4 indisleri). Birinci turda 3. indisteki 4 bulunur, ama bu ilk 4 olmayabilir. Kaydedilir ve sola bakılır. İkinci turda 1. indisteki 4 bulunur; o da kaydedilir. Üçüncü turda 0. indisteki 2, arananan küçüktür ve aralık sağa kayar. Aralık boşalınca son kaydedilen değer, yani 1 döner. 3 yoklama yapılmıştır ve süre yine *O(log n)*'dir.



Şekil 3.10: `ilkKonum`'un sonu. `bas = 1 > son = 0` olmuştur; `sonuc = 1`, ilk 4'ün indisidir.

Aynı fikir son konumu bulmak için de kullanılır: eşitlikte sonuç kaydedilir ve `bas = orta + 1` ile sağa devam edilir. İki metodun farkı, aranan değerin dizide kaç kez geçtiğini de verir. Örnekte son konum 4, ilk konum 1'dir; 4 değeri $4 - 1 + 1 = 4$ kez geçer. Bu sayım, tekrar sayısı ne olursa olsun $O(\log n)$ sürer.

> **Sık yapılan hata**
>
> İlk konumu, herhangi bir eşleşme bulduktan sonra sola doğru doğrusal tarayarak bulmak. Sonuç doğrudur, ama bütün öğeler eşitse tarama n adım sürer ve maliyet $O(n)$'ye çıkar. Eşitlikte arama aralığını daraltmak, maliyeti her durumda logaritmik tutar.

3.4 Hash ile arama

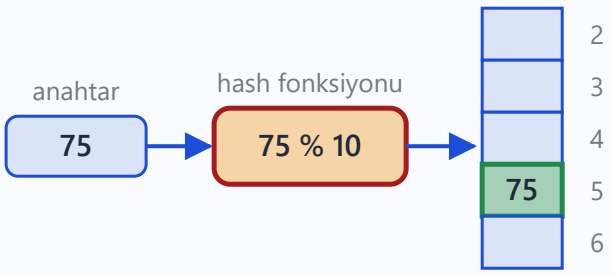
Şimdiye kadarki yöntemler aranan değerin yerini karşılaştırarak bulur. Her karşılaştırma bir miktar bilgi verir ve aday konumları azaltır. Karşılaştırmaya dayalı her yöntem için bir alt sınır vardır: sıralı bir dizide bile en kötü durumda yaklaşık $\log_2 n$ karşılaştırmadan kaçınılamaz. Hash ile arama bu sınırı başka bir yoldan aşar. Yeri aramaz, *hesaplar*.

3.4.1 Fikir: yeri hesaplamak

Hash ile arama, bir anahtarın kendi kilidine tam oturmasına benzer bir mantıkla çalışır. Veriler **anahtarlardan** ve onlara karşılık gelen **değerlerden** oluşur. Anahtar, bir **hash fonksiyonu** ile bir tablo indisine dönüştürülür. Bu indise anahtarın **hash değeri** denir ve verinin tablodaki yerini gösterir.

> **Tanım 3.5**
>
> **Hash tablosu** (*hash table*), m hücreli bir dizidir; her hücreye **kova** (*bucket*) denir. **Hash fonksiyonu** (*hash function*) h, her anahtarı 0 ile $m - 1$ arasında bir kova numarasına dönüştürür. Anahtar x, $h(x)$ numaralı kovaya yerleştirilir ve orada aranır.

Arama, anahtarın hash değerine karşılık gelen kovada yapılır. Kova boş değilse anahtar, kovadaki anahtarlarla karşılaştırılır. Eşleşme bulunursa arama biter. Aramanın maliyeti, hash değerinin hesaplanması ve o kovadaki birkaç karşılaştırmadan ibarettir. Bu maliyet tablodaki anahtar sayısından bağımsızdır. Bu yüzden iyi tasarlanmış bir hash tablosunda ortalama arama süresi *O(1)*'dir.



Şekil 3.11: Anahtar 75, *h(x) = x mod 10* hash fonksiyonuyla 5 numaralı kovaya yönlendirilir. Arama doğrudan bu kovaya bakar; diğer kovalara dokunulmaz.

Bu notun örneklerinde anahtarlar tam sayıdır ve hash fonksiyonu *h(x) = x % 10*'dur. Bu fonksiyon anahtarın son basamağını alır ve onu kova numarası yapar. Fonksiyon basittir, ama ilkeyi göstermeye yeter. Gerçek uygulamalarda kova sayısı çoğu zaman bir asal sayı ya da ikinin kuvveti seçilir ve fonksiyon anahtarın bütün bitlerini kullanır.

3.4.2 Avantajlar ve dezavantajlar

Hash ile aramanın avantajları şunlardır:

- Ortalama durumda çok hızlıdır: *O(1)*.
- Verilerin önceden sıralanmasına gerek yoktur.
- Büyük veri kümelerinde ve çok sayıda ekleme ile aramanın karışık yapıldığı uygulamalarda idealdir. Sıralı dizide yeni bir öğe eklemek öğeleri kaydırmayı gerektirir; hash tablosunda ekleme de ortalama *O(1)*'dir.
- Hash fonksiyonu iyi seçilmişse aranan öğeye doğrudan erişilir.

Dezavantajları da vardır:

- Hash fonksiyonu **çakışmalara** yol açabilir.
- Çakışma olduğunda ek işlem adımları gerekir. En kötü durumda bütün anahtarlar aynı kovaya düşer ve arama *O(n)*'ye çıkar.
- Ek bellek ister; tablonun bir kısmı her zaman boş tutulur.
- Tablo, anahtarları sıra gözetmeden dağıtır. Bu yüzden anahtarları sıralı gezmek, en küçük anahtarı bulmak ya da “50 ile 100 arasındaki anahtarlar” gibi aralık sorgularını yanıtlamak verimli değildir.

Tanım 3.6

Çakışma (*collision*), iki farklı anahtarın aynı hash değerine düşmesidir: $x \neq y$ iken $h(x) = h(y)$.

Çakışmalar kaçınılmazdır. Olası anahtarların sayısı kova sayısından çok büyüktür; güvercin yuvası ilkesine göre bazı anahtarlar aynı kovayı paylaşmak zorundadır. Örneğin bütün tam sayılar 10 kovaya dağıtılıyorsa son basamağı aynı olan her sayı aynı kovaya düşer. Bu yüzden hash tablosunun tasarımında iki soru vardır: çakışmaları azaltan bir hash fonksiyonu nasıl seçilir ve kaçınılmaz çakışmalar nasıl çözülür?

3.4.3 Hash fonksiyonu

Anahtar bir dizgi olduğunda önce onu bir sayıya çevirmek gerekir. En basit yol, karakterlerin sayısal kodlarını toplamaktır:

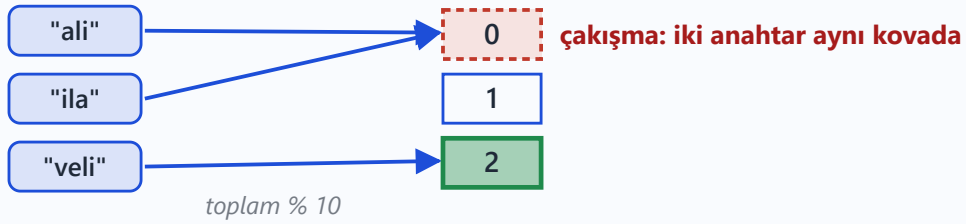
```
int hashFonksiyonu(String anahtar,
                   int m) {
  int toplam = 0;
  for (char k : anahtar.toCharArray())
    toplam += k;        // Unicode kodu
  return toplam % m;    // m: tablo boyu
}
```

Metot, anahtarın karakterleri üzerinde tek tek gezinir. Java'da `char` türü bir tam sayı gibi davranır; `toplam += k` satırı karakterin Unicode kodunu toplama ekler. Döngü bitince toplam, tablo boyu `m`'ye göre mod alınarak 0 ile $m - 1$ arasına indirilir. Metot $O(L)$ sürede çalışır; burada L anahtarın uzunluğudur.

Anahtar	Karakter kodları	Toplam *% 10*
`"ali"`	$97 + 108 + 105 = 310$	0
`"ila"`	$105 + 108 + 97 = 310$	0
`"veli"`	$118 + 101 + 108 + 105 = 432$	2

Tablo 3.8: Karakter toplamı hash fonksiyonu, $m = 10$.

Fonksiyon basittir ama zayıftır. Toplama işlemi sırayı görmez: harfleri aynı olan kelimeler her zaman aynı toplamı verir. “ali” ile “ila” bu yüzden 0 numaralı kovada çakışır. “kar” ile “ark” da aynı nedenle çakışır. Ayrıca kısa kelimelerin toplamları dar bir aralıkta toplanır; büyük bir tabloda kovaların çoğu hiç kullanılmaz.



Şekil 3.12: Karakter toplamı hash fonksiyonunda çakışma. “ali” ve “ila” aynı toplamı (310) verdiği için 0 numaralı kovaya düşer; “veli” 2 numaralı kovaya gider.

İyi bir hash fonksiyonu iki özelliği birlikte taşır: hızlı hesaplanır ve anahtarları kovalara düzgün dağıtır. Java'nın `String.hashCode()` metodu her karaktere konumuna bağlı bir ağırlık verir:

> $h(s) = \Sigma_{i=0}^{L-1} s[i] \cdot 31^{L-1-i}$

Bu toplamda karakterin yeri de önemlidir. “ali” için değer $97 \cdot 31^2 + 108 \cdot 31 + 105 = 96.670$, “ila” için 104.350'dir; iki kelime artık ayrışır. Toplam büyük olabileceği için `int` taşar, ama bu bir sorun değildir: taşma değerleri karıştırmaya devam eder. Kova numarası, bu değerin işareti düzeltildikten sonra *m*'ye göre modu alınarak bulunur.

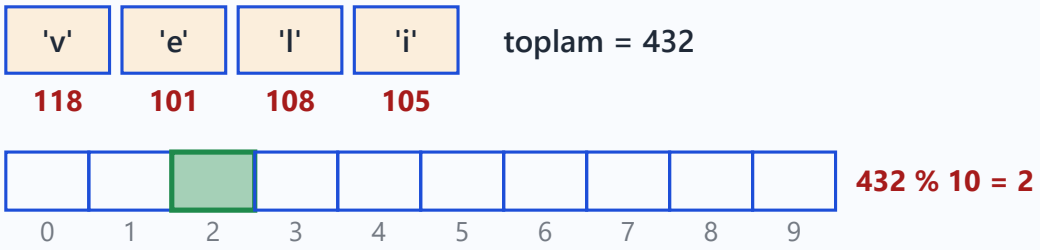
3.4.4 Kod izleme: hashFonksiyonu("veli", 10)

Örnek 3.7

`anahtar = "veli"`, `m = 10`. Başlangıçta `toplam = 0`.

1. `k = 'v'` : Unicode kodu 118 eklenir, `toplam = 118`.
2. `k = 'e'` : 101 eklenir, `toplam = 219`.
3. `k = 'l'` : 108 eklenir, `toplam = 327`.
4. `k = 'i'` : 105 eklenir, `toplam = 432`.
5. *432 % 10 = 2*: “veli” 2 numaralı kovaya düşer.

Aynı hesap “ali” ve “ila” için 310 verir; ikisi de 0 numaralı kovada çakışır.



Şekil 3.13: "veli" anahtarının karakter kodları ve toplamı. *432 mod 10 = 2* olduğu için anahtar 2 numaralı kovaya yerleşir.

3.4.5 Doğrusal yoklama ile arama

Çakışmayı çözmenin iki ana yolu vardır. Birincisinde her anahtar tablonun kendi hücrelerinden birine yerleşir; kovası doluysa başka bir boş kova aranır. Bu yaklaşıma **açık adresleme** (*open addressing*) denir. En basit açık adresleme yöntemi doğrusal yoklamadır.

> **Tanım 3.7**
>
> **Doğrusal yoklama** (*linear probing*): Anahtarın kovası *h(x)* doluysa sırasıyla *h(x) + 1*, *h(x) + 2*, … kovalarına bakılır; tablonun sonuna gelinince başa dönülür. Ekleme ilk boş kovaya yapılır. Arama da aynı sırayı izler ve anahtarı bulduğunda ya da boş bir kovaya ulaştığında durur.

Doğrusal yoklama kullanan bir tabloda arama şöyle yazılır:

```
int hashArama(Integer[] tablo, int anahtar) {
  int m = tablo.length;
  int indis = anahtar % m;   // hash
  while (tablo[indis] != null) {
    if (tablo[indis] == anahtar)
      return indis;          // bulundu
    indis = (indis + 1) % m; // sonraki kova
  }
  return -1;                 // boş kova: yok
}
```

Tablo `Integer[]` türündedir, böylece boş kova `null` ile gösterilebilir. Bu örnekte tablo yalnız anahtarları saklar; genel bir uygulamada her kova bir anahtar-değer çifti tutar. `m` tablo boyudur. İlk bakılacak kova `anahtar % m`'dir.

`while` döngüsü, `tablo[indis]` boş olana kadar sürer. Her turda kovadaki anahtar aranan anahtarla karşılaştırılır. Eşleşirse kovanın indisi döner; anahtar-değer tablosunda burada değer döndürülürdü. Eşleşme yoksa bir sonraki kovaya geçilir. `(indis + 1) % m` ifadesi, son kovadan sonra 0 numaralı ko-

vaya dönmeyi sağlar; tablo dairesel olarak taranır. Boş bir kovaya varılırsa anahtar tabloda yoktur ve metot *−1* döndürür.

Boş kovada durmanın mantığı eklemenin nasıl yapıldığından gelir. Bir anahtar eklenirken kendi kovasından başlayarak ilk boş kovaya yerleşir. Dolayısıyla anahtar tablodaysa, kendi kovasından onun yerine kadar olan bütün kovalar doludur. Arama sırasında bir boş kovaya rastlanırsa, anahtar daha ileride olamaz.

Sık yapılan hata

`tablo[indis] == anahtar` karşılaştırmasının `Integer` nesneleriyle güvenli olduğunu sanmak. Burada `anahtar` bir `int` olduğu için Java `tablo[indis]`'i kutudan çıkarıp sayı olarak karşılaştırır ve sonuç doğrudur. İki taraf da `Integer` olsaydı `==` nesne kimliklerini karşılaştırırdı; −128 ile 127 dışındaki değerlerde eşit sayılar bile "eşit değil" çıkabilirdi. İki nesne karşılaştırılırken `equals` kullanılmalıdır.

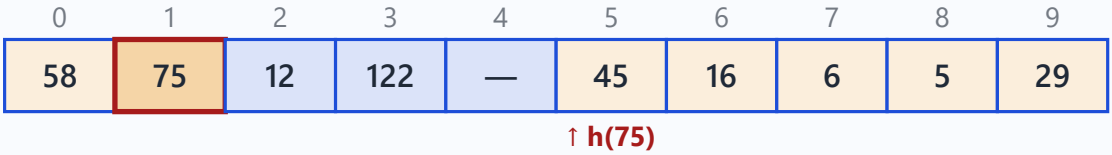
3.4.6 Doğrusal yoklama ile ekleme

Boş bir tabloya (*m = 10*) şu anahtarlar sırayla eklensin: 16, 12, 45, 29, 6, 122, 5, 58, 75. Hash fonksiyonu *h(x) = x % 10*'dur.

Anahtar	*h(x)*	Yoklanan kovalar	Yerleştiği kova	Yoklama
16	6	6	6	1
12	2	2	2	1
45	5	5	5	1
29	9	9	9	1
6	6	6 (dolu), 7	7	2
122	2	2 (dolu), 3	3	2
5	5	5, 6, 7 (dolu), 8	8	4
58	8	8, 9 (dolu), 0	0	3
75	5	5, 6, 7, 8, 9, 0 (dolu), 1	1	7

Tablo 3.9: Doğrusal yoklama ile ekleme.

İlk dört anahtar kendi kovalarına doğrudan yerleşir. 6'nın kovası 6'da 16 durduğu için bir sonraki kova olan 7'ye geçer. 122 de aynı biçimde 2'den 3'e kayar. 5'in durumu daha ilginçtir: 5, 6 ve 7 kovaları doludur, 5 ancak 8'e yerleşebilir. 58'in kendi kovası 8'i az önce 5 almıştır; 58, 9'u da dolu bulur ve tablonun sonundan başa dönerek 0'a yerleşir. Son anahtar 75'in kovası 5'tir ve 5'ten 0'a kadar bütün kovalar doludur. 75, yedi yoklamadan sonra 1 numaralı kovaya yerleşir.



Şekil 3.14: Dokuz ekleme sonunda tablo. 75'in hash değeri 5'tir, ama 1 numaralı kovaya yerleşmiştir. Yalnız 4 numaralı kova boştur; 5'ten 3'e kadar dolu kovalar tek bir küme oluşturur.

Son durumda dolu kovalar 5, 6, 7, 8, 9, 0, 1, 2, 3 sırasıyla bitişik bir küme oluşturmuştur; yalnız 4 numaralı kova boştur. Bu olguya **birincil kümelenme** (*primary clustering*) denir. Küme büyüdükçe, kümenin herhangi bir kovasına düşen yeni anahtar kümenin sonuna kadar yürümek zorunda kalır ve kümeyi bir kova daha büyütür. Büyük kümeler daha büyük olma eğilimindedir. Doğrusal yoklamanın tablo dolarken hızla yavaşlamasının nedeni budur.

3.4.7 Kod izleme: hashArama(tablo, 75)

Oluşan tabloda 75 aranıyor. Tablo `[58, 75, 12, 122, null, 45, 16, 6, 5, 29]` durumundadır.

Yoklama	`indis`	`tablo[indis]`	Karşılaştırma	Sonraki `indis`
1	5	45	*45 ≠ 75*	*(5 + 1) % 10 = 6*
2	6	16	*16 ≠ 75*	7
3	7	6	*6 ≠ 75*	8
4	8	5	*5 ≠ 75*	9
5	9	29	*29 ≠ 75*	*(9 + 1) % 10 = 0*
6	0	58	*58 ≠ 75*	1
7	1	75	*75 = 75*	`return 1`

Tablo 3.10: `hashArama(tablo, 75)` izlemesi.

Arama *75 % 10 = 5* numaralı kovadan başlar. 45, 16, 6, 5 ve 29 sırayla aşılır; 9 numaralı kovadan sonra sondan başa dönülür. 0 numaralı kovadaki 58 de aşılınca 1 numaralı kovada 75 bulunur ve metot 1 döndürür. Arama, eklemede olduğu gibi 7 yoklama sürmüştür: 75'in yerine kümenin sonunda ulaşılmıştır.



Şekil 3.15: `hashArama(tablo, 75)` sonunda `indis = 1`; 75 bulunmuştur.

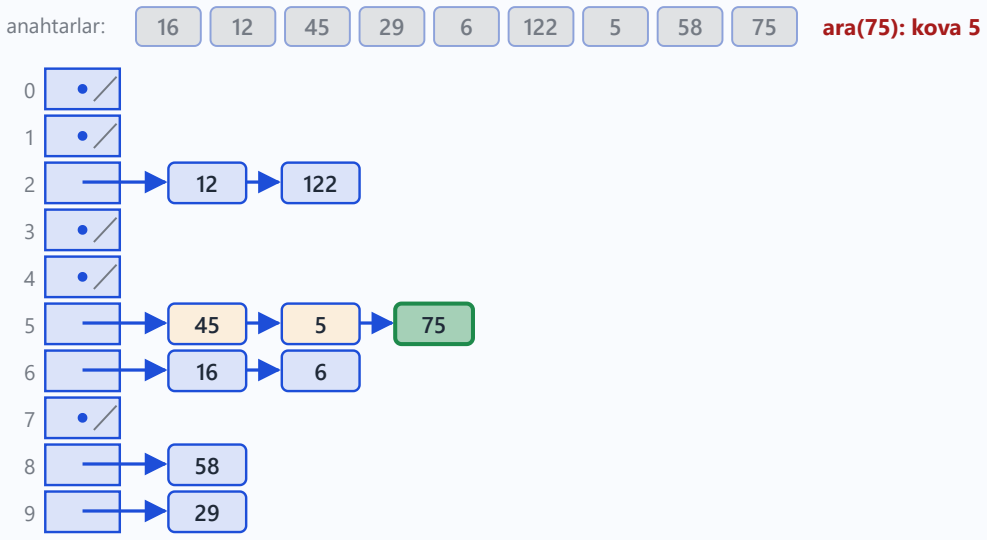
Başarısız bir arama daha pahalıdır. Aynı tabloda 35 aranırsa arama yine 5'ten başlar ve tek boş kova olan 4'e ulaşana kadar sürer: 5, 6, 7, 8, 9, 0, 1, 2, 3, 4 kovalarına bakılır. Dokuz anahtar karşılaştırması ve bir boş kova, toplam 10 yoklama yapılır. Bu tabloda başarılı aramaların ortalaması ise *22/9 ≈ 2,44* yoklamadır; çoğu anahtar kendi kovasına yakın durur.

3.4.8 Zincirleme

Çakışma çözümünün ikinci yolu, bir kovaya birden çok anahtar koymaktır.

> **Tanım 3.8**
>
> **Ayrı zincirleme** (*separate chaining*) yönteminde her kova bir bağlı listenin başını tutar. Aynı kovaya düşen anahtarlar bu listeye eklenir. Arama, anahtarın kovasındaki listeyi baştan sona gezer.

Aynı anahtarlar, aynı hash fonksiyonuyla zincirleme kullanan bir tabloya eklensin. 16, 12, 45 ve 29 boş kovalara düşer ve birer öğeli listeler oluşur. 6 eklenirken 6 numaralı kova doludur; bu bir çakışmadır, ama 6 başka bir kovaya kaymaz, 16'nın listesinin sonuna eklenir. 122, 2 numaralı kovanın listesine 12'den sonra eklenir. 5, kova 5'in listesinde 45'in arkasına geçer. 58 boş olan 8 numaralı kovaya yerleşir. Son olarak 75, kova 5'in listesine üçüncü öğe olarak eklenir.



Şekil 3.16: Zincirleme ile oluşan tablo. Kova 2'nin listesi 12 → 122, kova 5'inki 45 → 5 → 75, kova 6'nınki 16 → 6'dır. 75 aranırken kova 5'in listesi gezilir: 45 ≠ 75, 5 ≠ 75, 75 = 75.

Zincirlemede 75'i aramak için yalnız kova 5'in listesi gezilir: 45 ≠ 75, 5 ≠ 75, 75 = 75. Üç karşılaştırmada bulunur. Doğrusal yoklamadaki 7 yoklamaya göre bu çok daha azdır, çünkü zincirlemede kümelenme olmaz. Her anahtar yalnız kendi kovasıyla aynı hash değerini taşıyan anahtarlarla karşılaştırılır. Kova 6'ya düşen 16 ile kova 5'e düşen 75 hiçbir zaman karşılaşmaz.

Zincirlemenin başka üstünlükleri de vardır. Tablo hiçbir zaman “dolmaz”; listeler uzayabilir. Silme de kolaydır: öğe listeden çıkarılır. Doğrusal yoklamada ise bir anahtarı silip kovayı boşaltmak, arkasındaki anahtarlara giden yolu keser; silinen kovaya “silindi” işareti koymak gerekir. Zincirlemenin bedeli, her düğüm için ayrı bir nesne ve bir işaretçi saklamaktır. Bu, hem ek bellek hem de bellekte dağınık erişim demektir. Doğrusal yoklama ise anahtarları ardışık hücrelerde tuttuğu için önbelleği daha iyi kullanır.

Örnek 3.8

Zincirleme tablosunda 35 aranıyor. $h(35) = 5$; kova 5'in listesi 45 → 5 → 75'tir. 45 ≠ 35, 5 ≠ 35, 75 ≠ 35 ve liste biter. 3 anahtar karşılaştırmasından sonra aramanın başarısız olduğu anlaşılır. Aynı arama doğrusal yoklama tablosunda 9 anahtar karşılaştırması ve bir boş kova, toplam 10 yoklama gerektirmişti.

3.4.9 Yük faktörü ve karmaşıklık

Hash tablosunun başarımını belirleyen asıl büyüklük, tablonun ne kadar dolu olduğudur.

Tanım 3.9

Yük faktörü (*load factor*) $\alpha = n / m$'dir; burada n tablodaki anahtar sayısı, m kova sayısıdır.

Zincirlemede α, bir listenin ortalama uzunluğudur ve 1'i aşabilir. Doğrusal yoklamada her kova en fazla bir anahtar tuttuğu için $\alpha \le 1$'dir. Örnek tablolarımızda 10 kovaya 9 anahtar konmuştur: $\alpha = 0{,}9$.

Knuth, anahtarların kovalara rastgele ve birbirinden bağımsız dağıldığı düzgün hash varsayımı altında ortalama yoklama sayıları için şu yaklaşık formülleri verir:

Yöntem	Başarılı arama	Başarısız arama	$\alpha = 0{,}5$	$\alpha = 0{,}9$
Zincirleme	$\approx 1 + \alpha/2$	$\approx 1 + \alpha$	1,25 / 1,5	1,45 / 1,9
Doğrusal yoklama	$\approx \tfrac{1}{2}\,(1 + 1/(1-\alpha))$	$\approx \tfrac{1}{2}\,(1 + 1/(1-\alpha)^2)$	1,5 / 2,5	5,5 / 50,5

Tablo 3.11: Ortalama yoklama sayıları (düzgün hash varsayımı). Son iki sütunda önce başarılı, sonra başarısız arama verilmiştir.

Tablodaki sayılar formüllerden doğrudan hesaplanır. Örneğin doğrusal yoklamada $\alpha = 0{,}9$ için başarısız arama $\tfrac{1}{2}\,(1 + 1/0{,}1^2) = \tfrac{1}{2}\,(1 + 100) = 50{,}5$ yoklama ister. Zincirlemede maliyet α ile doğrusal olarak artar. Doğrusal yoklamada ise $1/(1-\alpha)^2$ terimi α 1'e yaklaşırken sınırsız büyür. Tablo yarı doluyken iki yöntem arasındaki fark küçüktür; yüzde doksan doluyken doğrusal yoklamada başarısız arama zincirlemeden yaklaşık 27 kat pahalıdır.



Şekil 3.17: Başarısız aramada ortalama yoklama sayısının yük faktörüyle değişimi. Zincirlemede eğri neredeyse düzdür; doğrusal yoklamada α 0,9'a yaklaşırken hızla yükselir (eğri 20 yoklamada kesilmiştir).

Formüller ortalama değerlerdir ve gerçek bir tablo bunlardan sapabilir. Örnek doğrusal yoklama tablomuzda $\alpha = 0{,}9$ iken başarılı aramaların ortalaması 2,44 yoklamaydı; formül 5,5 verir. Fark, dokuz anahtarlık küçük örnekte anahtarların çoğunun kendi kovasına yakın yerleşmiş olmasından gelir. Zincirleme tablosunda ise başarılı aramaların ortalaması $14/9 \approx 1{,}56$'dır ve formülün verdiği 1,45'e yakındır.

Teorem 3.3 (Hash ile aramanın ortalama maliyeti)

Düzgün hash varsayımı altında, yük faktörü sabit bir $\alpha_0 < 1$ değerinin altında tutulan bir hash tablosunda ortalama arama süresi $O(1)$'dir.

İspat. $\alpha \le \alpha_0$ ise zincirlemede ortalama yoklama en fazla $1 + \alpha_0$, doğrusal yoklamada en fazla $\tfrac{1}{2}\left(1 + 1/(1-\alpha_0)^2\right)$'dir. İki sınır da n'ye bağlı değildir; sabittir. Hash değerinin hesaplanması da anahtar uzunluğu sabit kabul edildiğinde sabit zaman alır. Toplam ortalama maliyet sabittir. □

Yük faktörünü sabit tutmanın yolu, tablo belli bir doluluğa ulaştığında onu büyütmektir. Yaygın uygulama şudur: α, 0,75 gibi bir eşiği aşınca iki kat büyüklükte yeni bir tablo ayrılır ve bütün anahtarlar yeni tabloya yeniden yerleştirilir. Hash değerleri kova sayısına bağlı olduğu için anahtarlar kopyalanmaz, yeniden hash'lenir. Bu işlem $O(n)$ sürer, ama seyrek yapılır; dinamik dizinin iki katına büyütülmesinde olduğu gibi ekleme başına amortize maliyeti sabittir. Java'nın `HashMap` sınıfı varsayılan olarak 0,75 eşiğini kullanır.

En kötü durum ise $O(n)$'dir. Kötü bir hash fonksiyonu bütün anahtarları aynı kovaya düşürebilir. Bu durumda arama tek bir uzun listeyi ya da kümeyi tarar; yani doğrusal aramaya dönüşür. $h(x) = x \bmod 10$ fonksiyonuyla 10, 20, 30, … gibi anahtarların hepsi 0 numaralı kovaya düşer. Doğrusal yoklamada α 1'e yaklaşırken kümelenme maliyeti hızla büyütür. Tablo tamamen dolarsa ve aranan anahtar tabloda yoksa boş kova hiç bulunamaz; `hashArama`'nın döngüsü hiç bitmez.

Sık yapılan hata

Doğrusal yoklama tablosunun tamamen dolmasına izin vermek. `hashArama` yalnız boş kovada durduğu için dolu bir tabloda olmayan bir anahtarı aramak sonsuz döngüye yol açar. Yük faktörü sınırlanmalı ya da döngü en fazla *m* yoklamadan sonra durdurulmalıdır.

3.5 Ara değer arama

İkili arama, aranan değer ne olursa olsun aralığın tam ortasına bakar. İnsanlar böyle aramaz. Telefon rehberinde "Yılmaz" aranırken rehber ortadan değil, sona yakın bir yerden açılır; "Akın" aranırken başa yakın bir yerden. Ara değer arama bu davranışı bir formüle döker.

3.5.1 Fikir: konumu tahmin etmek

Ara değer arama (*interpolation search*) sıralı veri kümelerinde kullanılır. Değerlerin aralık boyunca eşit aralıklarla dağıldığını varsayar: ardışık öğeler arasındaki fark yaklaşık aynıdır. Bu varsayım altında indis ile değer arasındaki ilişki bir doğruya yakındır. Aranan değerin konumu, aralığın en küçük ve en büyük öğesinden geçen doğru üzerinde tahmin edilir. Tahmin edilen konum yoklanır. Değer eşleşmezse arama, tıpkı ikili aramadaki gibi, tahminin solunda ya da sağındaki daha dar aralıkta sürer.

Tahmin formülü doğru denkleminden çıkar. Aralığın uçları *(bas, dizi[bas])* ve *(son, dizi[son])* noktalarıdır. Bu iki noktadan geçen doğru üzerinde değeri *aranan* olan noktanın indisi şöyledir:

Tanım 3.10

Ara değer aramada yoklanacak indis

> indis = bas + (aranan − dizi[bas]) · (son − bas) / (dizi[son] − dizi[bas])

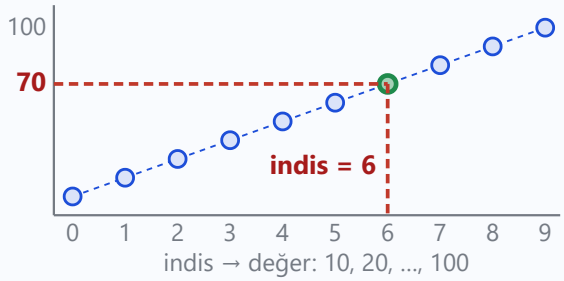
formülüyle hesaplanır. Bölme tam sayı bölmesidir; sonuç aşağı yuvarlanır.

Formülün yapısı sezgiyle okunabilir. *(aranan − dizi[bas]) / (dizi[son] − dizi[bas])* oranı, aranan değerin değer aralığında ne kadar ileride olduğunu gösterir; 0 ile 1 arasında bir sayıdır. Bu oran indis aralığının uzunluğu *son − bas* ile çarpılıp *bas*'a eklenince tahmini konum elde edilir. Aranan değer en küçük değere eşitse oran 0'dır ve *bas* yoklanır; en büyük değere eşitse oran 1'dir ve *son* yoklanır. İkili arama bu oranı her zaman ½ alan özel bir durum gibi düşünülebilir.

Örnek 3.9

`dizi = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}` dizisinde 70 aranıyor. *bas = 0*, *son = 9*, *dizi[bas] = 10*, *dizi[son] = 100*.

> indis = 0 + (70 − 10) · (9 − 0) / (100 − 10) = 60 · 9 / 90 = 540 / 90 = 6
>
> `dizi[6] = 70` : tek yoklamada bulunur. Dağılım tam düzgün olduğu için tahmin hatasızdır. İkili arama aynı aramada 4, 7, 5, 6 indislerini yoklar ve dördüncü yoklamada bulur.



Şekil 3.18: Düzgün dağılımda indis–değer ilişkisi bir doğrudur. Değer ekseninde 70'ten doğruya, oradan indis eksenine inilince tahmin 6 olur.

3.5.2 Kod: araDegerArama()

```
int araDegerArama(int[] dizi, int n,
                  int aranan) {
  int bas = 0, son = n - 1;
  while (bas <= son && aranan >= dizi[bas]
                    && aranan <= dizi[son]) {
    if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
    int pay = (aranan - dizi[bas]) * (son - bas);
    int indis = bas + pay / (dizi[son] - dizi[bas]);
    if (dizi[indis] == aranan) return indis;
    if (dizi[indis] < aranan) bas = indis + 1;
    else son = indis - 1;
  }
  return -1;
}
```

`bas` ve `son` yine arama aralığının sınırlarıdır. Döngü koşulu ikili aramadakinden zengindir: aralık boş olmamalı (`bas <= son`) ve aranan değer aralığın uç değerleri arasında kalmalıdır (`aranan >= dizi[bas]` ve `aranan <= dizi[son]`). Aranan değer uçların dışındaysa aralıkta olamaz; döngüye girilmez ve *−1* döner. Bu denetim aynı zamanda tahmini indisin aralık içinde kalmasını güvenceye alır. Aranan uçların dışında olsaydı oran 0'dan küçük ya da 1'den büyük çıkar ve indis dizinin dışına taşabilirdi.

Döngünün ilk satırı sıfıra bölmeyi önler. `dizi[son] == dizi[bas]` ise aralıktaki bütün değerler eşittir. Döngü koşulu sağlandığına göre aranan da bu değere eşittir ve `bas` döndürülebilir. Ardından pay ve indis hesaplanır. Pay ile paydanın ayrı yazılması, işlem sırasını açıkça belirler: önce çarpma, sonra

bölme. Önce bölme yapılsaydı tam sayı bölmesi oranı çoğu zaman 0'a yuvarlardı. Son olarak tahmini indisteki değer yoklanır. Eşitse indis döner. Küçükse aranan sağdadır, `bas = indis + 1`. Büyükse aranan soldadır, `son = indis - 1`. Aralık boşalırsa ya da aranan uçların dışına düşerse döngü biter ve *−1* döner.

3.5.3 Kod izleme: araDegerArama(dizi, 15, 24)

Dağılımı yaklaşık düzgün, ama tam düzgün olmayan bir dizi şudur:

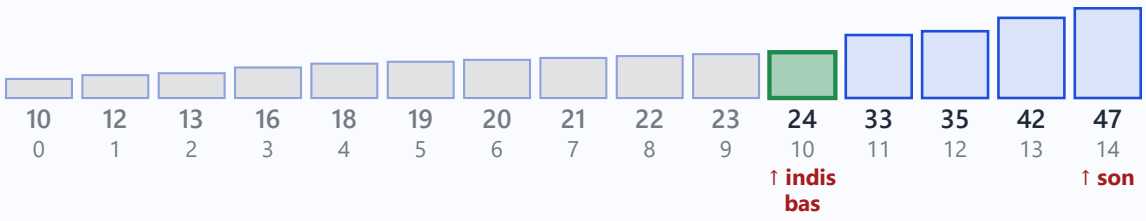
> `dizi = {10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47}`, $n = 15$

Aranan 24'tür. Dizinin ilk on bir öğesi sık, son dört öğesi seyrek dağılmıştır.

Yoklama	`bas`	`son`	Hesap	`indis`	`dizi[indis]`	İşlem
1	0	14	$0 + (24 - 10) \cdot 14 / (47 - 10) = 196/37$	5	19	`bas = 6`
2	6	14	$6 + (24 - 20) \cdot 8 / (47 - 20) = 6 + 32/27$	7	21	`bas = 8`
3	8	14	$8 + (24 - 22) \cdot 6 / (47 - 22) = 8 + 12/25$	8	22	`bas = 9`
4	9	14	$9 + (24 - 23) \cdot 5 / (47 - 23) = 9 + 5/24$	9	23	`bas = 10`
5	10	14	$10 + (24 - 24) \cdot 4 / (47 - 24) = 10 + 0/23$	10	24	`return 10`

Tablo 3.12: `araDegerArama(dizi, 15, 24)` izlemesi.

İlk tahmin $196/37 \approx 5{,}3$'tür ve aşağı yuvarlanarak 5 olur. `dizi[5] = 19` aranandan küçüktür; arama 6. indisten devam eder. Her yoklamada aralığın sol ucu yükselir, ama sağ uç hep 47'de kalır. 47 dizinin geri kalanından belirgin biçimde büyük olduğu için tahminleri sola çeker: ikinci yoklamada oran $4/27$, üçüncüde $2/25$'tir. Arama hedefe adım adım yaklaşır ve beşinci yoklamada 10. indiste 24'ü bulur.



Şekil 3.19: Ara değer aramanın sonu. Çubukların yüksekliği değeri gösterir. Yoklanan 5, 7, 8, 9 indisleri (gri) aranandan küçük çıkmış, 10. indiste 24 bulunmuştur.

Bu dizide ikili arama 24'ü 7, 11, 9, 10 indislerini yoklayarak dört adımda bulurdu. Ara değer arama burada ikili aramadan yavaştır. Dizinin dağılımı "yaklaşık düzgün" olsa da sondaki seyrek değerler tahmini bozmuştur. Ara değer aramanın hızı tamamen dağılımın düzgünlüğüne bağlıdır.

3.5.4 Karmaşıklık ve kötü durum

Durum	Mertebe
En iyi	*O(1)*
Ortalama (düzgün dağılım)	*O(log log n)*
En kötü (çarpık dağılım)	*O(n)*
Ek bellek	*O(1)*

Tablo 3.13: Ara değer aramanın maliyeti.

Düzgün ve rastgele dağılmış anahtarlarda ortalama yoklama sayısı *O(log log n)*'dir. Bu sonucu Perl, Itai ve Avni 1978'de kanıtlamıştır. Kanıt bu notun kapsamı dışındadır, ama sezgisi şöyle anlatılabilir. Düzgün dağılımda her tahminin hatası, kalan aralık boyunun karekökü mertebesindedir. Yani aralık her yoklamada yaklaşık n'den $\sqrt{n}$'ye iner: $n \to n^{1/2} \to n^{1/4} \to \dots$. Aralığın sabit boya inmesi için gereken adım sayısı $\log \log n$ mertebesindedir. İkili arama logaritmayı bir kez alır; ara değer arama iki kez.

Sayılarla bakınca fark çarpıcıdır. $n = 10^6$ için $\log_2 \log_2 n \approx \log_2 19{,}93 \approx 4{,}3$'tür; ikili arama ise yaklaşık 20 yoklama yapar. Ancak bu kazancın iki bedeli vardır:

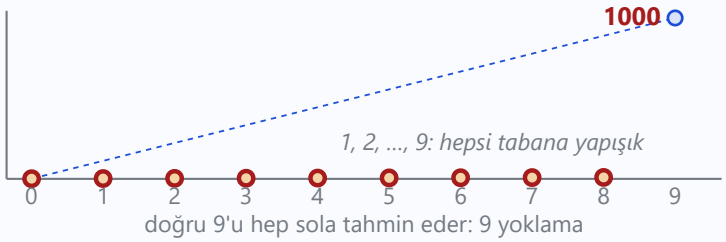
- Her yoklama ikili aramadakinden pahalıdır: bir çıkarma ve bölme yerine çarpma, bölme ve birkaç çıkarma içerir. Bellekten okuma yavaşsa, örneğin veri diskteyse, yoklama sayısını azaltmak bu bedele değer. Veri bellekteyse kazanç çoğu zaman küçüktür.
- Ara çarpım $(\text{aranan} - \text{dizi[bas]}) \cdot (\text{son} - \text{bas})$ büyük değerlerde `int`'i taşırabilir. Değerler 10^9, $\text{son} - \text{bas}$ ise 10^6 mertebesindeyse çarpım 10^{15} olur ve $2^{31} - 1$ sınırını çok aşar. Bu durumda çarpım `long` ile yapılmalıdır.

Örnek 3.10

`dizi = {1, 2, 3, 4, 5, 6, 7, 8, 9, 1000}` dizisinde 9 aranıyor. Dizi sıralıdır, ama son öğe ötekilerden çok büyüktür.

1. $\text{bas} = 0$: $\text{indis} = 0 + (9 - 1) \cdot 9 / (1000 - 1) = 72/999 \to 0$. `dizi[0] = 1 < 9`, `bas = 1`.
2. $\text{bas} = 1$: $\text{indis} = 1 + (9 - 2) \cdot 8 / 998 = 1 + 56/998 \to 1$. `dizi[1] = 2 < 9`, `bas = 2`.
3. Aynı biçimde pay paydadan hep küçük kalır ve her yoklamada indis yalnız bir artar: 2, 3, 4, 5, 6, 7.
4. $\text{bas} = 8$: $\text{indis} = 8 + 0 \cdot 1 / 991 = 8$. `dizi[8] = 9` : bulundu.

Yoklanan indisler 0, 1, 2, 3, 4, 5, 6, 7, 8'dir: **9 yoklama**. Ara değer arama doğrusal aramaya dönüşmüştür. İkili arama aynı aramayı 4, 7, 8 indisleriyle 3 yoklamada bitirir.



Şekil 3.20: Çarpık dağılım. Uç değer 1000, tahmin doğrusunu neredeyse yatay yapar; 1'den 9'a kadar bütün değerler tabana yapışık kalır ve her tahmin bir sonraki indise düşer.

Kötü durumun kaynağı, formülün uç değerlere güvenmesidir. Tek bir aykırı değer, aralığın geri kalanındaki bütün tahminleri bozar. Üstel büyüyen bir dizide, örneğin 2, 4, 8, ..., 256 dizisinde 32 aranırken de aynı durum görülür: tahminler 0, 1, 2, 3, 4 diye birer birer ilerler. En kötü durum *O(n)*'dir. Bu yüzden ara değer arama, dağılımı bilinen ve düzgün olan sayısal anahtarlarda kullanılır. Sıralı kimlik numaraları ya da düzenli aralıklarla alınmış zaman damgaları iyi örneklerdir. Pratikte ara değer adımlarını ikili arama adımlarıyla karıştıran melez yöntemler de kullanılır; böylece en kötü durum *O(log n)*'de tutulur.

Sık yapılan hata

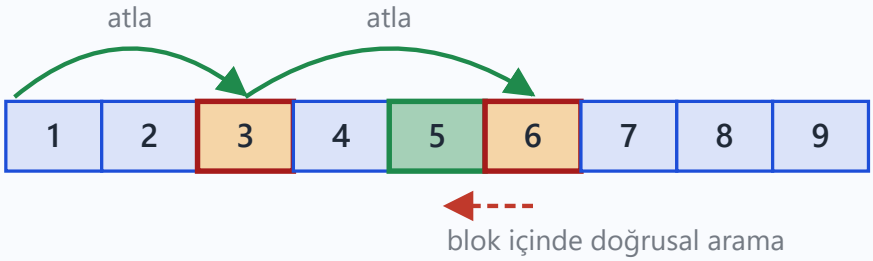
Ara değer aramada uç değer denetimini atlamak. Döngü koşulundan `aranan >= dizi[bas] && aranan <= dizi[son]` çıkarılırsa aralığın dışındaki bir değer için indis negatif ya da `son`'dan büyük çıkar ve dizi dışı erişim hatası oluşur. `dizi[son] == dizi[bas]` denetimi unutulursa da bütün öğeleri eşit bir aralıkta sıfıra bölme olur.

3.6 Atlayarak arama

İkili arama dizinin içinde büyük sıçramalar yapar: ortadan bir çeyreğe, oradan bir sekizde bire. Veri bellekteyse bu sıçramaların bedeli yoktur. Ama bazı ortamlarda geriye gitmek ya da uzak bir konuma atlamak pahalıdır. Atlayarak arama bu ortamlar için tasarlanmıştır.

3.6.1 Fikir: bloklar hâlinde ilerlemek

Atlayarak arama (*jump search*), engelli koşuda engelleri aşarak ilerleyen bir atlet gibi çalışır; dizi sıralı olmalıdır. Liste m büyüklüğünde bloklara ayrılır ve blok boyu genellikle $m = \sqrt{n}$ seçilir. Listenin başından büyük bir adım atılır ve o konumdaki öğe, yani bloğun son öğesi kontrol edilir. Bu öğe aranandan küçükse aranan bu blokta olamaz; bir sonraki bloğa atlanır. Bloğun son öğesi aranandan büyük ya da ona eşitse aranan, varsa, bu bloktadır. Arama o bloğun başına döner ve blok içinde doğrusal arama yapar.



Şekil 3.21: Dokuz öğeli dizide atlayarak arama ($m = 3$). Önce 3 ve 6 değerli blok sonlarına atlanır; 5 aranırken 6'da durulur ve bloğun içinde doğrusal arama yapılır.

Geri dönüş, ikili aramadaki gibi büyük sıçramalar gerektirmez. Veri hep ileri yönde okunur; yalnız son blokta bir kez, en fazla m adım geri gidilir. Bu yüzden atlayarak arama geriye gitmenin pahalı olduğu ortamlarda tercih edilir. Manyetik bant ileri yönde hızlı okunur, geri sarma yavaştır. Disk üzerindeki büyük bir dosya bloklar hâlinde okunur; her blok okuması pahalıdır ve atlanan bloklar hiç okunmaz.

3.6.2 Kod: atlayarakAraBul()

```
int atlayarakAraBul(int[] dizi, int eleman,
                    int n) {
  int blok = (int) Math.sqrt(n);
  int atlama = blok, konum = 0;
  while (dizi[Math.min(atlama, n) - 1] < eleman) {
    konum = atlama; atlama += blok; // atla
    if (konum >= n) return -1;
  }
  while (dizi[konum] < eleman) {     // doğrusal
    konum++;
    if (konum == Math.min(atlama, n)) return -1;
  }
  if (dizi[konum] == eleman) return konum;
  return -1;
}
```

Atlama mesafesi `Math.sqrt(n)` ile hesaplanır ve tam sayıya çevrilir; `blok` , $\lfloor\sqrt{n}\rfloor$'dir. `konum` o anki bloğun başını, `atlama` ise bir sonraki bloğun başını gösterir. Bloğun son öğesi `dizi[atlama - 1]` 'dir.

İlk döngü blokları atlar. Her turda bloğun son öğesine bakılır. `Math.min(atlama, n)` ifadesi, son bloğun tam dolu olmadığı durumda dizinin dışına çıkmayı önler; bu durumda dizinin son öğesine bakılır. Bloğun son öğesi aranandan küçükse aranan bu blokta değildir: `konum` bir sonraki bloğun başına, `atlama` da bir blok ileriye taşınır. `konum` dizinin sonunu geçmişse aranan, dizinin bütün öğelerinden büyüktür ve -1 döner.

İlk döngü bittiğinde aranan, varsa, `[konum, atlama)` aralığındadır. İkinci döngü bu aralığı doğrusal tarar. Taranan öğe aranandan küçük oldukça `konum` bir artar. Bloğun sonuna gelinmişse aranan bu blokta yoktur ve *−1* döner. Döngü, aranandan küçük olmayan bir öğede durur. Son satırdaki denetim bu öğenin aranana eşit olup olmadığına bakar: eşitse indis, değilse *−1* döner.

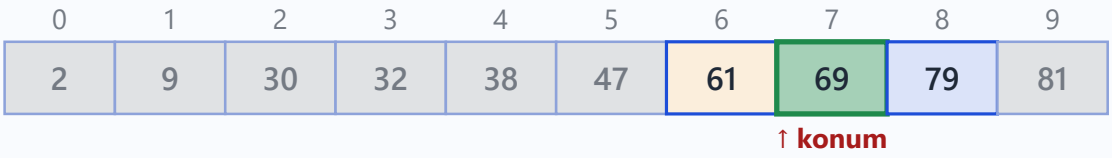
3.6.3 Kod izleme: atlayarakAraBul(dizi, 69, 10)

Dizi on öğelidir: `{2, 9, 30, 32, 38, 47, 61, 69, 79, 81}`. Aranan 69'dur.

Örnek 3.11

1. $n = 10 \rightarrow$ `blok = ⌊√10⌋ = 3` ; `atlama = 3` , `konum = 0` .
2. `dizi[min(3, 10) − 1] = dizi[2] = 30 < 69` → bir blok atla: `konum = 3` , `atlama = 6` .
3. `dizi[min(6, 10) − 1] = dizi[5] = 47 < 69` → bir blok atla: `konum = 6` , `atlama = 9` .
4. `dizi[8] = 79 ≥ 69` → atlama biter. Aranan `[6, 9)` bloğunda olabilir.
5. Doğrusal tarama: `dizi[6] = 61 < 69` → `konum = 7` ; 7, bloğun sonu olan 9'a eşit değildir.
6. `dizi[7] = 69` aranandan küçük değildir, döngü durur. Son denetimde `dizi[7] == 69` doğrudur; metot 7 döndürür.

Yoklanan indisler sırasıyla 2, 5, 8, 6, 7'dir: 5 yoklama. 8. indise bakıldıktan sonra 6'ya geri dönülmüştür; bu, aramadaki tek geri dönüştür.



Şekil 3.22: Atlayarak aramanın sonu. Atlanan ilk iki blok gridir; son bloğun içinde 61 geçilmiş, 69 bulunmuştur.

Başarısız bir arama da benzer yürür. Aynı dizide 50 aranırsa 2, 5 ve 8 indislerine bakılır; 79 ≥ 50 olduğu için `[6, 9)` bloğuna geçilir. `dizi[6] = 61` zaten 50'den büyüktür; doğrusal döngü hiç dönmez. Son denetimde 61 ≠ 50 olduğu için *−1* döner. Yoklanan indisler 2, 5, 8, 6'dır.

3.6.4 Neden √n?

Blok boyu *m* iken en kötü durumdaki yoklama sayısı şöyle bulunur. Atlama aşamasında en fazla *n/m* blok sonuna bakılır. Doğrusal aşamada bloğun içinde en fazla *m − 1* öğe taranır; bloğun son öğesine zaten bakılmıştı. Toplam:

> $f(m) = n/m + (m - 1)$

İki terim zıt yönde davranır. Bloklar küçükse atlama sayısı artar; $m = 1$ için arama doğrusal aramaya döner. Bloklar büyükse blok içi tarama uzar; $m = n$ için yine doğrusal aramaya dönülür. En iyi değer arada bir yerdedir.

Teorem 3.4 (Atlayarak aramada en iyi blok boyu)

$f(m) = n/m + m - 1$ fonksiyonu $m = \sqrt{n}$'de en küçük değerini alır. Bu değer $2\sqrt{n} - 1$'dir; dolayısıyla atlayarak aramanın en kötü durumu $O(\sqrt{n})$'dir.

İspat. f'nin türevi $f'(m) = -n/m^2 + 1$'dir. Türev sıfıra eşitlenirse $m^2 = n$, yani $m = \sqrt{n}$ bulunur. İkinci türev $f''(m) = 2n/m^3 > 0$ olduğundan bu nokta bir en küçük noktadır. Yerine koyunca $f(\sqrt{n}) = n/\sqrt{n} + \sqrt{n} - 1 = 2\sqrt{n} - 1$ elde edilir. □

Aynı sonuç türev kullanmadan da görülebilir. İki pozitif sayının çarpımı sabitse toplamları, sayılar eşit olduğunda en küçüktür. n/m ile m'nin çarpımı her zaman n'dir; bu yüzden toplam $n/m = m$ olduğunda, yani $m = \sqrt{n}$ iken en küçüktür.

n	$m = \sqrt{n}$	$\approx 2\sqrt{n} - 1$	İkili
100	10	19	7
10.000	100	199	14
10^6	1.000	1.999	20

Tablo 3.14: Atlayarak ve ikili aramanın en kötü durumdaki yoklama sayıları.

Karmaşıklık $O(\sqrt{n})$'dir: doğrusal aramanın $O(n)$'si ile ikili aramanın $O(\log n)$'si arasında kalır. Ek bellek $O(1)$'dir. Blok boyunun yanlış seçilmesi maliyeti hissedilir biçimde artırır. $n = 10.000$ için $m = 50$ seçilirse en kötü durum $200 + 49 = 249$ yoklama olur; en iyi seçim olan $m = 100$ ile 199'dur.

Blok bulunduktan sonra blok iki yönde taranabilir. Kod, bloğun başından ileri doğru tarar (69 aramasında 6 → 7). Bloğun sonundan geriye doğru da taranabilirdi (8 → 7). İki yol da en fazla $m - 1$ ek yoklama yapar. Bir iyileştirme daha mümkündür: blok bulunduktan sonra içeride doğrusal arama yerine yine atlayarak ($\sqrt{m}$ adımla) ya da ikili arama yapılabilir. Ancak ikili arama kullanılabilecekse atlayarak arama başından gereksizdir. Atlayarak arama yalnız geri gitmenin pahalı olduğu durumlarda anlamlıdır.

3.7 Üstel arama

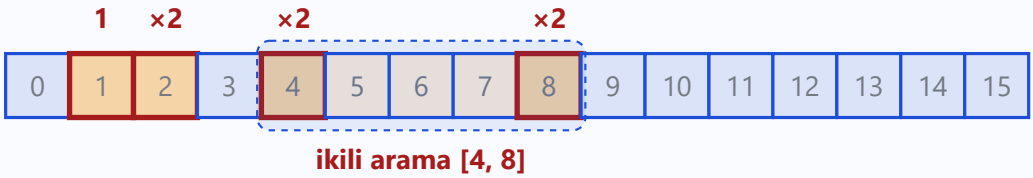
İkili arama dizinin boyunu bilmek zorundadır; ilk adımda `son = n - 1` atanır. Bazı durumlarda boy bilinmez ya da çok büyüktür. Örneğin sıralı bir veri akışı sürekli uzar ya da çok büyük bir dosyanın sonuna gitmek pahalıdır. Üstel arama önce aranan değeri içeren bir aralık bulur, sonra o aralıkta ikili arama yapar.

3.7.1 Fikir: sınırı katlamak

Üstel arama (*exponential search*) sıralı listeyi üstel olarak büyüyen parçalara ayırır. Sınır, yani `konum`, 1, 2, 4, 8, 16, ... diye her adımda ikiye katlanır. `dizi[konum]` aranandan büyük olunca durulur. Bu anda aranan, varsa, `[konum/2, konum]` aralığındadır, çünkü bir önceki sınır olan `konum/2`'deki değer aranandan büyük değildi. Bu aralıkta ikili arama yapılır. Yöntem, Bentley ve Yao'nun 1976'da sınırsız diziler için önerdiği aramaya dayanır.

Örnek 3.12

1024 öğeli bir listede aranan değer 600. indis civarında olsun. Sınır 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 değerlerinden geçer ve hepsinde değer aranandan küçüktür. Sınır 1024 olunca dizinin dışına çıkılır ve katlama durur. Arama *[512, min(1024, 1023)] = [512, 1023]* aralığında ikili aramayla sürer. Katlama aşaması 11 sınama yapar (0. indis dahil), ikili arama da 512 öğeli aralıkta en fazla 10 yoklama.



Şekil 3.23: Üstel arama: 1, 2, 4 ve 8 indisleri yoklanır; sınır her adımda iki katına çıkar. 8. indisteki değer aranandan büyük çıkınca ikili arama *[4, 8]* aralığında yapılır.

Yöntemin üstünlüğü, maliyetin *n*'ye değil aranan öğenin konumu *i*'ye bağlı olmasıdır. Dizinin boyu bilinmiyorsa (sınırsız ya da akan veri) ya da aranan öğe başa yakınsa üstel arama idealdir. Aranan 10. indisteyse dizinin bir milyon ya da bir milyar öğeli olması fark etmez; arama birkaç adımda biter.

3.7.2 Kod: ustelArama()

```
int ustelArama(int[] dizi, int aranan, int n) {
  if (dizi[0] == aranan) return 0;
  int konum = 1;
  while (konum < n && dizi[konum] <= aranan)
    konum *= 2;               // 1, 2, 4, 8, ...
  int bas = konum / 2, son = Math.min(konum, n - 1);
  while (bas <= son) {        // ikili arama
    int orta = (bas + son) / 2;
    if (dizi[orta] == aranan) return orta;
    if (dizi[orta] < aranan) bas = orta + 1;
    else son = orta - 1;
  }
  return -1;
}
```

Metot önce 0. indisi ayrıca denetler, çünkü katlama 1'den başlar ve 0. indis katlamada hiç yoklanmaz. Ardından `konum = 1` alınır. `dizi[konum] <= aranan` olduğu sürece `konum` 2 ile çarpılır. `konum < n` koşulu dizinin dışına çıkmayı önler. Döngü sonunda `konum`, aranan değerin olabileceği aralığın sonunu gösterir; `konum / 2` ise başını. Son katlamada `konum` dizinin dışına çıkmış olabileceği için üst sınır `Math.min(konum, n – 1)` ile alınır. Kalan kısım, yinelemeli ikili aramanın aynısıyla taranır.

Son aşamada ikili arama yerine `konum / 2`'den sona doğru doğrusal tarama yapılsaydı maliyet *O(i)* olurdu; aranan dizide yoksa dizinin sonuna kadar gidilirdi. Aralığın boyu *i* ile orantılıdır ve onu ikili aramayla taramak maliyeti *O(log i)*'ye indirir.

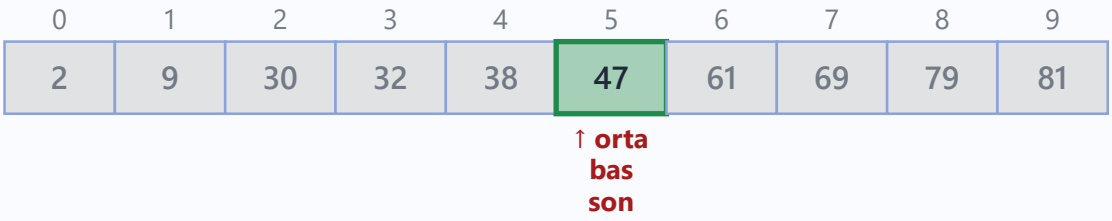
3.7.3 Kod izleme: ustelArama(dizi, 47, 10)

Dizi, atlayarak arama örneğindeki dizidir: `{2, 9, 30, 32, 38, 47, 61, 69, 79, 81}`. Aranan 47'dir.

Aşama	Yoklanan indis	Değer	Karar
ilk denetim	0	2	*2 ≠ 47*
katlama	1	9	*9 ≤ 47* → `konum = 2`
katlama	2	30	*30 ≤ 47* → `konum = 4`
katlama	4	38	*38 ≤ 47* → `konum = 8`
katlama	8	79	*79 > 47* → dur; aralık *[8/2, min(8, 9)] = [4, 8]*
ikili	6	61	`orta = (4 + 8)/2 = 6` ; *61 > 47* → `son = 5`
ikili	4	38	`orta = (4 + 5)/2 = 4` ; *38 < 47* → `bas = 5`
ikili	5	47	`orta = (5 + 5)/2 = 5` ; eşit → `return 5`

Tablo 3.15: `ustelArama(dizi, 47, 10)` izlemesi.

Toplam 8 yoklama yapılmıştır: 0, 1, 2, 4, 8, 6, 4, 5. 4. indise iki kez bakılmıştır. Katlama aşamasında 38 ≤ 47 sınaması yapılmış, ikili arama da aralığın alt ucu olan 4'ü yeniden yoklamıştır. Katlama aşaması 4. indisteki değerin aranandan büyük olmadığını zaten göstermişti; ikili arama `bas = konum / 2 + 1` ile başlatılsaydı bu tekrar önlenirdi. Bu dizide ikili arama 47'yi 3 yoklamada bulur: aranan dizinin başında değilse üstel arama ikili aramadan yavaştır.



Şekil 3.24: Üstel aramanın sonu. İkili aramanın üçüncü turunda `bas = son = orta = 5` olmuş ve 47 bulunmuştur.

3.7.4 Karmaşıklık

Teorem 3.5 (Üstel aramanın maliyeti)

Aranan değer $i \geq 1$ indisindeyse üstel arama $O(\log i)$ yoklama yapar. En kötü durumda ($i \approx n$) maliyet $O(\log n)$, en iyi durumda $O(1)$'dir; ek bellek $O(1)$'dir.

İspat. Katlama, `konum` ilk kez i'yi aştığında ya da dizinin sonuna ulaştığında durur. `konum` 1, 2, 4, … değerlerini aldığı için bu, $2^k > i$ olan en küçük k'de olur; $k = \lfloor \log_2 i \rfloor + 1$'dir. Katlama aşaması 1, 2, …, 2^k konumlarına, yani $k + 1$ konuma bakar. Bulunan aralık $[2^{k-1}, 2^k]$'dir ve en fazla $2^{k-1} + 1 \leq i + 1$ öğe içerir. Bu aralıkta ikili arama en fazla yaklaşık $\log_2 i + 1$ yoklama yapar. İki aşamanın toplamı yaklaşık $2 \log_2 i$, yani $O(\log i)$'dir. □

Üstel arama, aranan başa yakınsa ikili aramadan hızlıdır; sona yakınsa yaklaşık iki kat yavaştır. Tablo 3.16, bir milyon öğeli bir dizide farklı konumlar için iki yöntemi karşılaştırır.

i	Üstel ($\approx 2 \log_2 i$)	İkili ($\approx \log_2 n$)
10	≈ 7	20
1.000	≈ 20	20
500.000	≈ 38	20

Tablo 3.16: Üstel ve ikili aramanın yoklama sayıları, $n = 10^6$.

Kırılma noktası $i \approx \sqrt{n}$ civarındadır: $2 \log_2 i = \log_2 n$ eşitliği $i = \sqrt{n}$ verir. Bir milyon öğede bu nokta 1000'dir ve tablo da bunu gösterir.

Üstel aramanın başlıca kullanım alanları şunlardır: boyu bilinmeyen sıralı listeler, büyük bir dosyada başa yakın kayıtlar ve sıralı veri akışları. Aynı fikrin bir uygulaması "dörtnala" (*galloping*) aramadır. Timsort sıralama algoritması iki sıralı diziyi birleştirirken bazen bir dizinin uzun bir parçası ötekinin önüne geçer. Timsort bu parçanın sonunu tek tek değil, dörtnala aramayla bulur.

> **Sık yapılan hata**
>
> Üst sınırı `Math.min(konum, n - 1)` yerine doğrudan `konum` almak. Son katlamada `konum` dizinin boyunu aşmış olabilir; ikili arama bu durumda `dizi[orta]` ile dizi dışını okur ve `ArrayIndexOutOfBoundsException` fırlatılır. 10 öğeli dizide 81 aranırken katlama `konum = 16` ile durur; sınır alınmazsa ilk orta *(8 + 16)/2 = 12* olur.

3.8 Karşılaştırma ve seçim

Yedi yöntemin her biri belli bir durumda en iyisidir. Doğru seçim için önce yöntemlerin maliyetlerini yan yana koymak, sonra aynı veri üzerinde nasıl davrandıklarını görmek gerekir.

3.8.1 Karşılaştırma tablosu

Algoritma	Önkoşul	En iyi	Ortalama	En kötü	Ek bellek
Doğrusal	yok	*O(1)*	*O(n)*	*O(n)*	*O(1)*
İkili	sıralı	*O(1)*	*O(log n)*	*O(log n)*	*O(1)*
Hash (tablo)	hash tablosu	*O(1)*	*O(1)*	*O(n)*	*O(m)*
Ara değer	sıralı, düzgün	*O(1)*	*O(log log n)*	*O(n)*	*O(1)*
Atlayarak	sıralı	*O(1)*	*O(√n)*	*O(√n)*	*O(1)*
Üstel	sıralı	*O(1)*	*O(log i)*	*O(log n)*	*O(1)*

Tablo 3.17: Arama algoritmalarının karşılaştırılması. *i*: aranan öğenin indisi, *m*: kova sayısı.

Tabloyu okurken üç noktayı akılda tutmak gerekir. Birincisi, hash tablosunun ortalama *O(1)*'i yük faktörünün sabit tutulmasına bağlıdır; tablo dolmaya bırakılırsa bu güvence kaybolur. İkincisi, sıralı dizi isteyen yöntemlerde dizi baştan sıralı değilse bir kez *O(n log n)* sıralama maliyeti eklenir. Üçüncüsü, en kötü durum sütunu algoritmanın verdiği güvencedir. Hash tablosu ve ara değer arama ortalamada çok hızlıdır, ama kötü bir girdiye karşı hiçbir güvence vermez. İkili arama ise her girdide *O(log n)*'dir. Bu öngörülebilirlik, onu çoğu durumda varsayılan seçim yapar.

3.8.2 Aynı dizide yoklama sayıları

Mertebeler büyük *n* için konuşur. Küçük bir dizide yöntemlerin gerçekte kaç yoklama yaptığını görmek de öğreticidir. Atlayarak ve üstel arama örneklerindeki on öğeli dizi kullanılıyor:

```
dizi = {2, 9, 30, 32, 38, 47, 61, 69, 79, 81}
```

Aranan	Doğrusal	İkili	Atlayarak	Ara değer	Üstel	Sonuç
69	8 (0–7)	2 (4, 7)	5 (2, 5, 8, 6, 7)	1 (7)	7 (0, 1, 2, 4, 8, 6, 7)	7
47	6 (0–5)	3 (4, 7, 5)	5 (2, 5, 3, 4, 5)	1 (5)	8 (0, 1, 2, 4, 8, 6, 4, 5)	5
2	1 (0)	3 (4, 1, 0)	2 (2, 0)	1 (0)	1 (0)	0
50	10 (0–9)	4 (4, 7, 5, 6)	4 (2, 5, 8, 6)	1 (5)	8 (0, 1, 2, 4, 8, 6, 4, 5)	*−1*

Tablo 3.18: Aynı dizide yoklama sayıları. Her hücrede yoklama sayısı ve parantez içinde yoklanan indisler verilmiştir.

Tablodan birkaç ders çıkar. Dizi neredeyse düzgün dağıldığı için ara değer arama her sorguyu tek yoklamada sonuçlandırır. 69 için tahmin $0 + (69 - 2) \cdot 9 / 79 = 603/79 \approx 7{,}6 \to 7$'dir ve doğrudan hedefe düşer. 50 aranırken ara değer arama 5. indisteki 47'yi yoklar. Ardından `bas = 6` olur ve aranan, yeni aralığın alt ucu olan 61'den küçük kaldığı için döngü hemen biter; sonuç -1'dir.

İkili arama her sorguda en fazla 4 yoklama yapar; bu, $\lfloor \log_2 10 \rfloor + 1 = 4$ sınırıdır. Atlayarak arama bloğa göre değişir. 47 aranırken ikinci bloğun son öğesi tam olarak 47'dir, ama kod blok sonunda durmaz; bloğun başına dönüp 3, 4, 5 indislerini tarar. Üstel arama ise aranan değer dizinin başında olduğunda parlar: 2 için tek yoklama yeter. Aranan sona yaklaştıkça katlama ve ikili arama aşamaları birlikte ikili aramadan pahalı olur. 47 ve 50 aramalarında 4. indis iki kez yoklanır: bir kez katlamada, bir kez ikili aramada.

Bu kadar küçük bir dizide farklar birkaç yoklamadır. Dizi büyüdükçe mertebe farkları belirleyici olur. Bir milyon öğede doğrusal arama ortalama yarım milyon, atlayarak arama en fazla yaklaşık iki bin, ikili arama en fazla 20 yoklama yapar.

3.8.3 Hangi algoritma ne zaman?

Seçim, verinin özelliklerine ve sorguların biçimine bağlıdır:

- **Doğrusal arama:** sırasız ya da küçük veri, bağlı liste, tek seferlik arama.
- **İkili arama:** sıralı dizi ve çok sayıda sorgu; genel amaçlı varsayılan seçim.
- **Hash tablosu:** yalnız tam eşleşme sorgusu ve çok sayıda ekleme ile arama; sıra gerekmiyor.
- **Ara değer arama:** sıralı ve düzgün dağılmış sayısal anahtarlar, örneğin sıralı kimlik numaraları.
- **Atlayarak arama:** geriye gitmenin pahalı olduğu sıralı ortamlar, örneğin bant ya da blok blok okunan disk.
- **Üstel arama:** boyu bilinmeyen sıralı liste ya da aranan değerin başa yakın olduğu durumlar.

Bu kurallar bir soru sırasına dönüştürülebilir. İlk soru, verinin sıralı olup olmadığıdır. Veri sırasızsa iki yol vardır: tek ya da birkaç arama yapılacaksa doğrusal arama, çok sayıda arama yapılacaksa bir hash tablosu kurmak. Veri sıralıysa varsayılan seçim ikili aramadır. Dağılım düzgünse ara değer arama, boy bilinmiyorsa üstel arama, geri gitmek pahalıysa atlayarak arama düşünülür.

Örnek 3.13

Üç senaryoda seçim şöyle yapılır.

1. **Bir derleyicinin sembol tablosu.** Değişken adları sürekli eklenir ve aranır; sıraya ihtiyaç yoktur, yalnız "bu ad tanımlı mı?" sorusu sorulur. Hash tablosu en uygunudur: ekleme de arama da ortalama *O(1)*'dir.
2. **Bir sözlük uygulamasında önek araması.** "ara" ile başlayan bütün kelimeler isteniyor. Hash tablosu kelimeleri dağıttığı için bu sorguya yardım edemez. Sıralı kelime dizisinde ilk konum ikili aramasıyla "ara"nın ekleneceği yer bulunur ve oradan ileri okunur.
3. **Saniyede bir kayıt eklenen sıralı bir günlük dosyası.** Dosya sürekli büyür ve sorgular genellikle son kayıtlarla ilgilidir. Dosyanın sonundan geriye doğru üstel arama yapmak, aranan kayıt sona yakın olduğu için *O(log i)* maliyetle biter; dosyanın tam boyunu bilmek bile gerekmez.

3.9 Sık yapılan hatalar

Arama algoritmaları kısa kodlardır, ama hata yapmaya çok elverişlidir. Hataların çoğu sınır koşullarında ortaya çıkar ve küçük örneklerle yapılan denemelerde gözden kaçar.

Sık yapılan hata

Sırasız dizide ikili arama. Program hata vermez, ama yanlış sonuç döndürebilir. `{30, 10, 20}` dizisinde 30 aranırsa orta öğe 10'dur; 30 daha büyük olduğu için sağa gidilir, 20 de küçük kalır ve *−1* döner. Oysa 30, 0. indistedir. İkili, ara değer, atlayarak ve üstel arama sıralı dizi ister.

Sık yapılan hata

Sınırlarda *±1* hatası. `while (bas < son)` tek öğeli aralığı atlar. `son = orta` ataması ise aralığı küçültmeyebilir ve döngüyü sonsuza sokabilir.

Sık yapılan hata

Orta hesabında taşma. Çok büyük dizilerde `(bas + son) / 2` taşar ve negatif bir indis verir. `bas + (son - bas) / 2` taşmaz.

Sık yapılan hata

Ara değer aramada bölme. `dizi[son] == dizi[bas]` ise sıfıra bölme olur. Aranan uç değerlerin dışındaysa tahmini indis dizinin dışına çıkar. Bu yüzden önce aralık denetlenmelidir.

Sık yapılan hata

Dolu hash tablosu. Doğrusal yoklamada boş kova kalmazsa olmayan bir anahtarın araması hiç bitmez. Yük faktörü sınırlanmalı ya da *m* yoklamadan sonra durulmalıdır.

Sık yapılan hata

Üstel aramada sınır. `Math.min(konum, n - 1)` unutulursa `ArrayIndexOutOfBoundsException` fırlatılır. Son aşamada ikili arama yerine doğrusal tarama yapılırsa da *O(log i)* maliyeti bozulur ve *O(i)*'ye çıkar.

Bu hataların ortak bir dersi vardır: arama kodu yazıldıktan sonra sınır durumlarıyla denenmelidir. En az şu durumlar denenmelidir: boş dizi, tek öğeli dizi, aranan değerin ilk ya da son öğe olması, bütün değerlerden küçük ya da büyük olması ve iki öğe arasına düşmesi. İkili aramanın doğru yazımı zor olduğu için Java'da mümkün olduğunca `Arrays.binarySearch` ya da `Collections.binarySearch` kullanmak da iyi bir alışkanlıktır.

3.10 Özet

- **Arama problemi:** bir dizide aranan değerin indisini bulmak; yoksa *−1* döndürmek. Maliyet, yoklama (karşılaştırma) sayısıyla ölçülür.
- **Doğrusal arama:** baştan sona tek tek karşılaştırır; önkoşul yoktur. Başarılı aramada ortalama *(n + 1)/2*, en kötü durumda *n* karşılaştırma yapar: *O(n)*. Sırasız veride daha iyisi yoktur.
- **Bekçili doğrusal arama:** aranan değeri son hücreye yazarak döngüdeki `i < n` sınamasını kaldırır; karşılaştırma sayısı yaklaşık *2n*'den *n + 2*'ye iner, mertebe değişmez.
- **İkili arama:** sıralı dizide orta öğeyle karşılaştırır ve yarıyı eler. En fazla $\lfloor \log_2 n \rfloor + 1$ karşılaştırma yapar: *O(log n)*. Yinelemeli sürüm *O(1)*, özyinelemeli sürüm *O(log n)* ek bellek kullanır. Sıralı dizi, örtük bir dengeli ikili arama ağacıdır.
- **İkili aramada ayrıntılar:** orta hesabı taşmaya karşı `bas + (son - bas) / 2` ile yapılır; sınırlar *±1* ile güncellenir; ilk konum için eşitlikte sonuç kaydedilip sola devam edilir.
- **Hash ile arama:** yeri *h(anahtar)* ile hesaplar. Çakışmalar doğrusal yoklama ya da zincirleme ile çözülür. Yük faktörü $\alpha = n/m$ sabit tutulursa ortalama *O(1)*; en kötü durum *O(n)*. Doğrusal yoklamada birincil kümelenme α 1'e yaklaşırken maliyeti hızla büyütür.

- **Ara değer arama:** konumu uç değerlere göre orantıyla tahmin eder. Düzgün dağılımda *O(log log n)*, çarpık dağılımda *O(n)*'dir.
- **Atlayarak arama:** $\sqrt{n}$'lik blokları atlar, sonra bloğun içinde doğrusal arar. En kötü durum yaklaşık $2\sqrt{n} - 1$ yoklama, yani $O(\sqrt{n})$'dir; geri gitmenin pahalı olduğu ortamlarda anlamlıdır.
- **Üstel arama:** sınırı 1, 2, 4, … diye katlar, sonra *[konum/2, konum]* aralığında ikili arama yapar. Maliyet aranan öğenin indisine bağlıdır: *O(log i)*. Boyu bilinmeyen listelerde ve başa yakın aramalarda kullanılır.
- **Seçim:** veri sırasızsa doğrusal arama ya da hash tablosu; sıralıysa varsayılan ikili arama; düzgün dağılımda ara değer, bilinmeyen boyda üstel, pahalı geri dönüşte atlayarak arama.

3.11 Kavram listesi

arama problemi *search problem* · yoklama *probe* · en iyi durum *best case* · ortalama durum *average case* · en kötü durum *worst case* · doğrusal arama *linear search* · bekçi *sentinel* · ikili arama *binary search* · logaritmik *logarithmic* · böl ve fethet *divide and conquer* · azalt ve fethet *decrease and conquer* · yineleme bağıntısı *recurrence relation* · hash tablosu *hash table* · hash fonksiyonu *hash function* · kova *bucket* · çakışma *collision* · açık adresleme *open addressing* · doğrusal yoklama *linear probing* · birincil kümelenme *primary clustering* · zincirleme *(separate) chaining* · yük faktörü *load factor* · ara değer arama *interpolation search* · atlayarak arama *jump search* · üstel arama *exponential search*

3.12 Kendini Dene

1. `gozcuIleAra(dizi, 5, 9)` çağrısı `dizi = {12, 5, 9, 20, 7}` için çalışıyor. Metot ne döndürür ve toplam kaç karşılaştırma yapılır?

   Cevap

   Son hücreye bekçi olarak 9 yazılır, `sonEleman = 7`. Döngü 12 ≠ 9 ve 5 ≠ 9'dan sonra 2. indisteki gerçek 9'da durur: 3 karşılaştırma. 7 geri konur. `i < n - 1`, yani *2 < 4* doğrudur; kısa devre `||` nedeniyle ikinci koşul hiç sınanmaz. Metot 2 döndürür; toplam 4 karşılaştırma yapılır.

2. `{4, 9, 13, 20, 26, 31, 40, 48, 55, 61, 77}` dizisinde yinelemeli ikili arama 50'yi arıyor.

Hangi indisler yoklanır? Döngü bittiğinde `bas` ve `son` kaçtır ve `bas` neyi gösterir?

Cevap

Yoklanan indisler 5 (31 < 50, `bas = 6`), 8 (55 > 50, `son = 7`), 6 (40 < 50, `bas = 7`) ve 7'dir (48 < 50, `bas = 8`). Döngü `bas = 8`, `son = 7` ile biter ve *−1* döner. `bas`, 50'nin sırayı bozmadan ekleneceği konumdur: 48 ile 55 arasındaki 8. indis.

3. `ilkKonum({1, 3, 3, 3, 3, 3, 8}, 3)` çağrısında her turdaki `orta` ve `sonuc` değerleri nedir? Aynı dizide son 3'ün konumunu bulan sürüm hangi indisleri yoklar?

Cevap

Tur 1: `orta = 3`, eşit, `sonuc = 3`, `son = 2`. Tur 2: `orta = 1`, eşit, `sonuc = 1`, `son = 0`. Tur 3: `orta = 0`, 1 < 3, `bas = 1`. Döngü biter, 1 döner. Son konum sürümü eşitlikte `bas = orta + 1` ile sağa gider: 3, 5, 6 indislerini yoklar ve 5 döndürür. 3 değeri *5 − 1 + 1 = 5* kez geçer.

4. *m = 7*, *h(x) = x % 7* ve doğrusal yoklama kullanan boş bir tabloya 15, 22, 8, 29, 3 sırayla ekleniyor. Son tablo nedir? Ardından 36 aranırsa kaç kova yoklanır?

Cevap

15, 22, 8 ve 29'un dördü de 1 numaralı kovaya düşer: 15 → 1, 22 → 2, 8 → 3, 29 → 4 (yoklamalar 1, 2, 3, 4). 3'ün kovası 3'tür, ama 3 ve 4 doludur; 3, 5'e yerleşir. Tablo `[–, 15, 22, 8, 29, 3, –]` olur. 36 için *h = 1*: 1, 2, 3, 4, 5 kovalarındaki anahtarlar aşılır ve 6 numaralı boş kovada durulur. 6 yoklama yapılır, sonuç *−1*'dir.

5. Aynı beş anahtar *m = 7* kovalı bir zincirleme tablosuna eklenirse yük faktörü kaçtır, hangi listeler oluşur? 36 aranırken kaç anahtar karşılaştırılır?

Cevap

α = 5/7 ≈ 0,71. Kova 1'in listesi 15 → 22 → 8 → 29, kova 3'ünki yalnız 3'tür; diğer kovalar boştur. 36 aranırken kova 1'in listesindeki 4 anahtar karşılaştırılır ve liste biter. Bu örnekte zincirleme de yavaştır: sorunun kaynağı hash fonksiyonunun anahtarları tek kovada toplamasıdır, çakışma çözüm yöntemi değil.

6. 5, 10, 15, …, 100 dizisinde (20 öğe) ara değer arama 65'i ve 63'ü kaç yoklamada sonuçlandırır?

İkili arama 65 için hangi indisleri yoklar?

Cevap

65: *indis = 0 + (65 − 5) · 19 / 95 = 1140/95 = 12*, `dizi[12] = 65`; 1 yoklama. 63: *58 · 19 / 95 = 1102/95 ≈ 11,6* → 11, `dizi[11] = 60 < 63`, `bas = 12`. Artık *63 < dizi[12] = 65* olduğu için döngü biter: 1 yoklama, sonuç *−1*. İkili arama 65 için 9, 14, 11, 12 indislerini yoklar: 4 yoklama.

7. Atlayarak aramada *n = 25* ve *n = 400* için en iyi blok boyu ve en kötü durumdaki yoklama sayısı nedir? *n = 400* için blok boyu yanlışlıkla 10 seçilirse ne olur?

Cevap

n = 25: *m = 5*, *25/5 + 4 = 9* yoklama. *n = 400*: *m = 20*, *20 + 19 = 39* yoklama. *m = 10* seçilirse *400/10 + 9 = 49* yoklama olur; en iyi seçimden 10 fazla.

8. Bir milyon öğeli bir dizide `dizi[i] = 2i`'dir. Üstel arama 74'ü (indis 37) ararken hangi indisleri yoklar? İkili arama en fazla kaç yoklama yapardı?

Cevap

Önce 0. indis: 0 ≠ 74. Katlama 1, 2, 4, 8, 16, 32 indislerinde değer 74'ten küçük çıkar, 64. indiste 128 > 74 olur ve durulur. Aralık *[32, 64]*'tür. İkili arama 48, 39, 35, 37 indislerini yoklar ve 37'yi bulur. Toplam 12 yoklama: 0, 1, 2, 4, 8, 16, 32, 64, 48, 39, 35, 37. İkili arama en fazla $\lfloor \log_2 10^6 \rfloor + 1 = 20$ yoklama yapardı.

9. `bas = 1.500.000.000` ve `son = 1.600.000.000` iken Java'da `(bas + son) / 2` ve `bas + (son - bas) / 2` ne verir?

Cevap

Gerçek toplam 3.100.000.000'dir ve $2^{31} - 1$'i aşar. `int` toplamı $3.100.000.000 - 2^{32} = -1.194.967.296$ olur; ikiye bölünce *−597.483.648* çıkar ve `dizi[orta]` erişimi `ArrayIndexOutOfBoundsException` fırlatır. Güvenli yazım *1.500.000.000 + 50.000.000 = 1.550.000.000* verir.

3.13 İleri okuma

- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 3.1 ve 3.4: sıralı dizide ikili aramayla çalışan bir sembol tablosunu ve hash tablolarını (ayrı zincirleme, doğrusal yoklama) Java koduyla ve deneysel ölçümlerle anlatır.

- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2. baskı, 1998), Kesim 6.1, 6.2.1 ve 6.4: bekçili aramayı, ikili aramanın tarihini ve doğrusal yoklama formüllerinin çıkarılışını ayrıntılı biçimde verir.
- J. Bentley, *Programming Pearls* (2. baskı, 2000), 4. ve 5. kesimler: ikili aramanın doğru yazılmasını ve doğruluğunun gerekçelendirilmesini örnek olarak işler.
- J. Bloch, "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken", Google Research Blog, 2 Haziran 2006: orta hesabındaki taşma hatasının Java kütüphanesinde nasıl fark edildiğini anlatan kısa bir yazı.

3.14 Kaynaklar

1. J. L. Bentley, A. C. Yao, "An Almost Optimal Algorithm for Unbounded Searching", *Information Processing Letters* 5(3), 1976.
2. J. Bentley, *Programming Pearls*, 2. baskı, Addison-Wesley, 2000.
3. J. Bloch, "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken", Google Research Blog, 2006.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 2.3, 11).
5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. baskı, Addison-Wesley, 1998.
6. Y. Perl, A. Itai, H. Avni, "Interpolation Search — A Log Log N Search", *Communications of the ACM* 21(7), 1978.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 3.1 İkili arama, 3.4 Hash tabloları).
8. B. Shneiderman, "Jump Searching: A Fast Sequential Search Technique", *Communications of the ACM* 21(10), 1978.

KISIM II

Çizge ve Dizgi Algoritmaları

Bölüm 4–5

KISIM II · BÖLÜM 4

Çizge Algoritmaları

Çizgeler, yol ağlarından sosyal ağlara kadar düğümler ve bağlantılarla ifade edilebilen her yapıyı modellemekte kullanılır. Bölüm, çizgenin bellekte gösterimi, derinlik öncelikli (DFS) ve genişlik öncelikli (BFS) gezinme, Tarjan ve Kosaraju algoritmalarıyla güçlü bağlantılı bileşenler, topolojik sıralama ve iki parçalılık testiyle başlar. Ardından gevşetme işlemi üzerine kurulan Dijkstra, Bellman-Ford, Floyd-Warshall ve A* en kısa yol algoritmaları, kesim ve döngü özelliklerine dayanan Kruskal, Prim, Borůvka ve ters silme minimum kapsayan ağaç algoritmaları işlenir. Son kısımda artık ağ, artış yolu ve maksimum akış–minimum kesim teoremi ile Ford-Fulkerson, Edmonds-Karp, Dinic ve itme–yeniden etiketleme algoritmaları ele alınır.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum04/src`

BU BÖLÜMÜN NOTLARI

4.1 Çizge Gezinme Algoritmaları

4.2 En Kısa Yol Algoritmaları

4.3 Minimum Kapsayan Ağaç Algoritmaları

4.4 Ağ Akışı

KISIM II · BÖLÜM 4.1

Çizge Gezinme Algoritmaları

Şehirleri birbirine bağlayan yollar, bir sosyal ağdaki arkadaşlıklar, internetteki yönlendiriciler ve bir derleyicinin modülleri arasındaki bağımlılıklar aynı yapıyla anlatılır: nesneler ve aralarındaki bağlantılar. Bu yapının adı çizgedir. Bir çizge üzerinde soru sormanın ilk adımı, onda düzenli biçimde gezinmektir. Bu not önce çizge kavramlarını ve çizgenin bellekte tutulmasını kısaca ele alır. Ardından iki temel gezinme yöntemini ayrıntılı olarak işler: derinlik öncelikli arama (DFS) ve genişlik öncelikli arama (BFS). Son kısımda bu iki yöntemin üzerine kurulan algoritmalar yer alır: ağırlıksız çizgede en kısa yol, labirent çözme, Tarjan ve Kosaraju algoritmalarıyla güçlü bağlantılı bileşenler, topolojik sıralama ve iki parçalılık testi. Her algoritma için kod, adım adım izleme ve maliyet hesabı verilir.

ÖĞRENME ÇIKTILARI

Çizge kavramları
Yönlü, yönsüz, ağırlıklı, iki parçalı ve tam çizgeyi, ağacı ve basit olmayan çizgeyi ayırt etmek; bir çizgeyi komşuluk listesi ve komşuluk matrisiyle göstermek, iki gösterimin bellek ve zaman maliyetini karşılaştırmak.

DFS
Derinlik öncelikli aramayı özyinelemeli olarak ve açık bir yığınla yazmak; ziyaret sırasını adım adım çıkarmak; geri dönüşü ve iki sürümün neden farklı sıra verdiğini açıklamak.

BFS
Genişlik öncelikli aramayı kuyrukla yazmak; katman, ebeveyn ve ağırlıksız en kısa yol kavramlarını kullanmak; labirent örneğinde en kısa yolu bulmak.

Tarjan ve uygulamalar
Güçlü bağlantılı bileşenleri indeks ve enDüşükBağ değerleriyle bulmak; Kosaraju algoritmasıyla karşılaştırmak; topolojik sıralama ve iki parçalılık testini uygulamak; bütün gezinme algoritmalarının $O(V + E)$ zamanını gerekçelendirmek.

4.1.1 Çizgeler ve gösterimleri

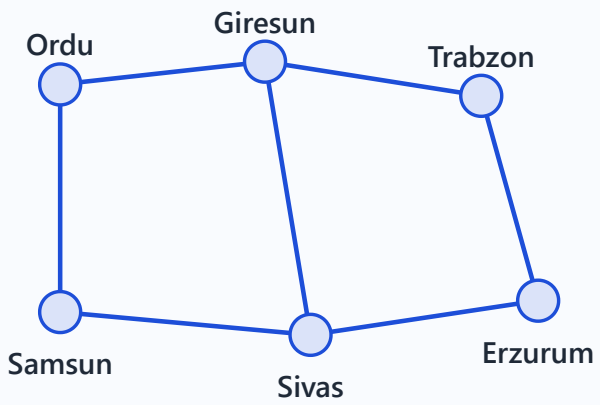
Dünyadaki pek çok düzen bir ağ gibidir. Şehirler yollarla, insanlar tanışıklıklarla, bilgisayarlar kablolarla birbirine bağlanır. Bu ağların ortak yanı, birkaç nesne ve bu nesnelerin bazı çiftleri arasındaki bağlantılardır. Nesnelerin ne olduğu ya da bağlantının fiziksel mi, mantıksal mı olduğu önemli değildir. Matematikte bu ortak yapıya çizge denir. Çizge algoritmaları da bu ağlar üzerinde sorulan soruları yanıtlar: İki şehir arasında yol var mı? En kısa yol hangisi? Ağ hangi gruplara ayrılır? Bir kablo koparsa ağ ikiye bölünür mü?

4.1.1.1 Çizge nedir?

Tanım 4.1.1

Bir **çizge** (*graph*) $G = (V, E)$ iki kümeden oluşur. V, **düğümlerin** (*vertex, node*) kümesidir. E ise **kenarların** (*edge*) kümesidir; her kenar iki düğümü birbirine bağlar. Bir kenarın bağladığı iki düğüme bu kenarın **uçları**, aralarında kenar bulunan iki düğüme de **komşu** denir.

Algoritmaların maliyeti çizgenin iki büyüklüğüne bağlıdır. Bu notta V harfi hem düğüm kümesini hem de düğüm sayısını, E harfi de hem kenar kümesini hem de kenar sayısını gösterir. Hangisinin kastedildiği bağlamdan anlaşılır. Karmaşıklıklar bu iki sayı cinsinden yazılır: $O(V + E)$, $O(V^2)$ gibi. Tek bir n yerine iki ayrı sayı kullanılmasının nedeni, aynı düğüm sayısında kenar sayısının çok farklı olabilmesidir. On düğümlü bir çizgede hiç kenar olmayabilir, ama kırk beş kenar da olabilir.



Şekil 4.1.1: Altı şehir ve aralarındaki yollar. Her şehir bir düğüm, her yol bir kenardır: $V = 6$, $E = 7$.

Şekil 4.1.1 küçük bir yol ağını gösterir. Ordu'dan Erzurum'a gitmek için Ordu–Giresun–Trabzon–Erzurum ya da Ordu–Samsun–Sivas–Erzurum gibi birkaç yol izlenebilir. Bir düğümden başlayıp her adımda bir kenar boyunca komşuya geçerek oluşturulan düğüm dizisine **yol** (*path*) denir. Yolun

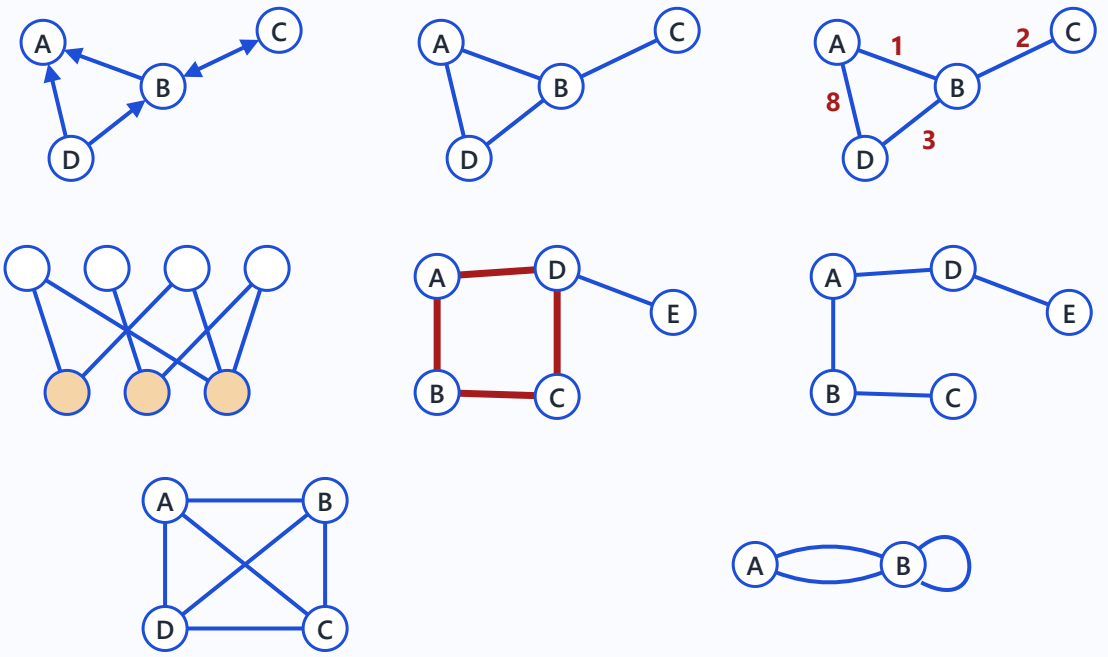
uzunluğu, içerdiği kenar sayısıdır; Ordu–Giresun–Trabzon–Erzurum yolunun uzunluğu 3'tür. Başladığı düğüme geri dönen ve kenar tekrarlamayan yola **döngü** (*cycle*) denir. Şekildeki Ordu–Giresun–Sivas–Samsun–Ordu dizisi bir döngüdür.

Bir düğüme bağlı kenar sayısına o düğümün **derecesi** (*degree*) denir. Şekilde Giresun'un derecesi 3'tür (Ordu, Trabzon, Sivas), Samsun'unki 2'dir. Derecelerin toplamı her zaman kenar sayısının iki katıdır, çünkü her kenar iki ucundaki düğümün derecesine birer katkı yapar. Şekilde dereceler 2, 3, 2, 2, 3, 2'dir; toplamları $14 = 2 \cdot 7$ eder. Bu basit gözlem, gezinme algoritmalarının maliyet hesabında doğrudan kullanılacak: komşu listelerinin toplam uzunluğu yönsüz bir çizgede $2E$'dir.

Herhangi iki düğümü arasında bir yol bulunan çizgeye **bağlı** (*connected*) çizge denir. Şekildeki ağ bağlıdır: her şehirden her şehre gidilebilir. Bağlı olmayan bir çizge, kendi içinde bağlı olan ve birbirine hiç kenarla bağlanmayan parçalara ayrılır. Bu parçaların her birine **bağlantılı bileşen** (*connected component*) denir.

4.1.1.2 Çizge türleri

Kenarların bir yönü, bir ağırlığı ya da özel bir düzeni olabilir. Bu özellikler hangi algoritmanın kullanılacağını belirler. Şekil 4.1.2 en sık karşılaşılan sekiz türü küçük örneklerle gösterir.



Şekil 4.1.2: Çizge türleri, soldan sağa ve yukarıdan aşağıya: yönlü, yönsüz, ağırlıklı, iki parçalı, döngülü, ağaç, tam ve basit olmayan çizge. İki parçalı çizgede üstteki dört düğüm bir kümeyi, alttaki üç düğüm öteki kümeyi oluşturur.

- **Yönlü çizge** (*directed graph*): Her kenarın bir başlangıcı ve bir bitişi vardır; kenar yalnız ok yönünde geçilir. $u \to v$ kenarı u'dan v'ye gitmeyi sağlar, ama tersini sağlamaz. Tek yönlü sokaklar, bir web sayfasından ötekine verilen bağlantılar ve "A görevi B'den önce bitmeli" türünden bağımlılıklar yönlü çizgedir. Yönlü çizgede bir düğüme giren kenar sayısına **giriş derecesi**, düğümden çıkan kenar sayısına **çıkış derecesi** denir.
- **Yönsüz çizge** (*undirected graph*): Kenarlar iki yönde de geçilir. Arkadaşlık ilişkisi simetrik olduğu için bir sosyal ağ yönsüz çizgeyle anlatılır. Yönsüz bir kenar, iki yönlü iki kenar gibi düşünülebilir; bellekte de çoğu zaman böyle tutulur.
- **Ağırlıklı çizge** (*weighted graph*): Her kenara bir sayı atanır. Bu sayı bir uzaklık, süre, maliyet ya da kapasite olabilir. Yol ağında kilometreler, bilgisayar ağında gecikme süreleri ağırlıktır. Ağırlıklı çizgede bir yolun uzunluğu kenar sayısıyla değil, ağırlıkların toplamıyla ölçülür.
- **İki parçalı çizge** (*bipartite graph*): Düğümler iki kümeye ayrılabilir ve her kenar farklı kümelerdeki iki düğümü bağlar; aynı kümedeki iki düğüm arasında kenar yoktur. Öğrenciler ile dersler, işçiler ile işler arasındaki eşleştirmeler bu yapıdadır.
- **Döngülü çizge** (*cyclic graph*): En az bir döngü içerir. Şekildeki örnekte A–D–C–B–A, başladığı yere dönen bir yoldur. Döngü içermeyen çizgeye **döngüsüz** (*acyclic*) denir. Döngüsüz yönlü çizgeler (*directed acyclic graph*, DAG) görev planlamada özel bir yer tutar.
- **Ağaç** (*tree*): Bağlı ve döngüsüz yönsüz çizgedir. Ağaçta herhangi iki düğüm arasında tam olarak bir yol vardır ve V düğümlü bir ağacın kenar sayısı her zaman $V - 1$'dir. İkili arama ağacı, heap ve dosya sistemi dizinleri bu yapının bilinen örnekleridir.
- **Tam çizge** (*complete graph*): Her düğüm çifti bir kenarla bağlıdır. n düğümlü tam çizgede $n(n-1)/2$ kenar vardır. Her düğüm öteki $n - 1$ düğüme bağlanır; her kenar böylece iki kez sayıldığı için toplam ikiye bölünür. Dört düğümde $4 \cdot 3/2 = 6$, on düğümde $10 \cdot 9/2 = 45$ kenar eder. Yönsüz basit bir çizgenin kenar sayısı bu değeri aşamaz.
- **Basit olmayan çizge**: Aynı iki düğüm arasında birden çok kenar (**çoklu kenar**) ya da bir düğümü kendisine bağlayan kenar (**öz döngü**, *self-loop*) içerir. Bu ikisini içermeyen çizgeye **basit çizge** denir. Aksi belirtilmedikçe bu notta çizgeler basittir; tek istisna, güçlü bağlantılı bileşen örneğindeki öz döngüdür.

Bu türler birbirini dışlamaz. Bir yol haritası hem yönlü (tek yönlü sokaklar), hem ağırlıklı (uzunluklar), hem de döngülü olabilir. Bir ağaç her zaman iki parçalıdır: kökten uzaklığı çift olan düğümler bir kümeye, tek olanlar öteki kümeye konursa her kenar farklı kümeleri bağlar. Bu gözlem, iki parçalılık testinin de temelidir.

4.1.1.3 Çizge algoritmalarının çeşitleri

Çizgeler üzerinde çalışan algoritmalar, yanıtladıkları soruya göre dört büyük gruba ayrılır:

1. **Gezinme** (*graph traversal*): Bir düğümden başlayıp ulaşılabilen bütün düğümlerde düzenli bir sırayla gezinmek. Derinlik öncelikli arama bir daldan sonuna kadar ilerler, genişlik öncelikli arama ise ağı katman katman tarar. Bu notun konusu bu gruptur.

2. **En kısa yol** (*shortest path*): Bir başlangıçtan öteki düğümlere giden en kısa yolları bulmak. Ağırlıklı çizgede Dijkstra algoritması bu işi yapar.
3. **Minimum kapsayan ağaç** (*minimum spanning tree*, MST): Bütün düğümleri en az toplam maliyetle birbirine bağlayan kenarları seçmek. Kruskal algoritması bunun bilinen bir örneğidir.
4. **Ağ akışı** (*network flow*): Kenarları kapasiteli bir ağda kaynaktan hedefe en çok ne kadar akış gönderilebileceğini bulmak.

Gezinme bu gruplar arasında en temel olanıdır, çünkü öteki algoritmaların çoğu içinde bir gezinme taşır. Dijkstra algoritması, kuyruğu öncelikli kuyrukla değiştirilmiş bir genişlik öncelikli aramaya benzer. Akış algoritmaları her turda bir gezinmeyle artırılabilir yol arar. Tablo 4.1.1 üç algoritmanın gündelik hayattaki karşılığını verir.

Algoritma	Gündelik örnek
DFS	Bir labirentten çıkış ararken bir koridoru sonuna kadar izleyip geri dönmek
BFS	Bir haberin şehirde kişiden kişiye yayılması: önce yakın çevre, sonra onların çevresi
Dijkstra	Bir kuryenin teslimatı en kısa sürede yapacağı rotayı bulmak

Tablo 4.1.1: Çizge algoritmalarının gündelik karşılıkları.

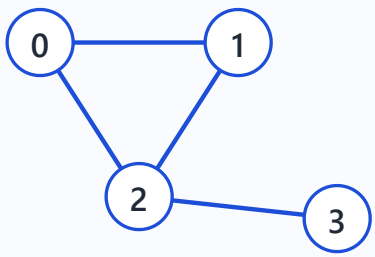
4.1.1.4 Çizgeyi bellekte tutmak

Bir çizgeyle çalışan programın ilk kararı, çizgeyi bellekte nasıl tutacağıdır. Bu karar algoritmanın hızını doğrudan etkiler. İki yaygın gösterim vardır: komşuluk listesi ve komşuluk matrisi. İkisinde de düğümler *0, 1, ..., V − 1* diye numaralanır; böylece bir düğüm, bir dizinin indisi olarak kullanılabilir.

Tanım 4.1.2

Komşuluk listesi (*adjacency list*): Her düğüm için o düğümün komşularını tutan bir liste. *V* tane liste bir dizide durur; `komsu[u]`, *u*'nun komşularıdır.

Komşuluk matrisi (*adjacency matrix*): *V* × *V* boyutunda bir tablo. *u* ile *v* arasında kenar varsa `M[u][v] = 1`, yoksa `M[u][v] = 0`'dır. Ağırlıklı çizgede 1 yerine kenarın ağırlığı yazılır.



Şekil 4.1.3: Dört düğümlü yönsüz örnek çizge: *V* = 4, *E* = 4. Kenarlar 0–1, 0–2, 1–2 ve 2–3'tür.

Şekil 4.1.3'teki çizgenin komşuluk listesi şöyledir:

```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

Yönsüz çizgede her kenar iki listede görünür. 0–1 kenarı hem 0'ın listesine 1 olarak, hem de 1'in listesine 0 olarak yazılır. Bu yüzden listelerin toplam uzunluğu $2E = 8$'dir. Yönlü çizgede ise $u \to v$ kenarı yalnız u'nun listesine eklenir ve toplam uzunluk E olur. Aynı çizgenin komşuluk matrisi Tablo 4.1.2'de verilmiştir.

	0	**1**	**2**	**3**
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Tablo 4.1.2: Şekildeki çizgenin komşuluk matrisi.

Yönsüz çizgenin matrisi köşegene göre simetriktir: `M[u][v] = M[v][u]`. Basit çizgede öz döngü olmadığı için köşegen sıfırdır. Matrisin bir satırındaki 1'lerin sayısı o düğümün derecesine eşittir; 2. satırda üç tane 1 vardır ve 2'nin derecesi 3'tür.

Java'da komşuluk listesi bir liste dizisi olarak kurulur. Aşağıdaki parça, kenarları `{u, v}` çiftleri hâlinde veren bir `kenarlar` dizisinden yönsüz bir çizgenin listesini oluşturur:

```
List<Integer>[] komsu = new List[n];
for (int v = 0; v < n; v++)
  komsu[v] = new ArrayList<>();
for (int[] k : kenarlar) {
  komsu[k[0]].add(k[1]);
  komsu[k[1]].add(k[0]);  // yönsüz: ters yön de eklenir
}
```

İlk döngü her düğüme boş bir liste açar: $O(V)$. İkinci döngü her kenar için iki ekleme yapar: $O(E)$. Kurulum $O(V + E)$ zaman alır. Kenarlar `(0, 1), (0, 2), (1, 2), (2, 3)` sırasıyla verilirse sonuç tam olarak yukarıdaki listedir. Yönlü çizgede ikinci `add` satırı yazılmaz. Bu notun bütün Java kodları bu biçimdeki `List<Integer>[] komsu` dizisini kullanır.

İki gösterimin maliyetleri Tablo 4.1.3'te karşılaştırılmıştır.

İşlem	Komşuluk listesi	Komşuluk matrisi
Bellek	$O(V + E)$	$O(V^2)$
***u*'nun bütün komşuları**	$O(\text{derece}(u))$	$O(V)$
(u, v) kenarı var mı?	$O(\text{derece}(u))$	$O(1)$
Gezinme (DFS ya da BFS)	$O(V + E)$	$O(V^2)$

Tablo 4.1.3: Komşuluk listesi ile komşuluk matrisinin maliyetleri.

Tablodaki her satırın bir gerekçesi vardır. Liste, düğüm başına bir liste başı ve kenar başına bir ya da iki öğe tutar; bu yüzden belleği $V + E$ ile orantılıdır. Matris ise kenar sayısından bağımsız olarak V^2 hücre ayırır. Bir düğümün komşularını gezmek, listede yalnız o düğümün listesini okumak demektir. Matriste ise satırın bütün V hücresine bakmak gerekir, çünkü hangi hücrelerin 1 olduğu önceden bilinmez. Belirli bir kenarın varlığını sormak matriste tek bir hücre okumaktır; listede ise u'nun listesini baştan taramak gerekir.

Örnek 4.1.1

$V = 1000$ düğümlü ve $E = 3000$ kenarlı yönsüz bir çizgede BFS'in yaklaşık iş miktarı iki gösterimle karşılaştırılsın.

1. **Komşuluk listesi.** BFS her düğümü bir kez kuyruktan alır ve her düğümün listesini bir kez okur. Listelerin toplam uzunluğu $2E = 6000$'dir. İş yaklaşık $V + 2E = 1000 + 6000 = 7000$ temel adımdır.
2. **Komşuluk matrisi.** Her düğüm için matrisin bir satırı baştan sona taranır. İş $V \cdot V = 1.000.000$ hücre okumasıdır.
3. **Oran.** $1.000.000 / 7000 \approx 143$. Matris bu çizgede yaklaşık 140 kat fazla iş yapar. Hücrelerin büyük çoğunluğu 0'dır ve bu sıfırları okumak boşa giden iştir.

Bir çizgenin kenar sayısı olabilecek en büyük değer olan $V(V - 1)/2$'nin çok altındaysa çizgeye **seyrek** (*sparse*), bu değere yakınsa **yoğun** (*dense*) denir. Gerçek ağların çoğu seyrektir. Bir yol ağında bir kavşağa bağlanan yol sayısı birkaç taneyle sınırlıdır; bir milyon kavşaklı ağda yaklaşık üç milyon yol olabilir. Komşuluk listesi bu ağı birkaç milyon öğeyle tutar. Matris ise 10^{12} hücre ister; her hücre tek bir bit olsa bile 125 gigabayt eder. Bu yüzden gezinme algoritmalarında varsayılan gösterim komşuluk listesidir. Matris, çizge küçük ve yoğunsa ya da "bu iki düğüm komşu mu?" sorusu çok sık soruluyorsa tercih edilir.

Sık yapılan hata

Yönsüz bir kenarı yalnız bir listeye eklemek. 0–1 kenarı yalnız `komsu[0]` 'a yazılırsa çizge, farkında olmadan $0 \to 1$ kenarlı yönlü bir çizgeye dönüşür. 1'den başlayan bir gezinme artık 0'a ulaşamaz ve bağlantılı bileşen sayısı yanlış çıkar. Yönsüz çizgede listelerin toplam uzunluğu $2E$

olmalıdır; bu, kurulumun doğruluğunu sınamanın kolay bir yoludur.

4.1.2 Çizge gezinme

4.1.2.1 Gezinme problemi

Tanım 4.1.3

Çizge gezinme (*graph traversal*), bir başlangıç düğümünden ulaşılabilen her düğümü düzenli bir sırayla **tam bir kez** ziyaret etmektir. Ziyaret, düğüm üzerinde yapılacak işi kapsar: düğümü yazdırmak, bir sayacı artırmak, bir diziye uzaklığını yazmak gibi.

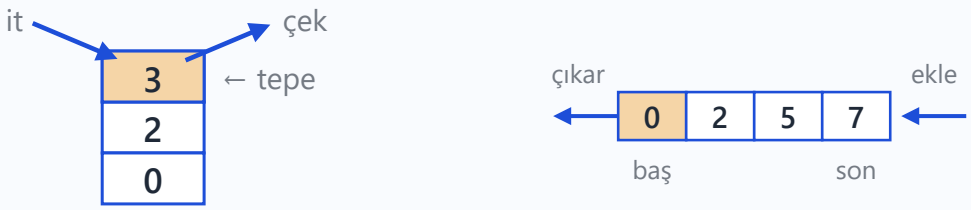
Gezinmenin amacı, çizgenin yapısını sistemli biçimde keşfetmektir. Hangi düğümlere ulaşılabildiği, kaç bileşen olduğu, bir döngü bulunup bulunmadığı gibi sorular bir gezinmeyle yanıtlanır. "Tam bir kez" koşulu iki şey söyler. Birincisi, ulaşılabilen hiçbir düğüm atlanmamalıdır. İkincisi, hiçbir düğüm iki kez işlenmemelidir.

İkinci koşul ağaçlarda kendiliğinden sağlanır. Bir ağacı kökten gezerken her düğüme tek bir yoldan ulaşılır, çünkü ağaçta döngü yoktur. Çizgede ise bir düğüme birçok yoldan ulaşılabilir ve döngüler bulunabilir. Şekil 4.1.3'teki çizgede 0'dan 1'e geçilir, 1'den 2'ye, 2'den yeniden 0'a. Hangi düğümlerin daha önce görüldüğü hatırlanmazsa algoritma 0–1–2 döngüsünde sonsuza kadar döner. Bu yüzden her gezinme algoritması bir **ziyaret dizisi** tutar: `ziyaret[v]`, v'nin daha önce işaretlenip işaretlenmediğini söyleyen bir `boolean` dizisidir. Bir düğüme ancak işaretli değilse ilerlenir.

Gezinme yalnız başlangıçtan *ulaşılabilen* düğümleri görür. Yönlü bir çizgede 0'dan 5'e yol yoksa 0'dan başlayan gezinme 5'i hiç ziyaret etmez. Bağlı olmayan bir çizgenin bütün düğümlerini gezmek için gezinme, henüz ziyaret edilmemiş her düğümden yeniden başlatılır. Yönsüz çizgede her başlatma yeni bir bağlantılı bileşeni keşfeder.

4.1.2.2 İki yaklaşım: yığın ve kuyruk

Gezinme sırasında algoritmanın önünde her an keşfedilmiş ama henüz işlenmemiş düğümlerden oluşan bir "bekleme listesi" vardır. Sıradaki düğüm bu listeden seçilir. Seçimin nasıl yapıldığı, gezinmenin karakterini belirler. İki temel yaklaşımın tek farkı, bekleme listesini tutan veri yapısıdır.



Şekil 4.1.4: Solda yığın: ekleme ve çıkarma aynı uçtan, tepeden yapılır (son giren ilk çıkar). Sağda kuyruk: sondan eklenir, baştan çıkarılır (ilk giren ilk çıkar).

- **Derinlik öncelikli arama** (*depth-first search*, DFS) bekleme listesini bir **yığında** (*stack*) tutar. Yığın son giren ilk çıkar (LIFO, *last in, first out*) kuralıyla çalışır. En son keşfedilen düğüm ilk işlenir; bu yüzden algoritma bir düğümden onun komşusuna, oradan da komşunun komşusuna doğru mümkün olduğunca derine iner. İlerleyecek yer kalmayınca bir önceki düğüme geri döner. Bu geri dönüşe **geri izleme** (*backtracking*) denir.
- **Genişlik öncelikli arama** (*breadth-first search*, BFS) bekleme listesini bir **kuyrukta** (*queue*) tutar. Kuyruk ilk giren ilk çıkar (FIFO, *first in, first out*) kuralıyla çalışır. Önce keşfedilen önce işlenir; bu yüzden algoritma önce başlangıca bir kenar uzaklıktaki düğümleri, sonra iki kenar uzaklıktakileri ziyaret eder. Gezinme düzey düzey ilerler.

Yığın ve kuyruk iki temel veri yapısıdır; ikisinde de ekleme ve çıkarma *O(1)* zaman alır. Java'da yığın için `ArrayDeque`'in `push` ve `pop` metotları, kuyruk için `Queue` arayüzünün `add` ve `poll` metotları kullanılır. Özyinelemeli yazılan DFS'te yığın açıkça görünmez. Onun yerine çağrı yığını kullanılır: her özyinelemeli çağrı yığına bir çerçeve iter, her dönüş bir çerçeve çeker.

4.1.3 Derinlik öncelikli arama (DFS)

Derinlik öncelikli arama, çizge algoritmalarının belki de en çok kullanılan yapı taşıdır. Döngü bulma, topolojik sıralama, güçlü bağlantılı bileşenler ve köprü bulma gibi pek çok algoritma, bir DFS'in üzerine birkaç satır eklenerek elde edilir. Bu kesim DFS'i önce bir benzetmeyle, sonra bir ağaç üzerinde, ardından özyinelemeli ve yığınlı iki Java koduyla ele alır.

4.1.3.1 Fikir: önce derine

DFS'in davranışı bir labirentteki gezgine benzer. Gezgin bir koridora girer ve çıkmaza varana kadar ilerler. Çıkmazda son geçtiği yol ayrımına geri döner ve oradan henüz denemediği bir koridora sapar. O koridor da bitince bir önceki ayrıma döner. Labirentin bütün koridorları denenene kadar bu böyle sürer. Gezginin işini kolaylaştıran iki şey vardır: geçtiği yerlere tebeşirle işaret koyar ki aynı koridoru iki kez dolaşmasın; geri dönüş noktalarını da sırasıyla aklında tutar.

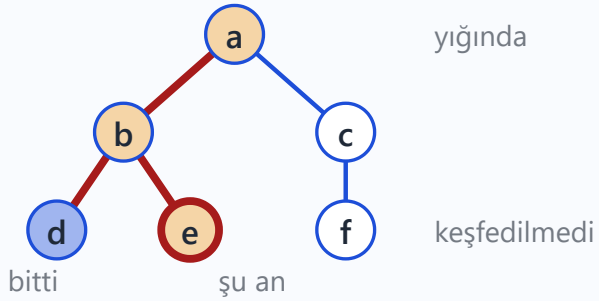
Tanım 4.1.4

Derinlik öncelikli arama (*depth-first search*, DFS), bir düğümü ziyaret ettikten sonra onun henüz ziyaret edilmemiş ilk komşusuna geçer ve aynı işi o komşudan sürdürür. Bir düğümün ziyaret edilmemiş komşusu kalmadığında, o düğüme gelinen düğüme geri döner ve oradaki sıradaki komşuyu dener.

Benzetmedeki tebeşir işaretleri `ziyaret[]` dizisidir. Geri dönüş noktalarını ise bir yığın hatırlar. Özyinelemeli sürümde bu yığın, programın çağrı yığınıdır: bir düğümün işi bitince onun çağrısı döner ve kontrol kendiliğinden bir önceki düğüme geçer. Yinelemeli sürümde açık bir `Deque` nesnesi kullanılır.

DFS sırasında bir düğüm üç durumdan birindedir. Bu durumlar renklerle anlatılır ve bu notun şekillerinde de aynı renkler kullanılır:

- **Beyaz:** düğüm henüz keşfedilmedi.
- **Sarı:** düğüm keşfedildi, ama işi sürüyor. Çağrısı yığında durur; komşularının bir kısmı henüz incelenmedi.
- **Koyu:** düğümün işi bitti; bütün komşuları incelendi ve çağrısı döndü.



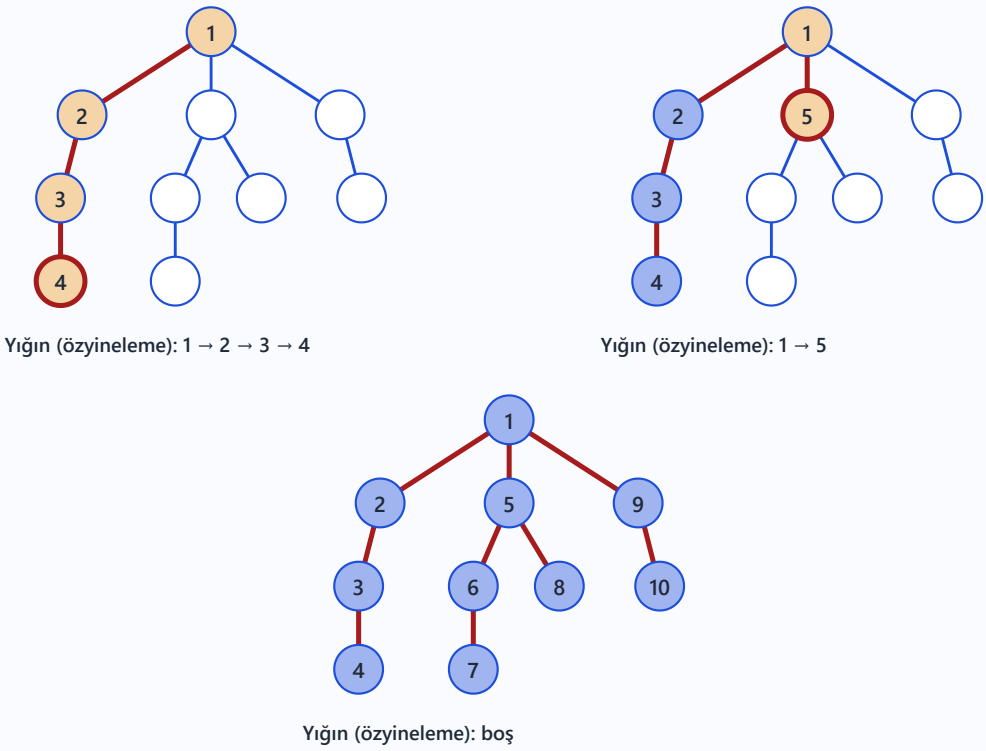
Şekil 4.1.5: DFS sırasında düğüm durumları. a ve b yığındadır (sarı), d'nin işi bitmiştir (koyu), e şu an işleniyor (kalın çerçeve), c ve f henüz keşfedilmemiştir (beyaz). Kalın kenarlar, ziyaret edilen düğümlere ilk ulaşılan kenarlardır.

Bir düğüme ilk kez ulaşılırken kullanılan kenarlar bir ağaç oluşturur. Bu ağaca **DFS ağacı** denir. Kökü başlangıç düğümüdür; her düğümün ebeveyni, onu keşfeden düğümdür. Şekil 4.1.5'te a–b, b–d ve b–e kenarları DFS ağacına aittir. Çizgenin öteki kenarları ağaca girmez, ama yine de incelenir ve önemli bilgiler taşır. Örneğin sarı bir düğüme giden kenar bir döngünün işaretidir.

DFS'in ziyaret sırası komşuların hangi sırayla denendiğine bağlıdır. Aynı çizgede komşu listeleri farklı sıralanırsa farklı ama yine geçerli bir DFS sırası çıkar. Bu notta aksi söylenmedikçe komşular, komşu listesindeki sırayla denenir. Yönsüz örneklerde listeler artan ya da alfabetik sıradadır.

4.1.3.2 Bir ağaçta ziyaret sırası

DFS'in “derine inip geri dönme” düzenini en açık gösteren örnek bir ağaçtır, çünkü ağaçta döngü yoktur ve hiçbir düğüme iki yoldan ulaşılmaz. Şekil 4.1.6 on düğümlü bir ağaçta kökten başlayan DFS'in üç anını gösterir. Düğümlerin üstündeki numaralar, DFS'in onları ziyaret ettiği sıradır.



Şekil 4.1.6: On düğümlü bir ağaçta DFS. Solda: kökten 4'e kadar en soldaki dal sonuna kadar izlenmiştir; yığında 1, 2, 3, 4 vardır. Ortada: 4, 3 ve 2'nin keşfedilmemiş komşusu kalmamış, geri dönülmüş ve 1'in sıradaki çocuğu 5 ziyaret edilmiştir. Sağda: gezinme bitmiş, yığın boşalmıştır.

Örnek 4.1.2

Ağacın kenarları 1–2, 1–5, 1–9, 2–3, 3–4, 5–6, 5–8, 6–7 ve 9–10'dur. Çocuklar soldan sağa denenir.

1. Kök 1 ziyaret edilir. Yığın: 1.
2. 1'in ilk keşfedilmemiş komşusu 2'dir; 2'ye inilir. Yığın: 1 → 2.
3. 2'den 3'e, 3'ten 4'e inilir. Yığın: 1 → 2 → 3 → 4. En soldaki dalın sonuna gelinmiştir.
4. 4'ün keşfedilmemiş komşusu yoktur; 4 biter ve 3'e dönülür. 3'ün de, 2'nin de başka keşfedilmemiş komşusu yoktur; ikisi de biter. Bu geri dönüş zinciri 1'e kadar sürer. Yığın: 1.
5. 1'in sıradaki çocuğu 5 ziyaret edilir. 5'ten 6'ya, 6'dan 7'ye inilir. Yığın: 1 → 5 → 6 → 7.
6. 7 ve 6 biter; 5'e dönülür. 5'in sıradaki çocuğu 8 ziyaret edilir. Yığın: 1 → 5 → 8.
7. 8 ve 5 biter; 1'e dönülür. 1'in son çocuğu 9, ardından 9'un çocuğu 10 ziyaret edilir. Yığın: 1 → 9 → 10.

8. 10, 9 ve 1 biter; yığın boşalır.

Ziyaret sırası 1, 2, 3, 4, 5, 6, 7, 8, 9, 10'dur. Düğümler bu örnekte zaten DFS sırasıyla numaralandığı için sıra, numaraların artan sırasıdır. Aynı ağaçta BFS'in sırası ise 1, 2, 5, 9, 3, 6, 8, 10, 4, 7 olurdu; önce kökün bütün çocukları gelirdi.

Örnekte yığının en çok dört düğüm tuttuğu görülür. Bu sayı kökten en uzak yaprağa giden yolun düğüm sayısıdır. DFS'in yığını hiçbir zaman en uzun yoldan daha derine inmez, çünkü yığında her an kökten şu anki düğüme giden ağaç yolu durur.

4.1.3.3 Özyinelemeli sözde kod

DFS'in tanımı kendisini kullanır: bir düğümden DFS yapmak, onu işaretleyip her keşfedilmemiş komşusundan DFS yapmaktır. Bu yüzden algoritma en doğal hâliyle özyinelemeli yazılır. Kaynaklarda iki yazım yaygındır. İlki, İngilizce kaynaklarda sık görülen biçimdir:

```
procedure DFS(G, v) is
  label v as discovered
  for all directed edges from v to w
      that are in G.adjacentEdges(v) do
    if vertex w is not labeled
        as discovered then
      call DFS(G, w)
```

İkincisi, aynı fikrin denetimi çağrının başına taşıyan Türkçe yazımıdır:

```
DFS(düğüm, ziyaretEdilenler):
  eğer düğüm ziyaretEdilenler
        içinde değilse:
    düğüm'ü ziyaretEdilenler'e ekle
    düğüm'ü işle (örneğin, yazdır)
    her bir komşu düğüm için:
      DFS(komşu, ziyaretEdilenler)
```

İki sürüm aynı ziyaret sırasını üretir. Fark, “ziyaret edildi mi?” denetiminin nerede yapıldığıdır. İlk sürüm bir komşuyu *çağırmadan önce* denetler; işaretli komşu için hiç çağrı yapılmaz. İkinci sürüm her komşu için çağrı yapar ve denetimi çağrının başında yapar; işaretli bir düğüm için açılan çağrı hemen döner. İkinci sürüm daha kısa görünür, ama gereksiz çağrılar yapar. Her kenar için bir çağrı açılır ve çağrı sayısı $O(E)$ olur. Mertebe değişmez, ama sabit çarpan büyür. Java kodu ilk sürümü izler.

Sözde kodlar tek bir başlangıç düğümünden ulaşılabilen düğümleri gezer. Bağlı olmayan bir çizgenin bütün düğümlerini gezmek için DFS, ziyaret edilmemiş her düğümden yeniden başlatılır. Yönsüz çizgede her başlatma bir bağlantılı bileşeni bütünüyle gezer.

4.1.3.4 Kod: dfs()

```
void dfs(int v, boolean[] ziyaret) {
  ziyaret[v] = true;
  System.out.print(v + " ");
  for (int w : komsu[v])
    if (!ziyaret[w])
      dfs(w, ziyaret);
}
```

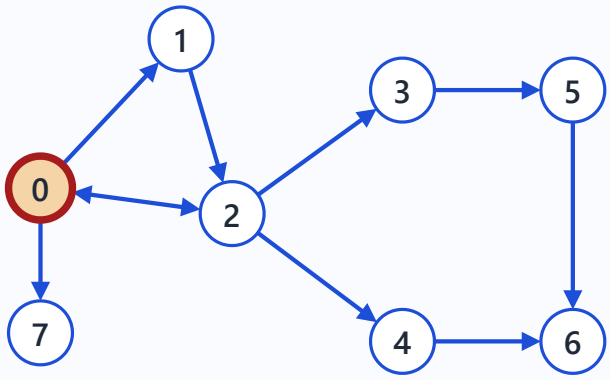
Kod, sözde kodun ilk sürümünün Java karşılığıdır ve satır satır şöyle çalışır:

- `ziyaret[v] = true;` düğümü işaretler. İşaret, düğüm ziyaret edilir edilmez konur. Böylece komşulardan biri *v*'ye geri dönen bir kenar taşısa bile *v* ikinci kez çağrılmaz.
- `System.out.print(v + " ");` düğüm üzerindeki işi yapar; burada iş, düğümü yazdırmaktır. Bu satır, çağrının başında olduğu için düğümler **keşif sırasıyla** yazılır. Aynı satır döngüden sonraya konsaydı düğümler **bitiş sırasıyla** yazılırdı; topolojik sıralamada bu ikinci sıra kullanılır.
- `for (int w : komsu[v])` *v*'nin komşu listesinde baştan sona gezinir.
- `if (!ziyaret[w]) dfs(w, ziyaret);` keşfedilmemiş komşuya iner. Özyinelemeli çağrı döndüğünde *w*'den ulaşılabilen her düğüm ziyaret edilmiştir ve döngü sıradaki komşuyla sürer. Geri dönüş tam olarak bu çağrı dönüşüdür.

Metot `ziyaret` dizisini parametre olarak alır. Dizi bir kez oluşturulur ve bütün çağrılar aynı diziyi paylaşır. Bu, bağlı olmayan çizgede DFS'i yeniden başlatmayı kolaylaştırır: aynı dizi, ikinci başlatmada birinci başlatmanın işaretlerini korur.

4.1.3.5 Kod izleme: dfs(0)

İzlemede kullanılan çizge yönlüdür. Kenarları *0→1*, *0→2*, *0→7*, *1→2*, *2→0*, *2→3*, *2→4*, *3→5*, *4→6* ve *5→6*'dır. Komşu listeleri bu sırayla dolar: `komsu[0] = [1, 2, 7]`, `komsu[2] = [0, 3, 4]`. 6 ve 7'nin çıkan kenarı yoktur. Çizgede *0 → 1 → 2 → 0* ve *0 → 2 → 0* döngüleri vardır; ziyaret dizisi olmadan DFS bu döngülerde sonsuza kadar dönerdi.



Şekil 4.1.7: İzlemenin başı: `dfs(0)` çağrıldı, 0 işaretlendi ve yazdırıldı. 0 ile 2 arasındaki çift oklu çizgi iki ayrı kenardır: *0→2* ve *2→0*.

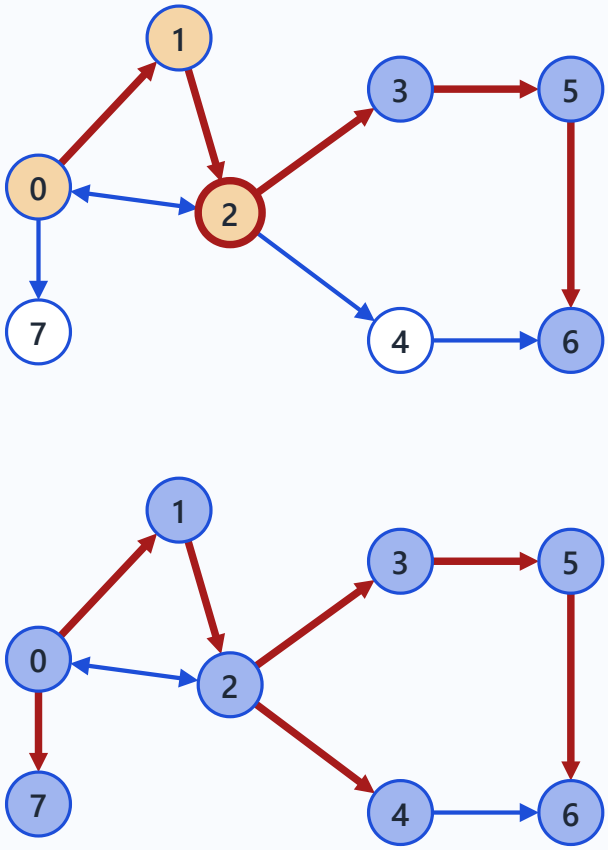
Adım	Olay	Çağrı yığını	Çıktı
1	`dfs(0)` : 0 işaretlenir, yazdırılır	0	0
2	0'ın komşusu 1 işaretsiz → `dfs(1)`	0, 1	0 1
3	1'in komşusu 2 işaretsiz → `dfs(2)`	0, 1, 2	0 1 2
4	2'nin komşusu 0 işaretli → atlanır	0, 1, 2	0 1 2
5	2'nin komşusu 3 işaretsiz → `dfs(3)`	0, 1, 2, 3	0 1 2 3
6	3'ün komşusu 5 işaretsiz → `dfs(5)`	0, 1, 2, 3, 5	0 1 2 3 5
7	5'in komşusu 6 işaretsiz → `dfs(6)`	0, 1, 2, 3, 5, 6	0 1 2 3 5 6
8	6, 5 ve 3 biter; kontrol 2'ye döner	0, 1, 2	0 1 2 3 5 6
9	2'nin komşusu 4 işaretsiz → `dfs(4)`	0, 1, 2, 4	0 1 2 3 5 6 4
10	4'ün komşusu 6 işaretli → atlanır	0, 1, 2, 4	0 1 2 3 5 6 4
11	4, 2 ve 1 biter; kontrol 0'a döner	0	0 1 2 3 5 6 4
12	0'ın komşusu 2 işaretli → atlanır	0	0 1 2 3 5 6 4
13	0'ın komşusu 7 işaretsiz → `dfs(7)`	0, 7	0 1 2 3 5 6 4 7
14	7 ve 0 biter; `dfs(0)` döner	boş	0 1 2 3 5 6 4 7

Tablo 4.1.4: `dfs(0)` izlemesi. Çağrı yığını soldan sağa büyür; en sağdaki çağrı şu an çalışandır.

İzlemenin üç anı ayrıca dikkat ister. **4. adımda** 2'den 0'a giden kenar incelenir. 0 işaretlidir, üstelik hâlâ çağrı yığınındadır, yani sarıdır. Bu kenar bir döngünün kanıtıdır: yığında 0'dan 2'ye giden bir ağaç yolu (0 → 1 → 2) vardır ve *2 → 0* kenarı bu yolu kapatır. Kod burada yalnız atlar; döngü bulan sürüm burada "döngü var" diyebilir.

8. adımda üç çağrı art arda döner. 6'nın hiç çıkan kenarı yoktur; 5'in tek komşusu 6, 3'ün tek komşusu 5'tir. Kontrol 2'deki `for` döngüsüne, kaldığı yere döner ve sıradaki komşu 4 denenir. Geri dönüşün koddaki karşılığı budur: ayrı bir "geri dön" komutu yoktur, çağrıların dönmesi yeterlidir.

10. adımda 4'ten 6'ya giden kenar incelenir. 6 işaretlidir, ama yığında değildir; işi çoktan bitmiştir. Bu kenar bir döngü oluşturmaz: 6'dan 4'e dönen bir yol yoktur. Yönlü çizgede yalnız yığındaki düğüme giden kenar döngü belirtisidir; işi bitmiş düğüme giden kenar döngü belirtisi değildir.



Şekil 4.1.8: Solda 8. adım: 6, 5 ve 3 bitmiştir (koyu); kontrol 2'ye dönmüştür ve yığında 0, 1, 2 vardır. Sağda izlemenin sonu: kalın kenarlar DFS ağacını oluşturur (*0→1*, *1→2*, *2→3*, *3→5*, *5→6*, *2→4*, *0→7*). İnce kalan *0→2*, *2→0* ve *4→6* kenarları ağaca girmemiştir.

Çıktı `0 1 2 3 5 6 4 7`'dir. Çağrı yığını en çok altı çerçeveye ulaşmıştır (7. adım: 0, 1, 2, 3, 5, 6). DFS ağacında 0'dan 6'ya giden yol *0 → 1 → 2 → 3 → 5 → 6*'dır ve beş kenar uzunluğundadır. Oysa çizgede 0'dan 6'ya üç kenarlık bir yol vardır: *0 → 2 → 4 → 6*. Bu fark, DFS'in en kısa yolu bulmadığının ilk işaretidir.

4.1.3.6 Yığınla DFS

Özyinelemeli DFS kısa ve anlaşılırdır, ama bir zayıflığı vardır: çağrı yığınının derinliği DFS yolunun uzunluğu kadardır. Çok uzun bir yol içeren çizgede bu derinlik on binleri bulabilir. Çağrı yığını sınırlı bir bellek alanıdır ve aşıldığında Java `StackOverflowError` fırlatır. JDK 21'in varsayılan ayarlarıyla yapılan bir denemede, 100.000 düğümlü bir yol çizgesinde özyinelemeli DFS yaklaşık 10.000 çağrı

derinliğinde bu hatayla durmuştur. Çözüm, çağrı yığınının işini programın kendi oluşturduğu açık bir yığın nesnesine devretmektir. Bu nesne dinamik bellekte durur ve boyu yalnız toplam bellekle sınırlıdır. Önce İngilizce kaynaklardaki yaygın sözde kod:

```
procedure DFS_iterative(G, v) is
  let S be a stack
  S.push(v)
  while S is not empty do
    v = S.pop()
    if v is not labeled as discovered then
      label v as discovered
      for all edges from v to w
          in G.adjacentEdges(v) do
        S.push(w)
```

Aynı algoritmanın Türkçe yazımı:

```
DFS(başlangıç_düğüm):
  boş bir yığın oluştur
  başlangıç_düğüm'ü yığına ekle
  döngü yığın boş değilken:
    mevcut_düğüm = yığından çıkar
    eğer mevcut_düğüm ziyaretEdilenler
          içinde değilse:
      mevcut_düğüm'ü ziyaretEdilenler'e ekle
      her bir komşu_düğüm için:
        komşu_düğüm'ü yığına ekle
```

Java karşılığı:

```
void dfsYigin(int s) {
  boolean[] ziyaret = new boolean[n];
  Deque<Integer> yigin = new ArrayDeque<>();
  yigin.push(s);
  while (!yigin.isEmpty()) {
    int v = yigin.pop();
    if (!ziyaret[v]) {
      ziyaret[v] = true;
      System.out.print(v + " ");
      for (int w : komsu[v])
        yigin.push(w);
    }
  }
}
```

Kod üç noktada özyinelemeli sürümden ayrılır:

- **Bekleme listesi açıktır.** `ArrayDeque` bir yığın olarak kullanılır: `push` tepeye ekler, `pop` tepeden çeker. Döngü, yığın boşalana kadar sürer.
- **İşaretleme çekerken yapılır.** Bir düğüm yığına itilirken değil, yığından çekildiğinde işaretlenir. Bu yüzden henüz çekilmemiş bir düğüm, farklı komşulardan birden çok kez itilebilir. Çekildiğinde işaretliyse hiçbir şey yapılmaz. Yığına toplam $1 + \Sigma$ *derece* itme yapılır; bu da yönlü çizgede $1 + E$, yönsüz çizgede $1 + 2E$ eder. Yığının boyu $O(E)$'ye kadar çıkabilir.
- **Sıra terse döner.** Komşular listedeki sırayla itilir, ama yığından en son itilen önce çıkar. Listesi `[1, 2, 7]` olan 0'dan sonra ilk çekilen düğüm 7'dir, 1 değil. Bu yüzden ziyaret sırası özyinelemeli sürümün sırasından farklıdır.

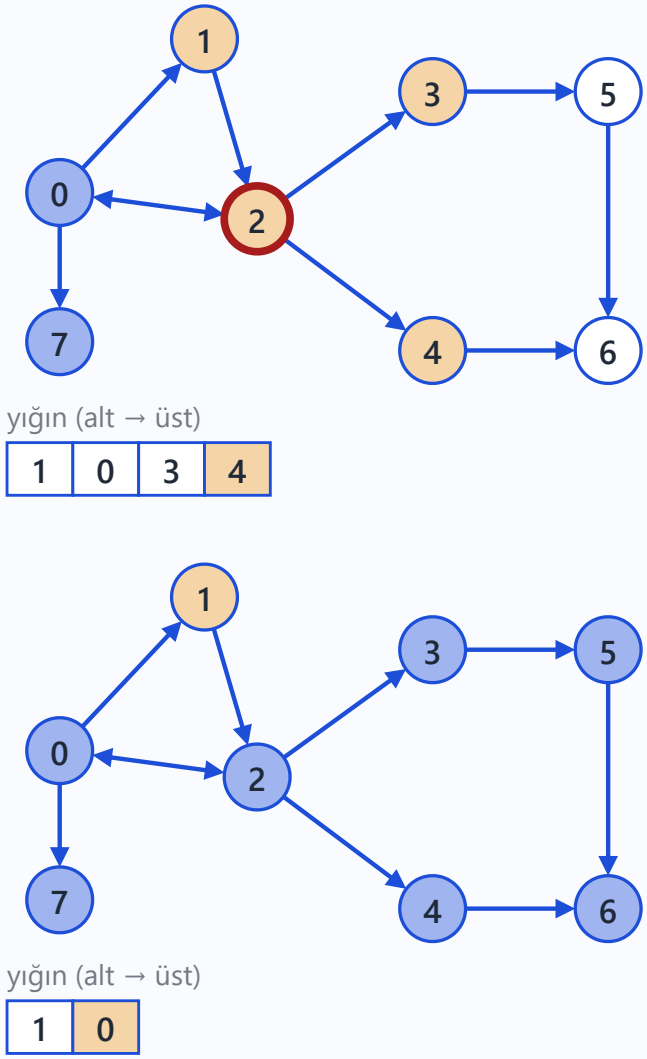
İşaretlemenin neden çekerken yapıldığını küçük bir örnek gösterir. 0'ın komşuları 1 ve 2, 2'nin komşusu da 1 olsun. Düğümler itilirken işaretlenseydi 0 işlenirken 1 ve 2 birlikte işaretlenirdi. 2 çekildiğinde komşusu 1 zaten işaretli olduğu için yığına girmezdi ve 1, 2'nin altından değil, 0'ın ikinci çocuğu olarak işlenirdi. Gezinme her düğümü yine bir kez ziyaret ederdi, ama 2'den başlayan dal sonuna kadar izlenmemiş olurdu; sıra artık derinlik öncelikli olmazdı. Çekerken işaretlemek, bir düğümün en son keşfedildiği yerden işlenmesini sağlar. DFS'in "önce derine" kuralı da budur.

4.1.3.7 Kod izleme: dfsYigin(0)

Aynı yönlü çizgede `dfsYigin(0)` çağrısını izleyelim. Tabloda yığın soldan sağa, alttan tepeye doğru yazılmıştır; en sağdaki öğe tepedir.

Çekilen	İşlem	İşlemden sonra yığın	Çıktı
—	başlangıç: 0 itilir	0	
0	işaretle, yaz; 1, 2, 7 itilir	1, 2, 7	0
7	işaretle, yaz; komşusu yok	1, 2	0 7
2	işaretle, yaz; 0, 3, 4 itilir	1, 0, 3, 4	0 7 2
4	işaretle, yaz; 6 itilir	1, 0, 3, 6	0 7 2 4
6	işaretle, yaz; komşusu yok	1, 0, 3	0 7 2 4 6
3	işaretle, yaz; 5 itilir	1, 0, 5	0 7 2 4 6 3
5	işaretle, yaz; 6 itilir	1, 0, 6	0 7 2 4 6 3 5
6	zaten işaretli, atlanır	1, 0	0 7 2 4 6 3 5
0	zaten işaretli, atlanır	1	0 7 2 4 6 3 5
1	işaretle, yaz; 2 itilir	2	0 7 2 4 6 3 5 1
2	zaten işaretli, atlanır	boş	0 7 2 4 6 3 5 1

Tablo 4.1.5: `dfsYigin(0)` izlemesi. Her satır bir `pop` işlemidir.



Şekil 4.1.9: Solda 2 çekilmiş ve komşuları 0, 3, 4 itilmiştir; 0 işaretli olduğu hâlde yığına girmiştir. Sağda 6 ikinci kez çekilmiştir (önce 4'ten, sonra 5'ten itilmişti); işaretli olduğu için atlanır. Yığında 1 ve 0 kalmıştır.

Çıktı `0 7 2 4 6 3 5 1`'dir. Özyinelemeli sürüm aynı çizgede `0 1 2 3 5 6 4 7` yazmıştı. İki sıra farklıdır, ama ikisi de geçerli bir DFS sırasıdır: yığınlı sürüm, komşu listelerinin ters çevrildiği bir çizgede özyinelemeli DFS'in vereceği sırayı verir. Yığına toplam 11 itme yapılmıştır: başlangıçtaki 0 ve on kenarın her biri için bir itme (*1 + E = 11*). 6, 0 ve 2 yığına ikişer kez girmiştir. Yığın en çok dört öğe tutmuştur.

Özyinelemeli sürümle aynı sıra istenirse komşular *ters sırada* itilir. 0'ın komşuları 7, 2, 1 sırasıyla itilirse tepeye 1 gelir ve ilk çekilen 1 olur. Bu durumda iki sürüm aynı ziyaret sırasını üretir.

Örnek 4.1.3

Yönsüz bir çizgenin kenarları A–B, A–C, B–D, C–D, C–E, D–F, E–F ve F–G'dir; komşu listeleri alfabetiktir. A'dan özyinelemeli DFS ve yığınla DFS sıraları bulunsun.

Özyinelemeli DFS. A'nın ilk komşusu B'ye inilir. B'nin listesi `[A, D]` 'dir; A işaretli olduğu için D'ye inilir. D'nin listesi `[B, C, F]` 'dir; B işaretli, C'ye inilir. C'nin listesi `[A, D, E]` 'dir; A ve D işaretli, E'ye inilir. E'den F'ye (C işaretli), F'den G'ye (D ve E işaretli) inilir. G'nin tek komşusu F işaretlidir. Bundan sonra çağrılar sırayla döner ve yeni düğüm çıkmaz. Sıra **A B D C E F G**'dir; çağrı yığını en çok yedi çerçeveye ulaşır.

Yığınla DFS. Komşular alfabetik sırayla itilir:

1. A çekilir, yazılır; B ve C itilir. Yığın: B, C.
2. C çekilir, yazılır; A, D, E itilir. Yığın: B, A, D, E.
3. E çekilir, yazılır; C, F itilir. Yığın: B, A, D, C, F.
4. F çekilir, yazılır; D, E, G itilir. Yığın: B, A, D, C, D, E, G.
5. G çekilir, yazılır; F itilir. Yığın: B, A, D, C, D, E, F (yedi öğe).
6. F ve E işaretlidir, atlanır. D çekilir, yazılır; B, C, F itilir.
7. F ve C atlanır. B çekilir, yazılır; A ve D itilir.
8. Kalan D, A, C, D, A, B öğelerinin hepsi işaretlidir; tek tek atlanır ve yığın boşalır.

Sıra **A C E F G D B**'dir. Her düğüm bir kez işlenir ve işlenirken bütün komşularını iter. Komşu listelerinin toplam uzunluğu $2E = 16$'dır; başlangıçtaki itmeyle birlikte toplam **17 itme** yapılır.

4.1.3.8 Karmaşıklık

Teorem 4.1.1 (DFS'in zamanı)

Komşuluk listesiyle tutulan bir çizgede DFS $O(V + E)$ zamanda çalışır. Komşuluk matrisiyle tutulan çizgede zaman $O(V^2)$'dir.

İspat. Maliyet iki kalemde toplanır. Birinci kalem düğümlerdir. Bir düğüm için `dfs` en fazla bir kez çağrılır, çünkü çağrı yalnız işaretsiz düğüm için yapılır ve çağrının ilk işi düğümü işaretlemektir. Her çağrının döngü dışındaki işi sabittir; düğümler toplamı $O(V)$ eder. İkinci kalem kenarlardır. Her çağrı kendi düğümünün komşu listesinde bir kez gezinir ve her komşu için sabit iş yapar. Bir düğümün listesinde yalnız bir kez gezinildiğinden, bütün listelerin toplam uzunluğu kadar iş yapılır. Bu uzunluk yönlü çizgede E, yönsüz çizgede $2E$'dir; kenarlar toplamı $O(E)$ eder. Toplam $O(V + E)$'dir. Matriste bir düğümün komşularını bulmak, derecesinden bağımsız olarak V hücrelik bir satırı taramaktır. V düğüm için V satır taranır ve toplam $O(V^2)$ olur. ■

$O(V + E)$, çizgenin kendisini okumak için gereken zamandır. Her düğüme ve her kenara en az bir kez bakmadan bütün çizge hakkında bir şey söylenemez. Bu yüzden bu sınır gezinme için en iyisidir ve bu tür algoritmalara **doğrusal zamanlı** çizge algoritmaları denir.

Ek bellek de iki kalemdir. `ziyaret[]` dizisi *O(V)* yer tutar. Yığının boyu sürüme göre değişir. Özyinelemeli sürümde çağrı yığını en çok DFS ağacındaki en uzun yol kadar derinleşir; bu en kötü durumda *V*'dir, yani *O(V)*. Örneğin bir yol çizgesinde (0–1–2–…) yığın bütün düğümleri tutar. Çekerken işaretleyen yığınlı sürümde ise bir düğüm birden çok kez itilebildiği için yığın *O(E)*'ye kadar büyüyebilir. Yoğun bir çizgede *E*, V^2 mertebesinde olabileceğinden bu fark önemsiz değildir. Yığındaki tekrarları önleyen daha ayrıntılı bir sürüm de yazılabilir. O sürümde yığının her öğesi, düğümle birlikte "sıradaki komşunun indisini" tutar. Böylece özyinelemeli DFS birebir taklit edilir ve *O(V)* ek bellek yeter.

4.1.3.9 DFS ne işe yarar, ne yapamaz?

DFS'in gücü, gezinme sırasında topladığı yapısal bilgiden gelir. Bir düğümün ne zaman keşfedildiği, ne zaman bittiği ve bir kenarın sarı mı yoksa koyu bir düğüme mi gittiği, pek çok soruyu yanıtlamaya yeter:

- **Ulaşılabilirlik ve bağlantılı bileşenler.** *s*'den DFS yapıldıktan sonra `ziyaret[t]` doğruysa *t*'ye ulaşılabilir. DFS'in kaç kez başlatıldığı, yönsüz çizgedeki bağlantılı bileşen sayısını verir.
- **Döngü tespiti.** Yönlü çizgede çağrı yığınındaki (sarı) bir düğüme giden kenara **geri kenar** (*back edge*) denir. Geri kenar, hedef düğümden kaynak düğüme inen ağaç yoluyla birlikte bir döngü oluşturur. Tersine, çizgede bir döngü varsa döngüde ilk keşfedilen düğüm bitmeden döngünün öteki bütün düğümleri keşfedilir ve döngünün o düğüme dönen kenarı bir geri kenar olur. Dolayısıyla yönlü bir çizgede döngü olması, DFS'te geri kenar bulunmasıyla eşdeğerdir.
- **Topolojik sıralama.** Döngüsüz yönlü bir çizgede düğümlerin bitiş sırası tersine çevrilince bir topolojik sıra elde edilir.
- **Güçlü bağlantılı bileşenler.** Tarjan ve Kosaraju algoritmaları DFS üzerine kuruludur.
- **Köprü ve eklem noktaları, bulmaca çözme.** Köprüler Tarjan'ın enDüşükBağ fikriyle bulunur. Sudoku ve sekiz vezir gibi bulmacaları çözen geri izleme (*backtracking*) algoritmaları da örtük bir çizgede yapılan DFS'tir.

DFS'in yapamadığı da bir o kadar önemlidir: **DFS en kısa yolu bulmaz.** DFS bir hedefe ilk ulaştığında izlediği yol, uzun bir dolambaç olabilir. Yönlü örnekte DFS 6'ya 0 → 1 → 2 → 3 → 5 → 6 yoluyla, beş kenarda ulaşmıştı; en kısa yol üç kenarlıktı. Örnek 4.1.3'teki yönsüz çizgede DFS ağacı A'dan G'ye A–B–D–C–E–F–G yolunu, yani altı kenarı izler; en kısa yol ise A–B–D–F–G'dir ve dört kenarlıktır. En az kenarlı yolu bulmak için genişlik öncelikli arama gerekir.

4.1.4 Genişlik öncelikli arama (BFS)

DFS bir dalı sonuna kadar izlerken BFS bütün dalları aynı anda, adım adım ilerletir. Bu fark küçük görünür, ama önemli bir sonucu vardır: BFS, ağırlıksız bir çizgede başlangıçtan her düğüme giden en az kenarlı yolu bulur. Navigasyonda "en az aktarmalı güzergâh", sosyal ağda "iki kişi arasındaki tanışıklık zincirinin uzunluğu" ve oyunlarda "en az hamlede çözüm" bu özellikle hesaplanır.

4.1.4.1 Fikir: katman katman

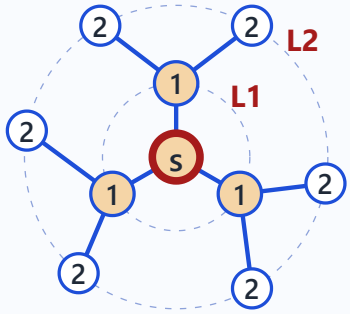
Durgun bir suya atılan taşın oluşturduğu dalgalar BFS'i iyi anlatır. Dalga önce taşın hemen çevresine ulaşır, sonra bir halka dışarı, sonra bir halka daha. Bir haberin yayılması da böyledir: haberi alan kişi onu bütün arkadaşlarına söyler, onlar da kendi arkadaşlarına. Haberi ilk duyanlar başlangıca bir adım uzaktakilerdir, sonra iki adım uzaktakiler duyar.

Tanım 4.1.5

Genişlik öncelikli arama (*breadth-first search*, BFS), başlangıç düğümü s'den ulaşılabilen düğümleri s'ye uzaklıklarına göre ziyaret eder: önce s'yi, sonra s'ye bir kenar uzaklıktaki düğümleri, sonra iki kenar uzaklıktakileri. Keşfedilen düğümler bir kuyrukta bekler; kuyruğun ilk giren ilk çıkar kuralı bu sırayı korur.

Tanım 4.1.6

BFS'in **katmanları** (*levels*) şöyle tanımlanır: $L_0 = \{s\}$; L_1, s'nin komşularıdır; genel olarak L_{k+1}, L_k'daki düğümlerin henüz hiçbir katmana girmemiş komşularıdır. L_k'daki her düğümün s'ye uzaklığı, yani s'den ona giden en kısa yolun kenar sayısı, tam olarak k'dir.



Şekil 4.1.10: BFS katmanları. s merkezdedir; uzaklığı 1 olan düğümler iç halkada (L_1), uzaklığı 2 olanlar dış halkadadır (L_2).

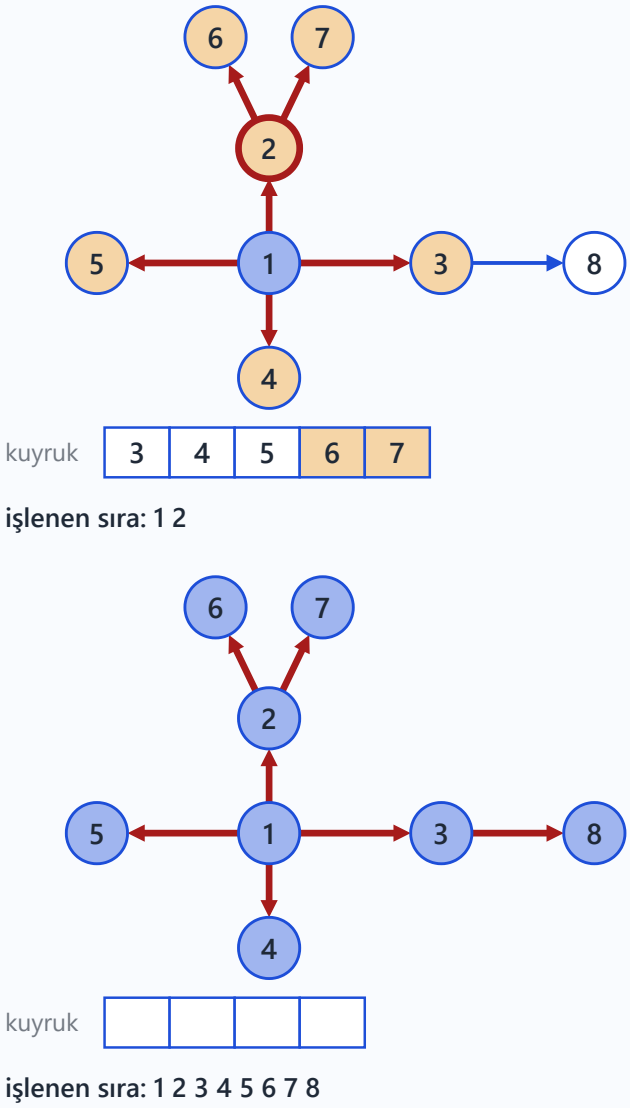
BFS'in iki kuralı vardır. Birincisi, kuyruk sırayı korur: önce keşfedilen önce işlenir. L_1'deki bütün düğümler kuyruğa L_2'dekilerden önce girdiği için önce çıkar. İkincisi, düğüm kuyruğa **girerken** işaretlenir. Böylece birden çok komşusu bir düğümü görse bile düğüm kuyruğa en fazla bir kez girer. Bu kural yığınlı DFS'teki "çekerken işaretle" kuralının tersidir; amacı kuyruğun gereksiz yere büyümesini önlemektir.

4.1.4.2 Katmanlara ayrılan bir örnek

Sekiz düğümlü yönsüz bir çizge alınsın: 1 düğümü 2, 3, 4 ve 5'e; 2 düğümü 6 ve 7'ye; 3 düğümü 8'e bağlıdır. BFS 1'den başlar. Tablo 4.1.6 her adımda kuyruktan alınan düğümü, kuyruğa eklenen düğümleri ve kuyruğun durumunu gösterir.

Adım	Alınan	Yeni keşfedilen	Kuyruk	Katman
0	—	1	1	$1 \to L_0$
1	1	2, 3, 4, 5	2, 3, 4, 5	$2, 3, 4, 5 \to L_1$
2	2	6, 7	3, 4, 5, 6, 7	$6, 7 \to L_2$
3	3	8	4, 5, 6, 7, 8	$8 \to L_2$
4	4	—	5, 6, 7, 8	
5	5	—	6, 7, 8	
6	6	—	7, 8	
7	7	—	8	
8	8	—	boş	

Tablo 4.1.6: 1'den BFS. Kuyruk soldan çıkar, sağdan girer.



Şekil 4.1.11: Solda 2. adımın sonu: 1 ve 2 işlenmiş (koyu), 3, 4, 5, 6, 7 kuyrukta bekliyor (sarı). Sağda gezinmenin sonu: işlenme sırası 1, 2, 3, 4, 5, 6, 7, 8; kalın oklar her düğümü keşfeden kenarı gösterir.

Ziyaret sırası 1, 2, 3, 4, 5, 6, 7, 8'dir. Önce uzaklığı 1 olanlar (2, 3, 4, 5), sonra uzaklığı 2 olanlar (6, 7, 8) gelir. 6 ve 7, 3'ten önce keşfedilmiştir, ama kuyruğun sonuna eklendikleri için 3, 4 ve 5 işlendikten sonra işlenirler. Bir katmanın hepsi bitmeden sonraki katmana geçilmez; BFS'in “katman katman” ilerlemesi budur. Bu örnekte çizge bir ağaçtır ve her düğümü keşfeden kenarlar çizgenin bütün kenarlarıdır.

4.1.4.3 Sözde kod

```
procedure BFS(G, root) is
  let Q be a queue
  label root as explored
  Q.enqueue(root)
  while Q is not empty do
    v := Q.dequeue()
    for all edges from v to w
        in G.adjacentEdges(v) do
      if w is not labeled as explored then
        label w as explored
        w.parent := v
        Q.enqueue(w)
```

Türkçe yazımı:

```
BFS(başlangıç_düğüm):
  boş bir kuyruk oluştur
  başlangıç_düğüm'ü keşfedildi işaretle
  başlangıç_düğüm'ü kuyruğa koy
  döngü kuyruk boş değilken:
    mevcut_düğüm = kuyruktan al
    her bir komşu_düğüm için
      eğer komşu_düğüm keşfedilmemiş ise
        komşu_düğüm'ü keşfedildi işaretle
        komşu_düğüm'ün ebeveyni = mevcut_düğüm
        komşu_düğüm'ü kuyruğa koy
```

Sözde kodda iki ayrıntı önemlidir. Birincisi `w.parent := v` satırıdır: her düğüm, onu keşfeden düğümü ebeveyni olarak kaydeder. Bu ebeveyn bağlantıları bir ağaç oluşturur; bu ağaca **BFS ağacı** denir. Hedeften ebeveynler boyunca köke doğru gidilince başlangıçtan hedefe giden en kısa yol tersten elde edilir. İkincisi, işaretleme kuyruğa *eklerken* yapılır, kuyruktan çıkarırken değil. Aksi hâlde bir düğümü, kuyruktan çıkana kadar geçen sürede onu gören her komşusu yeniden ekler.

4.1.4.4 Kod: bfs()

```
void bfs(int s) {
  boolean[] ziyaret = new boolean[n];
  Queue<Integer> kuyruk = new LinkedList<>();
  ziyaret[s] = true; kuyruk.add(s);
  while (!kuyruk.isEmpty()) {
    int v = kuyruk.poll();
    System.out.print(v + " ");
    for (int w : komsu[v])
      if (!ziyaret[w]) {
        ziyaret[w] = true;
        kuyruk.add(w);
      }
  }
}
```

- `boolean[] ziyaret = new boolean[n];` bütün düğümleri işaretsiz başlatır. Java dizileri `false` ile dolu oluşturulur.
- `Queue<Integer> kuyruk = new LinkedList<>();` kuyruğu oluşturur. `LinkedList`, `Queue` arayüzünü uygular; `add` sona ekler, `poll` baştan çıkarır, ikisi de *O(1)*'dir. `ArrayDeque` de aynı işi görür.
- `ziyaret[s] = true; kuyruk.add(s);` başlangıcı işaretler ve kuyruğa koyar. İşaret, kuyruğa girişle aynı anda konur.
- `int v = kuyruk.poll();` kuyruğun başındaki düğümü çıkarır; ardından düğüm yazdırılır. Düğümler kuyruktan çıktıkları sırayla yazıldığı için çıktı, BFS ziyaret sırasıdır.
- İç döngü *v*'nin komşuları üzerinde gezinir. İşaretsiz komşu işaretlenir ve kuyruğun sonuna eklenir; işaretli komşu atlanır.

Zaman çözümlemesi DFS'inkiyle aynıdır. Her düğüm kuyruğa en fazla bir kez girer ve bir kez çıkar: *O(V)*. Her düğümün komşu listesinde, düğüm kuyruktan çıktığında bir kez gezinilir: toplam *O(E)*. BFS komşuluk listesiyle *O(V + E)*, matrisle $O(V^2)$ zamanda çalışır. Ek bellek `ziyaret[]` için *O(V)*, kuyruk için en geniş katman kadardır; bu da en kötü durumda *O(V)*'dir.

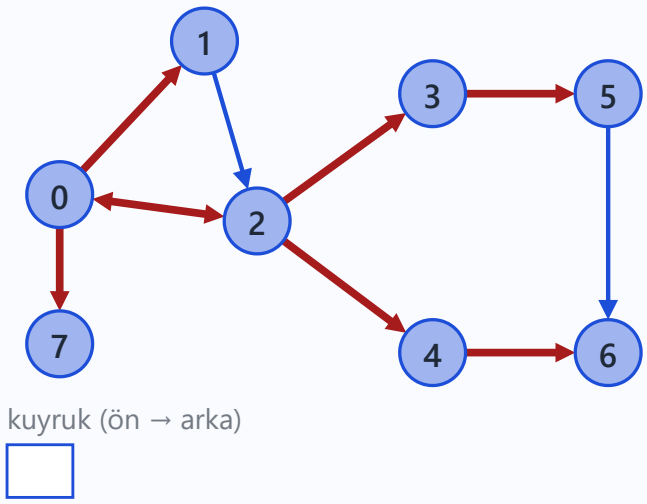
4.1.4.5 Kod izleme: bfs(0)

Kodu DFS izlemesinde kullanılan yönlü çizgede çalıştıralım: `komsu[0] = [1, 2, 7]`, `komsu[1] = [2]`, `komsu[2] = [0, 3, 4]`, `komsu[3] = [5]`, `komsu[4] = [6]`, `komsu[5] = [6]`; 6 ve 7'nin komşusu yoktur.

Alınan v	Komşu taraması	Kuyruk (sonra)	Çıktı
—	0 işaretlenir, kuyruğa girer	0	
0	1, 2, 7 işaretsiz → işaretle, ekle	1, 2, 7	0
1	2 işaretli → atla	2, 7	0 1

Alınan ∨	Komşu taraması	Kuyruk (sonra)	Çıktı
2	0 işaretli → atla; 3, 4 → işaretle, ekle	7, 3, 4	0 1 2
7	komşu yok	3, 4	0 1 2 7
3	5 → işaretle, ekle	4, 5	0 1 2 7 3
4	6 → işaretle, ekle	5, 6	0 1 2 7 3 4
5	6 işaretli → atla	6	0 1 2 7 3 4 5
6	komşu yok	boş	0 1 2 7 3 4 5 6

Tablo 4.1.7: `bfs(0)` izlemesi. Her satır bir `poll` işlemi ve ardından gelen komşu taramasıdır.



Şekil 4.1.12: `bfs(0)` bittiğinde kalın kenarlar BFS ağacını oluşturur: *0→1, 0→2, 0→7, 2→3, 2→4, 3→5, 4→6*. Ağaç DFS ağacından sığ ve geniştir.

Çıktı `0 1 2 7 3 4 5 6`'dır ve katmanlar *{0}*, *{1, 2, 7}*, *{3, 4}*, *{5, 6}*'dır. 2, 1'in komşusu olarak da görülür, ama 0 işlenirken zaten işaretlendiği için kuyruğa ikinci kez girmez. Aynı durum 6 için de geçerlidir: 4 onu işaretleyip eklemiştir, 5 onu yalnız görür. Kuyruk en çok üç düğüm tutmuştur. 6'nın uzaklığı 3'tür (*0 → 2 → 4 → 6*); DFS ağacı aynı düğüme beş kenarda ulaşmıştı.

Sık yapılan hata

BFS'te düğümü kuyruktan çıkarırken işaretlemek. Bu durumda işaretsiz bir düğümü, kuyrukta beklediği sürede onu gören her komşusu yeniden ekler. Sonuç çoğu zaman yine doğrudur, çünkü çıkarılan düğüm işaretliyse atlanır; ama kuyruk gereksiz yere büyür. Beş düğümlü tam çizgede 0'dan başlayan doğru BFS kuyruğa 5 ekleme yapar ve kuyruk en çok 4 düğüm tutar. Çıkarırken işaretleyen sürüm aynı çizgede 11 ekleme yapar ve kuyruk 7 düğüme ulaşır. Genel olarak ekleme sayısı *O(V)*'den *O(E)*'ye çıkar. Doğrusu eklerken işaretlemektir.

4.1.4.6 Uzaklık ve ebeveyn dizileri

BFS'in en kısa yol özelliğini kullanmak için ziyaret dizisi yerine iki dizi tutulur. `uzaklik[v]`, *s*'den *v*'ye en az kenar sayısını; ebeveyn dizisi (kodda `ata[]`) ise BFS ağacında *v*'nin ebeveynini, yani *v*'yi keşfeden düğümü tutar. `uzaklik[v] == -1` "henüz keşfedilmedi" anlamına gelir; bu yüzden ayrı bir `ziyaret[]` dizisine gerek kalmaz.

```
void bfsUzaklik(int s) {
  Arrays.fill(uzaklik, -1);
  Arrays.fill(ata, -1); uzaklik[s] = 0;
  Queue<Integer> kuyruk = new LinkedList<>();
  kuyruk.add(s);
  while (!kuyruk.isEmpty()) {
    int v = kuyruk.poll();
    for (int w : komsu[v])
      if (uzaklik[w] == -1) {  // yeni
        uzaklik[w] = uzaklik[v] + 1;
        ata[w] = v;
        kuyruk.add(w);
      }
  }
}
```

- `Arrays.fill(uzaklik, -1);` bütün düğümleri keşfedilmemiş sayar.
- `Arrays.fill(ata, -1); uzaklik[s] = 0;` bütün ebeveynleri "yok" (*−1*) yapar ve başlangıcın uzaklığını 0'a çeker. Başlangıcın ebeveyni de *−1* kalır; yol kurulurken bu değer köke varıldığını gösterir. `ata[]`'nın tamamının doldurulması, ulaşılamayan düğümlerin de *−1* taşımasını sağlar; bunun neden gerektiği yol kurulurken görülecek.
- `if (uzaklik[w] == -1)` komşunun yeni olup olmadığını sorar. Yeni komşu için `uzaklik[w] = uzaklik[v] + 1` yazılır: *w*, *v*'nin katmanının bir sonraki katmanındadır. `ata[w] = v`, *w*'ye *v*'den gelindiğini kaydeder.

Kodu 1'den 8'e numaralı yönsüz örnek çizgede izleyelim (1–2, 1–3, 1–4, 1–5, 2–6, 2–7, 3–8). Tabloda diziler 1–8 indisleriyle yazılmıştır; "—" henüz keşfedilmemiş düğümdür (`uzaklik = -1`, `ata = -1`).

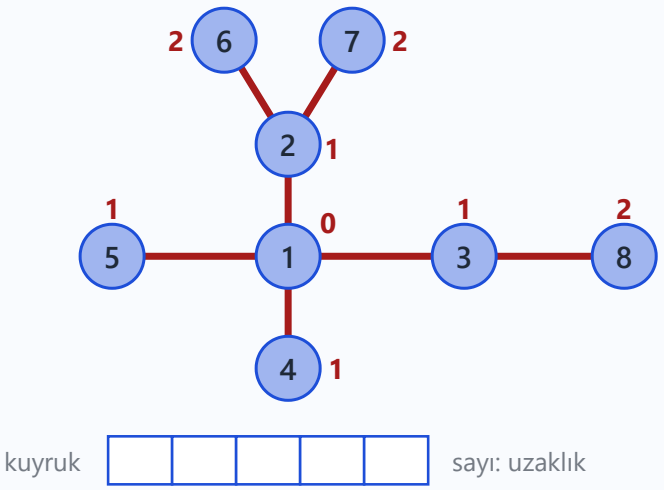
Alınan	Kuyruk (sonra)	1	2	3	4	5	6	7	8
—	1	0/−1	—	—	—	—	—	—	—
1	2, 3, 4, 5	0/−1	1/1	1/1	1/1	1/1	—	—	—
2	3, 4, 5, 6, 7	0/−1	1/1	1/1	1/1	1/1	2/2	2/2	—
3	4, 5, 6, 7, 8	0/−1	1/1	1/1	1/1	1/1	2/2	2/2	2/3
4, 5, 6, 7, 8	boş	değişiklik yok: her birinin tek komşusu zaten keşfedilmiş							

Tablo 4.1.8: `bfsUzaklik(1)` izlemesi. Hücrelerde `uzaklik/ata` yazılıdır.

2 kuyruktan alındığında komşusu 1 zaten keşfedilmiştir (`uzaklik[1] = 0`), bu yüzden atlanır. 6 ve 7 ise yenidir ve `uzaklik[2] + 1 = 2` değerini alır. Kuyruk boşaldığında sonuç Tablo 4.1.9'daki gibidir.

düğüm	1	2	3	4	5	6	7	8
uzaklık	0	1	1	1	1	2	2	2
ebeveyn	—	1	1	1	1	2	2	3

Tablo 4.1.9: Kaynak 1 için BFS sonucu.



Şekil 4.1.13: `bfsUzaklik(1)` bittiğinde kalın kenarlar ebeveyn bağlantılarıdır ve BFS ağacını oluşturur. Örneğin 8'in ebeveyni 3, 3'ün ebeveyni 1'dir; 1'den 8'e en kısa yol 1 → 3 → 8'dir ve uzunluğu `uzaklik[8] = 2` 'dir.

Kuyruktan çıkan düğümlerin uzaklıkları sırasıyla 0, 1, 1, 1, 1, 2, 2, 2'dir; dizi hiç azalmaz. Bu, BFS'in temel özelliğidir ve en kısa yol teoreminin ispatında kullanılır.

4.1.4.7 Ağırlıksız en kısa yol

Teorem 4.1.2 (BFS ve en kısa yol)

Ağırlıksız bir çizgede *s*'den BFS yapıldıktan sonra, *s*'den ulaşılabilen her *v* için `uzaklik[v]`, *s*'den *v*'ye giden en kısa yolun kenar sayısına eşittir. Ebeveyn bağlantıları *v*'den *s*'ye izlenerek bu uzunlukta bir yol elde edilir.

İspat. Önce iki gözlem yapılır. (1) Kuyruğa giren her düğümün uzaklığı, onu ekleyen düğümün uzaklığından bir fazladır. (2) Kuyruktaki uzaklıklar her an azalmayan sıradadır ve ilk ile son öğe arasındaki fark en fazla 1'dir. İkinci gözlem tümevarımla gösterilir: başta kuyrukta yalnız *s* vardır. Uzaklığı *d* olan baştaki düğüm çıkarılıp yeni düğümler *d + 1* uzaklığıyla sona eklendiğinde kuyrukta en küçük değer en az *d*, en büyük değer en fazla *d + 1* kalır. Şimdi teoremin kendisi gösterilsin. Bir *v* için

`uzaklik[v]`'nin gerçek en kısa uzaklık *δ(v)*'den küçük olamayacağı açıktır, çünkü ebeveyn zinciri *s*'den *v*'ye `uzaklik[v]` kenarlı gerçek bir yoldur. Tersini göstermek için `uzaklik[v] > δ(v)` olan ve *δ* değeri en küçük olan *v* alınsın. *s*'den *v*'ye en kısa bir yolda *v*'den önceki düğüm *u* olsun; *δ(u) = δ(v) − 1*'dir ve *v*'nin seçimi gereği `uzaklik[u] = δ(u)`'dur. *u* kuyruktan çıkarıldığında *v* henüz keşfedilmemişse *u* onu keşfeder ve `uzaklik[v] = δ(u) + 1 = δ(v)` olur; çelişki. *v* daha önce keşfedilmişse onu ekleyen düğüm *u*'dan önce kuyruktan çıkmıştır. Gözlem (2) gereği o düğümün uzaklığı en fazla *δ(u)*'dur; o zaman `uzaklik[v] ≤ δ(u) + 1 = δ(v)` olur; yine çelişki. ■

Teoremin günlük dildeki karşılığı şudur: BFS bir düğüme *ilk* ulaştığı anda ona en kısa yoldan ulaşmıştır. Daha kısa bir yol olsaydı, o yolun son düğümünden önceki düğüm daha önce kuyruktan çıkardı ve hedefi daha önce keşfederdi. Bu yüzden BFS bir düğümün uzaklığını bir kez yazar ve hiçbir zaman düzeltmez.

Yolun kendisi ebeveyn dizisinden geri kurulur:

```
List<Integer> yol(int[] ata, int s, int t) {
  LinkedList<Integer> yol = new LinkedList<>();
  for (int v = t; v != -1; v = ata[v])
    yol.addFirst(v);   // hedeften geriye
  if (yol.getFirst() != s)
    return null;       // t'ye ulaşılamadı
  return yol;
}
```

Döngü hedef *t*'den başlar ve her adımda ebeveyne geçer; *−1*'e gelince durur. Düğümler hedeften kaynağa doğru bulunduğu için listenin *başına* eklenir (`addFirst`); böylece liste kaynaktan hedefe doğru sıralı çıkar. Kaynak 1, hedef 8 için döngü 8'i ekler, `ata[8] = 3`'e geçer, 3'ü ekler, `ata[3] = 1`'e geçer, 1'i ekler ve `ata[1] = -1` ile durur. Liste `[1, 3, 8]`'dir; ilk öğe kaynaktır ve yol döndürülür. Yolun düğüm sayısı `uzaklik[t] + 1` olduğu için yol kurma *O(uzaklik[t])* zaman alır.

Son `if`, hedefe hiç ulaşılamadığı durumu yakalar. *t* keşfedilmediyse `ata[t] = -1`'dir; döngü yalnız *t*'yi ekleyip durur. Listenin ilk öğesi *t*'dir, *s* değil; metot `null` döndürür. Bu denetimin doğru çalışması, `ata[]`'nın BFS başında *−1* ile doldurulmasına bağlıdır.

Sık yapılan hata

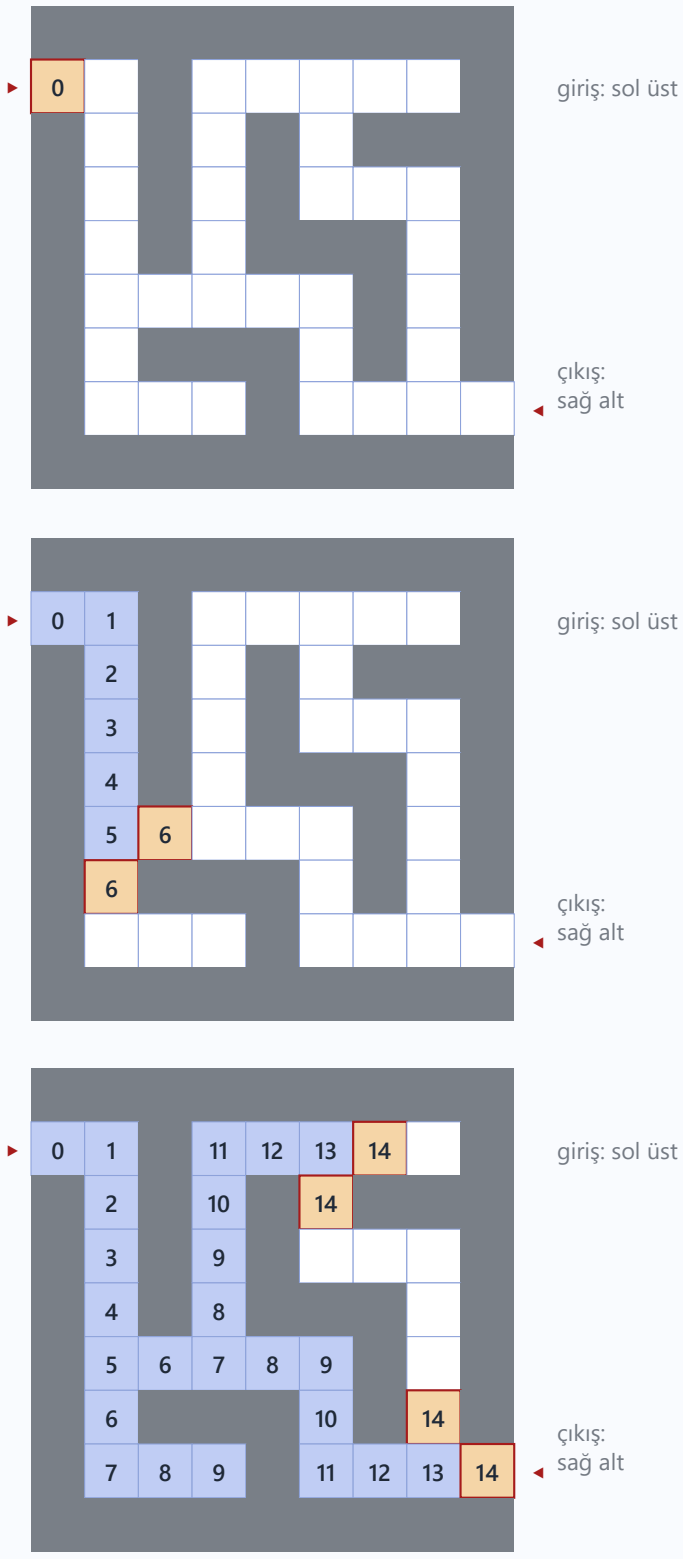
BFS'te yalnız `uzaklik[]`'yı *−1* ile doldurup `ata[]`'yı Java'nın varsayılan değerleriyle bırakmak. `int` dizisi 0'larla oluşturulduğu için ulaşılamayan bir *t* için `ata[t] = 0` kalır ve yol kurma döngüsü *t*'den 0'a geçer. Kaynak 0 ise döngü `[0, t]` listesini kurar ve var olmayan bir yolu döndürür. Örneğin 0–1, 1–3 kenarlı dört düğümlü çizgede 2'ye hiçbir kenar yoktur, ama 0'dan yapılan BFS'ten sonra `yol(ata, 0, 2)` bu durumda `[0, 2]` döndürür. Kaynak 0 değilse ve 0 da ulaşılamıyorsa döngü 0'da kendi üzerine döner ve hiç bitmez. `ata[]` da *−1* ile doldurulunca aynı çağrı doğru olarak `null` döndürür.

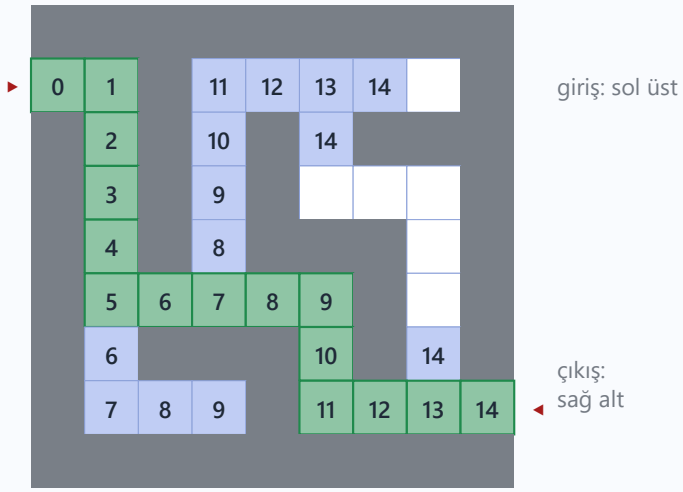
Teorem yalnız **ağırlıksız** çizgede, ya da bütün ağırlıkların eşit olduğu çizgede geçerlidir. BFS kenar sayısını en aza indirir, ağırlık toplamını değil. Kenarları $s \to t$ (ağırlık 10), $s \to a$ (ağırlık 1) ve $a \to t$ (ağırlık 1) olan bir çizge alınsın. BFS t'yi ilk katmanda, s'nin doğrudan komşusu olarak bulur. Bulunan yol tek kenarlıdır, ama maliyeti 10'dur. En ucuz yol $s \to a \to t$'dir ve maliyeti 2'dir. Ağırlıklı çizgede en kısa yol için Dijkstra algoritması gerekir.

4.1.4.8 BFS ile labirent çözmek

Kareli bir labirent doğrudan bir çizgeye çevrilebilir. Her boş hücre bir düğümdür. Yukarı, aşağı, sağa ya da sola bitişik iki boş hücre arasında bir kenar vardır. Duvar hücreleri çizgeye hiç girmez. Böylece "girişten çıkışa en az adım" sorusu, "bu çizgede iki düğüm arasındaki en kısa yol" sorusuna dönüşür. Çizge ağırlıksızdır, çünkü her adımın maliyeti aynıdır; doğru araç BFS'tir.

Şekil 4.1.14'teki labirent 9 × 9 hücreliktir. Hücrelerin 47'si duvar, 34'ü boştur; bitişik boş hücre çiftlerinin sayısı da 34'tür. Çizgede 34 düğüm ve 34 kenar vardır. Giriş sol üstte, çıkış sağ alttadır. Bir düğümün komşuları, dört yöndeki boş hücrelerdir. Komşular kod içinde ayrıca bir listeye yazılmaz; bir hücrenin dört komşusunun satır ve sütun numarası doğrudan hesaplanır. Böyle tanımlanan çizgeye **örtük çizge** denir.


0
giriş: sol üst
çıkış:
sağ alt
0 1
2
3
4
5 6
6
giriş: sol üst
çıkış:
sağ alt
0 1 11 12 13 14
2 10 14
3 9
4 8
5 6 7 8 9
6 10 14
7 8 9 11 12 13 14
giriş: sol üst
çıkış:
sağ alt



Şekil 4.1.14: Labirentte BFS. Hücrelerdeki sayı girişten o hücreye en az adım sayısıdır; sarı hücreler o anki sınırdır. Sol üstte başlangıç: giriş uzaklık 0 ile kuyruktadır. Sağ üstte uzaklık 6: yol 5. hücrede çatallanmış, dalga hem sağa hem aşağı yayılmıştır. Sol altta uzaklık 14: çıkışa ulaşılmıştır. Sağ altta ebeveyn bağlantılarıyla geri kurulan en kısa yol (yeşil, 15 hücre).

Gezinme şöyle ilerler. Giriş hücresi uzaklık 0 ile kuyruğa konur. İlk adımlarda labirent tek bir koridordur ve dalga aşağı doğru iner; her hücreye yazılan sayı bir öncekinden bir fazladır. Uzaklık 5'teki hücrede yol çatallanır. Dalga artık iki kola ayrılır: biri sağa ve oradan yukarıdaki odaya, öteki aşağıya ilerler. BFS bu iki kolu sırayla değil, aynı anda ilerletir; kuyrukta iki kolun uzaklığı 6 olan hücreleri, ardından uzaklığı 7 olan hücreleri yan yana bekler. Bu yüzden BFS'e “var olan bütün yolları eşzamanlı deneyen” arama da denir.

Uzaklık 10'da dalga hem yukarıdaki koridora, hem de aşağıdaki alt koridora ulaşır. Üst kol bir süre çıkmaz sokaklarda dolaşırken alt koridor çıkışa yaklaşır. Çıkış hücresine 14 adımda ulaşılır. Teorem gereği bu değer kesin olarak en kısadır; daha kısa bir yol olsaydı çıkış daha önceki bir katmanda keşfedilirdi. Arama bu noktada durdurulabilir. Durdurulmazsa uzaklığı 15, 16 ve 17 olan altı hücre daha keşfedilir; bu hücreler en kısa yol için gereksizdir. En kısa yol, çıkıştan başlayıp ebeveyn bağlantıları girişe kadar izlenerek bulunur. Yol 15 hücreden, yani 14 adımdan oluşur.

DFS aynı labirentte de çıkışı bulur, ama bulduğu yolun en kısa olduğu garanti değildir. Çatallanma noktasında önce yukarıdaki kola girerse, o kolun bütün çıkmazlarını dolaştıktan sonra çıkışa uzun bir dolambaçla varabilir. BFS'in bedeli ise bellektir: dalganın bütün sınırını aynı anda kuyrukta tutar.

4.1.4.9 BFS ile DFS'in karşılaştırılması

	DFS	BFS
Yardımcı yapı	Yığın (LIFO) ya da özyineleme	Kuyruk (FIFO)
Gezinme biçimi	Bir dalı sonuna kadar izler, sonra geri döner	Katman katman, uzaklık sırasıyla
Zaman	$O(V + E)$ liste; $O(V^2)$ matris	$O(V + E)$ liste; $O(V^2)$ matris
Ek bellek	$O(V)$: en uzun yol yığında	$O(V)$: en geniş katman kuyrukta

	DFS	BFS
En kısa yol	Hayır	Evet (ağırlıksız çizgede)
Ürettiği ağaç	Derin ve ince	Sığ ve geniş
Tipik kullanım	Döngü, bileşen, topolojik sıra, SCC, geri izleme	En az adım, düzey, iki parçalılık testi, yayılma

Tablo 4.1.10: DFS ile BFS'in karşılaştırılması.

Aynı yönlü çizgede, aynı kaynak 0'dan DFS `0 1 2 3 5 6 4 7`, BFS ise `0 1 2 7 3 4 5 6` yazar. İkisi de bütün düğümleri bir kez ziyaret eder ve ikisinin de zamanı aynıdır; fark ziyaret sırasında ve ürettikleri ağaçtadır.

Bellek karşılaştırması çizgenin biçimine bağlıdır. 15 düğümlü tam bir ikili ağaç (dört düzey) kökten gezilsin. BFS'te son düzeydeki sekiz yaprak aynı anda kuyrukta bulunur; kuyruk en çok 8 düğüm tutar. Özyinelemeli DFS'te ise çağrı yığını kökten yaprağa giden yol kadar, 4 çerçeve derinleşir. Ağaç 20 düzeyli olsaydı, yani 1.048.575 düğümü olsaydı, BFS kuyruğu 524.288 düğüme ulaşırdı; DFS'in yığını ise 20 çerçevede kalırdı. Tersine, uzun bir yol çizgesinde DFS'in yığını bütün düğümleri tutar, BFS'in kuyruğu ise uçtan başlayınca tek düğümle yetinir. İkisinin de en kötü durumu $O(V)$'dir; hangisinin daha az bellek kullanacağını çizgenin derinliği ve genişliği belirler.

Örnek 4.1.4

Örnek 4.1.3'teki yönsüz çizgede (A–B, A–C, B–D, C–D, C–E, D–F, E–F, F–G; komşular alfabetik) A'dan BFS ile ziyaret sırası, uzaklıklar ve A'dan G'ye en kısa yol bulunsun.

1. A işaretlenir, kuyruğa girer. Kuyruk: A.
2. A alınır; B ve C keşfedilir (uzaklık 1, ebeveyn A). Kuyruk: B, C.
3. B alınır; A işaretli, D keşfedilir (uzaklık 2, ebeveyn B). Kuyruk: C, D.
4. C alınır; A ve D işaretli, E keşfedilir (uzaklık 2, ebeveyn C). Kuyruk: D, E.
5. D alınır; B ve C işaretli, F keşfedilir (uzaklık 3, ebeveyn D). Kuyruk: E, F.
6. E alınır; C ve F işaretli. Kuyruk: F.
7. F alınır; D ve E işaretli, G keşfedilir (uzaklık 4, ebeveyn F). Kuyruk: G.
8. G alınır; F işaretli. Kuyruk boşalır.

Ziyaret sırası **A B C D E F G**'dir. Uzaklıklar A 0, B 1, C 1, D 2, E 2, F 3, G 4'tür. F'yi E değil D keşfetmiştir, çünkü D kuyruktan E'den önce çıkar. G'den ebeveynler izlenir: G'nin ebeveyni F, F'ninki D, D'ninki B, B'ninki A. Ters çevrilince en kısa yol **A–B–D–F–G** bulunur; dört kenarlıdır. DFS'in aynı çizgede izlediği A–B–D–C–E–F–G yolu ise altı kenarlıdır.

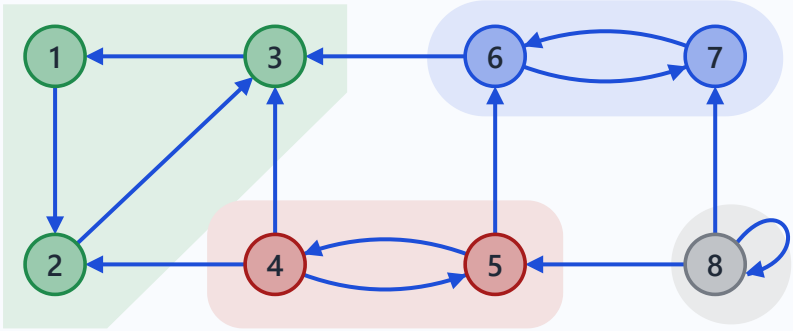
4.1.5 Güçlü bağlantılı bileşenler

Yönsüz bir çizgede bağlılık basit bir kavramdır: *u*'dan *v*'ye yol varsa *v*'den *u*'ya da vardır. Yönlü çizgede bu simetri bozulur. Tek yönlü sokaklarla dolu bir şehirde bir mahalleden ötekine gidilebilir, ama aynı yoldan geri dönülemeyebilir. Hangi düğümlerin birbirine karşılıklı olarak ulaşabildiğini bulmak, yönlü çizgelerin yapısını anlamanın ilk adımıdır.

4.1.5.1 Tanım

Tanım 4.1.7

Yönlü bir çizgede *u*'dan *v*'ye ve *v*'den *u*'ya yol varsa *u* ile *v* **karşılıklı ulaşılabilirdir**. Her düğümün öteki her düğüme ulaşabildiği ve bu özelliği bozmadan büyütülemeyen (maksimal) düğüm kümesine **güçlü bağlantılı bileşen** (*strongly connected component*, SCC) denir.



Şekil 4.1.15: Sekiz düğümlü yönlü bir çizge ve dört güçlü bağlantılı bileşeni: *{1, 2, 3}*, *{4, 5}*, *{6, 7}* ve *{8}*. Aynı renkteki düğümler aynı bileşendedir.

Şekil 4.1.15'teki çizgenin kenarları *1→2*, *2→3*, *3→1*, *4→2*, *4→3*, *4→5*, *5→6*, *5→4*, *6→3*, *6→7*, *7→6*, *8→5*, *8→7* ve *8→8*'dir. 1, 2 ve 3 bir döngü üzerindedir (*1 → 2 → 3 → 1*); her biri ötekilere ulaşır. 4 ile 5 arasında iki yönde de kenar vardır; 6 ile 7 de öyle. 8 ise tek başına bir bileşendir: 8'den öteki bütün düğümlere ulaşılır, ama hiçbir düğümden 8'e ulaşılamaz.

Tanımdan birkaç sonuç çıkar. Karşılıklı ulaşılabilirlik bir denklik bağıntısıdır: her düğüm kendisine ulaşır (sıfır uzunluklu yol), bağıntı simetriktir ve geçişlidir ($u \rightleftarrows v$ ve $v \rightleftarrows w$ ise $u \rightleftarrows w$). Bu yüzden güçlü bağlantılı bileşenler düğüm kümesini ayrık parçalara böler; her düğüm tam olarak bir bileşene aittir. Tek bir düğüm de bileşen olabilir; döngüye hiç katılmayan her düğüm böyledir.

Sık karşılaşılan bir yanılgı, "bir düğüm ve ondan ulaşılabilen bütün düğümler" kümesini bileşen sanmaktır. 8'den bütün düğümlere ulaşılır, ama *{1, …, 8}* bir SCC değildir, çünkü örneğin 1'den 8'e dönülemez. SCC iki yönlü ulaşılabilirlik ister.

Her bileşen tek bir düğüme indirgenip bileşenler arasındaki kenarlar korunursa **bileşen çizgesi** elde edilir. Şekildeki örnekte bileşen çizgesinin kenarları şunlardır: *{8} → {4, 5}*, *{8} → {6, 7}*, *{4, 5} → {1, 2, 3}*, *{4, 5} → {6, 7}* ve *{6, 7} → {1, 2, 3}*. Bileşen çizgesi her zaman döngüsüzdür. İki bileşen arasında iki yönde de yol olsaydı, bu bileşenlerin düğümleri karşılıklı ulaşılabilir olurdu ve iki bileşen aslında tek bir bileşen olurdu. Bu yapı pek çok uygulamada kullanılır: sosyal ağlarda birbirini takip eden toplulukları bulmak, bir derleyicide birbirine karşılıklı bağımlı modülleri birlikte derlemek, mantıkta 2-SAT problemini çözmek bunlardan birkaçıdır.

Bileşenleri bulmanın en kaba yolu, her düğümden bir gezinme yapıp karşılıklı ulaşılabilirliği tek tek denetlemektir; bu *O(V · (V + E))* zaman alır. Robert Tarjan 1972'de bütün bileşenleri **tek bir DFS** ile, *O(V + E)* zamanda bulan bir algoritma yayımladı.

4.1.5.2 Tarjan algoritmasının fikri

Tarjan algoritması DFS sırasında her düğüme iki sayı atar ve keşfedilen düğümleri ayrı bir yığında tutar.

> **Tanım 4.1.8**
>
> **indeks(v)**, *v*'nin keşif sırasıdır: ilk keşfedilen düğümün indeksi 1, ikincinin 2'dir. **enDüşükBağ(v)** (*low-link*), *v*'nin DFS alt ağacından en fazla bir ağaç dışı kenarla, hâlâ yığında olan düğümlere ulaşarak erişilebilen en küçük indekstir. Başlangıçta *enDüşükBağ(v) = indeks(v)*'dir.

Yığın, DFS'in çağrı yığınından farklıdır. Bir düğüm keşfedildiğinde bu yığına itilir, ama işi bittiğinde hemen çekilmez. Bileşeni belirlenene kadar yığında kalır. Böylece yığında, o ana kadar keşfedilmiş ve henüz hiçbir bileşene atanmamış düğümler durur.

DFS bir *v* düğümünün kenarlarını incelerken üç durumla karşılaşır:

1. **Ağaç kenarı** *v → w*, *w* henüz keşfedilmemiş: *w*'den özyinelemeli olarak devam edilir. Çağrı dönünce *enDüşükBağ(v) = min(enDüşükBağ(v), enDüşükBağ(w))* yapılır. *w*'nin alt ağacı eski bir düğüme ulaşabiliyorsa *v* de o düğüme ulaşabilir.
2. **Yığındaki** bir *w*'ye kenar: *w* keşfedilmiş ve henüz bir bileşene atanmamıştır. *enDüşükBağ(v) = min(enDüşükBağ(v), indeks(w))* yapılır. *v*, kendisinden önce keşfedilmiş ve bileşeni hâlâ açık olan bir düğüme ulaşabiliyor.
3. **Bileşeni bulunmuş** bir *w*'ye kenar: *w* keşfedilmiş ve yığından çekilmiştir. Bu kenar **yok sayılır**. *w*'nin bileşeni kapanmıştır; *w*'den *v*'ye dönen bir yol yoktur ve *v* o bileşene katılamaz.

v'nin bütün kenarları incelendikten sonra karar verilir. *enDüşükBağ(v) = indeks(v)* ise *v*'nin alt ağacından *v*'den daha eski bir yığın düğümüne ulaşılamıyordur. *v* bir bileşenin **köküdür**, yani o bileşende ilk keşfedilen düğümdür. Yığından *v*'ye kadar (*v* dahil) çekilen bütün düğümler bir SCC oluşturur.

enDüşükBağ(v) < indeks(v) ise *v*'nin alt ağacından daha eski bir yığın düğümüne geri dönülebiliyordur; *v* o düğümle aynı bileşendedir ve kök değildir. *v* yığında kalır ve bileşeni, kök bittiğinde birlikte çekilir.

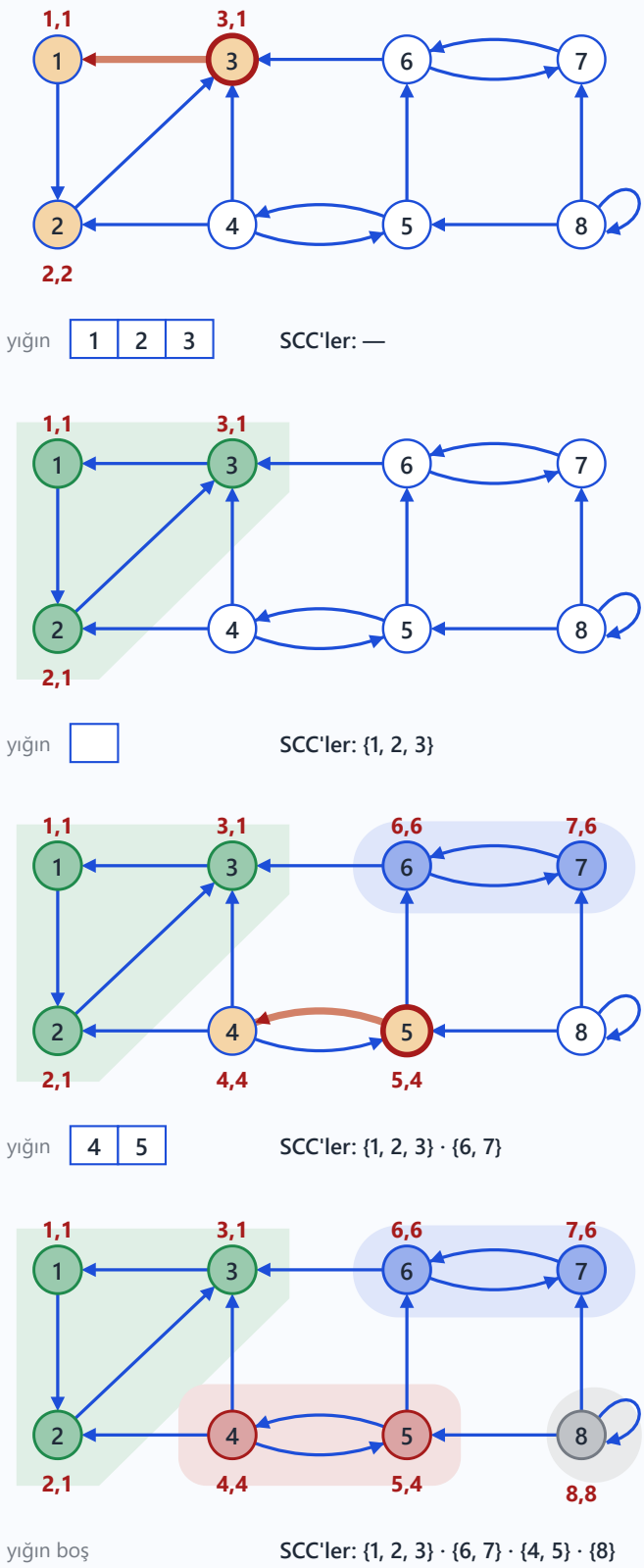
Kök koşulunun neden doğru olduğunu kısaca şöyle gerekçelendirmek mümkün: Bir bileşende ilk keşfedilen düğüm *r* olsun. Bileşenin öteki bütün düğümleri *r*'den ulaşılabilir olduğu için *r*'nin alt ağacına düşer ve *r* bitmeden keşfedilir. Bileşenin hiçbir düğümü *r*'den eski bir yığın düğümüne ulaşamaz; ulaşsaydı o düğüm de *r*'ye ulaşır, *r* ondan ulaşılır ve aynı bileşende olurdu. Bu yüzden *enDüşükBağ(r) = indeks(r)* kalır. Bileşenin öteki düğümleri ise *r*'ye geri dönebildikleri için daha küçük bir enDüşükBağ değeri alır.

4.1.5.3 Tarjan adım adım

Şekil 4.1.15'teki çizgede Tarjan algoritması düğümleri 1'den 8'e sırayla dener. İndeksler 1'den başlar. Her düğümün yanına "indeks, enDüşükBağ" çifti yazılır. Tablo 4.1.11 algoritmanın bütün adımlarını verir.

Adım	Olay	Yığın (sonra)	Bulunan SCC
1	SCC(1): indeks = enDüşükBağ = 1	1	
2	1 → 2 ağaç kenarı; SCC(2): 2, 2	1, 2	
3	2 → 3 ağaç kenarı; SCC(3): 3, 3	1, 2, 3	
4	3 → 1: 1 yığında → enDüşükBağ(3) = min(3, 1) = 1	1, 2, 3	
5	3 biter; 1 < 3, kök değil. 2'ye dönüş: enDüşükBağ(2) = min(2, 1) = 1	1, 2, 3	
6	2 biter, kök değil. 1'e dönüş: enDüşükBağ(1) = 1. 1'de enDüşükBağ = indeks → kök	boş	*{1, 2, 3}*
7	SCC(4): 4, 4	4	
8	4 → 2 ve 4 → 3 yok sayılır (bileşenleri bulunmuş); 4 → 5: SCC(5): 5, 5	4, 5	
9	5 → 6: SCC(6): 6, 6	4, 5, 6	
10	6 → 3 yok sayılır; 6 → 7: SCC(7): 7, 7	4, 5, 6, 7	
11	7 → 6: 6 yığında → enDüşükBağ(7) = min(7, 6) = 6	4, 5, 6, 7	
12	7 biter, kök değil. 6'ya dönüş: enDüşükBağ(6) = 6 değişmez; 6 köktür	4, 5	*{6, 7}*
13	5'e dönüş: enDüşükBağ(5) = min(5, 6) = 5. 5 → 4: 4 yığında → enDüşükBağ(5) = min(5, 4) = 4	4, 5	
14	5 biter, kök değil. 4'e dönüş: enDüşükBağ(4) = min(4, 4) = 4; 4 köktür	boş	*{4, 5}*
15	SCC(8): 8, 8	8	
16	8 → 5 ve 8 → 7 yok sayılır; öz döngü 8 → 8: 8 yığında, enDüşükBağ = min(8, 8) = 8	8	
17	8'de enDüşükBağ = indeks → kök	boş	*{8}*

Tablo 4.1.11: Sekiz düğümlü çizgede Tarjan algoritması. Yığın soldan sağa büyür.



Şekil 4.1.16: Tarjan algoritmasının dört anı; düğümlerin yanında "indeks, enDüşükBağ" yazılıdır. Sol üstte 3 → 1 kenarı 3'ün enDüşükBağ değerini 1'e indirir. Sağ üstte 1 kök olarak bulunmuş ve *{1, 2, 3}* yığından çekilmiştir. Sol altta 5 → 4 kenarı 5'in değerini 4'e indirir. Sağ altta dört bileşen bulunmuştur.

Tablodaki üç an algoritmanın bütün mantığını gösterir. 4. adımda 3 → 1 kenarı, yığındaki eski bir düğüme dönüşü yakalar ve 3'ün enDüşükBağ değerini 1'e indirir. Bu değer 5. ve 6. adımlarda çağrı dönüşleriyle 2'ye ve 1'e taşınır. Sonuçta yalnız 1'de indeks ile enDüşükBağ eşit kalır; 1 köktür ve yığında 1'in üstündeki her düğüm onun bileşenidir.

8. adımda 4 → 2 ve 4 → 3 kenarları yok sayılır. 2 ve 3 keşfedilmiştir, ama yığında değildir; bileşenleri kapanmıştır. Bu kenarlar dikkate alınsaydı 4'ün enDüşükBağ değeri 2'ye inerdi ve 4 kök sayılmazdı. Oysa 4, 2'ye ulaşsa da 2'den 4'e dönen bir yol yoktur; 4 yeni bir bileşenin ilk düğümüdür.

12. adımda 6 kök olarak bulunur ve *{6, 7}* çekilir, ama 5 ve 4 yığında kalır. 5 ile 4'ün bileşeni daha sonra, 14. adımda kapanır. Bileşenler bulunma sırasıyla *{1, 2, 3}*, *{6, 7}*, *{4, 5}*, *{8}*'dir. Bileşen çizgesindeki kenarlarla karşılaştırılınca bu sıranın **ters topolojik sıra** olduğu görülür: her bileşen, ondan kenar alan bütün bileşenlerden önce bulunur. *{1, 2, 3}*'ten çıkan hiçbir kenar yoktur ve ilk o bulunur; *{8}*'e ise hiçbir kenar girmez ve en son o bulunur. Bu bir rastlantı değildir: bir bileşen ancak ondan ulaşılabilen bütün bileşenler bulunduktan sonra kapanabilir.

4.1.5.4 Sözde kod ve Java kodu

Algoritmanın sözde kodu iki parçadır. Dış döngü ziyaret edilmemiş her düğümden `SCC` çağırır:

```
TARJAN_SCC(G):
  indeks = 1
  yığın = [ ]
  SCCler = [ ]
  her bir v için V içinde:
    eğer v.indeks tanımlı değilse:
      SCC(v)
  döndür SCCler
```

Asıl iş özyinelemeli `SCC` yordamındadır:

```
SCC(v):
  v.indeks = v.enDüşükBağ = indeks++
  yığın.ekle(v)
  her bir (v, w) için E içinde:
    eğer w.indeks tanımlı değilse:
      SCC(w)
      v.enDüşükBağ = min(v.enDüşükBağ, w.enDüşükBağ)
    değilse eğer w yığında ise:
      v.enDüşükBağ = min(v.enDüşükBağ, w.indeks)
  eğer v.enDüşükBağ == v.indeks:
    scc = [ ]
    döngü w ≠ v olduğu sürece:  // en az bir kez
      w = yığın.çıkar()
      scc.ekle(w)
    SCCler.ekle(scc)
```

Java kodu sözde kodun birebir karşılığıdır:

```
void scc(int v) {
  indeks[v] = dusuk[v] = ++sayac;
  yigin.push(v); yiginda[v] = true;
  for (int w : komsu[v]) {
    if (indeks[w] == 0) {
      scc(w);
      dusuk[v] = Math.min(dusuk[v], dusuk[w]);
    } else if (yiginda[w])
      dusuk[v] = Math.min(dusuk[v], indeks[w]);
  }
  if (dusuk[v] == indeks[v]) {
    int w;
    do {
      w = yigin.pop(); yiginda[w] = false;
      System.out.print(w + " ");
    } while (w != v);
    System.out.println();
  }
}
```

- `indeks[v] = dusuk[v] = ++sayac;` keşif sırasını atar. `sayac` 0'dan başlar ve önce artırıldığı için ilk düğümün indeksi 1 olur. Bu tercih, `indeks[w] == 0` denetimini "henüz keşfedilmedi" anlamında kullanmayı sağlar; ayrı bir ziyaret dizisine gerek kalmaz. İndeksleri 0'dan başlatmak da aynı bileşenleri verir; o durumda keşfedilmemiş düğüm için başka bir işaret, örneğin -1 gerekir.
- `yigin.push(v); yiginda[v] = true;` düğümü yığına iter. `yiginda[]` dizisi, "w yığında mı?" sorusunu $O(1)$ zamanda yanıtlar. Yığını baştan sona aramak her kenarda $O(V)$ sürerdi.
- `if (indeks[w] == 0)` dalı ağaç kenarıdır: özyinelemeli çağrıdan sonra `dusuk[v]`, `dusuk[w]` ile güncellenir.
- `else if (yiginda[w])` dalı yığındaki düğüme giden kenardır: `dusuk[v]`, `indeks[w]` ile güncellenir. Bileşeni bulunmuş w için iki koşul da yanlıştır ve hiçbir şey yapılmaz.
- Son `if` kök denetimidir. `do … while` döngüsü yığından v'ye kadar bütün düğümleri çeker ve yazdırır. Döngü `do … while` biçimindedir, çünkü en az bir düğüm (v'nin kendisi) her zaman çekilir. Her bileşen ayrı bir satıra yazılır.

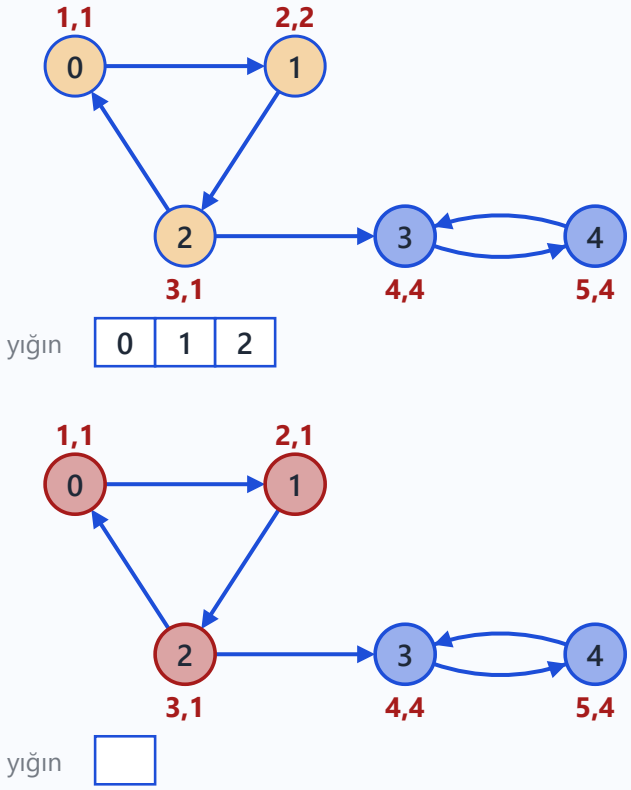
4.1.5.5 Kod izleme: scc(0)

İzlemenin çizgesi beş düğümlüdür: $0 \to 1$, $1 \to 2$, $2 \to 0$, $2 \to 3$, $3 \to 4$, $4 \to 3$. `komsu[2] = [0, 3]` 'tür.

Adım	Olay	Değişen değer	Yığın (sonra)	Çıktı
1	`scc(0)`	0: 1/1	0	
2	0 → 1 keşfedilmemiş → `scc(1)`	1: 2/2	0, 1	

Adım	Olay	Değişen değer	Yığın (sonra)	Çıktı
3	1 → 2 keşfedilmemiş → `scc(2)`	2: 3/3	0, 1, 2	
4	2 → 0: 0 yığında	`dusuk[2] = min(3, 1) = 1`	0, 1, 2	
5	2 → 3 keşfedilmemiş → `scc(3)`	3: 4/4	0, 1, 2, 3	
6	3 → 4 keşfedilmemiş → `scc(4)`	4: 5/5	0, 1, 2, 3, 4	
7	4 → 3: 3 yığında	`dusuk[4] = min(5, 4) = 4`	0, 1, 2, 3, 4	
8	`dusuk[4] = 4 ≠ indeks[4] = 5` : kök değil, döner		0, 1, 2, 3, 4	
9	`scc(4)` döndü	`dusuk[3] = min(4, 4) = 4`	0, 1, 2, 3, 4	
10	`dusuk[3] == indeks[3] = 4` : 3 köktür; 4 ve 3 çekilir		0, 1, 2	4 3
11	`scc(3)` döndü	`dusuk[2] = min(1, 4) = 1`	0, 1, 2	4 3
12	`dusuk[2] = 1 ≠ 3` : kök değil, döner	`dusuk[1] = min(2, 1) = 1`	0, 1, 2	4 3
13	`dusuk[1] = 1 ≠ 2` : kök değil, döner	`dusuk[0] = min(1, 1) = 1`	0, 1, 2	4 3
14	`dusuk[0] == indeks[0] = 1` : 0 köktür; 2, 1, 0 çekilir		boş	4 3 / 2 1 0

Tablo 4.1.12: `scc(0)` izlemesi. `indeks/dusuk` değerleri değiştikleri satırda yazılmıştır.



Şekil 4.1.17: Solda 10. adım: 3 kök olarak bulunmuş, *{3, 4}* yığından çekilmiştir; yığında 0, 1, 2 kalmıştır. Sağda izlemenin sonu: iki bileşen bulunmuştur. Düğüm yanındaki sayılar `indeks/dusuk` değerleridir.

Program iki satır yazar: önce `4 3`, sonra `2 1 0`. Bileşen içindeki düğümler yığından çekildikleri sırayla, yani keşif sırasının tersiyle yazılır. 4. adımdaki 2 → 0 kenarı bir geri kenardır ve `dusuk[2]`'yi 1'e indirir. 11. adımda `scc(3)` dönünce `dusuk[3] = 4` değeri 2'nin değerini değiştirmez, çünkü 2 zaten 1'e ulaşmıştır. *{3, 4}* bileşeni 2'den ulaşılan ama 2'ye geri dönemeyen bir bileşendir ve ilk o kapanır.

Örnek 4.1.5

Yönlü çizge *0→1, 1→2, 2→0, 1→3, 3→4, 4→5, 5→3, 5→6* verilsin (`komsu[1] = [2, 3]`, `komsu[5] = [3, 6]`). Tarjan algoritması 0'dan başlatılır ve indeksler 1'den başlar.

1. 0, 1, 2 sırayla keşfedilir: indeksleri 1, 2, 3. 2 → 0 kenarı yığındaki 0'a gider; `dusuk[2] = 1`. 2 kök değildir; dönüşte `dusuk[1] = 1` olur.
2. 1 → 3 ağaç kenarıdır. 3, 4, 5 sırayla keşfedilir: indeksleri 4, 5, 6. 5 → 3 kenarı yığındaki 3'e gider; `dusuk[5] = 4`.
3. 5 → 6 ağaç kenarıdır; 6'nın indeksi 7'dir. 6'nın çıkan kenarı yoktur, `dusuk[6] = indeks[6] = 7` : 6 köktür. İlk bileşen *{6}*'dır.
4. 5'e dönülür: `dusuk[5] = min(4, 7) = 4`; 5 kök değildir. 4'e dönüş `dusuk[4] = 4`, 3'e dönüş `dusuk[3] = 4` verir. 3'te indeks ile enDüşükBağ eşittir: ikinci bileşen *{3, 4, 5}*'tir.

5. 1'e dönülür: `dusuk[1] = min(1, 4) = 1`. 0'a dönülür ve 0 köktür: üçüncü bileşen *{0, 1, 2}*'dir.

Bileşenler bulunma sırasıyla *{6}*, *{3, 4, 5}*, *{0, 1, 2}*'dir. Bileşen çizgesi *{0, 1, 2}* → *{3, 4, 5}* → *{6}*'dır; bulunma sırası bunun tersidir.

4.1.5.6 Karmaşıklık

Teorem 4.1.3 (Tarjan'ın maliyeti)

Komşuluk listesiyle tutulan bir çizgede Tarjan algoritması bütün güçlü bağlantılı bileşenleri $O(V + E)$ zamanda ve $O(V)$ ek bellekle bulur.

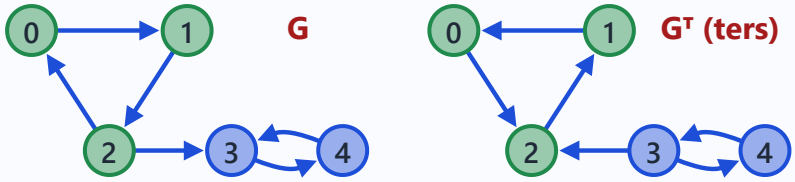
İspat. Algoritma bir DFS'tir ve DFS'in sayımı aynen geçerlidir. Her düğüm için `scc` bir kez çağrılır; düğüm yığına bir kez itilir ve bir kez çekilir. Düğüm başına iş sabittir: $O(V)$. Her kenar, kaynağının komşu listesinde gezinilirken bir kez incelenir ve her incelemede yapılan iş (bir `min` ve `yiginda[]` denetimi) sabittir: $O(E)$. Çekme döngüsünün toplam çalışma sayısı da çekilen düğüm sayısıdır, yani V'dir. Ek bellek `indeks`, `dusuk`, `yiginda` dizileri ve yığın için $O(V)$, özyineleme derinliği için yine $O(V)$'dir. ■

Sık yapılan hata

Tarjan kodunda `else if (yiginda[w])` denetimini atlayıp keşfedilmiş her w için `dusuk[v] = min(dusuk[v], indeks[w])` yapmak. Bileşeni bulunmuş bir düğüme giden kenar o zaman yanlışlıkla "geri dönüş" sayılır. Sekiz düğümlü örnekte 4 → 2 kenarı enDüşükBağ(4)'ü 2'ye, 6 → 3 kenarı enDüşükBağ(6)'yı 3'e, 8 → 5 kenarı enDüşükBağ(8)'i 5'e indirir. 4, 6 ve 8 artık kök sayılmaz. İlk bileşen *{1, 2, 3}* doğru bulunur, ama *{4, 5}*, *{6, 7}* ve *{8}* hiç çekilmez; 4'ten 8'e bütün düğümler algoritma bittiğinde yığında kalır.

4.1.5.7 Kosaraju algoritması

Güçlü bağlantılı bileşenleri bulan ikinci klasik yöntem Kosaraju algoritmasıdır. Tarjan'dan daha kolay anlaşılır, ama iki DFS ve çizgenin bir kopyasını ister. Algoritma, bir çizgenin **tersi** (*transpose*) üzerinde çalışır: ters çizge aynı düğümleri içerir ve her $u \to v$ kenarı $v \to u$ olur. Bir çizge ile tersinin güçlü bağlantılı bileşenleri aynıdır, çünkü karşılıklı ulaşılabilirlik okların yönünden etkilenmez.



Şekil 4.1.18: Solda bir çizge, sağda tersi. Bileşen renkleri aynı kalır, yalnız oklar döner.

Algoritmanın üç adımı vardır:

1. Çizgede DFS yapılır ve her düğüm **işi bittiğinde** bir yığına itilir. Yığının tepesinde en son biten düğüm durur.
2. Çizgenin tersi oluşturulur.
3. Yığından düğümler sırayla çekilir. Çekilen düğüm henüz ziyaret edilmemişse ters çizgede ondan bir DFS yapılır. Bu DFS'in ulaştığı düğümler bir SCC'dir.

Ters çizgeyi kuran parça:

```
List<Integer>[] ters = new List[n];
for (int v = 0; v < n; v++)
  ters[v] = new ArrayList<>();
for (int v = 0; v < n; v++)
  for (int w : komsu[v])
    ters[w].add(v);       // v → w kenarı w → v olur
```

Algoritmanın Java kodu:

```
void kosaraju() {
  for (int v = 0; v < n; v++)  // 1. tur
    if (!ziyaret[v]) dfs1(v);
  Arrays.fill(ziyaret, false); // 2. tur
  while (!yigin.isEmpty()) {
    int v = yigin.pop();
    if (!ziyaret[v]) {   // yeni SCC
      dfs2(v); System.out.println(); }
  }
}
void dfs1(int v) {
  ziyaret[v] = true;
  for (int w : komsu[v])
    if (!ziyaret[w]) dfs1(w);
  yigin.push(v);       // v bitti
}
```

İkinci turdaki `dfs2`, ters çizgede çalışan sıradan bir DFS'tir:

```
void dfs2(int v) {     // ters çizgede DFS
  ziyaret[v] = true;
  System.out.print(v + " ");
  for (int w : ters[v])
    if (!ziyaret[w]) dfs2(w);
}
```

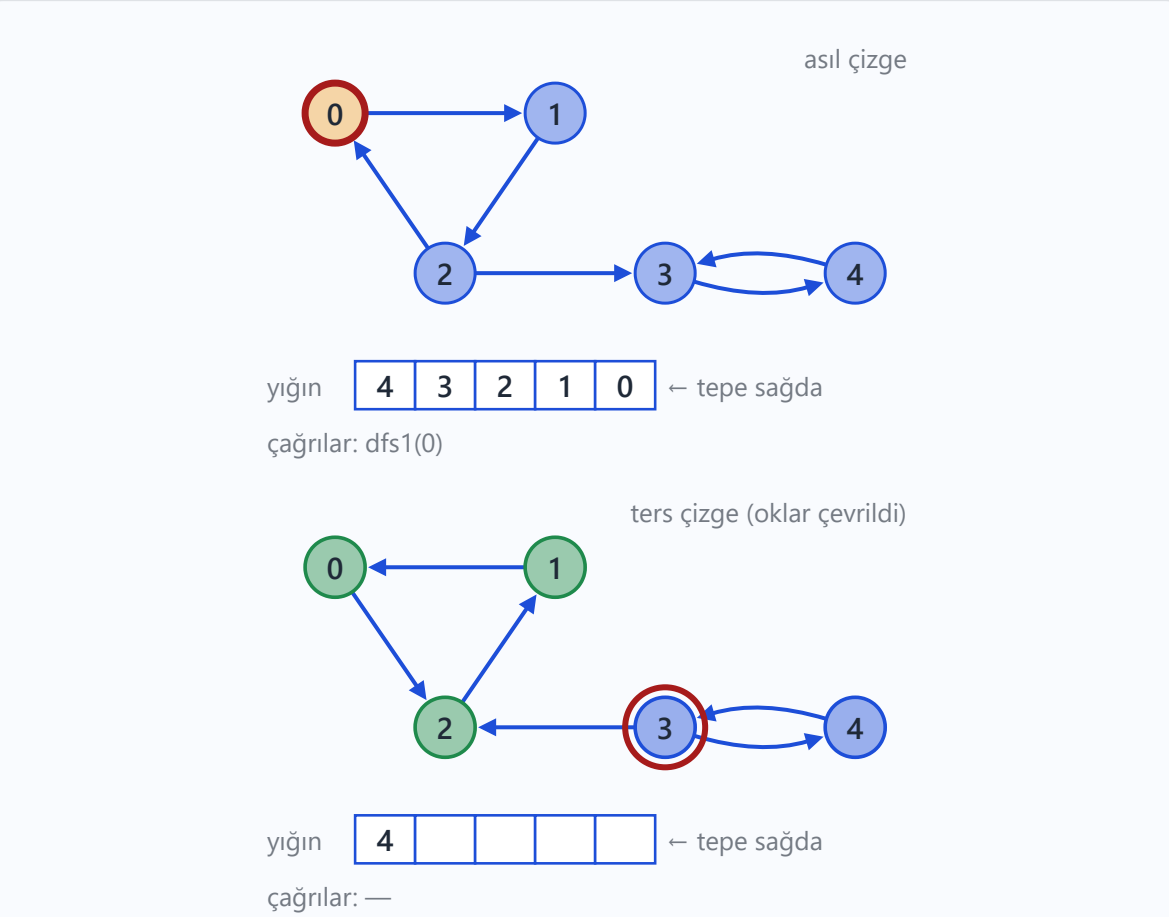
`dfs1`, `dfs`'ten tek bir satırla ayrılır: düğüm, komşularının hepsi bittikten sonra yığına itilir (`yigin.push(v); // v bitti`). Bu satır bitiş sırasını kaydeder. `kosaraju()` önce ziyaret edilmemiş her düğümden `dfs1` başlatır. Sonra ziyaret dizisini temizler ve yığını boşaltır. Yığından çekilen ziyaret edilmemiş her düğüm yeni bir bileşenin başlangıcıdır.

Algoritmanın neden doğru çalıştığı bileşen çizgesiyle açıklanır. Birinci turda en son biten düğüm, bileşen çizgesinde hiç kenar almayan bir *kaynak* bileşendedir. Ters çizgede bu bileşenin bütün dış kenarları içe döner. Bu yüzden ters çizgede ondan başlayan DFS bileşenden dışarı çıkamaz ve tam olarak o bileşeni gezer. Yığından sonraki ziyaret edilmemiş düğüm, kalan bileşenler arasında yine bir kaynak bileşendedir. Ters çizgede ondan çıkan kenarlar ya kendi bileşenine ya da zaten bulunmuş bileşenlere gider; ziyaretli düğümlere de girilmez. Böylece her DFS tam bir bileşen bulur.

Kosaraju algoritmasını Tarjan izlemesindeki beş düğümlü çizgede (*0→1*, *1→2*, *2→0*, *2→3*, *3→4*, *4→3*) izleyelim.

Tur	Olay	Yığın (sonra)	Bulunan SCC
1	`dfs1(0)` → `dfs1(1)` → `dfs1(2)`; 2 → 0 ziyaretli, atlanır; `dfs1(3)` → `dfs1(4)`	boş	
1	4'ün tek komşusu 3 ziyaretli: 4 biter, itilir	4	
1	3 biter, itilir	4, 3	
1	2, 1, 0 sırayla biter ve itilir	4, 3, 2, 1, 0	
2	0 çekilir; ters çizgede `dfs2(0)`: 0 → 2 → 1	4, 3, 2, 1	*{0, 2, 1}*
2	1 ve 2 çekilir; ziyaretli, atlanır	4, 3	
2	3 çekilir; `dfs2(3)`: 3 → 2 ters kenarı ziyaretli 2'ye gider; 3 → 4	4	*{3, 4}*
2	4 çekilir; ziyaretli, atlanır	boş	

Tablo 4.1.13: Kosaraju algoritmasının izlemesi. Yığın soldan sağa büyür; tepe sağdadır.



Şekil 4.1.19: Solda birinci turun sonu: bütün düğümler bitmiş ve bitiş sırasıyla yığına itilmiştir; tepede en son biten 0 vardır. Sağda ikinci tur: ters çizgede *{0, 1, 2}* bulunmuş, 3'ten başlayan DFS 3 → 2 ters kenarını atlayıp 4'e ulaşmıştır.

Program iki satır yazar: `0 2 1` ve `3 4`. Ters çizgede 3'ün komşu listesi `[2, 4]`'tür; 2 → 3 kenarı ters çizgede 3 → 2 olmuştur. Bu kenar 3'ü bir önceki bileşene bağlar, ama 2 ziyaretli olduğu için DFS oraya geçmez. Birinci turda 0 en son bittiği için kaynak bileşen olan *{0, 1, 2}*'den başlanmıştır. Bu bileşen, asıl çizgede *{3, 4}*'e kenar verir; ters çizgede bu kenar içe döndüğü için `dfs2(0)` 3'e ulaşamaz.

Sekiz düğümlü örnekte Kosaraju algoritması da aynı dört bileşeni bulur. Birinci turun bitiş sırası 3, 2, 1, 7, 6, 5, 4, 8'dir. İkinci turda bileşenler *{8}*, *{4, 5}*, *{6, 7}*, *{1, 2, 3}* sırasıyla bulunur. Bu sıra Tarjan'ın sırasının tam tersidir: Kosaraju bileşenleri topolojik sırayla, Tarjan ters topolojik sırayla verir.

	Tarjan	**Kosaraju**
DFS sayısı	1	2 (biri ters çizgede)
Ek yapı	indeks ve enDüşükBağ dizileri, yığın	ters çizge (*O(V + E)* bellek), yığın
Zaman	*O(V + E)*	*O(V + E)*
Bileşenlerin sırası	Ters topolojik	Topolojik

Tablo 4.1.14: Tarjan ile Kosaraju algoritmalarının karşılaştırılması.

İki algoritmanın mertebesi aynıdır. Tarjan çizgede bir kez gezinir ve kopya gerektirmez; pratikte genellikle daha hızlıdır. Kosaraju'nun mantığı daha kolay kavranır ve doğruluğu daha kolay gerekçelendirilir. Ters çizgeyi kurmak $O(V + E)$ zaman ve bellek ister, ama mertebeyi değiştirmez.

4.1.6 Gezinmenin uygulamaları

DFS ve BFS tek başlarına yalnız düğümlerde bir sırayla gezinir. Gezinmeye birkaç satır eklenince çizgenin yapısıyla ilgili pek çok soru aynı $O(V + E)$ zamanda yanıtlanır. Bu kesim en sık kullanılan uygulamaları ele alır.

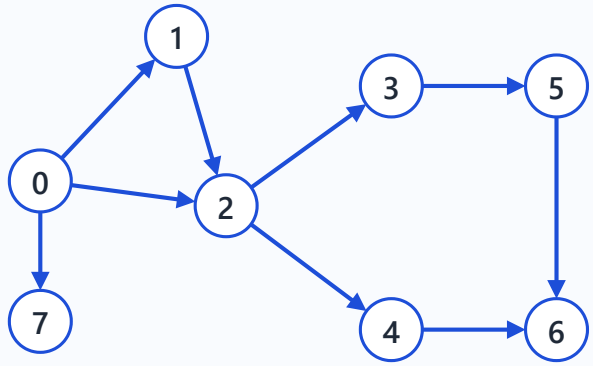
4.1.6.1 Topolojik sıralama

Tanım 4.1.9

Döngüsüz yönlü bir çizgenin (DAG) **topolojik sıralaması** (*topological sort*), düğümlerin öyle bir sıralanışıdır ki her $u \to v$ kenarı için u, v'den önce gelir.

Kenarlar "önce bu, sonra şu" türünden ön koşulları anlatır. Ön koşullu dersler, bir projenin derlenme sırası, bir tablolama programında hücrelerin hesaplanma sırası ve bir şantiyedeki işlerin planlanması topolojik sıralama problemleridir. Örneğin "Veri Yapıları" dersi "Programlama" dersinden sonra alınmalıysa, ders planı bu iki dersi bu sırayla içermelidir.

Topolojik sıra yalnız döngüsüz yönlü çizgede vardır. Çizgede $u \to v \to \ldots \to u$ biçiminde bir döngü olsaydı, u'nun hem bu döngüdeki her düğümden önce hem de sonra gelmesi gerekirdi. Tersine, her DAG'ın en az bir topolojik sırası vardır. Ayrıca topolojik sıra çoğu zaman tek değildir: aralarında yol olmayan iki düğüm herhangi bir sırada yazılabilir.



Şekil 4.1.20: Sekiz düğümlü döngüsüz yönlü çizge: *0→1, 0→2, 0→7, 1→2, 2→3, 2→4, 3→5, 4→6, 5→6*. Giriş dereceleri 0:0, 1:1, 2:2, 3:1, 4:1, 5:1, 6:2, 7:1'dir.

Şekil 4.1.20'deki çizge, DFS izlemesindeki yönlü çizgeden *2 → 0* kenarı çıkarılarak elde edilmiştir. O kenar çizgeye bir döngü katıyordu; kenar çıkınca çizge döngüsüz olur. Topolojik sıralamanın iki klasik yöntemi vardır: BFS tabanlı Kahn algoritması ve DFS tabanlı ters bitiş sırası. Bu çizgede ikisinin sonucu farklıdır, ama ikisi de geçerlidir:

Yöntem	Sıra
Kahn	`0 1 7 2 3 4 5 6`
DFS (ters bitiş)	`0 7 1 2 4 3 5 6`

Tablo 4.1.15: Aynı DAG'ın iki topolojik sırası.

İki sırada da dokuz kenarın her biri soldan sağa gider. Örneğin *4 → 6* ve *5 → 6* kenarları için 6 iki sırada da en sondadır. 7 ise yalnız 0'dan sonra gelmek zorundadır; Kahn onu 1'den sonra, DFS 0'ın hemen arkasına yazar.

4.1.6.2 Kahn algoritması ve kod izleme

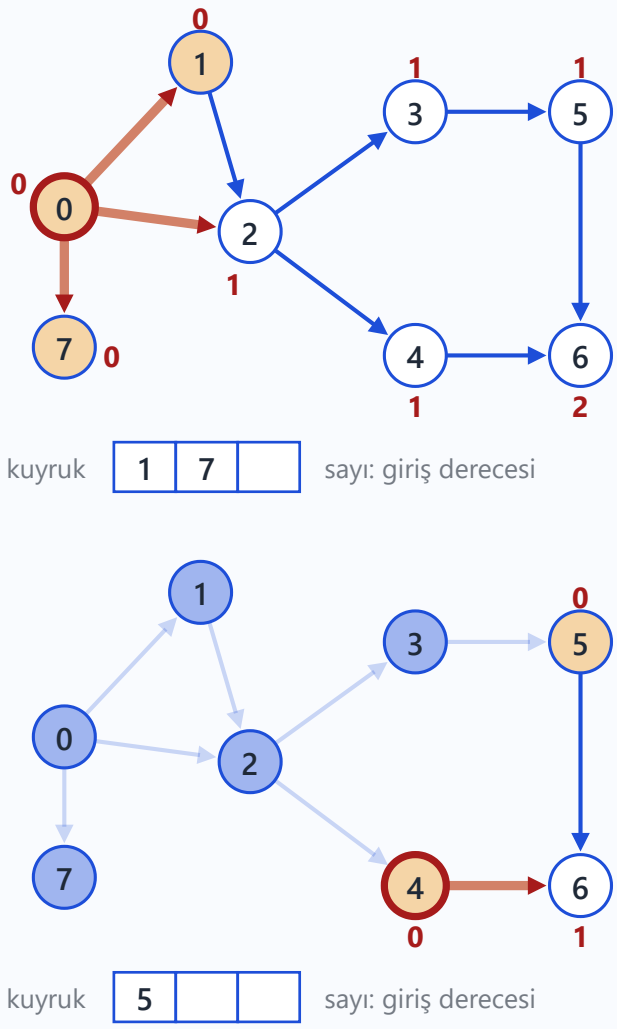
Kahn algoritmasının fikri, ön koşulu kalmamış işleri sırayla yapmaktır. Giriş derecesi 0 olan bir düğümün hiçbir ön koşulu yoktur; ilk o yazılabilir. Yazılan düğüm ve kenarları çizgeden silinmiş sayılır. Bu silme, bazı düğümlerin giriş derecesini 0'a indirir; o düğümler de artık yazılabilir. Hazır düğümler bir kuyrukta bekler.

```
void kahn() {
  int[] giris = new int[n];
  for (int v = 0; v < n; v++)
    for (int w : komsu[v]) giris[w]++;
  Queue<Integer> kuyruk = new LinkedList<>();
  for (int v = 0; v < n; v++)
    if (giris[v] == 0) kuyruk.add(v);
  while (!kuyruk.isEmpty()) {
    int v = kuyruk.poll();
    System.out.print(v + " ");
    for (int w : komsu[v])
      if (--giris[w] == 0) kuyruk.add(w);
  }
}
```

- İlk iç içe döngü her kenarın hedefinin giriş derecesini bir artırır. Bütün komşu listelerinde bir kez gezinildiği için bu kısım *O(V + E)*'dir.
- İkinci döngü giriş derecesi 0 olan bütün düğümleri kuyruğa koyar.
- Ana döngü kuyruktan bir düğüm alır ve yazar. Çıkan her kenar için hedefin derecesi azaltılır: `--giris[w]`. Kenarın kendisi silinmez, yalnız sayaç azaltılır. Derecesi 0'a inen hedef kuyruğa eklenir.

Alınan	Azalan dereceler	`giris` (sonra)	Kuyruk (sonra)	Çıktı
—	dereceler sayılır; 0 kuyruğa girer	0 1 2 1 1 1 2 1	0	
0	1 → 0, 2 → 1, 7 → 0	0 0 1 1 1 1 2 0	1, 7	0
1	2 → 0	0 0 0 1 1 1 2 0	7, 2	0 1
7	çıkan kenar yok	aynı	2	0 1 7
2	3 → 0, 4 → 0	0 0 0 0 0 1 2 0	3, 4	0 1 7 2
3	5 → 0	0 0 0 0 0 0 2 0	4, 5	0 1 7 2 3
4	6 → 1	0 0 0 0 0 0 1 0	5	0 1 7 2 3 4
5	6 → 0	0 0 0 0 0 0 0 0	6	0 1 7 2 3 4 5
6	çıkan kenar yok	aynı	boş	0 1 7 2 3 4 5 6

Tablo 4.1.16: `kahn()` izlemesi. `giris` dizisi 0'dan 7'ye indislidir.



Şekil 4.1.21: Solda 0 çıkarılmış, kenarları silinmiş sayılmıştır; derecesi 0'a inen 1 ve 7 kuyruğa girmiştir. Sağda 4 çıkarılmıştır; 6'nın derecesi 1'e inmiştir, ama 5 → 6 kenarı hâlâ beklediği için 6 kuyruğa giremez.

Çıktı `0 1 7 2 3 4 5 6`'dır. 4 çıkarıldığında 6'nın derecesi yalnız 1'e iner: 6'nın iki ön koşulundan biri (4) tamamlanmış, öteki (5) hâlâ bekliyor. 6 ancak 5 çıkarıldıktan sonra kuyruğa girer. Kahn algoritmasının özü budur: bir düğüm, bütün ön koşulları yazılmadan yazılmaz.

Kahn algoritması aynı zamanda bir **döngü testidir**. Çizgede döngü varsa döngüdeki düğümlerin hiçbirinin derecesi 0'a inmez, çünkü her biri döngüdeki bir önceki düğümü bekler. Kuyruk, bütün düğümler yazılmadan boşalır. Bu yüzden yazılan düğüm sayısı V'den azsa çizgede döngü vardır. Örnek çizgeye $2 \to 0$ kenarı geri eklenirse 0'ın giriş derecesi 1 olur ve başlangıçta derecesi 0 olan hiçbir düğüm kalmaz. Kuyruk boş başlar, hiçbir düğüm yazılmaz ve algoritma döngüyü bildirir.

Algoritmanın zamanı $O(V + E)$'dir: her düğüm kuyruğa bir kez girip çıkar, her kenar bir kez sayılır ve bir kez azaltılır. Kuyruk yerine başka bir yapı da kullanılabilir. Örneğin hazır düğümler bir öncelikli kuyrukta tutulursa her adımda numarası en küçük hazır düğüm seçilir. Bu yolla sözlük sırasına göre en küçük topolojik sıra elde edilir; zaman ise $O((V + E) \log V)$'ye çıkar.

4.1.6.3 DFS ile topolojik sıralama

DFS tabanlı yöntemde düğümler, işleri bittiği anda bir yığına itilir. Gezinme bitince yığından çekilen sıra bir topolojik sıradır. Bu, Kosaraju algoritmasının birinci turuyla aynı iştir:

```
void dfsBitis(int v) {
  ziyaret[v] = true;
  for (int w : komsu[v])
    if (!ziyaret[w]) dfsBitis(w);
  yigin.push(v);          // bitince it
}
```

Ana program ziyaret edilmemiş her düğümden `dfsBitis` çağırır, sonra yığını boşaltarak düğümleri yazar. Şekil 4.1.20'deki çizgede `dfsBitis(0)` önce 1'e, oradan 2'ye, 3'e, 5'e ve 6'ya iner. 6'nın çıkan kenarı yoktur; ilk biten odur. Ardından 5 ve 3 biter. 2'nin sıradaki komşusu 4'tür; 4'ün komşusu 6 zaten bitmiştir ve 4 biter. Sonra 2, 1, 7 ve 0 biter. Bitiş sırası 6, 5, 3, 4, 2, 1, 7, 0'dır; yığından çekilen sıra bunun tersi olan `0 7 1 2 4 3 5 6`'dır.

Teorem 4.1.4 (Ters bitiş sırası)

Döngüsüz yönlü bir çizgede DFS yapılırsa her $u \to v$ kenarı için v, u'dan önce biter. Dolayısıyla bitiş sırasının tersi bir topolojik sıradır.

İspat. $u \to v$ kenarı, u'nun komşu listesinde gezinilirken incelenir. O anda v'nin üç durumu olabilir. v beyazsa (keşfedilmemişse) u'dan ona inilir ve v'nin çağrısı u'nunkinden önce döner. v koyuysa (bitmişse) zaten u'dan önce bitmiştir. v sarı olamaz: sarı olsaydı v çağrı yığınında, yani u'nun bir atası olurdu; v'den u'ya ağaç yolu ve $u \to v$ kenarı bir döngü oluştururdu. Çizge döngüsüz olduğu için bu olanaksızdır. Her durumda v, u'dan önce biter ve ters sırada u önce gelir. ■

İspattaki "sarı olamaz" adımı aynı zamanda döngü testini verir: DFS sırasında sarı bir düğüme giden kenar bulunursa çizge döngülüdür ve topolojik sıra yoktur. DFS yöntemi de *O(V + E)* zamanda çalışır. Kahn yöntemi sırayı baştan sona üretir. Hazır işleri kuyrukta tuttuğu için paralel planlamaya da uygundur. DFS yöntemi ise mevcut bir DFS koduna tek satır eklenerek elde edilir.

4.1.6.4 Bağlantılı bileşenler

Yönsüz bir çizgenin kaç bağlantılı bileşeni olduğu, gezinmenin kaç kez başlatıldığı sayılarak bulunur. Ziyaret edilmemiş her düğümden yeni bir DFS (ya da BFS) başlatılır; her başlatma yeni bir bileşeni bütünüyle gezer.

```
int bilesenSayisi() {
  boolean[] ziyaret = new boolean[n];
  int sayac = 0;
  for (int v = 0; v < n; v++)
    if (!ziyaret[v]) {   // yeni bileşen
      dfs(v, ziyaret);
      sayac++;
    }
  return sayac;
}
```

Bütün başlatmalar aynı `ziyaret` dizisini paylaştığı için her düğüm yalnız bir kez ziyaret edilir ve toplam zaman yine *O(V + E)*'dir. Dokuz düğümlü ve 0–1, 1–2, 3–4, 5–6, 6–7, 7–5 kenarlı bir çizgede metot 4 döndürür: bileşenler *{0, 1, 2}*, *{3, 4}*, *{5, 6, 7}* ve tek başına kalan *{8}*'dir. Her başlatmada ziyaret edilen düğümlere bir bileşen numarası yazılırsa "iki düğüm aynı bileşende mi?" sorusu sonradan *O(1)* zamanda yanıtlanır.

Aynı sayım yönlü çizgede bileşen sayısını vermez. Yönlü çizgede bir DFS'in ziyaret ettiği küme, başlangıç düğümünün seçimine bağlıdır. Kenarları *0→1*, *2→0*, *3→2*, *5→4* olan altı düğümlü bir çizge alınsın. Düğümler 0'dan 5'e doğru denenirse DFS beş kez başlatılır: 0'dan yalnız 1'e ulaşılır, 2, 3, 4 ve 5 ayrı ayrı başlatılır. Düğümler 5'ten 0'a doğru denenirse iki başlatma yeter: 5'ten 4'e, 3'ten 2, 0 ve 1'e ulaşılır. Oysa bu çizgenin kenar yönleri göz ardı edildiğinde 2 bileşeni, güçlü bağlantılı bileşen olarak ise 6 bileşeni vardır. Yönlü çizgede doğru araç Tarjan ya da Kosaraju algoritmasıdır.

4.1.6.5 İki parçalılık testi

Bir çizgenin iki parçalı olup olmadığını sınamak, düğümleri iki renge, komşu düğümler farklı renkte olacak biçimde boyamaya çalışmaktır. BFS bu boyamayı doğal olarak yapar: başlangıç bir renge, onun komşuları öteki renge, onların yeni komşuları yine ilk renge boyanır. Başka bir deyişle çift katmanlar bir renk, tek katmanlar öteki renk alır.

Teorem 4.1.5 (İki parçalılık)

Bir çizge ancak ve ancak tek uzunlukta bir döngü içermiyorsa iki parçalıdır.

İspat. İki parçalı bir çizgede her kenar renk değiştirir. Bir döngü başladığı düğüme, yani başladığı renge döner; bu yüzden döngüde çift sayıda renk değişimi, yani çift sayıda kenar vardır. Ters yönde, tek uzunlukta döngü içermeyen bağlı bir çizgede BFS katmanları kullanılır. Aynı katmandaki iki düğüm u ile w arasında bir kenar olsaydı, s'den u'ya ve w'ye giden BFS ağacı yolları ayrıldıkları ortak atadan itibaren eşit uzunlukta olurdu. Bu iki yol ve u–w kenarı birlikte tek uzunlukta bir döngü oluştururdu. Öyleyse her kenar ardışık iki katmanı bağlar ve çift/tek katman boyaması geçerlidir. Bağlı olmayan çizgede aynı akıl yürütme her bileşene ayrı ayrı uygulanır. ■

```
boolean ikiParcaliMi(int s) {
  Arrays.fill(renk, -1);
  renk[s] = 0; kuyruk.add(s);
  while (!kuyruk.isEmpty()) {
    int v = kuyruk.poll();
    for (int w : komsu[v])
      if (renk[w] == -1) {
        renk[w] = 1 - renk[v];
        kuyruk.add(w);
      } else if (renk[w] == renk[v])
        return false;  // aynı renk
  }
  return true;
}
```

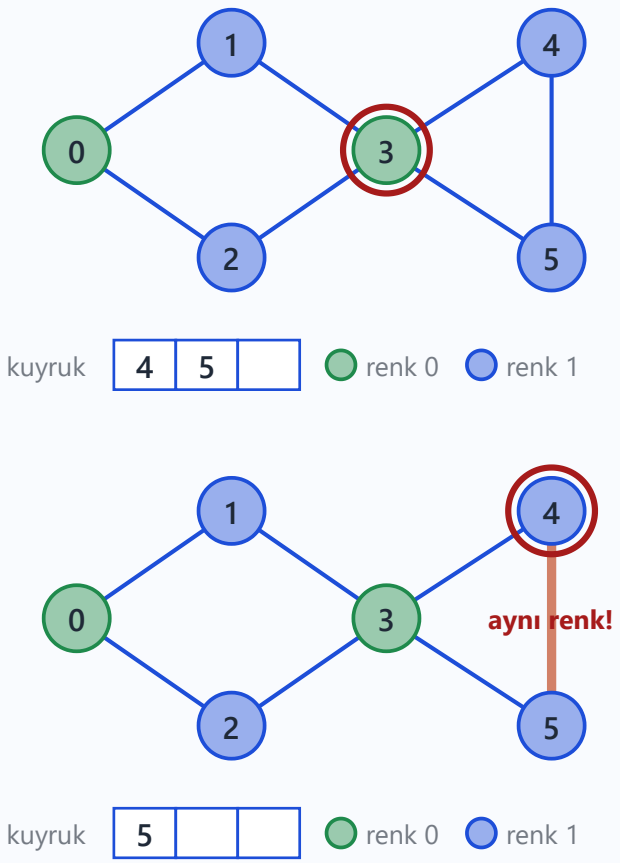
`renk[v] == -1` boyanmamış düğümdür. Boyanmamış komşu, `1 - renk[v]` ile karşı renge boyanır ve kuyruğa eklenir; renk dizisi burada ziyaret dizisinin işini de görür. Boyanmış bir komşu v ile aynı renkteyse bir kenarın iki ucu aynı kümeye düşmüştür ve metot hemen `false` döndürür. Kuyruk sorunsuz boşalırsa boyama geçerlidir ve `true` döner. Metot yalnız s'nin bileşenini sınar; bütün çizge için her boyanmamış düğümden yeniden başlatılır.

İzlemenin çizgesi altı düğümlü ve yönsüzdür: kenarlar 0–1, 0–2, 1–3, 2–3, 3–4, 3–5, 4–5. Çizgede 0–1–3–2 döngüsü (dört kenar) ve 3–4–5 üçgeni (üç kenar) vardır.

Alınan (renk)	Komşular	`renk` (sonra)	Kuyruk (sonra)
—	0, renk 0 ile kuyruğa girer	0 −1 −1 −1 −1 −1	0
0 (0)	1, 2 boyanmamış → renk 1	0 1 1 −1 −1 −1	1, 2
1 (1)	0 karşı renkte; 3 → renk 0	0 1 1 0 −1 −1	2, 3
2 (1)	0 ve 3 karşı renkte	aynı	3
3 (0)	1, 2 karşı renkte; 4, 5 → renk 1	0 1 1 0 1 1	4, 5

Alınan (renk)	Komşular	`renk` (sonra)	Kuyruk (sonra)
4 (1)	3 karşı renkte; **5 aynı renkte** → `false`	0 1 1 0 1 1	5

Tablo 4.1.17: `ikiParcaliMi(0)` izlemesi. `renk` dizisi 0'dan 5'e indislidir.



Şekil 4.1.22: Solda 3 çıkarılmış, komşuları 4 ve 5 karşı renge boyanmıştır. Sağda 4 çıkarılmıştır: komşusu 5 de aynı renktedir. 4–5 kenarının iki ucu aynı kümeye düşmüştür; 3–4–5 üçgeni tek uzunlukta bir döngüdür.

0–1–3–2 döngüsü dört kenarlıdır ve boyamayı bozmaz: 0 ile 3 aynı renk, 1 ile 2 öteki renk alır. Sorun 3–4–5 üçgenindedir; üç kenarlık bir döngü iki renkle boyanamaz. 4–5 kenarı silinseydi metot `true` döndürürdü. Test bir BFS olduğu için *O(V + E)* zamanda çalışır.

Örnek 4.1.6

A–B, A–C, B–D, C–D, C–E, D–F, E–F, F–G kenarlı çizge iki parçalı mıdır?

Çözüm: A'dan BFS uzaklıkları A 0, B 1, C 1, D 2, E 2, F 3, G 4'tür. Çift uzaklıktakiler *{A, D, E, G}*, tek uzaklıktakiler *{B, C, F}*'dir. Sekiz kenarın her biri bu iki kümeyi bağlar; örneğin C–D kenarı 1. ve 2. katmanları, E–F kenarı 2. ve 3. katmanları bağlar. Aynı katmanda iki komşu yoktur, çizge iki parçalıdır. Çizgedeki döngüler dört kenarlı A–B–D–C–A ve C–D–F–E–C ile altı kenarlı A–B–D–F–E–C–A'dır; hepsi çift uzunluktadır.

4.1.6.6 Döngü, köprü ve eklem noktası

Döngü tespiti iki çizge türünde farklı yapılır. Yönsüz çizgede DFS sırasında ziyaret edilmiş bir komşuya rastlanırsa ve bu komşu düğümün ebeveyni değilse bir döngü vardır. Ebeveyn istisnası gereklidir, çünkü yönsüz bir kenar iki yönde de listelenir ve çocuktan ebeveyne dönen kenar bir döngü değildir. Yönlü çizgede ise yalnız çağrı yığınındaki, yani sarı bir düğüme giden kenar döngü gösterir; işi bitmiş bir düğüme giden kenar döngü göstermez. DFS izlemesinde *2 → 0* kenarı sarı 0'a gidiyordu ve döngü belirtisiydi; *4 → 6* kenarı ise bitmiş 6'ya gidiyordu ve değildi.

Yönsüz bağlı bir çizgede silindiğinde çizgeyi ikiye bölen kenara **köprü** (*bridge*), silindiğinde çizgeyi bölen düğüme **eklem noktası** (*articulation point*) denir. Bir ağdaki köprüler ve eklem noktaları, tek bir arızanın ağı bölebileceği zayıf yerlerdir. İkisi de Tarjan'ın enDüşükBağ fikriyle tek bir DFS'te bulunur. Yönsüz çizgede *keşif(v)*, *v*'nin keşif sırası; *enDüşükBağ(w)* ise *w*'nin alt ağacından tek bir geri kenarla ulaşılabilen en küçük keşif değeridir. DFS ağacındaki *v*–*w* kenarı (*w* çocuk) için:

- *enDüşükBağ(w) > keşif(v)* ise *v*–*w* bir köprüdür: *w*'nin alt ağacından *v*'ye ya da *v*'nin atalarına dönen bir geri kenar yoktur ve kenar silinince alt ağaç kopar.
- *enDüşükBağ(w) ≥ keşif(v)* ise *v* bir eklem noktasıdır (kök hariç; kök için iki ya da daha çok çocuğu olması koşulu aranır). Bu durumda alt ağaç en fazla *v*'ye dönebilir; *v* silinince yine kopar.

A–G harfli örnek çizgede tek köprü F–G'dir. DFS sırası A, B, D, C, E, F, G olduğundan keşif değerleri A 1, B 2, D 3, C 4, E 5, F 6, G 7'dir. G'nin tek komşusu F olduğu için *enDüşükBağ(G) = 7 > keşif(F) = 6* olur ve F–G köprüdür. Öteki her kenar bir döngü üzerindedir; örneğin E–F kenarı, C–D–F–E döngüsünün parçasıdır ve silinse bile E'den F'ye C ve D üzerinden gidilebilir. F de bir eklem noktasıdır: silinirse G yalnız kalır.

4.1.6.7 Yinelemeli derinleştirme ve rastgele yürüyüş

Yinelemeli derinleştirme (*iterative deepening DFS*, IDDFS), derinlik sınırı 0, 1, 2, … diye artırılan bir dizi DFS yapar. Her turda DFS, sınırdan daha derine inmez. Hedef ilk kez *d* sınırlı turda bulunursa en sığ çözüm *d* derinliktedir. Yöntem DFS'in az bellek kullanmasını BFS'in en sığ çözümü bulma güvencesiyle birleştirir. İlk bakışta üst katmanların her turda yeniden gezilmesi israf gibi görünür. Ama düğümlerin büyük çoğunluğu en derin katmandadır. Her düğümün 10 çocuğu olduğu bir ağaçta 5 derinliğe kadar BFS 111.111 düğüm ziyaret eder. IDDFS ise toplam 123.456 ziyaret yapar; fark yalnız yüzde on bir kadardır. Satranç ve bulmaca çözücülerinin taradığı devasa örtük çizgelerde bu yöntem yaygındır.

Rastgele yürüyüş (*random walk*), her adımda bulunulan düğümün komşularından birine rastgele geçer. Bir gezinme değildir, çünkü bütün düğümleri ziyaret etme güvencesi yoktur; ama çok büyük ağlarda örnekleme yapmanın ucuz bir yoludur. Bir web sayfasının PageRank değeri, web çizgesinde uzun süre rastgele yürüyen birinin o sayfada bulunma olasılığına dayanır.

4.1.7 Sık yapılan hatalar

Sık yapılan hata

Ziyaret dizisini unutmak. Döngülü bir çizgede ziyaret dizisi olmadan DFS sonsuz özyinelemeye, BFS sonsuz kuyruğa girer. "BFS döngülü çizgede çalışmaz" yanılgısının kaynağı budur. İşaretleme yapıldığında iki algoritma da döngülü çizgede sorunsuz çalışır.

Sık yapılan hata

BFS'te çıkarırken işaretlemek. Aynı düğüm birçok komşudan kuyruğa eklenir. Sonuç doğru kalabilir, ama kuyruk *O(E)*'ye büyür. Doğrusu düğümü kuyruğa eklerken işaretlemektir.

Sık yapılan hata

DFS'ten en kısa yol beklemek. DFS'in hedefe ilk ulaştığı yol en kısa yol değildir. Ağırlıksız çizgede BFS, ağırlıklı çizgede Dijkstra algoritması gerekir. BFS de ağırlıklı çizgede en kısa yolu bulmaz; yalnız en az kenarlı yolu bulur.

Sık yapılan hata

Yığınla DFS'in sırasını özyinelemeli sürümle aynı sanmak. Komşular listedeki sırayla itilince ziyaret sırası tersine döner: yönlü örnekte yığınla DFS `0 7 2 …`, özyinelemeli DFS `0 1 2 …` yazar. Özyinelemeli sıra isteniyorsa komşular ters sırada itilir.

Sık yapılan hata

Tarjan'da "yığında mı?" denetimini atlamak. `min(…, indeks(w))` güncellemesi yalnız yığındaki *w* için yapılır; bileşeni bulunmuş *w* yok sayılır. Aksi hâlde kökler kaybolur ve bazı bileşenler hiç çıkarılmaz.

Sık yapılan hata

Bağlı olmayan çizgeyi tek kaynaktan gezmek. Tek bir kaynaktan yapılan gezinme yalnız o kaynaktan ulaşılabilen düğümleri görür. Bütün çizge için gezinme, ziyaret edilmemiş her düğümden yeniden başlatılır. Aynı nedenle BFS'ten sonra yol kurulurken ulaşılamayan hedefler için `ata[]` 'nın -1 taşıması gerekir.

Sık yapılan hata

Gösterimi gözden kaçırmak. DFS ve BFS'in $O(V + E)$ zamanı komşuluk listesi içindir. Aynı kod komşuluk matrisiyle yazılırsa her düğümün satırı baştan sona taranır ve zaman $O(V^2)$ olur. Seyrek bir çizgede bu fark yüz kattan fazla olabilir.

4.1.8 Özet

- **Çizge:** $G = (V, E)$, düğümler ve onları bağlayan kenarlar. Yönlü, yönsüz, ağırlıklı, iki parçalı, döngülü, tam çizge ve ağaç başlıca türlerdir. n düğümlü tam çizgede $n(n-1)/2$ kenar vardır; yönsüz çizgede derecelerin toplamı $2E$'dir.
- **Gösterim:** komşuluk listesi $O(V + E)$, komşuluk matrisi $O(V^2)$ bellek ister. Gezinme listeyle $O(V + E)$, matrisle $O(V^2)$ zaman alır. Seyrek çizgelerde liste tercih edilir.
- **Gezinme:** ulaşılabilen her düğümü tam bir kez ziyaret etmek. Döngüler yüzünden ziyaret dizisi şarttır. Bekleme listesi yığınsa DFS, kuyruksa BFS elde edilir.
- **DFS:** bir dalı sonuna kadar izler, sonra geri döner. Özyinelemeli sürümde geri dönüşü çağrı yığını yapar. Ziyaret sırası komşu sırasına bağlıdır; bir düğüme ilk ulaşılan kenarlar DFS ağacını oluşturur.
- **Yığınla DFS:** çekerken işaretler; bir düğüm yığına birden çok kez girebilir ve yığın $O(E)$'ye büyüyebilir. Komşular listedeki sırayla itilirse sıra özyinelemeli sürümün tersine döner. Derin çizgelerde `StackOverflowError` riskini ortadan kaldırır.
- **BFS:** kuyrukla katman katman ilerler; düğümü kuyruğa eklerken işaretler. `uzaklik[]` ve `ata[]` dizileriyle ağırlıksız çizgede en kısa yolları bulur; yol, ebeveynler hedeften geriye izlenerek kurulur. Ağırlıklı çizgede en kısa yolu bulmaz.
- **Labirent:** boş hücreler düğüm, bitişik boş hücreler kenardır. BFS bütün yolları eşzamanlı dener ve çıkışa ilk ulaştığında bulduğu yol en kısadır.
- **Güçlü bağlantılı bileşen:** karşılıklı ulaşılabilir düğümlerin maksimal kümesidir. Bileşenler düğümleri ayrık parçalara böler; bileşen çizgesi döngüsüzdür.
- **Tarjan:** indeks keşif sırasıdır; enDüşükBağ alt ağaçtan yığındaki düğümlere ulaşarak erişilen en küçük indekstir. $enDüşükBağ(v) = indeks(v)$ ise v köktür ve yığından v'ye kadar çekilenler bir bileşendir. Bileşeni bulunmuş düğüme giden kenar yok sayılır. Zaman $O(V + E)$'dir; bileşenler ters topolojik sırada çıkar.

- **Kosaraju:** önce bitiş sırasını yığına kaydeden bir DFS, sonra ters çizgede yığın sırasıyla ikinci bir DFS. İkinci turdaki her DFS bir bileşen bulur. Zaman yine *O(V + E)*'dir.
- **Uygulamalar:** topolojik sıra (Kahn ya da ters bitiş sırası; yazılan düğüm sayısı *V*'den azsa döngü vardır), bağlantılı bileşen sayısı, BFS ile iki renge boyayarak iki parçalılık testi, döngü tespiti, köprü ve eklem noktası, yinelemeli derinleştirme.

4.1.9 Kavram listesi

çizge *graph* · düğüm *vertex / node* · kenar *edge* · yönlü çizge *directed graph* · ağırlıklı çizge *weighted graph* · iki parçalı çizge *bipartite graph* · komşuluk listesi *adjacency list* · komşuluk matrisi *adjacency matrix* · çizge gezinme *graph traversal* · derinlik öncelikli arama *depth-first search (DFS)* · genişlik öncelikli arama *breadth-first search (BFS)* · sözde kod *pseudocode* · geri izleme *backtracking* · geri kenar *back edge* · ata *ancestor* · ebeveyn *parent* · en kısa yol *shortest path* · bileşen *connected component* · güçlü bağlantılı bileşen *strongly connected component (SCC)* · topolojik sıralama *topological sort*

4.1.10 Kendini Dene

1. Yönsüz bir çizgenin kenarları 0–1, 0–4, 1–2, 1–4, 2–3, 3–4, 3–5'tir; komşu listeleri artan sıradadır. `dfs(0)` ve `bfs(0)` ne yazdırır? BFS'e göre 0'dan 5'e en kısa yol hangisidir?

   Cevap

   Komşu listeleri 0: [1, 4], 1: [0, 2, 4], 2: [1, 3], 3: [2, 4, 5], 4: [0, 1, 3], 5: [3]'tür. DFS 0'dan 1'e, 2'ye, 3'e iner; 3'ün komşusu 2 işaretli olduğu için 4'e geçer. 4'ün bütün komşuları işaretlidir, 3'e dönülür ve 5 ziyaret edilir: `0 1 2 3 4 5`. BFS önce 1 ve 4'ü, sonra 1'den 2'yi, 4'ten 3'ü, en son 3'ten 5'i keşfeder: `0 1 4 2 3 5`. Uzaklıklar 0, 1, 2, 2, 1, 3'tür. 3'ü 2 değil 4 keşfeder, çünkü 4 kuyruktan 2'den önce çıkar. Ebeveynler izlenince en kısa yol 0–4–3–5 bulunur (3 kenar).

2. Yönlü çizge *0→1*, *0→2*, *1→3*, *2→3*, *3→4*, *2→4* verilsin; komşu listeleri bu kenar sırasıyla dolar. `dfsYigin(0)` ne yazdırır ve kaç itme yapar? Özyinelemeli `dfs(0)` ne yazdırır?

   Cevap

   `komsu[0] = [1, 2]`, `komsu[2] = [3, 4]`'tür. Yığınla DFS 0'dan sonra en son itilen 2'yi çeker; 2'den sonra en son itilen 4'ü, sonra 3'ü (3'ün komşusu 4 itilir, ama işaretlidir), en son 1'i yazar: `0 2 4 3 1`. Toplam itme *1 + E = 7*'dir. Özyinelemeli DFS `0 1 3 4 2` yazar. Yığınlı sürümde komşular ters sırada itilirse o da `0 1 3 4 2` yazar.

3. Yedi düğümlü yönsüz çizgenin kenarları 0–1, 0–2, 1–3, 2–3, 3–4, 5–6'dır. `bfsUzaklik(0)`'dan sonra `uzaklik[]` ne olur? `yol(ata, 0, 4)` ve `yol(ata, 0, 6)` ne döndürür?

   Cevap

   `uzaklik = [0, 1, 1, 2, 3, -1, -1]`; 5 ve 6 ulaşılamaz. `ata[4] = 3`, `ata[3] = 1`, `ata[1] = 0` olduğundan `yol(ata, 0, 4)` `[0, 1, 3, 4]` döndürür. `ata[6] = -1` olduğundan döngü yalnız 6'yı ekler; listenin ilk öğesi 0 değildir ve metot `null` döndürür. `ata[]` *−1* ile doldurulmasaydı `ata[6] = 0` kalır ve metot var olmayan `[0, 6]` yolunu döndürürdü.

4. Yönlü çizge *0→1*, *1→2*, *2→0*, *2→3*, *3→4*, *4→5*, *5→3*, *4→6* verilsin (`komsu[2] = [0, 3]`, `komsu[4] = [5, 6]`). `scc(0)` hangi satırları yazdırır? Her düğümün son `indeks/dusuk` değeri nedir?

   Cevap

   0, 1, 2, 3, 4, 5, 6 sırayla keşfedilir; indeksleri 1'den 7'ye kadardır. 2 → 0 kenarı `dusuk[2]`'yi 1'e, 5 → 3 kenarı `dusuk[5]`'i 4'e indirir. 6'nın çıkan kenarı yoktur ve ilk kök odur. Sonra 3 kök olur, en son 0. Çıktı üç satırdır: `6`, `5 4 3`, `2 1 0`. Son değerler 0: 1/1, 1: 2/1, 2: 3/1, 3: 4/4, 4: 5/4, 5: 6/4, 6: 7/7'dir.

5. DAG *0→2*, *1→2*, *1→3*, *2→4*, *3→4*, *4→5*, *3→5* için Kahn algoritması ve ters bitiş sırası hangi sırayı verir? Çizgeye *5→1* kenarı eklenirse Kahn ne yazar?

   Cevap

   Başta 0 ve 1'in giriş derecesi 0'dır. Kahn `0 1 2 3 4 5` yazar: 0'dan sonra 2'nin derecesi 1'e iner, 1'den sonra 2 ve 3 hazır olur, 3'ten sonra 4, 4'ten sonra 5 hazır olur. DFS'te bitiş sırası 5, 4, 2, 0, 3, 1'dir; tersi `1 3 0 2 4 5`'tir. *5→1* eklenince 1'in derecesi 1 olur. Kuyrukta yalnız 0 vardır; 0 yazılınca 2'nin derecesi 1'e iner ve kuyruk boşalır. Yalnız `0` yazılır; 6'dan az düğüm yazıldığı için çizgede döngü vardır (*1 → 3 → 5 → 1*).

6. Beş düğümlü bir halka (0–1–2–3–4–0) ile altı düğümlü bir halka (0–1–2–3–4–5–0) iki parçalı mıdır? `ikiParcaliMi(0)` ilkinde hangi kenarda `false` döndürür?

   Cevap

   Altı düğümlü halka iki parçalıdır: renkler 0, 1, 0, 1, 0, 1 sırasıyla değişir. Beş düğümlü halka tek uzunlukta bir döngüdür ve iki parçalı değildir. BFS 0'ı renk 0'a, 1 ve 4'ü renk 1'e, 2 ve 3'ü renk 0'a boyar. 2 kuyruktan alındığında komşusu 3 de renk 0'dadır: metot 2–3 kenarında `false` döndürür.

7. Beş düğümlü tam çizgede 0'dan BFS yapılıyor. Eklerken işaretleyen doğru sürüm ile çıkarırken işa-

retleyen sürüm kuyruğa kaçar ekleme yapar?

Cevap

Doğru sürüm her düğümü bir kez ekler: 5 ekleme. Çıkarırken işaretleyen sürümde 0 çıkınca 1, 2, 3, 4 eklenir; 1 çıkınca henüz işaretsiz 2, 3, 4 yeniden eklenir; 2 çıkınca 3 ve 4; 3 çıkınca 4. Toplam $1 + 4 + 3 + 2 + 1 = 11$ ekleme yapılır ve kuyruk 7 öğeye ulaşır. Ziyaret sırası yine 0 1 2 3 4'tür, ama iş ve bellek boşa harcanmıştır.

8. $V = 500$ düğümlü, $E = 2000$ kenarlı yönsüz bir çizgede komşuluk listelerindeki toplam öğe sayısı ve komşuluk matrisinin hücre sayısı nedir? BFS'in yaklaşık iş miktarı iki gösterimde kaç kat farklıdır?

Cevap

Listelerde $2E = 4000$ öğe, matriste $V^2 = 250.000$ hücre vardır. BFS listeyle yaklaşık $V + 2E = 4500$, matrisle $V^2 = 250.000$ adım yapar; oran yaklaşık 56'dır.

4.1.11 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 20: BFS ve DFS'i beyaz–gri–siyah renklendirmeyle, keşif ve bitiş zamanlarıyla, kenar sınıflandırmasıyla ve doğruluk ispatlarıyla verir; topolojik sıralama ve Kosaraju algoritması da buradadır.
- R. Sedgewick, K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 4.1 ve 4.2: yönsüz ve yönlü çizgelerde gezinmeyi Java koduyla anlatır; labirent, bileşenler ve Kosaraju–Sharir algoritması için uygulamalı örnekler içerir.
- J. Kleinberg, É. Tardos, *Algorithm Design* (2006), Bölüm 3: BFS katmanlarını, iki parçalılık testini ve DAG'larda topolojik sıralamayı kısa ve okunaklı ispatlarla ele alır.
- R. E. Tarjan, “Depth-First Search and Linear Graph Algorithms” (1972): DFS'in doğrusal zamanlı çizge algoritmalarının temeli olduğunu gösteren ve enDüşükBağ fikrini tanıtan özgün makale.

4.1.12 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 20: Elementary Graph Algorithms).
2. A. B. Kahn, “Topological Sorting of Large Networks”, *Communications of the ACM* 5(11), 1962.
3. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2006 (Bölüm 3: Graphs).
4. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 4.1–4.2).
5. R. E. Tarjan, “Depth-First Search and Linear Graph Algorithms”, *SIAM Journal on Computing* 1(2), 1972.

KISIM II · BÖLÜM 4.2

En Kısa Yol Algoritmaları

Bir haritada iki nokta arasındaki en kısa güzergâhı bulmak, bir ağda paketin en ucuz yoldan gitmesini sağlamak ya da bir oyunda karakteri engellerin çevresinden hedefe ulaştırmak aynı problemin farklı yüzleridir. Hepsi ağırlıklı bir çizgede en kısa yolu arar. Not, problemi ve bütün algoritmaların ortak işlemi olan gevşetmeyi tanımlayarak başlar. Ardından dört algoritmayı aynı sırayla ele alır. Dijkstra algoritması negatif olmayan ağırlıklarda en yakın düğümü kesinleştirerek ilerler. Bellman-Ford algoritması bütün kenarları tekrar tekrar gevşetir; negatif kenarı kaldırır ve negatif döngüyü yakalar. Floyd-Warshall algoritması bütün düğüm çiftleri arasındaki uzaklıkları üç iç içe döngüyle tek seferde hesaplar. A* araması ise hedefe olan uzaklığı tahmin ederek aramayı hedefe yöneltir. Her algoritma örnek çizgeler üzerinde adım adım çalıştırılır ve Java kodu satır satır izlenir. Not, ağırlıksız çizgelerde genişlik öncelikli aramayla ve algoritmaların karşılaştırmasıyla kapanır.

ÖĞRENME ÇIKTILARI

Kavramlar
Tek kaynaklı, tüm çiftler ve tek çift en kısa yol problemlerini ayırt etmek; gevşetme işlemini, uzaklık tahminini (d) ve önceki düğüm dizisini (π) açıklamak.

Dijkstra ve Bellman-Ford
İki algoritmayı örnek çizgeler üzerinde adım adım çalıştırmak, uzaklık ve önceki tablolarını doldurmak, Java kodlarını izlemek ve karmaşıklıklarını gerekçesiyle vermek.

Negatif kenar ve döngü
Dijkstra algoritmasının negatif kenarda neden yanıldığını bir örnekle göstermek; negatif döngüyü Bellman-Ford ve Floyd-Warshall ile tespit etmek.

Floyd-Warshall ve A*
Tüm çiftler için D ve Π matrislerini ara düğüm sırasıyla güncellemek. A* aramasını $f = g + h$ ile uygulamak, sezgiselin kabul edilebilir ve tutarlı olmasının anlamını açıklamak ve problem türüne göre algoritma seçmek.

4.2.1 En kısa yol problemi

Çizge gezinme algoritmaları bir düğümden hangi düğümlere ulaşılabildiğini söyler. Gerçek uygulamalarda çoğu zaman bu yetmez: ulaşmanın en ucuz yolu da istenir. Kenarlara uzunluk, süre ya da maliyet gibi sayılar eklendiğinde soru "ulaşılır mı?" olmaktan çıkar, "en az kaça ulaşılır?" olur. Bu kesim problemi tanımlar, türlerini ayırır ve bütün algoritmaların üzerine kurulduğu ortak işlemi, gevşetmeyi anlatır.

4.2.1.1 Yol ağı bir çizgedir

Bir şehrin yol haritası doğal bir çizgedir. Kavşaklar **düğüm** (*vertex*), kavşakları bağlayan yollar **kenar** (*edge*) olur. Her yolun bir uzunluğu ya da geçiş süresi vardır; bu sayıya kenarın **ağırlığı** (*weight*) denir. Kenarlarına ağırlık yazılmış çizgeye **ağırlıklı çizge** (*weighted graph*) adı verilir. Tek yönlü sokaklar varsa kenarların yönü de önemlidir; bu durumda çizge yönlüdür.

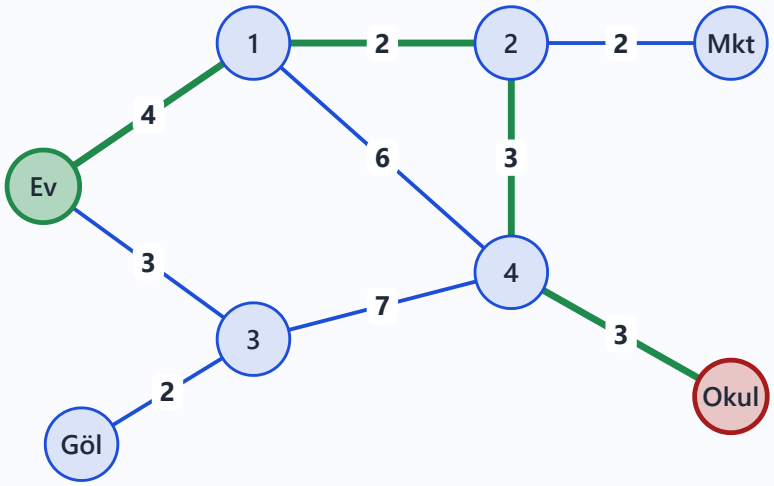
Çizgede bir **yol**, art arda gelen kenarlarla birbirine bağlanan düğümler dizisidir. Yolun uzunluğu, üzerindeki kenar ağırlıklarının toplamıdır. Bu tanımda kenar sayısı değil, ağırlıkların toplamı önemlidir. İki kenarlı bir yol, beş kenarlı bir yoldan daha uzun olabilir.

Tanım 4.2.1

$G = (V, E)$ ağırlıklı bir çizge ve $w(u, v)$, (u, v) kenarının ağırlığı olsun. $p = \langle v_0, v_1, \ldots, v_k \rangle$ yolunun **uzunluğu** $w(p) = w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$ toplamıdır.

s'den v'ye **en kısa yol** (*shortest path*), bu iki düğüm arasındaki yollar içinde uzunluğu en küçük olandır. Bu en küçük uzunluğa s'den v'ye **uzaklık** (*distance*) denir ve $\delta(s, v)$ ile gösterilir. s'den v'ye hiç yol yoksa $\delta(s, v) = \infty$ alınır.

Şekil 4.2.1 küçük bir yol ağını gösterir. Ev'den Okul'a giden birkaç yol vardır. Ev–1–2–4–Okul yolu dört kenarlıdır ve uzunluğu $4 + 2 + 3 + 3 = 12$'dir. Ev–1–4–Okul yolu yalnız üç kenarlıdır, ama 1–4 kenarı 6 birim tuttuğu için uzunluğu $4 + 6 + 3 = 13$ olur. Ev–3–4–Okul yolu da üç kenarlıdır ve o da 13 eder. En kısa yol, en az kenarlı yol değil, dört kenarlı Ev–1–2–4–Okul yoludur: $\delta(\text{Ev}, \text{Okul}) = 12$.



Şekil 4.2.1: Küçük bir yol ağı. Düğümler kavşak, kenarlar yol, sayılar yol uzunluğudur. Yeşil kenarlar Ev'den Okul'a en kısa yolu (12) gösterir. Daha az kenarlı Ev–1–4–Okul yolu 13 tutar.

Örnek, problemin özünü gösterir. Ağırlıksız bir çizgede en kısa yol en az kenarlı yoldur ve genişlik öncelikli arama bunu doğrudan bulur. Ağırlıklı bir çizgede ise kenar saymak yetmez; ağırlıkların toplamı karşılaştırılmalıdır. Bu yüzden ağırlıklı çizgeler için ayrı algoritmalara gerek vardır.

En kısa yol problemi gündelik teknolojinin pek çok yerinde karşımıza çıkar. Navigasyon uygulamaları yol ağında en kısa ya da en hızlı güzergâhı hesaplar. İnternet yönlendiricileri paketleri hangi komşuya ileteceklerine en kısa yol hesabıyla karar verir. OSPF (*Open Shortest Path First*) protokolü Dijkstra algoritmasını, RIP (*Routing Information Protocol*) protokolü ise Bellman-Ford'un dağıtık bir biçimini kullanır. Oyunlarda karakterlerin engellerin çevresinden dolaşması A* aramasıyla yapılır. Lojistikte araçların dağıtım rotaları da en kısa yol hesaplarına dayanır.

> **Sık yapılan hata**
>
> Ağırlıklı bir çizgede en kısa yolu en az kenarlı yol sanmak. Şekil 4.2.1'deki Ev–1–4–Okul yolu üç kenarlıdır, ama uzunluğu 13'tür; dört kenarlı Ev–1–2–4–Okul yolu ise 12 tutar. Kenar sayısı ancak bütün ağırlıklar eşitse belirleyicidir.

4.2.1.2 Problem türleri

"En kısa yolu bul" isteği üç farklı biçimde gelebilir. Hangi biçimin istendiği, kullanılacak algoritmayı da belirler.

- **Tek kaynaklı en kısa yollar** (*single-source shortest paths*): bir *s* kaynağından çizgedeki bütün düğümlere uzaklıklar istenir. Sonuç, her düğüm için bir sayı içeren bir dizidir. Dijkstra, Bellman-Ford ve ağırlıksız çizgede genişlik öncelikli arama bu problemi çözer.

- **Tüm çiftler arası en kısa yollar** (*all-pairs shortest paths*): her *(u, v)* düğüm çifti için uzaklık istenir. Sonuç *n* × *n* boyutlu bir matristir. Floyd-Warshall algoritması bu problem için tasarlanmıştır.
- **Tek çift** (*single-pair*): yalnız bir *s* başlangıcından bir *t* hedefine en kısa yol istenir. Tek kaynaklı bir algoritma bu işi de görür, ama gereğinden fazla düğüm açar. A* araması hedefi bildiği için aramayı ona doğru yöneltir.

Problem türlerinin yanında çizgenin özellikleri de seçimi etkiler. En önemli özellik, negatif ağırlıklı kenar olup olmadığıdır. Ağırlıklar maliyet olarak düşünülünce negatif değer garip görünebilir. Oysa kazanç ya da geri ödeme taşıyan kenarlarda negatif ağırlık doğaldır: bir ulaşım ağında indirim sağlayan aktarma, bir finans çizgesinde kâr getiren işlem böyle kenarlardır. Tablo 4.2.1 bölümdeki algoritmaları ne buldukları ve kısıtlarıyla birlikte listeler.

Algoritma	Ne bulur?	Kısıt / not
Dijkstra	kaynaktan tüm düğümlere	kenar ağırlıkları negatif olmamalı
Bellman-Ford	kaynaktan tüm düğümlere	negatif kenarda çalışır, negatif döngüyü bulur; Dijkstra'dan yavaş
Floyd-Warshall	tüm çiftler arası	negatif kenarda çalışır; negatif döngüyü köşegenden tespit eder
A*	başlangıçtan hedefe	sezgisel tahminle aramayı hızlandırır
BFS	kaynaktan tüm düğümlere	yalnız ağırlıksız (ya da eşit ağırlıklı) çizgeler

Tablo 4.2.1: En kısa yol algoritmalarının çözdüğü problemler ve kısıtları.

4.2.1.3 Temel işlem: gevşetme

Bu bölümdeki algoritmaların hepsi aynı fikirle çalışır. Her düğüm için o ana kadar bulunmuş en iyi yolun uzunluğu bir tahmin olarak tutulur. Algoritma ilerledikçe daha kısa yollar bulunur ve tahminler küçülür. Sonunda her tahmin gerçek uzaklığa eşit olur. Her düğüm için iki bilgi saklanır:

- *d[v]*: kaynaktan *v*'ye şimdiye kadar bulunan en kısa yolun uzunluğu. Başta kaynakta 0, diğer bütün düğümlerde ∞'dur. ∞ "henüz yol bulunmadı" demektir.
- *π[v]*: bu yolda *v*'den hemen önce gelen düğüm. Bu düğüme **önceki** (*predecessor*) düğüm denir. Başta tanımsızdır ve "/" ya da "–" ile gösterilir.

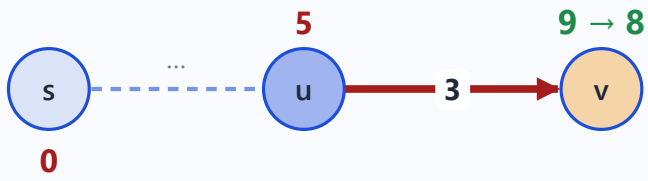
Tahminleri küçülten tek işlem **gevşetmedir**. Bir *(u, v)* kenarı gevşetilirken şu soru sorulur: kaynaktan *u*'ya bilinen yolu alıp sonuna *(u, v)* kenarını eklersek, *v* için bildiğimizden daha kısa bir yol çıkar mı? Çıkarsa *v*'nin tahmini ve önceki düğümü güncellenir.

Tanım 4.2.2

(u, v) kenarının **gevşetilmesi** (*relaxation*): *d[u] + w(u, v) < d[v]* ise *d[v] ← d[u] + w(u, v)* ve *π[v] ← u* yapılır; değilse hiçbir şey değişmez.

```
RELAX(u, v, w):
  eğer d[u] + w(u, v) < d[v]:
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Sözde kod tanımın kendisidir. İkinci satır, *u* üzerinden gelen yolun daha kısa olup olmadığını sorar. Kısaysa üçüncü satır yeni uzunluğu yazar, dördüncü satır da yolun artık *u*'dan geldiğini kaydeder. İşlem bir karşılaştırma ve en çok iki atamadan oluşur; süresi *O(1)*'dir.



Şekil 4.2.2: Bir gevşetme adımı. Kaynaktan *u*'ya bilinen yol 5, *(u, v)* kenarı 3 birimdir. *v*'nin tahmini 9'du; *5 + 3 = 8 < 9* olduğu için tahmin 8'e iner ve *π[v] = u* olur.

Örnek 4.2.1

Şekil 4.2.2'deki durumda *(u, v)* kenarı iki kez üst üste gevşetiliyor. Her gevşetmenin sonucu bulunacak.

1. **Birinci gevşetme.** *d[u] = 5*, *w(u, v) = 3*, *d[v] = 9*. Aday uzunluk *5 + 3 = 8*'dir. *8 < 9* olduğundan *d[v] = 8* ve *π[v] = u* olur.
2. **İkinci gevşetme.** Aday uzunluk yine 8'dir, ama artık *d[v] = 8*'dir. *8 < 8* yanlış olduğundan hiçbir şey değişmez.
3. **Sonuç.** Aynı kenarı tekrar gevşetmek zarar vermez; yalnız boşa bir karşılaştırma yapılır. *d[u]* küçülmedikçe ikinci gevşetme bir şey kazandırmaz.

Gevşetmenin iki önemli özelliği vardır. Birincisi, *d[v]* yalnız azalabilir; hiçbir gevşetme bir tahmini büyütmez. İkincisi, *d[v]* hiçbir zaman gerçek uzaklık *δ(s, v)*'nin altına inmez. Bunun nedeni basittir: *d[v]*'ye yazılan her değer, kaynaktan *v*'ye gerçekten var olan bir yolun uzunluğudur. Hiçbir yol en kısa yoldan kısa olamayacağına göre tahmin de uzaklığın altına düşemez. Tahmin yukarıdan aşağı doğru iner ve doğru değere ulaşınca orada durur.

Algoritmalar arasındaki fark gevşetmede değil, kenarların *hangi sırayla* ve *kaç kez* gevşetildiğindedir. Dijkstra her kenarı bir kez, akıllıca seçilmiş bir sırada gevşetir. Bellman-Ford bütün kenarları sıra gözetmeden tekrar tekrar gevşetir. Floyd-Warshall aynı işlemi düğüm çiftleri üzerinde, ara düğüm ekleyerek yapar. A* ise Dijkstra'nın sırasını hedefe olan tahminle değiştirir.

Sık yapılan hata

Gevşetmede yalnız *d[v]*'yi güncelleyip *π[v]*'yi unutmak. Uzaklıklar doğru çıkar, ama yolun kendisi geri kurulamaz. En kısa yolun yalnız uzunluğu değil, hangi düğümlerden geçtiği de isteniyorsa iki atama birlikte yapılmalıdır.

4.2.1.4 En kısa yolların özellikleri

Algoritmaların neden doğru çalıştığını anlamak için en kısa yolların dört özelliğini bilmek gerekir. Bu özellikler hem algoritmaların tasarımında hem de doğruluk kanıtlarında kullanılır.

Optimal alt yapı. En kısa yolun her parçası da en kısa yoldur. $s \rightsquigarrow u \rightsquigarrow v$ yolu s'den v'ye en kısa yolsa, bu yolun $s \rightsquigarrow u$ parçası da s'den u'ya en kısa yoldur. Kanıt kısa bir çelişkiyle yapılır. s'den u'ya daha kısa bir yol olsaydı, eski parça çıkarılıp yerine bu yol konabilirdi. O zaman v'ye de daha kısa bir yol elde edilirdi; bu, yolun en kısa olmasıyla çelişir. **Optimal alt yapı** (*optimal substructure*), dinamik programlamanın ve açgözlü algoritmaların ortak dayanağıdır. Bellman-Ford ve Floyd-Warshall bu özelliği dinamik programlamayla, Dijkstra açgözlü seçimle kullanır.

Üçgen eşitsizliği. Her (u, v) kenarı için $\delta(s, v) \le \delta(s, u) + w(u, v)$ sağlanır. Eşitsizliğin sağ tarafı, u'ya en kısa yoldan gidip (u, v) kenarını kullanan belirli bir yolun uzunluğudur. En kısa yol bu yoldan uzun olamaz. Gevşetme bu eşitsizliğin tahminler üzerinde bozulduğu kenarları bulup düzeltir: $d[v] > d[u] + w(u, v)$ olan her kenar bir düzeltme fırsatıdır.

Negatif döngü. Toplam ağırlığı negatif olan bir döngüye **negatif döngü** (*negative cycle*) denir. Kaynaktan böyle bir döngüye ulaşılabiliyorsa, döngüde her tur atıldığında yol biraz daha kısalır. Yol istenildiği kadar kısaltılabildiği için en kısa yol tanımsız kalır; uzaklık $-\infty$ olarak yazılır. Bu durum yalnız döngüdeki düğümleri değil, döngüden geçerek ulaşılan bütün düğümleri etkiler.

Basit yol. Negatif döngü yoksa en kısa yol bir döngü içermez. Pozitif ağırlıklı bir döngüyü yoldan çıkarmak yolu kısaltır, sıfır ağırlıklı bir döngüyü çıkarmak ise uzunluğu değiştirmez. Her iki durumda da döngüsüz bir en kısa yol vardır. Döngü içermeyen yol her düğümden en çok bir kez geçer. V düğümlü bir çizgede böyle bir yol en çok V düğüm, dolayısıyla en çok $V - 1$ kenar içerir. Bellman-Ford'un $V - 1$ tur dönmesinin nedeni budur.

Tanım 4.2.3

Kaynaktan ulaşılabilen her $v \neq s$ düğümü için $(\pi[v], v)$ kenarları alınırsa, kökü s olan bir ağaç elde edilir. Bu ağaçta kökten her düğüme giden tek yol, o düğüme bir en kısa yoldur. Bu ağaca **en kısa yol ağacı** (*shortest-path tree*) denir.

En kısa yol ağacı, bütün en kısa yolları tek bir yapıda toplar. Her düğüm yalnız kendi önceki düğümünü sakladığı için yolların hepsi V boyutlu tek bir π dizisine sığar. Bir yolu okumak için hedeften başlanır ve π işaretçileri kaynağa varana kadar izlenir. Bu yolla elde edilen düğümler ters sıradadır; yolu

kaynaktan hedefe yazmak için sıra çevrilir.

Sık yapılan hata

Negatif kenarı negatif döngüyle karıştırmak. Negatif kenar tek başına en kısa yolu tanımsız yapmaz; Bellman-Ford ve Floyd-Warshall böyle çizgelerde doğru sonuç verir. Sorun, toplamı negatif olan bir döngünün kaynaktan ulaşılabilir olmasıdır. Yönsüz bir çizgede ise tek bir negatif kenar bile döngü oluşturur: $u \to v \to u$ yolu iki kat negatif ağırlık taşır.

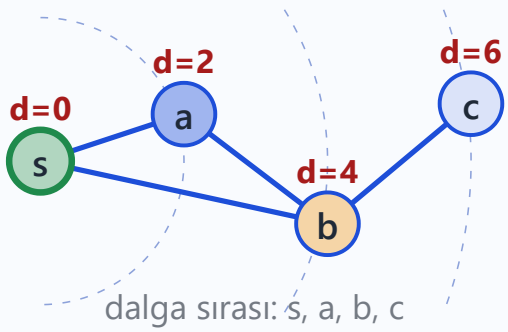
4.2.2 Dijkstra algoritması

Dijkstra algoritması tek kaynaklı en kısa yol probleminin en çok kullanılan çözümüdür. Edsger W. Dijkstra algoritmayı 1956'da geliştirdi ve 1959'da üç sayfalık kısa bir makalede yayımladı. Algoritma, kaynaktan diğer bütün düğümlere en kısa yolları bulur. Tek bir koşulu vardır: kenar ağırlıklarının hiçbiri negatif olmamalıdır. Bu koşul sağlandığında hızlıdır ve büyük yol ağlarında bile pratikte kullanılabilir.

4.2.2.1 Açgözlü seçim ve su dalgası

Algoritmanın fikrini bir benzetme iyi anlatır. Durgun bir göle taş atıldığında dalga her yöne aynı hızla yayılır. Dalga önce taşa en yakın noktaya, sonra bir sonraki en yakın noktaya ulaşır. Kenarları su kanalları, ağırlıkları kanal uzunlukları gibi düşünelim. Kaynaktan çıkan dalga düğümlere tam olarak uzaklık sırasıyla varır. Bir düğüme ilk ulaşan dalga, oraya en kısa yoldan gelmiştir; daha sonra gelen hiçbir dalga daha erken varamaz.

Dijkstra algoritması bu dalganın hareketini adım adım taklit eder. Düğümler iki gruba ayrılır. Uzaklığı kesinleşmiş düğümler S kümesindedir; dalga onlara çoktan ulaşmıştır. Diğer düğümlerin yalnız bir tahmini vardır. Her adımda, kesinleşmemiş düğümler arasından tahmini en küçük olan seçilir ve kesinleştirilir. Sonra bu düğümden çıkan kenarlar gevşetilir; böylece komşuların tahminleri güncellenir.



Şekil 4.2.3: Su dalgası benzetmesi. Kaynak s'den çıkan dalga düğümlere uzaklık sırasıyla ulaşır: s (0), a (2), b (4), c (6). Dijkstra düğümleri bu sırayla kesinleştirir.

Her adımda "şu an en iyi görünen" seçeneği alan ve bu kararı bir daha değiştirmeyen algoritmalara **açgözlü algoritma** (*greedy algorithm*) denir. Dijkstra açgözlüdür: seçilen düğümün uzaklığını kesin kabul eder ve ona bir daha dokunmaz. Açgözlü kararlar çoğu problemde yanlış sonuç verir. Dijkstra'da ise doğrudur, çünkü ağırlıklar negatif değildir. Bu koşulun neden yeterli olduğu ve bozulunca ne olduğu kesimin sonunda gösterilir.

4.2.2.2 Algoritmanın adımları

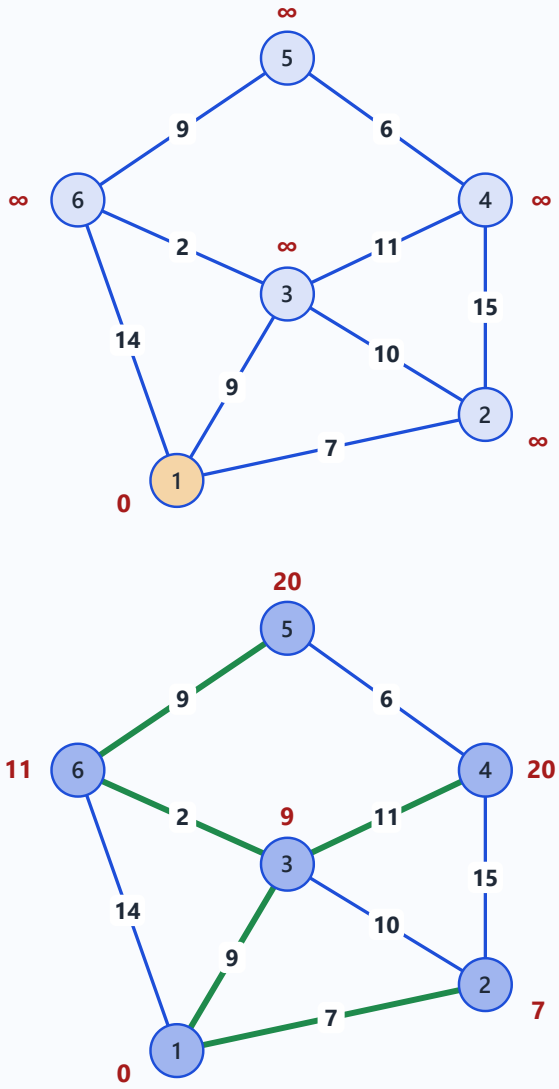
Algoritma beş adımda özetlenir:

1. **İlklendirme.** Kaynağın uzaklığı 0, diğer bütün düğümlerin uzaklığı ∞ yapılır. Önceki düğümler tanımsızdır ve kesinleşmiş düğüm kümesi boştur: $d[s] = 0$, $d[v] = \infty$, $\pi[v]$ = tanımsız, $S = \emptyset$.
2. **Seçim.** Kesinleşmemiş düğümler arasından d değeri en küçük olan u seçilir ve S'ye eklenir. Seçim bir dizi taranarak ya da bir **öncelikli kuyruk** (*priority queue*) ile yapılır.
3. **Aday uzaklıklar.** u'nun henüz kesinleşmemiş her komşusu v için aday uzaklık $d[u] + w(u, v)$ hesaplanır.
4. **Gevşetme.** Aday uzaklık $d[v]$'den küçükse $d[v]$ ve $\pi[v]$ güncellenir.
5. **Tekrar.** Bütün düğümler kesinleşene ya da kalan düğümlerin hepsinin uzaklığı ∞ olana kadar 2.–4. adımlar tekrarlanır. Kaynaktan ulaşılamayan düğümlerin uzaklığı ∞ olarak kalır.

Her düğüm bir kez seçilir ve seçildiği anda çıkan kenarları bir kez gevşetilir. Yönlü bir çizgede her kenar toplamda bir kez, yönsüz bir çizgede iki uçtan birer kez ele alınır. Algoritmanın bütün işi bu seçimler ve gevşetmelerden oluşur.

4.2.2.3 Örnek 1: yönsüz çizge

İlk örnek altı düğümlü yönsüz bir çizgedir; kaynak 1 numaralı düğümdür. Kenarlar ve ağırlıkları şunlardır: 1–2 (7), 1–3 (9), 1–6 (14), 2–3 (10), 2–4 (15), 3–4 (11), 3–6 (2), 4–5 (6), 5–6 (9). Şekil 4.2.4 çizgenin başlangıç ve bitiş durumlarını yan yana gösterir.



Şekil 4.2.4: Dijkstra Örnek 1. Solda başlangıç: kaynak 1'in uzaklığı 0, diğerleri ∞. Sağda bitiş: bütün düğümler kesinleşmiş, kırmızı sayılar uzaklıklar, yeşil kenarlar en kısa yol ağacıdır.

Algoritma altı tur döner; her turda bir düğüm kesinleşir. Tablo 4.2.2 her turdan sonraki uzaklık tahminlerini verir. Kalın değerler o turda değişen tahminlerdir.

Tur	Seçilen (d)	Gevşetmeler	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]
başlangıç	—	—	0	∞	∞	∞	∞	∞
1	1 (0)	2: ∞ → 7; 3: ∞ → 9; 6: ∞ → 14	0	**7**	**9**	∞	∞	**14**
2	2 (7)	3: 17 ≥ 9, değişmez; 4: ∞ → 22	0	7	9	**22**	∞	14
3	3 (9)	4: 22 → 20; 6: 14 → 11	0	7	9	**20**	∞	**11**
4	6 (11)	5: ∞ → 20	0	7	9	20	**20**	11
5	4 (20)	5: 26 ≥ 20, değişmez	0	7	9	20	20	11

Tur	Seçilen (d)	Gevşetmeler	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]
6	5 (20)	kesinleşmemiş komşu yok	0	7	9	20	20	11

Tablo 4.2.2: Dijkstra Örnek 1'in tur tur izlemesi (kaynak 1). Her satır, seçilen düğümün kenarları gevşetildikten sonraki durumdur.

Turların ayrıntısı şöyledir. Birinci turda yalnız kaynağın uzaklığı sonludur; 1 seçilir. 1'in üç komşusu vardır ve hepsinin tahmini ∞'dur. Kenar ağırlıkları doğrudan yeni tahminler olur: $d[2] = 7$, $d[3] = 9$, $d[6] = 14$. Üçünün de önceki düğümü 1'dir.

İkinci turda kesinleşmemiş düğümlerin en küçük tahmini 7'dir; 2 seçilir. 1 zaten kesinleştiği için atlanır. 2–3 kenarı $7 + 10 = 17$ önerir; bu 9'dan büyük olduğu için 3'ün tahmini değişmez. 2–4 kenarı $7 + 15 = 22$ önerir ve 4'e ilk kez bir yol bulunur: $d[4] = 22$, $\pi[4] = 2$.

Üçüncü turda 3 seçilir ($d = 9$). Bu tur örneğin en öğretici adımıdır. 3–4 kenarı $9 + 11 = 20$ önerir ve bu, bir önceki turda bulunan 22'den küçüktür. Böylece 4'ün tahmini 20'ye iner ve önceki düğümü 2'den 3'e geçer. 3–6 kenarı da $9 + 2 = 11$ önerir ve 6'nın 14 olan tahminini düşürür. 1'den 6'ya doğrudan kenar 14 tutar; 3 üzerinden dolaşmak 11 tutar. Doğrudan kenar her zaman en kısa yol değildir.

Dördüncü turda en küçük tahmin 11'dir; 6 seçilir. 6–5 kenarı 5'e ilk yolu verir: $11 + 9 = 20$. Beşinci turda 4 ve 5'in tahminleri eşittir (20). Eşitlikte küçük numaralı düğüm seçilmiştir; 4 kesinleşir. 4–5 kenarı $20 + 6 = 26$ önerir, 20'yi geçemez. Son turda 5 seçilir ve kesinleşmemiş komşusu kalmadığı için algoritma biter.

Sonuçta uzaklıklar 2: 7, 3: 9, 6: 11, 4: 20, 5: 20'dir. Önceki dizisi $\pi[2] = 1$, $\pi[3] = 1$, $\pi[4] = 3$, $\pi[5] = 6$, $\pi[6] = 3$'tür. Bu beş kenar en kısa yol ağacını oluşturur. Örneğin 5'e en kısa yol, işaretçiler geriye izlenerek bulunur: $5 \leftarrow 6 \leftarrow 3 \leftarrow 1$. Yol $1 \to 3 \to 6 \to 5$, uzunluğu $9 + 2 + 9 = 20$'dir. Düğümlerin kesinleşme sırası 1, 2, 3, 6, 4, 5'tir; bu sıra uzaklıkların küçükten büyüğe sırasıyla aynıdır.

Örnek 4.2.2

Aynı çizgede kaynak 5 alınırsa uzaklıklar ne olur?

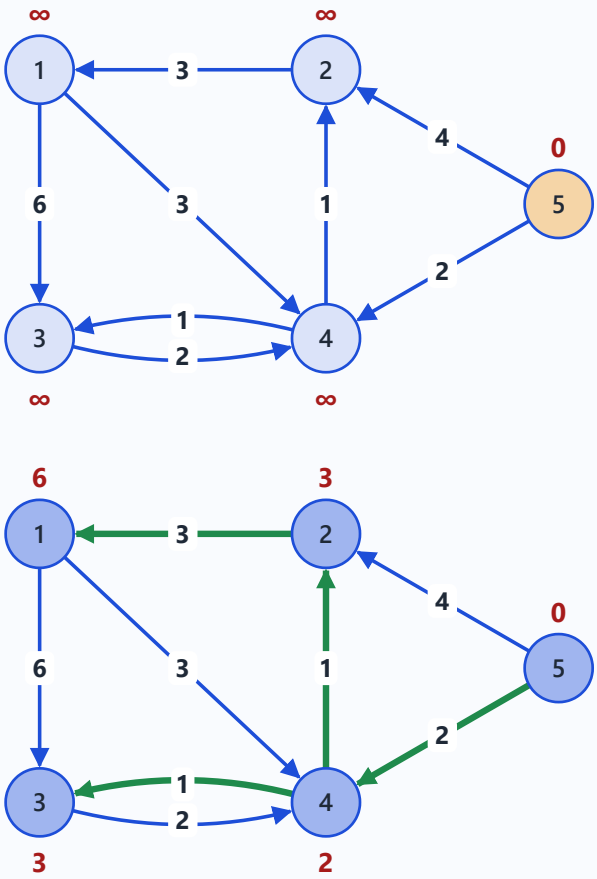
1. **5 seçilir (0).** Komşular: 4: ∞ → 6, 6: ∞ → 9.
2. **4 seçilir (6).** 2: $6 + 15 = 21$, 3: $6 + 11 = 17$.
3. **6 seçilir (9).** 1: $9 + 14 = 23$; 3: $9 + 2 = 11 < 17$, tahmin 11'e iner.
4. **3 seçilir (11).** 1: $11 + 9 = 20 < 23$, tahmin 20'ye iner; 2: $11 + 10 = 21$, eşit olduğu için değişmez.
5. **1 seçilir (20), sonra 2 seçilir (21).** Yeni gevşetme yoktur.

Sonuç: d = 1: 20, 2: 21, 3: 11, 4: 6, 5: 0, 6: 9. Örneğin 3'e yol $5 \to 6 \to 3 = 9 + 2$, 2'ye yol $5 \to 4 \to 2 = 6 + 15$'tir.

4.2.2.4 Örnek 2: yönlü çizge, Q ve S kümeleri

İkinci örnek beş düğümlü yönlü bir çizgedir ve kaynak 5'tir. Kenarlar: *5 → 2* (4), *5 → 4* (2), *4 → 2* (1), *4 → 3* (1), *3 → 4* (2), *2 → 1* (3), *1 → 3* (6), *1 → 4* (3). Yönlü çizgede kenar yalnız ok yönünde kullanılır. Örneğin *2 → 1* kenarı 2'den 1'e gitmeye izin verir, tersine izin vermez. Aynı çizge Bellman-Ford ve Floyd-Warshall örneklerinde de kullanılacak; böylece üç algoritmanın sonuçları karşılaştırılabilir.

Bu kez algoritmanın durumu iki kümeyle izlenir. *Q*, henüz kesinleşmemiş düğümlerin öncelikli kuyruğudur ve düğümleri *d* değerine göre sıralı tutar. *S* ise kesinleşmiş düğümlerin listesidir. Her adımda *Q*'nun başındaki düğüm alınır, *S*'ye eklenir ve kenarları gevşetilir.



Şekil 4.2.5: Dijkstra Örnek 2. Solda yönlü çizge ve başlangıç uzaklıkları (kaynak 5). Sağda son durum: yalnız önceki kenarları (yeşil) kalır ve en kısa yol ağacını oluşturur.

Adım	Q'dan alınan	Gevşetmeler	Q (adımdan sonra)	S
başlangıç	—	—	5 (0), 1 (∞), 2 (∞), 3 (∞), 4 (∞)	∅
1	5, d = 0	(5, 2): ∞ → 4; (5, 4): ∞ → 2	4 (2, π 5), 2 (4, π 5), 1 (∞), 3 (∞)	5
2	4, d = 2	(4, 2): 4 → 3; (4, 3): ∞ → 3	2 (3, π 4), 3 (3, π 4), 1 (∞)	5, 4
3	2, d = 3	(2, 1): ∞ → 6	3 (3, π 4), 1 (6, π 2)	5, 4, 2
4	3, d = 3	(3, 4): 4 zaten S'de	1 (6, π 2)	5, 4, 2, 3

Adım	Q'dan alınan	Gevşetmeler	Q (adımdan sonra)	S
5	1, d = 6	(1, 3), (1, 4): ikisi de S'de	boş	5, 4, 2, 3, 1

Tablo 4.2.3: Dijkstra Örnek 2'de Q ve S kümelerinin değişimi. Q'daki düğümler (d, π) değerleriyle, küçük d başta olacak biçimde yazılmıştır.

İkinci adım dikkat ister. 4 kesinleşince *4 → 2* kenarı *2 + 1 = 3* önerir. 2'nin tahmini birinci adımda doğrudan kenarla 4 olmuştu; şimdi 3'e iner. Kaynaktan 2'ye doğrudan kenar 4 tutar, 4 üzerinden dolaşmak ise 3 tutar. Üçüncü adımda 2 ile 3'ün tahminleri eşittir (3); küçük numaralı 2 önce alınır. Dördüncü ve beşinci adımlarda yeni gevşetme olmaz, çünkü kenarların ucundaki düğümler zaten kesinleşmiştir.

Son durumda uzaklıklar ve önceki düğümler şöyledir:

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Tablo 4.2.4: Dijkstra Örnek 2'nin sonucu (kaynak 5).

Yeşil kenarlar iki yol verir: *5 → 4 → 2 → 1* (uzunluk *2 + 1 + 3 = 6*) ve *5 → 4 → 3* (uzunluk *2 + 1 = 3*). *5 → 2* kenarı en kısa yol ağacına girmez; çünkü 2'ye 4 üzerinden daha kısa bir yol vardır.

4.2.2.5 Sözde kod ve tembel silme

Algoritmanın öncelikli kuyruklu sözde kodu aşağıdadır. Kuyrukta *(düğüm, uzaklık)* çiftleri tutulur; `çıkar()` işlemi uzaklığı en küçük çifti döndürür.

```
DIJKSTRA(G, kaynak):
  mesafeler = [ ]; ziyaretEdildi = [ ]
  öncelikKuyruğu = { }
  her bir v için V içinde:
    mesafeler[v] = sonsuz
    ziyaretEdildi[v] = 0
  mesafeler[kaynak] = 0
  öncelikKuyruğu.ekle(kaynak, 0)
  döngü öncelikKuyruğu boş değil iken:
    u = öncelikKuyruğu.çıkar()   // en küçük mesafe
    eğer ziyaretEdildi[u] == 0:
      ziyaretEdildi[u] = 1
      her bir (u, v) için G[u] içinde:
        eğer mesafeler[u] + ağırlık(u, v) < mesafeler[v]:
          mesafeler[v] = mesafeler[u] + ağırlık(u, v)
          öncelikKuyruğu.ekle(v, mesafeler[v])
  döndür mesafeler
```

İlk altı satır dizileri hazırlar: her düğümün uzaklığı sonsuz, ziyaret bilgisi 0 olur. Kaynağın uzaklığı 0 yapılır ve kaynak kuyruğa eklenir. Ana döngü kuyruk boşalana kadar döner. Her turda en küçük uzaklıklı kayıt çıkarılır. Düğüm daha önce işlenmemişse işaretlenir ve komşuları gevşetilir. Bir komşunun uzaklığı küçülürse komşu, yeni uzaklığıyla kuyruğa eklenir.

Kodda bir ayrıntı göze çarpar: bir düğümün uzaklığı küçüldüğünde kuyruktaki eski kaydı silinmez, yeni bir kayıt eklenir. Böylece aynı düğüm kuyrukta birden çok kez bulunabilir. Eski kayıt daha büyük bir uzaklık taşıdığı için kuyruktan daha geç çıkar. Çıktığında düğüm zaten işaretlenmiştir ve `ziyaretEdildi` denetimi onu atlar. Bu yönteme **tembel silme** (*lazy deletion*) denir.

Tembel silmenin nedeni, Java'nın `PriorityQueue` sınıfında bir öğenin önceliğini azaltan bir işlemin bulunmamasıdır. Kuyruktan rastgele bir öğeyi silmek de doğrusal zaman alır. Yeni kayıt eklemek ise *O(log n)* sürer. Bedeli, kuyruğun biraz büyümesidir: her gevşetme bir kayıt ekleyebildiği için kuyrukta en çok *E* kayıt birikir. Yolu da bulmak isteniyorsa gevşetme sırasında `önceki[v] = u` ataması eklenir.

> **Sık yapılan hata**
>
> Tembel silmeli kodda `ziyaretEdildi` denetimini atlamak. Eski kayıt çıktığında düğüm ikinci kez işlenir ve komşuları gereksiz yere yeniden gevşetilir. Ağırlıklar negatif değilse sonuç yine doğru çıkar, ama iş artar. Denetim, her düğümün tam bir kez işlenmesini sağlar.

4.2.2.6 Java kodu ve izleme

Sözde kodun Java karşılığı aşağıdadır. Çizge `komsu` adlı bir komşuluk listesinde tutulur: `komsu.get(u)`, *u*'dan çıkan kenarların listesidir. Her `Kenar` nesnesinin `hedef` ve `agirlik` alanları vardır. Düğümler 1'den `V`'ye kadar numaralıdır; bu yüzden diziler `V + 1` boyutludur ve 0 indisi kullanılmaz. `SONSUZ`, `Integer.MAX_VALUE` değerini taşıyan bir sabittir.

```
void dijkstra(int kaynak) {
  int[] uzaklik = new int[V + 1];
  int[] onceki = new int[V + 1];
  boolean[] ziyaret = new boolean[V + 1];
  Arrays.fill(uzaklik, SONSUZ);
  uzaklik[kaynak] = 0;
  PriorityQueue<int[]> kuyruk =
    new PriorityQueue<>((a, b) -> a[1] - b[1]);
  kuyruk.add(new int[] { kaynak, 0 });
  while (!kuyruk.isEmpty()) {
    int u = kuyruk.poll()[0];
    if (ziyaret[u]) continue; // eski kayıt
    ziyaret[u] = true;
    for (Kenar k : komsu.get(u)) {
      int v = k.hedef, w = k.agirlik;
      if (!ziyaret[v]
          && uzaklik[u] + w < uzaklik[v]) {
        uzaklik[v] = uzaklik[u] + w;
        onceki[v] = u;
        kuyruk.add(new int[] { v, uzaklik[v] });
      }
    }
  }
}
```

Kod satır satır şöyle okunur:

- **Satır 2–6:** `uzaklik`, `onceki` ve `ziyaret` dizileri oluşturulur. `uzaklik` sonsuzla doldurulur, kaynağınki 0 yapılır. `boolean` dizisi Java'da kendiliğinden `false` ile başlar.
- **Satır 7–9:** Kuyruk `int[]` çiftleri tutar: `a[0]` düğüm, `a[1]` o anki uzaklıktır. Karşılaştırıcı `(a, b) -> a[1] - b[1]` çiftleri uzaklığa göre sıralar; böylece `poll()` en küçük uzaklıklı çifti verir. Başta kuyrukta yalnız `{kaynak, 0}` vardır.
- **Satır 11–13:** En küçük çift çıkarılır ve düğümü `u`'ya alınır. Düğüm daha önce işlenmişse bu eski bir kayıttır ve `continue` ile atlanır. İşlenmemişse işaretlenir; bu an düğümün kesinleştiği andır.
- **Satır 14–21:** `u`'dan çıkan her kenar için komşu `v` ve ağırlık `w` alınır. Komşu kesinleşmemişse ve `u` üzerinden gelen yol daha kısaysa gevşetme yapılır: uzaklık ve önceki güncellenir, yeni çift kuyruğa eklenir.

Karşılaştırıcıdaki çıkarma işlemi bir taşma tuzağı taşır. Uzaklıklardan biri `SONSUZ` olsaydı `a[1] - b[1]` taşabilir ve yanlış işaret verebilirdi. Bu kodda kuyruğa yalnız sonlu uzaklıklar eklendiği için sorun çıkmaz. Daha güvenli bir yazım `Integer.compare(a[1], b[1])` ya da `Comparator.comparingInt(a -> a[1])` olurdu.

Kod, Örnek 2'deki çizgede `dijkstra(5)` ile çalıştırıldığında kuyruk Tablo 4.2.5'teki gibi değişir. Kuyruk sütunu, her işlemden sonra kuyruktaki kayıtları küçükten büyüğe sırayla gösterir.

poll()	Sonuç	İncelenen kenarlar	Kuyruk (sonra)
—	başlangıç	—	(5, 0)
(5, 0)	5 kesinleşir	5→2: 0 + 4 < ∞, uzaklik[2] = 4; 5→4: 0 + 2 < ∞, uzaklik[4] = 2	(4, 2) (2, 4)
(4, 2)	4 kesinleşir	4→2: 2 + 1 < 4, uzaklik[2] = 3; 4→3: 2 + 1 < ∞, uzaklik[3] = 3	(2, 3) (3, 3) (2, 4)
(2, 3)	2 kesinleşir	2→1: 3 + 3 < ∞, uzaklik[1] = 6	(3, 3) (2, 4) (1, 6)
(3, 3)	3 kesinleşir	3→4: ziyaret[4] = true, koşul yanlış	(2, 4) (1, 6)
(2, 4)	eski kayıt, atlanır	—	(1, 6)
(1, 6)	1 kesinleşir	1→3 ve 1→4: ikisi de ziyaret edilmiş	boş

Tablo 4.2.5: `dijkstra(5)` izlemesi (Örnek 2). Çiftler (düğüm, uzaklık) biçimindedir.

İzlemede tembel silme açıkça görülür. 2 numaralı düğüm kuyruğa iki kez girmiştir: önce *(2, 4)* olarak, sonra daha kısa yol bulununca *(2, 3)* olarak. Yeni kayıt önce çıkar ve 2'yi kesinleştirir. Eski *(2, 4)* kaydı beşinci `poll()` çağrısında çıkar; `ziyaret[2]` zaten `true` olduğu için atlanır. Toplam altı `poll()` çağrısı yapılır: beş düğüm için birer tane ve bir eski kayıt için bir tane. Döngü bittiğinde `uzaklik = [6, 3, 3, 2, 0]` ve `onceki = [2, 4, 4, 5, -]` olur (indis 1–5); bu, el ile bulunan sonuçla aynıdır.

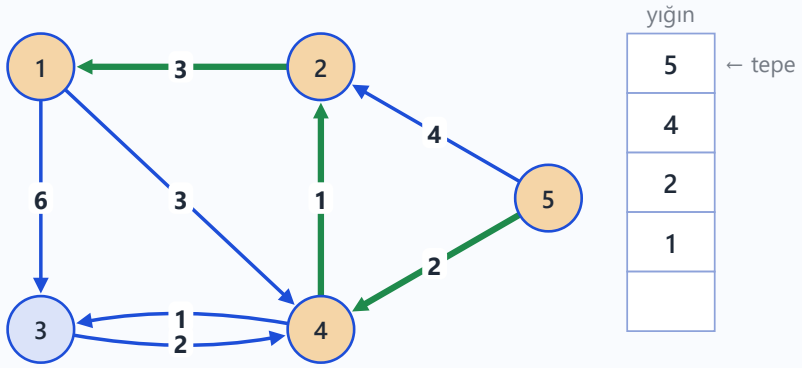
4.2.2.7 Yolu geri kurmak

Dijkstra uzaklıkları ve önceki dizisini birlikte üretir. Uzaklık dizisi “ne kadar?” sorusunu, önceki dizisi “hangi yoldan?” sorusunu cevaplar. Örnek 2'nin sonucu *π = [2, 4, 4, 5, /]* idi. Hedef 1 için işaretçiler geriye izlenir: *π[1] = 2*, *π[2] = 4*, *π[4] = 5*. 5 kaynaktır, izleme burada durur. Toplanan düğümler *1, 2, 4, 5* sırasındadır, yani hedeften kaynağa doğrudur. Sıra çevrilince yol *5 → 4 → 2 → 1* olur ve uzunluğu *2 + 1 + 3 = 6 = d[1]*'dir.

Sırayı çevirmenin doğal yolu bir **yığın** (*stack*) kullanmaktır. Yığın son giren ilk çıkar (LIFO) ilkesiyle çalışır. Hedeften kaynağa doğru itilen düğümler kaynaktan hedefe doğru çekilir.

```
void yolYaz(int[] onceki, int kaynak, int hedef) {
  Deque<Integer> yigin = new ArrayDeque<>();
  for (int v = hedef; v != kaynak; v = onceki[v])
    yigin.push(v);          // hedeften geriye
  yigin.push(kaynak);
  while (!yigin.isEmpty())
    System.out.print(yigin.pop() + " ");
}
```

İkinci satır boş bir yığın oluşturur; Java'da `ArrayDeque` sınıfı yığın olarak kullanılır. `for` döngüsü `v = hedef` ile başlar. `v` kaynağa eşit olmadıkça `v`'yi yığına iter ve bir önceki düğüme geçer. Döngü kaynağı itmez; çünkü `v` kaynağa eşit olduğunda koşul yanlış olur. Beşinci satır kaynağı ayrıca iter. Son iki satır yığını boşaltarak düğümleri yazdırır. Tablo 4.2.6 `yolYaz(onceki, 5, 1)` çağrısının izlemesidir.



Şekil 4.2.6: `yolYaz(onceki, 5, 1)` çağrısında kaynak itildikten sonraki durum. Yığının tepesinde (en üstte) kaynak 5, dibinde hedef 1 vardır. Yeşil kenarlar 5'ten 1'e en kısa yoldur.

Adım	`v`	İşlem	Yığın	Çıktı
1	1	1 ≠ 5, push(1); v = onceki[1] = 2	[1]	
2	2	2 ≠ 5, push(2); v = onceki[2] = 4	[2, 1]	
3	4	4 ≠ 5, push(4); v = onceki[4] = 5	[4, 2, 1]	
4	5	5 = kaynak, döngü biter; push(5)	[5, 4, 2, 1]	
5–8	—	dört kez pop() ve yaz	[ ]	5 4 2 1

Tablo 4.2.6: `yolYaz(onceki, 5, 1)` izlemesi; `onceki = [2, 4, 4, 5, /]` (indis 1–5). Yığın tepesi solda yazılmıştır.

Yöntemin maliyeti yolun kenar sayısıyla orantılıdır. Yol en çok $V - 1$ kenarlı olduğu için süre ve yığının kapladığı yer $O(V)$'dir. Önceki dizisinin kendisi de $O(V)$ yer tutar. Bütün en kısa yolların tek bir dizide saklanabilmesi, optimal alt yapının pratik bir sonucudur: her yol, kendinden bir önceki düğümün yolunun bir kenar uzatılmış hâlidir.

Sık yapılan hata

`yolYaz` 'ı kaynaktan ulaşılamayan bir hedefle çağırmak. Böyle bir hedefin önceki değeri hiç atanmamıştır; Java'da 0 kalır ya da −1 gibi bir işaret değeri taşır. Döngü kaynağa hiç ulaşamaz: ya sonsuza kadar döner ya da dizinin dışına çıkıp hata verir. Çağırmadan önce `uzaklik[hedef] != SONSUZ` denetlenmelidir.

4.2.2.8 Karmaşıklık

Dijkstra'nın maliyeti iki işlemden gelir: en küçük uzaklıklı düğümü seçmek ve kenarları gevşetmek. Seçim V kez, gevşetme en çok E kez yapılır. Toplam süre, bu iki işlemin hangi veri yapısıyla yapıldığına bağlıdır.

- **Dizi.** Uzaklıklar sırasız bir dizide tutulur. En küçüğü bulmak bütün diziyi taramak demektir: her seçim $O(V)$, V seçim $O(V^2)$ tutar. Gevşetme diziye tek bir yazmadır, $O(1)$. Toplam $O(V^2 + E) = O(V^2)$'dir, çünkü basit bir çizgede $E < V^2$'dir.
- **İkili heap.** Öncelikli kuyruk ikili heap ile kurulur. En küçüğü çıkarmak ve yeni kayıt eklemek $O(\log V)$ sürer. V çıkarma ve en çok E ekleme yapılır: toplam $O((V + E) \log V)$. Tembel silmede kuyrukta E kayıt bulunabilir; $\log E \le \log V^2 = 2 \log V$ olduğundan sınır değişmez.
- **Fibonacci heap.** Bu yapıda en küçüğü çıkarmak amortize $O(\log V)$, önceliği azaltmak amortize $O(1)$ sürer. Toplam $O(E + V \log V)$'dir. Kuramsal olarak en iyi sınırdır, ama yapının sabitleri büyük olduğu için pratikte ikili heap çoğu zaman daha hızlıdır.

Seçim yapısı	Seçim	Gevşetme	Toplam	Uygun çizge
Dizi (doğrusal tarama)	$V \times O(V)$	$E \times O(1)$	$O(V^2)$	yoğun ($E \approx V^2$)
İkili heap	$O(\log V)$	$O(\log V)$	$O((V + E) \log V)$	seyrek ($E \ll V^2$)
Fibonacci heap	$O(\log V)$ amortize	$O(1)$ amortize	$O(E + V \log V)$	kuramsal en iyi

Tablo 4.2.7: Seçim yapısına göre Dijkstra'nın maliyeti.

Hangi yapının daha iyi olduğu çizgenin yoğunluğuna bağlıdır. **Yoğun** bir çizgede kenar sayısı V^2 mertebesindedir. Heap sürümü bu durumda $O(V^2 \log V)$ tutar ve dizi sürümünden yavaştır. **Seyrek** bir çizgede ise kenar sayısı V mertebesindedir ve heap sürümü açıkça kazanır. Yol ağları seyrektir: bir kavşaktan çoğu zaman üç dört yol çıkar.

Örnek 4.2.3

Bir yol ağında $V = 10^4$ kavşak ve $E = 5 \cdot 10^4$ yol vardır. İki sürümün yaklaşık işlem sayıları karşılaştırılacak.

1. **Dizi sürümü.** $V^2 = 10^8$ işlem.
2. **Heap sürümü.** $V + E = 6 \cdot 10^4$; $\log_2 10^4 \approx 13{,}3$, yukarı yuvarlanınca 14. Toplam yaklaşık $6 \cdot 10^4 \cdot 14 \approx 8{,}4 \cdot 10^5$ işlem.
3. **Karşılaştırma.** Heap sürümü yaklaşık 120 kat az işlem yapar. Çizge seyrek olduğu için heap'in her işlemdeki $\log V$ bedeli, dizinin her seçimde bütün düğümleri taramasından çok daha ucuzdur.

Bellek tarafında `uzaklik`, `onceki` ve `ziyaret` dizileri $O(V)$ yer tutar. Tembel silmeli kuyrukta en çok E kayıt birikebilir. Toplam ek bellek $O(V + E)$'dir. Önceliği azaltma işlemi olan bir kuyrukta her düğüm bir kez bulunur ve kuyruk $O(V)$ yer tutar.

4.2.2.9 Doğruluk ve negatif kenar sorunu

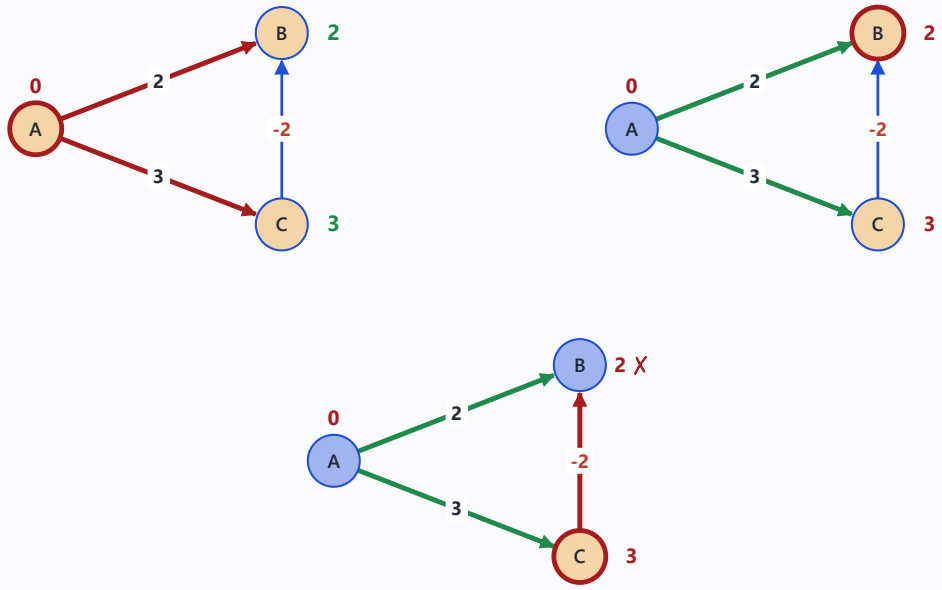
Dijkstra bir düğümü seçtiği anda onun uzaklığını kesin kabul eder. Bu kararın neden doğru olduğu aşağıdaki teoremle gösterilir.

Teorem 4.2.1 (Dijkstra'nın doğruluğu)

Bütün kenar ağırlıkları negatif değilse, Dijkstra algoritması bir u düğümünü seçip S'ye eklediği anda $d[u] = \delta(s, u)$'dur.

İspat. Teoremin yanlış olduğunu ve u'nun, seçildiği anda $d[u] > \delta(s, u)$ olan ilk düğüm olduğunu varsayalım. Kaynaktan u'ya bir en kısa yol P alalım. P kaynaktan, yani S'nin içinden başlar ve S'nin dışındaki u'da biter. Dolayısıyla P üzerinde S'nin dışındaki ilk düğüm vardır; buna y, ondan önceki düğüme x diyelim. x daha önce kesinleşmiştir ve varsayım gereği $d[x] = \delta(s, x)$'tir. x kesinleşirken (x, y) kenarı gevşetildiği için $d[y] \le \delta(s, x) + w(x, y) = \delta(s, y)$ olur; yani $d[y] = \delta(s, y)$. y, P üzerinde u'dan önce gelir ve ağırlıklar negatif olmadığı için $\delta(s, y) \le \delta(s, u)$'dur. Buradan $d[y] = \delta(s, y) \le \delta(s, u) < d[u]$ çıkar. Oysa algoritma u'yu seçtiğine göre $d[u] \le d[y]$ olmalıdır. Çelişki. □

Kanıtın kilit adımı $\delta(s, y) \le \delta(s, u)$ eşitsizliğidir. Yol y'den u'ya devam ederken uzunluğu azalmaz, çünkü eklenen her kenar negatif değildir. Kısacası, S'nin dışından geçen her yol en az $d[u]$ uzunluğundadır; yol ancak uzayabilir. Negatif bir kenar bu varsayımı bozar. Sonradan işlenen bir düğüm, negatif bir kenarla kesinleşmiş bir düğüme daha kısa bir yol açabilir. Dijkstra kesinleşmiş düğümlere dönmediği için bu yolu hiç görmez.



Şekil 4.2.7: Dijkstra'nın negatif kenarda yanılması. Solda: A kesinleşir; $d[B] = 2$, $d[C] = 3$. Ortada: en küçük tahmin B'nindir; B kesinleşir. Sağda: C işlenince $3 + (-2) = 1 < 2$ bulunur, ama B artık değiştirilemez.

Örnek 4.2.4

Kenarlar $A \to B$ (2), $A \to C$ (3) ve $C \to B$ (−2) olan çizgede kaynak A'dır. Dijkstra'nın bulduğu $d[B]$ ile gerçek uzaklık karşılaştırılacak.

1. **A kesinleşir.** Gevşetmeler: $d[B] = 2$, $d[C] = 3$.
2. **B seçilir.** En küçük tahmin B'nindir (2). Dijkstra B'yi kesinleştirir; yani "B'ye daha kısa yol olamaz" diye karar verir.
3. **C seçilir.** $C \to B$ kenarı $3 + (-2) = 1 < 2$ önerir. Ama B zaten S'dedir ve gevşetilmez.
4. **Sonuç.** Dijkstra $d[B] = 2$ verir; gerçek uzaklık $\delta(A, B) = 1$'dir ($A \to C \to B$). Hata sessizdir: algoritma hiçbir uyarı vermeden yanlış sayı üretir.

Akla gelen bir çözüm, bütün ağırlıklara aynı sabiti ekleyerek negatif kenarları ortadan kaldırmaktır. Bu çözüm işe yaramaz. Sabit her kenara eklendiği için çok kenarlı yollar daha fazla ceza alır. Örnekte bütün ağırlıklara 2 eklenirse $A \to B$ yolu 4, $A \to C \to B$ yolu $5 + 0 = 5$ olur. Artık doğrudan kenar daha kısa görünür ve en kısa yol değişmiş olur. Sabit ekleme, kenar sayısı sabit olan yapılarda sıralamayı korur: örneğin her kapsayan ağaçta $V - 1$ kenar bulunur. En kısa yollarda ise kenar sayısı yoldan yola değişir. Negatif kenar varsa doğru araç Bellman-Ford algoritmasıdır.

Sık yapılan hata

Ağırlığı 0 olan kenarları da sorun sanmak. Teoremin koşulu "negatif olmayan" ağırlıktır; sıfır bu koşula uyar. Kanıttaki $\delta(s, y) \le \delta(s, u)$ eşitsizliği sıfır ağırlıklı kenarlarda da geçerlidir. Dijkstra yalnız kesin negatif kenarlarda yanılır.

4.2.3 Bellman-Ford algoritması

Dijkstra hızlıdır, ama negatif kenara dayanamaz. Bellman-Ford algoritması bu eksiği kapatır. Algoritma tek kaynaklı en kısa yolları bulur, negatif ağırlıklı kenarlarla doğru çalışır ve kaynaktan ulaşılabilen negatif döngüleri tespit eder. Bunun bedeli hızdır: Bellman-Ford $O(V \cdot E)$ zaman alır ve Dijkstra'dan belirgin biçimde yavaştır.

Algoritma adını Richard Bellman (1958) ve Lester Ford Jr.'dan (1956) alır. Edward F. Moore da 1959'da aynı yöntemi bağımsız olarak yayımlamıştır; bu yüzden bazı kaynaklarda Bellman-Ford-Moore algoritması diye geçer.

4.2.3.1 Fikir ve adımlar

Bellman-Ford hiçbir düğümü kesinleştirmez ve düğüm seçmez. Bunun yerine bütün kenarları sırayla gevşetir ve bu taramayı $V - 1$ kez tekrarlar. Bir taramaya **tur** denir. Her turda bir kenarın gevşemesi, bir düğümün tahminini düşürür. Bu düşüş, o düğümden çıkan kenarların sonraki gevşetmelerinde komşulara da yansır. İyileşme böylece yol boyunca ilerler.

Bu yaklaşım açgözlü değildir; bir **dinamik programlamadır** (*dynamic programming*). i. tur, en çok i kenarlı en kısa yolları, bir önceki turun bulduğu en çok $i - 1$ kenarlı yollardan kurar. Algoritmanın dört adımı şunlardır:

1. **İlklendirme.** Kaynağa uzaklık 0, diğer düğümlere ∞ atanır; önceki düğümler tanımsızdır.
2. **Turlar.** Kenarlar tek tek incelenir ve her biri gevşetilir; bu tur $V - 1$ kez tekrarlanır. Bir turdaki kenar sırası bütün turlarda aynıdır. Sıra sonucu değiştirmez, yalnız sonuca kaç turda ulaşılacağını etkiler.
3. **Yayılma.** Bir düğümün uzaklığı güncellenirse, sonraki incelemelerde onun komşularının uzaklıkları da güncellenebilir. İyileşme, yol boyunca her turda en az bir kenar ilerler.
4. **Denetim.** Negatif döngü kontrolü için bütün kenarlar bir kez daha incelenir. Hâlâ gevşeyen bir kenar varsa kaynaktan ulaşılabilen bir negatif döngü vardır. Döngü yoksa $V - 1$ turdan sonra hiçbir kenar gevşemez.

Bellman-Ford düğüm seçmediği için komşuluk listesine de gerek duymaz. Çizgeyi bir kenar listesi olarak tutmak yeter: her kayıt bir (u, v, w) üçlüsüdür. Algoritma bu listeyi baştan sona okur ve her üçlüyü gevşetir.

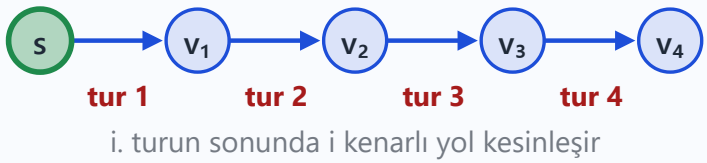
4.2.3.2 Neden V − 1 tur? Karmaşıklık

Tur sayısının nereden geldiği, en kısa yolların bir özelliğine dayanır. Negatif döngü yoksa her düğüme döngüsüz bir en kısa yol vardır ve bu yol en çok $V - 1$ kenarlıdır. Algoritmanın her turu bu yolları bir kenar daha uzatır.

> **Teorem 4.2.2 (Bellman-Ford değişmezi)**
>
> Kaynaktan ulaşılabilen negatif döngü yoksa, i. turun sonunda, en kısa yollarından biri en çok i kenarlı olan her v düğümü için $d[v] = \delta(s, v)$'dir. Buna göre $V - 1$ turdan sonra bütün uzaklıklar doğrudur.

İspat. v'ye en kısa yol $s = v_0, v_1, \ldots, v_k = v$ ve $k \le i$ olsun. j üzerinden tümevarım yapalım. Başlangıçta $d[v_0] = 0 = \delta(s, s)$'dir. $j - 1$. turun sonunda $d[v_{j-1}] = \delta(s, v_{j-1})$ olduğunu varsayalım. j. turda (v_{j-1}, v_j) kenarı da gevşetilir. Gevşetmeden sonra $d[v_j] \le \delta(s, v_{j-1}) + w(v_{j-1}, v_j) = \delta(s, v_j)$ olur; eşitlik, optimal alt yapıdan gelir. Tahmin uzaklığın altına inemeyeceği için $d[v_j] = \delta(s, v_j)$'dir. $j = k \le i$ için iddia elde edilir. Negatif döngü yoksa her düğümün en kısa yolu en çok $V - 1$ kenarlıdır; $i = V - 1$ alınınca bütün uzaklıklar doğru çıkar. □

Kanıtın ana fikri şudur: j. turda yolun j. kenarı mutlaka gevşetilir. Kenarların hangi sırada incelendiği önemli değildir; sıra ancak işin daha erken bitmesini sağlayabilir. Uygun bir sırada, bir tur içinde birkaç kenar art arda iyileşebilir ve yol tek turda tamamlanabilir. En kötü sırada ise her tur yalnız bir kenar ilerler ve $V - 1$ turun hepsi gerekir.



Şekil 4.2.8: Beş düğümlü bir zincirde en kısa yol dört kenarlıdır. *i*. turun sonunda yolun ilk *i* kenarı kesinleşmiştir; dört tur bütün yolu tamamlar.

Süre kolayca hesaplanır. Her tur E kenarı inceler ve her inceleme $O(1)$ sürer. $V - 1$ tur ve bir denetim turu, toplam $V \cdot E$ inceleme eder: $O(V \cdot E)$. Ek bellek yalnız uzaklık ve önceki dizileridir: $O(V)$.

Çizge	Zaman
Seyrek, $E \approx V$	$O(V^2)$
Yoğun, $E \approx V^2$	$O(V^3)$
En iyi durum (erken durma)	$O(E)$

Tablo 4.2.8: Çizgenin yoğunluğuna göre Bellman-Ford'un süresi.

Tablonun son satırı basit bir iyileştirmeden gelir. Bir turda hiçbir d değeri değişmezse sonraki turlar da hiçbir şeyi değiştiremez. Çünkü bir tur, yalnız bir önceki turda değişen tahminler sayesinde yeni bir gevşetme bulabilir. Değişiklik olmayan bir turdan sonra döngüden çıkılabilir. Kenarlar uygun sıradaysa ilk tur her şeyi bulur, ikinci tur bunu doğrular: süre $O(E)$'ye iner.

```
int[] bellmanFordErken(int V, List<Kenar> kenarlar, int kaynak) {
  int[] uzaklik = new int[V];
  Arrays.fill(uzaklik, SONSUZ);
  uzaklik[kaynak] = 0;
  for (int i = 1; i < V; i++) {
    boolean degisti = false;
    for (Kenar k : kenarlar) {
      int u = k.bas, v = k.son, w = k.agirlik;
      if (uzaklik[u] != SONSUZ && uzaklik[u] + w < uzaklik[v]) {
        uzaklik[v] = uzaklik[u] + w;
        degisti = true;
      }
    }
    if (!degisti) break;   // bu tur hiçbir şeyi değiştirmedi
  }
  return uzaklik;
}
```

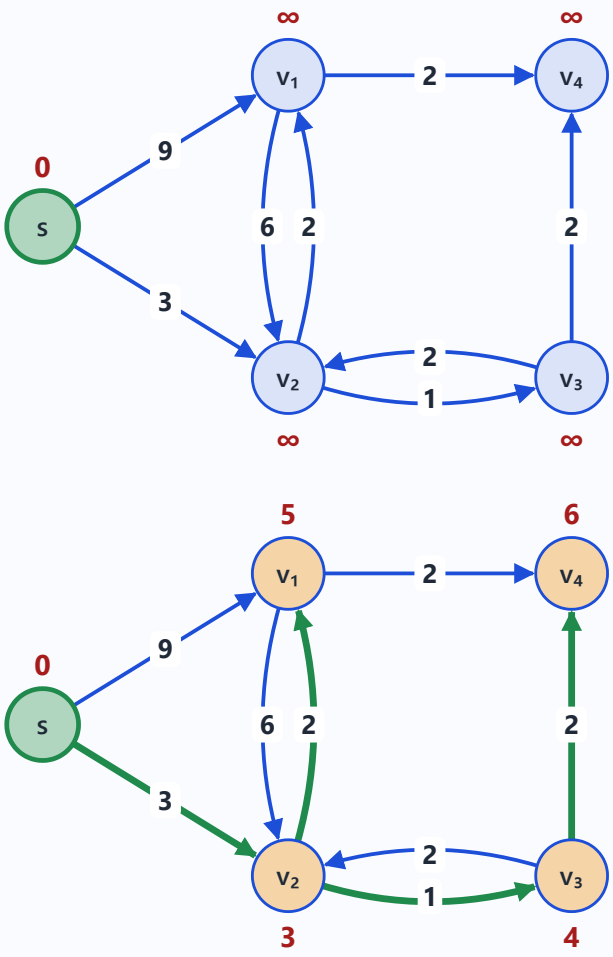
Kodda her turun başında `degisti` bayrağı `false` yapılır. Turdaki herhangi bir gevşetme bayrağı `true` yapar. Tur bittiğinde bayrak hâlâ `false` ise `break` döngüyü sonlandırır. Erken durma sonucu değiştirmez; yalnız gereksiz turları atlar. En kötü durum yine $O(V \cdot E)$'dir, çünkü her turda bir şey de-

ğişebilir.

> **Sık yapılan hata**
>
> Bellman-Ford'un işi tek turda bitirdiğini sanmak. Kenar sırası şanslıysa bu doğrudur, ama genelde değildir. Negatif kenarlı Örnek 2'de beş turun hepsi gerekir. Ayrıca denetim turu, $V - 1$ turdan ayrı bir taramadır; tur sayısına katılmaz.

4.2.3.3 Örnek 1: kenar kenar gevşetme

İlk örnek beş düğümlü yönlü bir çizgedir. Kaynak s'dir ve kenarlar her turda şu sırayla incelenir: (s, v_1) 9, (s, v_2) 3, (v_1, v_2) 6, (v_1, v_4) 2, (v_2, v_1) 2, (v_2, v_3) 1, (v_3, v_2) 2, (v_3, v_4) 2.



Şekil 4.2.9: Bellman-Ford Örnek 1. Solda başlangıç durumu, sağda sonuç. Yeşil kenarlar en kısa yol ağacıdır: $s \to v_2 \to v_1$ (5), $s \to v_2 \to v_3 \to v_4$ (6).

Kenar	Aday	Sonuç	d[v_1]	d[v_2]	d[v_3]	d[v_4]
başlangıç	—	—	∞	∞	∞	∞

Kenar	Aday	Sonuç	$d[v_1]$	$d[v_2]$	$d[v_3]$	$d[v_4]$
(s, v_1) 9	$0 + 9 = 9 < \infty$	$d[v_1] = 9$	**9**	∞	∞	∞
(s, v_2) 3	$0 + 3 = 3 < \infty$	$d[v_2] = 3$	9	**3**	∞	∞
(v_1, v_2) 6	$9 + 6 = 15 \geq 3$	değişmez	9	3	∞	∞
(v_1, v_4) 2	$9 + 2 = 11 < \infty$	$d[v_4] = 11$	9	3	∞	**11**
(v_2, v_1) 2	$3 + 2 = 5 < 9$	$d[v_1] = 5$	**5**	3	∞	11
(v_2, v_3) 1	$3 + 1 = 4 < \infty$	$d[v_3] = 4$	5	3	**4**	11
(v_3, v_2) 2	$4 + 2 = 6 \geq 3$	değişmez	5	3	4	11
(v_3, v_4) 2	$4 + 2 = 6 < 11$	$d[v_4] = 6$	5	3	4	**6**

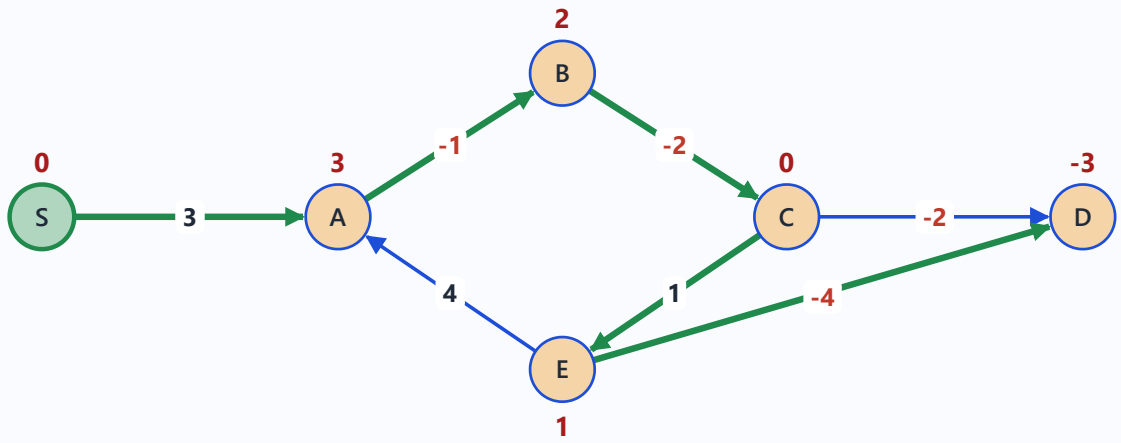
Tablo 4.2.9: Bellman-Ford Örnek 1'in birinci turu, kenar kenar.

Birinci turda sekiz kenardan altısı gevşer. Bazı tahminler aynı tur içinde iki kez değişir. v_1 önce doğrudan kenarla 9 olur, sonra v_2 üzerinden 5'e iner. v_4 önce 11 olur, sonra v_3 üzerinden 6'ya iner. Bu, kenar sırasının işe yaradığı bir durumdur. v_2'nin değeri, ondan çıkan kenarlar incelenmeden önce bulunmuştur; bu yüzden iyileşme aynı tur içinde yola yayılır.

İkinci turda sekiz kenarın hiçbiri gevşemez. Erken durmalı sürüm burada durur. Sonuç $d[v_1] = 5$, $d[v_2] = 3$, $d[v_3] = 4$, $d[v_4] = 6$'dır. Erken durma olmasaydı üç tur ve bir denetim turu daha boşa dönerdi. Bu sırada bütün iş tek turda bitmiştir.

4.2.3.4 Örnek 2: negatif kenarlar ve en kötü sıra

İkinci örnekte altı düğüm ve dört negatif kenar vardır. Kaynak S'dir. Kenarlar her turda şu sırayla incelenir: *(E, D)* −4, *(E, A)* 4, *(C, D)* −2, *(C, E)* 1, *(B, C)* −2, *(A, B)* −1, *(S, A)* 3. Bu sıra bilerek seçilmiştir: en kısa yolun kenarlarını sondan başa doğru dizer. Bu, Bellman-Ford için en kötü sıradır.



Şekil 4.2.10: Bellman-Ford Örnek 2'nin sonucu (kaynak S). Kırmızı kenar ağırlıkları negatiftir. Yeşil kenarlar en kısa yol ağacıdır; D'ye en kısa yol E üzerinden gelir (−3).

Tur	Gevşeyen kenarlar	S	A	B	C	D	E
başlangıç	—	0	∞	∞	∞	∞	∞
1	(S, A): ∞ → 3	0	**3**	∞	∞	∞	∞
2	(A, B): ∞ → 2	0	3	**2**	∞	∞	∞
3	(B, C): ∞ → 0	0	3	2	**0**	∞	∞
4	(C, D): ∞ → −2; (C, E): ∞ → 1	0	3	2	0	**−2**	**1**
5	(E, D): −2 → −3	0	3	2	0	**−3**	1
denetim	hiçbiri	0	3	2	0	−3	1

Tablo 4.2.10: Bellman-Ford Örnek 2'nin tur tur izlemesi.

Birinci turda yalnız *(S, A)* kenarı işe yarar. Listedeki diğer kenarların başlangıç düğümleri incelendikleri anda hâlâ ∞'dur. Örneğin *(A, B)* kenarı listede *(S, A)*'dan önce gelir. İncelendiğinde A'nın tahmini henüz ∞'dur ve kenar boşa geçer. Her tur yolu tam olarak bir kenar uzatır: A, B, C sırayla bulunur. Dördüncü turda C'den çıkan iki kenar birden gevşer. Beşinci turda D, E üzerinden daha kısa bir yol bulur: $1 + (-4) = -3 < -2$. Doğrudan $C \to D$ kenarı −2 verirken $C \to E \to D$ yolu $1 - 4 = -3$ verir.

$V - 1 = 5$ turun hepsi kullanılmıştır. Denetim turunda hiçbir kenar gevşemez; demek ki negatif döngü yoktur. Gerçekten de çizgedeki tek döngü $A \to B \to C \to E \to A$'dır ve ağırlığı $-1 - 2 + 1 + 4 = +2$'dir. Negatif kenarlar olduğu hâlde döngü pozitiftir.

Örnek 4.2.5

Aynı kenarlar ters sırada, yani *(S, A), (A, B), (B, C), (C, E), (C, D), (E, A), (E, D)* sırasıyla incelenirse erken durmalı Bellman-Ford kaç tur döner?

1. **Birinci tur.** *(S, A)*: A = 3. Hemen ardından *(A, B)*: B = 2. *(B, C)*: C = 0. *(C, E)*: E = 1. *(C, D)*: D = −2. *(E, A)* 5 önerir ve 3'ü geçemez. Son kenar *(E, D)*: D = −3. Altı kenar gevşer.
2. **İkinci tur.** Hiçbir kenar gevşemez; `degisti` `false` kalır ve döngü biter.
3. **Sonuç.** İki tur yeter; sonuç en kötü sıradakiyle aynıdır. Kenar sırası sonucu değil, tur sayısını değiştirir. Bu iki sayı, notun erken durmalı kodunun Java ile çalıştırılmasıyla da doğrulanmıştır: en kötü sırada 5 tur, iyi sırada 2 tur.

4.2.3.5 Örnek 3: Dijkstra'nın çizgesi

Üçüncü örnek, Dijkstra Örnek 2'deki yönlü çizgedir; kaynak yine 5'tir. Negatif kenar yoktur, bu yüzden iki algoritmanın aynı sonucu vermesi gerekir. Kenarlar her turda şu sırayla incelenir: *(1, 3), (1, 4), (2, 1), (3, 4), (4, 2), (4, 3), (5, 2), (5, 4)*.

Tur	Gevşeyen kenarlar	d[1..5]	π[1..5]
başlangıç	—	∞, ∞, ∞, ∞, 0	/, /, /, /, /
1	(5, 2): ∞ → 4; (5, 4): ∞ → 2	∞, 4, ∞, 2, 0	/, 5, /, 5, /

Tur	Gevşeyen kenarlar	d[1..5]	π[1..5]
2	(2, 1): ∞ → 7; (4, 2): 4 → 3; (4, 3): ∞ → 3	7, 3, 3, 2, 0	2, 4, 4, 5, /
3	(2, 1): 7 → 6	6, 3, 3, 2, 0	2, 4, 4, 5, /
4	hiçbiri	6, 3, 3, 2, 0	2, 4, 4, 5, /

Tablo 4.2.11: Bellman-Ford Örnek 3 (kaynak 5). Her tur sonunda d ve π.

Birinci turda kaynaktan çıkan iki kenar listenin sonundadır; yalnız onlar gevşer. İkinci turda üç gevşetme olur. *(2, 1)* kenarı 1'e ilk yolu verir: *4 + 3 = 7*. Hemen sonra *(4, 2)* kenarı 2'nin tahminini 3'e düşürür. Ama *(2, 1)* listede *(4, 2)*'den önce geldiği için bu iyileşme 1'e ancak üçüncü turda yansır: *3 + 3 = 6*. Dördüncü turda hiçbir şey değişmez.

Sonuç *d = [6, 3, 3, 2, 0]* ve *π = [2, 4, 4, 5, /]*'dir. Dijkstra'nın bulduğu sonuçla tıpatıp aynıdır. Fark yapılan iştedir. Dijkstra her kenara bir kez bakmıştı; Bellman-Ford dört turda 32 kenar incelemesi yapar ve denetim turuyla bu sayı 40'a çıkar.

Denetim turu, her kenar için bir kez daha *d[v] > d[u] + w(u, v)* sorusunu sorar. Tablo 4.2.12 Örnek 3'ün sonucunda bu sorunun sekiz kenar için de "hayır" cevabını verdiğini gösterir.

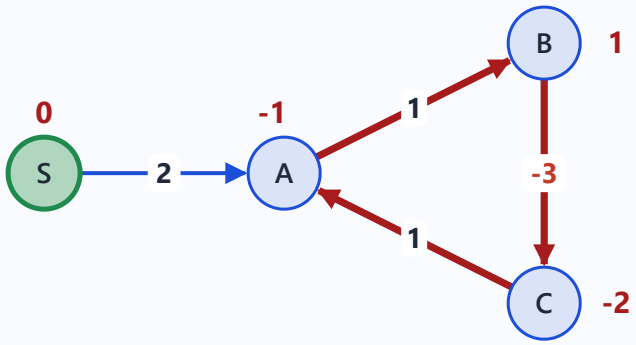
Kenar	d[v]	d[u] + w(u, v)	Sonuç
(1, 3)	d[3] = 3	6 + 6 = 12	3 ≯ 12 ✓
(1, 4)	d[4] = 2	6 + 3 = 9	2 ≯ 9 ✓
(2, 1)	d[1] = 6	3 + 3 = 6	6 ≯ 6 ✓
(3, 4)	d[4] = 2	3 + 2 = 5	2 ≯ 5 ✓
(4, 2)	d[2] = 3	2 + 1 = 3	3 ≯ 3 ✓
(4, 3)	d[3] = 3	2 + 1 = 3	3 ≯ 3 ✓
(5, 2)	d[2] = 3	0 + 4 = 4	3 ≯ 4 ✓
(5, 4)	d[4] = 2	0 + 2 = 2	2 ≯ 2 ✓

Tablo 4.2.12: Örnek 3'ün sonucunda negatif döngü denetimi.

Eşitlik sağlanan kenarlar dikkat çekicidir: *(2, 1)*, *(4, 2)*, *(4, 3)* ve *(5, 4)*. Bunlar tam olarak en kısa yol ağacının kenarlarıdır. Ağaçtaki her kenar için *d[v] = d[u] + w(u, v)* olur, çünkü *v*'nin değeri bu kenardan gelmiştir. Ağaç dışındaki kenarlarda ise eşitsizlik kesindir.

4.2.3.6 Negatif döngü tespiti

Denetim turunun gücü negatif döngülü bir çizgede ortaya çıkar. Şekil 4.2.11 dört düğümlü bir örnektir. Kaynak S'den A'ya 2 ağırlıklı bir kenar vardır. *A → B → C → A* döngüsünün ağırlığı ise *1 + (−3) + 1 = −1*'dir. Kenar sırası *(S, A), (A, B), (B, C), (C, A)*'dır.



Şekil 4.2.11: Negatif döngülü çizge ve $V - 1 = 3$ turdan sonraki tahminler. Kırmızı döngü $A \to B \to C \to A$'nın ağırlığı −1'dir.

Tur	S	A	B	C
1	0	1	3	0
2	0	0	2	−1
3	0	−1	1	−2

Tablo 4.2.13: Negatif döngülü çizgede tur sonu tahminleri.

Birinci turda A önce $S \to A$ kenarıyla 2 olur. Tur bitmeden döngü dolaşılır: B = 3, C = 0 bulunur ve $C \to A$ kenarı A'yı 1'e indirir. Sonraki her turda döngüdeki üç değer birer azalır. Bu düşüş hiç durmaz, çünkü döngüde her tur atıldığında yol 1 kısalır. Normal bir çizgede değerler en geç $V - 1$ turda sabitlenir; burada sabitlenmezler.

$V - 1 = 3$ turdan sonra $d = [0, -1, 1, -2]$'dir. Denetim turu (S, A) ile başlar: $0 + 2 = 2$, A'nın −1 olan değerinden küçük değildir. Sonra (A, B) gelir: $-1 + 1 = 0 < 1$. Kenar hâlâ gevşiyor; demek ki negatif döngü vardır. Döngüdeki ve döngüden ulaşılan bütün düğümlerin en kısa yolu tanımsızdır (−∞). Bulunan d değerleri bu düğümler için anlamsızdır; yalnız algoritmanın kaçıncı turda durdurulduğunu yansıtır.

Teorem 4.2.3 (Negatif döngü denetimi)

Bellman-Ford'un $V - 1$ turundan sonra bir kenar hâlâ gevşiyorsa, kaynaktan ulaşılabilen bir negatif döngü vardır. Tersine, böyle bir döngü varsa en az bir kenar gevşer.

İspat. Birinci yön önceki teoremden çıkar: negatif döngü olmasaydı $V - 1$ turdan sonra bütün tahminler doğru olurdu ve üçgen eşitsizliği hiçbir kenarın gevşemesine izin vermezdi. Ters yön için ulaşılabilir bir negatif döngü $v_0, v_1, \ldots, v_k = v_0$ alalım ve hiçbir kenarın gevşemediğini varsayalım. O zaman dön-

günün her kenarı için $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ sağlanır. Bu k eşitsizlik toplanınca sol ve sağdaki d toplamları aynı düğümleri içerdiği için birbirini götürür. Geriye $0 \le$ döngünün ağırlığı kalır. Bu, döngünün negatif olmasıyla çelişir. Döngü kaynaktan ulaşılabildiği için tahminler sonludur ve toplama geçerlidir. □

Örnek 4.2.6

Bir döngü $A \to B$ (3), $B \to C$ (−5) ve $C \to A$ (w) kenarlarından oluşuyor ve A kaynaktan ulaşılabiliyor. w tam sayısının hangi değerlerinde denetim turu negatif döngü bildirir?

1. **Döngü ağırlığı.** $3 + (-5) + w = w - 2$.
2. **Negatiflik koşulu.** $w - 2 < 0$, yani $w < 2$. Tam sayılarda $w \le 1$.
3. **Sınır durumu.** $w = 2$'de döngünün ağırlığı 0'dır. Döngüde dolaşmak yolu kısaltmaz; negatif döngü yoktur ve denetim bir şey bildirmez.

4.2.3.7 Sözde kod, Java kodu ve izleme

Bellman-Ford'un sözde kodu iki bölümden oluşur: $V - 1$ turluk dış döngü ve ardından gelen tek bir denetim turu.

```
BELLMAN_FORD(G, kaynak):
  mesafeler = [ ]
  her bir v için V içinde:
    mesafeler[v] = sonsuz
  mesafeler[kaynak] = 0
  döngü i = 1 den |V| - 1:
    her bir (u, v, ağırlık) için E içinde:
      eğer mesafeler[u] + ağırlık < mesafeler[v]:
        mesafeler[v] = mesafeler[u] + ağırlık
  her bir (u, v, ağırlık) için E içinde:
    eğer mesafeler[u] + ağırlık < mesafeler[v]:
      hata "Negatif ağırlıklı döngü var"
  döndür mesafeler
```

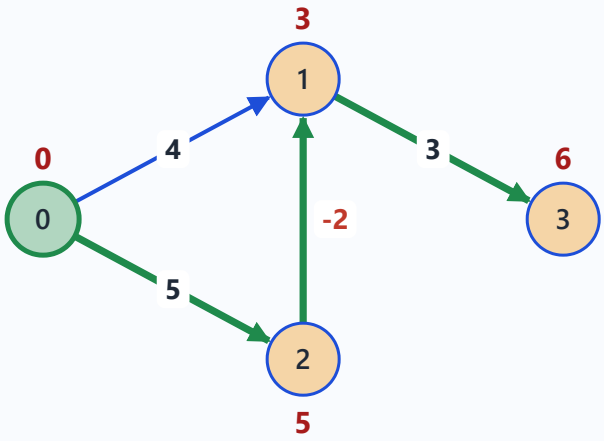
Dış döngü $|V| - 1$ kez döner; iç döngü her turda kenar listesinin tamamını gezer. Son dört satır denetim turudur. Denetim, gevşetmenin koşulunu sorar ama atama yapmaz; koşul doğruysa hata bildirir. Java karşılığında kenarlar `kenarlar` adlı bir listede tutulur. Her `Kenar` nesnesinin `bas`, `son` ve `agirlik` alanları vardır. Düğümler 0'dan `V - 1`'e kadar numaralıdır.

```
void bellmanFord(int kaynak) {
  int[] uzaklik = new int[V];
  int[] onceki = new int[V];
  Arrays.fill(uzaklik, SONSUZ);
  uzaklik[kaynak] = 0;
  for (int i = 1; i < V; i++) {
    for (Kenar k : kenarlar) {
      int u = k.bas, v = k.son, w = k.agirlik;
      if (uzaklik[u] != SONSUZ
          && uzaklik[u] + w < uzaklik[v]) {
        uzaklik[v] = uzaklik[u] + w;
        onceki[v] = u;
      }
    }
  }
  for (Kenar k : kenarlar) { // denetim
    int u = k.bas, v = k.son, w = k.agirlik;
    if (uzaklik[u] != SONSUZ
        && uzaklik[u] + w < uzaklik[v])
      throw new IllegalStateException(
          "Negatif döngü");
  }
}
```

- **Satır 2–5:** Diziler oluşturulur, uzaklıklar sonsuzla doldurulur ve kaynağınki 0 yapılır.
- **Satır 6:** `i` 1'den `V - 1`'e kadar döner; toplam $V - 1$ tur.
- **Satır 7–13:** Her kenar için `u`, `v` ve `w` alınır ve gevşetme yapılır. Koşulun ilk parçası `uzaklik[u] != SONSUZ`'dur. `uzaklik[u]` sonsuzsa toplama hiç yapılmaz.
- **Satır 16–21:** Denetim turu aynı koşulu sorar. Koşul doğruysa `IllegalStateException` fırlatılır.

Koşuldaki `uzaklik[u] != SONSUZ` denetimi süs değildir. `SONSUZ` burada `Integer.MAX_VALUE`, yani 2.147.483.647'dir. Java'da bu sayıya pozitif bir şey eklenince sonuç taşar ve negatif bir sayıya döner: `Integer.MAX_VALUE + 3` işlemi −2.147.483.646 verir. Denetim olmasaydı ulaşılmamış bir düğümden çıkan kenar, bu negatif sayı yüzünden "daha kısa" görünürdü. Dijkstra'nın kodunda bu denetim gerekmiyordu, çünkü kuyruktan yalnız sonlu uzaklıklı düğümler çıkar. Bellman-Ford ise henüz ulaşılmamış düğümlerin kenarlarını da gevşetmeye çalışır.

Tablo 4.2.14 kodun dört düğümlü bir çizgedeki izlemesidir. Kaynak 0, kenar listesi *(2, 1, −2), (0, 1, 4), (0, 2, 5), (1, 3, 3)* sırasındadır.



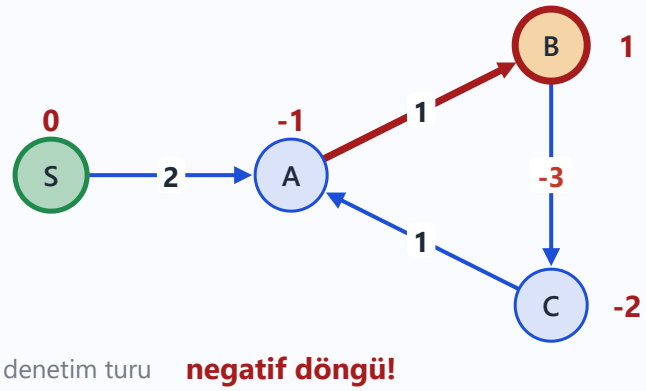
Şekil 4.2.12: `bellmanFord(0)` izlemesinin çizgesi ve sonucu. 1'e en kısa yol, negatif *2 → 1* kenarını kullanır: *0 → 2 → 1*, uzunluk *5 − 2 = 3*.

Tur	Kenar	Koşul	Sonuç	uzaklik
—	—	—	başlangıç	[0, ∞, ∞, ∞]
1	(2, 1, −2)	uzaklik[2] = ∞	atlanır	[0, ∞, ∞, ∞]
1	(0, 1, 4)	0 + 4 < ∞	uzaklik[1] = 4, onceki[1] = 0	[0, 4, ∞, ∞]
1	(0, 2, 5)	0 + 5 < ∞	uzaklik[2] = 5, onceki[2] = 0	[0, 4, 5, ∞]
1	(1, 3, 3)	4 + 3 < ∞	uzaklik[3] = 7, onceki[3] = 1	[0, 4, 5, 7]
2	(2, 1, −2)	5 − 2 = 3 < 4	uzaklik[1] = 3, onceki[1] = 2	[0, 3, 5, 7]
2	(0, 1, 4)	4 ≥ 3	değişmez	[0, 3, 5, 7]
2	(0, 2, 5)	5 ≥ 5	değişmez	[0, 3, 5, 7]
2	(1, 3, 3)	3 + 3 = 6 < 7	uzaklik[3] = 6	[0, 3, 5, 6]
3	dört kenar	hiçbiri doğru değil	değişmez	[0, 3, 5, 6]
denetim	dört kenar	hiçbiri doğru değil	istisna yok	[0, 3, 5, 6]

Tablo 4.2.14: `bellmanFord(0)` izlemesi.

Birinci turda *(2, 1)* kenarı boşa geçer, çünkü listenin başındadır ve incelendiğinde 2'nin değeri hâlâ ∞'dur. İkinci turda bu kenar işe yarar: negatif ağırlık 1'in değerini 4'ten 3'e indirir. Aynı turun sonunda *(1, 3)* kenarı bu iyileşmeyi 3'e taşır: 7'den 6'ya. Üçüncü tur hiçbir şeyi değiştirmez ve denetim turu istisna fırlatmaz. Sonuç `[0, 3, 5, 6]`'dır.

Aynı kod negatif döngülü çizgede çalıştırıldığında farklı biter. S, A, B, C düğümleri 0, 1, 2, 3 numarasını alsın. Kenarlar *(0, 1, 2), (1, 2, 1), (2, 3, −3), (3, 1, 1)* sırasındadır. Birinci turda dört kenarın dördü de gevşer: A = 2, B = 3, C = 0, sonra A = 1. İkinci ve üçüncü turlarda üçer kenar gevşer; döngüdeki değerler her turda bir azalır. Üç tur sonunda `uzaklik = [0, -1, 1, -2]`'dir. Denetimde ilk kenar *(0, 1)* için *0 + 2 = 2*, −1'den küçük değildir. İkinci kenar *(1, 2)* için *−1 + 1 = 0 < 1*'dir; koşul doğrudur ve `IllegalStateException("Negatif döngü")` fırlatılır.



Şekil 4.2.13: Denetim turunda $A \to B$ kenarı hâlâ gevşiyor: $-1 + 1 = 0 < 1$. Kod negatif döngü istisnası fırlatır.

Sık yapılan hata

Denetim turunu atlayıp $V - 1$ turun sonucunu doğrudan kullanmak. Negatif döngülü bir çizgede algoritma yine bir dizi döndürür; dizi sıradan sayılarla doludur ve yanlış olduğu anlaşılmaz. Negatif döngülü örnekteki `[0, -1, 1, -2]` dizisinin ancak denetimle anlamsız olduğu anlaşılır.

4.2.4 Floyd-Warshall algoritması

Dijkstra ve Bellman-Ford tek bir kaynaktan çalışır. Bazı uygulamalar ise bütün düğüm çiftleri arasındaki uzaklıkları ister: bir haritadaki bütün şehirler arasındaki mesafe tablosu ya da bir ağdaki bütün yönlendiricilerin birbirine uzaklığı gibi. Floyd-Warshall algoritması bu tüm çiftler problemini tek seferde çözer. Kodu yalnız üç iç içe döngüden oluşur. Negatif kenarlarla çalışır ve negatif döngüyü de tespit eder.

4.2.4.1 Tüm çiftler problemi

Algoritmanın tarihi ilginçtir. Bernard Roy 1959'da, Stephen Warshall 1962'de ve Robert Floyd yine 1962'de aynı fikri birbirinden bağımsız olarak yayımladı. Warshall'ın makalesi bir ilişkinin geçişli kapanışını, yani hangi düğümün hangisine ulaşabildiğini hesaplıyordu. Floyd aynı yapıyı en kısa yollara uyguladı. Algoritma bugün ikisinin adıyla anılır.

Floyd-Warshall bir dinamik programlama algoritmasıdır. Sonuçlar $V \times V$ boyutlu bir D matrisinde tutulur; $D[i][j]$, i'den j'ye o ana kadar bulunan en kısa uzaklıktır. Yolları geri kurmak için ikinci bir Π matrisi de tutulur. Tek kaynaklı algoritmalardaki π dizisinin her kaynak için ayrı bir satır alan hâlidir.

Algoritma dört adımda çalışır:

1. **İlklendirme.** D kenar ağırlıklarıyla doldurulur: köşegen $D[i][i] = 0$, doğrudan kenar yoksa ∞.

2. **Ara düğüm.** $k = 1, 2, \ldots, n$ sırayla ele alınır. Her k için bütün (i, j) çiftleri gezilir.
3. **Karşılaştırma.** k üzerinden geçen yolun uzunluğu $D[i][k] + D[k][j]$ hesaplanır ve mevcut $D[i][j]$ ile karşılaştırılır.
4. **Güncelleme.** Yeni yol daha kısaysa değer matrise yazılır ve Π matrisi de güncellenir.

Üçüncü adımdaki karşılaştırma, gevşetmenin düğüm çiftleri üzerindeki biçimidir. $i \rightsquigarrow k$ ve $k \rightsquigarrow j$ yolları birleştirilir; ortaya çıkan yol bilinen $i \rightsquigarrow j$ yolundan kısaysa onun yerini alır.

4.2.4.2 Alt problem ve yineleme bağıntısı

Dinamik programlamanın ilk işi alt problemi tanımlamaktır. Floyd-Warshall'da alt problem, bir yolun hangi ara düğümleri kullanabileceğini sınırlar. Bir yolun **ara düğümleri**, başlangıç ve bitiş dışındaki düğümleridir.

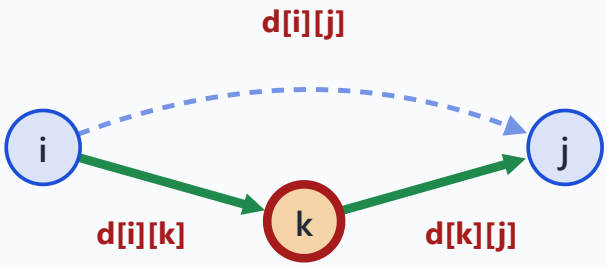
> **Tanım 4.2.4**
>
> $D^{(k)}[i][j]$, ara düğümlerinin hepsi $\{1, 2, \ldots, k\}$ kümesinden seçilen $i \rightsquigarrow j$ yolları arasında en kısasının uzunluğudur. $D^{(0)}$'da hiçbir ara düğüme izin yoktur; bu matris yalnız doğrudan kenarları içerir. $D^{(n)}$'de bütün düğümler ara düğüm olabilir; bu matris aranan sonuçtur.

$D^{(k-1)}$ bilindiğinde $D^{(k)}$ nasıl bulunur? i'den j'ye, ara düğümleri $\{1, \ldots, k\}$'dan gelen bir en kısa yol p düşünelim. İki durum vardır.

- ***k*, *p* üzerinde değildir.** O zaman p'nin bütün ara düğümleri $\{1, \ldots, k-1\}$'dendir ve uzunluğu $D^{(k-1)}[i][j]$'dir.
- ***k*, *p* üzerindedir.** Negatif döngü yoksa p basit bir yoldur ve k'den yalnız bir kez geçer. p, $i \rightsquigarrow k$ ve $k \rightsquigarrow j$ parçalarına ayrılır. İki parçanın ara düğümleri de $\{1, \ldots, k-1\}$'dendir. Optimal alt yapı gereği her parça kendi başına en kısadır. Uzunluk $D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$'dir.

En kısa yol bu iki seçenekten küçük olanıdır:

$$D^{(k)}[i][j] = \min\left(D^{(k-1)}[i][j],\ D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \right)$$



Şekil 4.2.14: Floyd-Warshall bağıntısı. Kesikli ok bilinen $i \rightsquigarrow j$ yoludur. Yeşil oklar k üzerinden geçen aday yoldur: $D[i][k] + D[k][j]$. İkisinden kısa olan saklanır.

Bağıntı, k'nin ya yolda olmadığını ya da yolun k'den tam bir kez geçtiğini söyler. Ara düğüm kümesi her adımda bir düğüm büyür: önce yalnız 1 kullanılabilir, sonra 1 ve 2, en sonunda bütün düğümler. n adımın sonunda her çift için bütün olası yollar hesaba katılmış olur.

4.2.4.3 Karmaşıklık ve döngü sırası

Algoritmanın süresi kod yapısından okunur. Üç iç içe `for` döngüsü vardır ve her biri V kez döner. En dış döngü ara düğümleri (k), ikinci döngü kaynak düğümleri (i), en iç döngü hedef düğümleri (j) gezer. En içteki karşılaştırma $O(1)$ sürer ve V^3 kez yapılır. Süre çizgenin kenar sayısından bağımsızdır: $\Theta(V^3)$. Bu, en iyi ve en kötü durumun aynı olduğu, sıkı bir sınırdır.

Bellek $\Theta(V^2)$'dir. Bağıntıya bakınca her k için yeni bir matris gerekiyormuş gibi görünür. Gerekmez: D yerinde güncellenebilir. k. turda k. satır ve k. sütun değişmez. Örneğin $D[i][k]$ için aday $D[i][k] + D[k][k] = D[i][k] + 0$'dır ve mevcut değeri geçemez. Bu turda okunan bütün $D[i][k]$ ve $D[k][j]$ değerleri bu yüzden $D^{(k-1)}$'deki değerleriyle aynıdır.

Yöntem	Zaman
Floyd-Warshall	$O(V^3)$
$V \times$ Dijkstra (dizi)	$O(V^3)$
$V \times$ Dijkstra (heap)	$O(V(V + E) \log V)$
$V \times$ Bellman-Ford	$O(V^2 E)$

Tablo 4.2.15: Tüm çiftler arası en kısa yollar için seçenekler.

Tablo, Floyd-Warshall'ın her zaman en hızlı seçenek olmadığını gösterir. Seyrek ve negatif kenarsız bir çizgede her düğümden bir kez heap'li Dijkstra çalıştırmak daha hızlıdır. Örneğin $V = 200$ ve $E = 800$ için $V^3 = 8 \cdot 10^6$ iken $V(V + E) \log_2 V \approx 1{,}5 \cdot 10^6$'dır. Floyd-Warshall'ın üstünlüğü başka yerdedir. Negatif kenarlarla çalışır, kodu çok kısadır ve sabit çarpanı küçüktür. Yoğun çizgelerde ve birkaç yüz düğüme kadar olan problemlerde pratik bir seçimdir. Negatif kenarlı bir çizgede V kez Bellman-Ford çalıştırmak ise $O(V^2 E)$ tutar; yoğun çizgede bu $O(V^4)$'tür.

Döngülerin sırası keyfî değildir: k en dışta olmalıdır. Bağıntı $D^{(k)}$'yı $D^{(k-1)}$'den kurar. k. tur başladığında bütün çiftler $\{1, \ldots, k-1\}$ ara düğümleriyle hesaplanmış olmalıdır. Döngüler `i`, `j`, `k` sırasına getirilirse her (i, j) hücresi bir kez, diğer hücreler hazır olmadan hesaplanır.

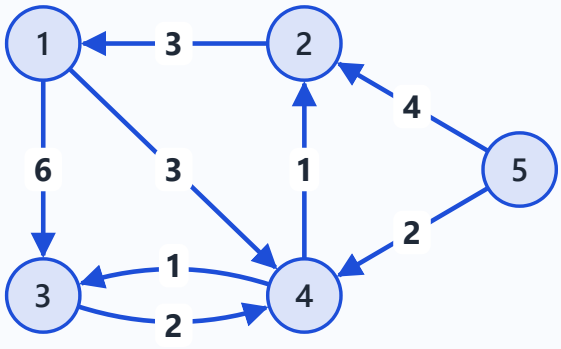
Örnek 4.2.7

Aşağıdaki Floyd-Warshall örneğinin çizgesinde döngüler `i, j, k` sırasıyla (k en içte) çalıştırılırsa ne olur?

1. **3 ⇝ 1 hücresi.** Doğru sonuç $D[3][1] = 6$'dır; yol $3 \to 4 \to 2 \to 1$'dir ve iki ara düğüm kullanır.
2. **Hesap anı.** Yanlış sırada $D[3][1]$, satır 3'ün ilk hücresi olarak tek bir kez hesaplanır. O anda yalnız doğrudan kenarlar bilinir: $D[3][4] = 2$'dir, ama $D[4][1]$ henüz ∞'dur; çünkü satır 4 daha sonra işlenecektir. $D[3][2]$ de henüz hesaplanmamıştır ve ∞'dur.
3. **Sonuç.** Hiçbir k için $D[3][k] + D[k][1]$ sonlu çıkmaz ve hücre ∞ kalır. Python ile yapılan bir denemede bu sıra, matrisin diğer bütün hücrelerini doğru bulup yalnız bu hücreyi kaçırmıştır. Döngü sayısı değişmez, süre yine $\Theta(V^3)$'tür; değişen yalnız sonucun doğruluğudur.

4.2.4.4 Örnek: D ve Π matrisleri

Örnek çizge, Dijkstra Örnek 2 ve Bellman-Ford Örnek 3'te kullanılan beş düğümlü yönlü çizgedir. Bu kez bir kaynak seçilmez; bütün çiftler hesaplanır. Π matrisinde $\Pi[i][j]$, i'den j'ye en kısa yolda j'den önceki düğümdür. Başlangıçta doğrudan kenar varsa $\Pi[i][j] = i$, yoksa "/" yazılır.



Şekil 4.2.15: Floyd-Warshall örneğinin çizgesi. Kenarlar: $1 \to 3$ (6), $1 \to 4$ (3), $2 \to 1$ (3), $3 \to 4$ (2), $4 \to 2$ (1), $4 \to 3$ (1), $5 \to 2$ (4), $5 \to 4$ (2).

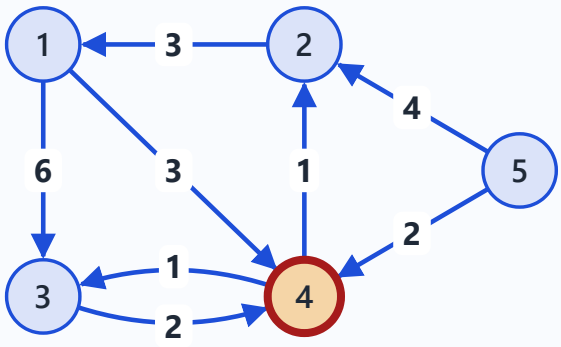
	1	2	3	4	5
1	0 / –	∞ / –	6 / 1	3 / 1	∞ / –
2	3 / 2	0 / –	∞ / –	∞ / –	∞ / –
3	∞ / –	∞ / –	0 / –	2 / 3	∞ / –

	1	2	3	4	5
4	∞ / –	1 / 4	1 / 4	0 / –	∞ / –
5	∞ / –	4 / 5	∞ / –	2 / 5	0 / –

Tablo 4.2.16: $D^{(0)}$ ve $\Pi^{(0)}$ (ilklendirme). Satır i, sütun j; her hücrede "D / Π"; Π tanımsızsa "–" yazılmıştır.

Ara düğümler sırayla eklendikçe matris değişir. Her turda değişen hücreler aşağıda listelenmiştir.

- ***k = 1.*** 1'e gelen tek kenar $2 \to 1$'dir; bu yüzden yalnız 2. satır değişebilir. $2 \rightsquigarrow 3$: $D[2][1] + D[1][3] = 3 + 6 = 9$. $2 \rightsquigarrow 4$: $3 + 3 = 6$. İki hücrenin Π değeri $\Pi[1][j] = 1$ olur.
- ***k = 2.*** 2'ye gelen kenarlar 4'ten ve 5'ten gelir. $4 \rightsquigarrow 1$: $1 + 3 = 4$. $5 \rightsquigarrow 1$: $4 + 3 = 7$. $5 \rightsquigarrow 3$: $4 + 9 = 13$. Son hücre, bir önceki turda bulunan $2 \rightsquigarrow 3$ yolunu kullanır; bu yol zaten 1 üzerinden geçer. Böylece $5 \rightsquigarrow 3$ yolu iki ara düğüm kullanır: $5 \to 2 \to 1 \to 3$.
- ***k = 3.*** 3'ten çıkan tek kenar $3 \to 4$'tür ve 3 üzerinden hiçbir yol kısalmaz.
- ***k = 4.*** 4 çizgenin merkezidir; sekiz hücre kısalır: $1 \rightsquigarrow 2$ (4), $1 \rightsquigarrow 3$ (6 → 4), $2 \rightsquigarrow 3$ (9 → 7), $3 \rightsquigarrow 1$ (6), $3 \rightsquigarrow 2$ (3), $5 \rightsquigarrow 1$ (7 → 6), $5 \rightsquigarrow 2$ (4 → 3), $5 \rightsquigarrow 3$ (13 → 3). Örneğin $5 \rightsquigarrow 3$ için $D[5][4] + D[4][3] = 2 + 1 = 3 < 13$'tür.
- ***k = 5.*** 5'e hiçbir kenar gelmez; bütün $i \neq 5$ için $D[i][5] = \infty$'dur. 5 üzerinden hiçbir yol kısalmaz.



Şekil 4.2.16: $k = 4$ turu. Kalın çerçeveli ara düğüm 4, kendisine gelen ve kendisinden çıkan kenarlar sayesinde sekiz yolu birden kısaltır.

	1	2	3	4	5
1	0 / –	4 / 4	4 / 4	3 / 1	∞ / –
2	3 / 2	0 / –	7 / 4	6 / 1	∞ / –
3	6 / 2	3 / 4	0 / –	2 / 3	∞ / –
4	4 / 2	1 / 4	1 / 4	0 / –	∞ / –
5	6 / 2	3 / 4	3 / 4	2 / 5	0 / –

Tablo 4.2.17: Son durum $D^{(5)}$ ve $\Pi^{(5)}$; her hücrede "D / Π"; Π tanımsızsa "–" yazılmıştır.

Son matristen birkaç sonuç okunur. 5. sütun 5'in kendisi dışında baştan sona ∞'dur: 5'e hiçbir düğümden ulaşılamaz. 5. satır *[6, 3, 3, 2, 0]*'dır. Bu, Dijkstra'nın ve Bellman-Ford'un kaynak 5 için bulduğu uzaklık dizisinin tıpatıp aynısıdır. Floyd-Warshall bir kaynak seçmeden çalıştığı hâlde, matrisin her satırı o satırdaki düğümün tek kaynaklı sonucunu verir. Köşegenin hepsi 0'dır; negatif döngü yoktur.

4.2.4.5 Yolun okunması

Π[i][j], *i* ⇝ *j* en kısa yolunda *j*'den önceki düğümdür. Yol, tek kaynaklı durumdaki gibi hedeften geriye okunur. Fark, bütün okumanın *i*. satırda yapılmasıdır: *j*'den önceki düğüm *Π[i][j]*, ondan önceki *Π[i][Π[i][j]]*, ve böyle sürer.

Örnek 4.2.8

Son *Π* matrisinden *5* ⇝ *1* ve *1* ⇝ *2* yolları okunacak.

1. ***5* ⇝ *1*.** 5. satırda okunur: *Π[5][1] = 2*, *Π[5][2] = 4*, *Π[5][4] = 5*. 5 kaynaktır, okuma durur. Yol *5 → 4 → 2 → 1*, uzunluğu *D[5][1] = 6*'dır.
2. ***1* ⇝ *2*.** 1. satırda okunur: *Π[1][2] = 4*, *Π[1][4] = 1*. Yol *1 → 4 → 2*, uzunluğu *3 + 1 = 4 = D[1][2]*.
3. **Denetim.** *3* ⇝ *1*: *Π[3][1] = 2*, *Π[3][2] = 4*, *Π[3][4] = 3*. Yol *3 → 4 → 2 → 1*, uzunluğu *2 + 1 + 3 = 6 = D[3][1]*.

Π güncellemesinin kuralı da buradan anlaşılır. *k* üzerinden daha kısa bir yol bulunduğunda *Π[i][j] ← Π[k][j]* yapılır. Yeni yolun son parçası *k* ⇝ *j* yoludur. Bu yüzden *j*'den önceki düğüm, *k* ⇝ *j* yolunda *j*'den önceki düğümle aynıdır. *Π[i][j] ← k* yazmak yanlış olurdu; *k*, *j*'ye doğrudan bağlı olmayabilir.

4.2.4.6 Sözde kod, Java kodu ve izleme

Sözde kod iki bölümdür. Birinci bölüm matrisi ilklendirir.

```
FLOYD_WARSHALL(G):
  mesafeler = [ ][ ]
  her bir u için V içinde:
    her bir v için V içinde:
      eğer u == v:
        mesafeler[u][v] = 0
      yoksa eğer (u, v) E içinde:
        mesafeler[u][v] = ağırlık(u, v)
      yoksa:
        mesafeler[u][v] = sonsuz
```

İkinci bölüm üç iç içe döngüdür; ardından köşegen denetlenir.

```
  her bir k için V içinde:
    her bir i için V içinde:
      her bir j için V içinde:
        eğer m[i][k] + m[k][j] < m[i][j]:
          m[i][j] = m[i][k] + m[k][j]
  her bir i için V içinde:
    eğer mesafeler[i][i] < 0:
      hata "Negatif ağırlıklı döngü var"
  döndür mesafeler
```

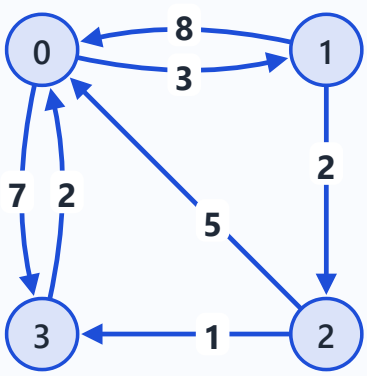
Negatif döngü denetimi Bellman-Ford'unkinden daha basittir. Negatif bir döngü üzerindeki bir *i* düğümü, döngüyü dolaşarak kendine negatif uzunlukta bir yol bulur. Algoritma bu yolu da hesaplar ve *D[i][i]* 0'ın altına iner. Bu yüzden köşegende negatif bir değer görülmesi negatif döngünün işaretidir.

Java kodunda matris yerinde güncellenir; ayrı bir $D^{(k)}$ kopyası tutulmaz. `onceki` matrisi sınıfın bir alanıdır ve başlangıçta `onceki[i][j] = i` değerlerini taşır.

```
void floydWarshall(int[][] d) {  // V x V
  for (int k = 0; k < V; k++)       // ara
    for (int i = 0; i < V; i++)     // kaynak
      for (int j = 0; j < V; j++)   // hedef
        if (d[i][k] != SONSUZ
            && d[k][j] != SONSUZ
            && d[i][k] + d[k][j] < d[i][j]) {
          d[i][j] = d[i][k] + d[k][j];
          onceki[i][j] = onceki[k][j];
        }
  for (int i = 0; i < V; i++)
    if (d[i][i] < 0)
      throw new IllegalStateException(
          "Negatif döngü");
}
```

- **Satır 2–4:** Üç döngü; açıklamalar her döngünün rolünü yazar: ara, kaynak, hedef.
- **Satır 5–7:** Önce iki parçanın da sonlu olduğu denetlenir. `SONSUZ` `Integer.MAX_VALUE` olduğu için iki sonsuzun ya da sonsuzla bir sayının toplamı taşar ve negatif bir sayıya dönebilir. Denetim bu taşmayı önler.
- **Satır 8–9:** Yeni uzunluk yazılır. `onceki[i][j] = onceki[k][j]` : yeni yolda *j*'den önceki düğüm, $k \rightsquigarrow j$ yolundakiyle aynıdır.
- **Satır 11–14:** Köşegende negatif değer varsa istisna fırlatılır.

Kod dört düğümlü bir çizgede izlenir. Düğümler 0–3 numaralıdır. Kenarlar $0 \to 1$ (3), $0 \to 3$ (7), $1 \to 0$ (8), $1 \to 2$ (2), $2 \to 0$ (5), $2 \to 3$ (1), $3 \to 0$ (2)'dir.



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

Şekil 4.2.17: `floydWarshall` izlemesinin çizgesi ve dört tur sonundaki `d` ve `onceki` matrisleri.

k	Hücre	Hesap	Yeni d	Yeni onceki
0	d[1][3]	d[1][0] + d[0][3] = 8 + 7 = 15 < ∞	15	onceki[0][3] = 0
0	d[2][1]	d[2][0] + d[0][1] = 5 + 3 = 8 < ∞	8	onceki[0][1] = 0
0	d[3][1]	d[3][0] + d[0][1] = 2 + 3 = 5 < ∞	5	onceki[0][1] = 0
1	d[0][2]	d[0][1] + d[1][2] = 3 + 2 = 5 < ∞	5	onceki[1][2] = 1
1	d[3][2]	d[3][1] + d[1][2] = 5 + 2 = 7 < ∞	7	onceki[1][2] = 1
2	d[0][3]	d[0][2] + d[2][3] = 5 + 1 = 6 < 7	6	onceki[2][3] = 2
2	d[1][0]	d[1][2] + d[2][0] = 2 + 5 = 7 < 8	7	onceki[2][0] = 2
2	d[1][3]	d[1][2] + d[2][3] = 2 + 1 = 3 < 15	3	onceki[2][3] = 2
3	d[1][0]	d[1][3] + d[3][0] = 3 + 2 = 5 < 7	5	onceki[3][0] = 3
3	d[2][0]	d[2][3] + d[3][0] = 1 + 2 = 3 < 5	3	onceki[3][0] = 3
3	d[2][1]	d[2][3] + d[3][1] = 1 + 5 = 6 < 8	6	onceki[3][1] = 0

Tablo 4.2.18: `floydWarshall` izlemesinde değişen hücreler (sırasıyla).

On bir güncelleme yapılır ve sekiz hücre değişir; bazı hücreler birden çok kez kısalır: `d[1][0]` önce 8'dir, *k = 2*'de 7'ye, *k = 3*'te 5'e iner. Son yol *1 → 2 → 3 → 0* yoludur ve *2 + 1 + 2 = 5* tutar. `d[1][3]` ise önce 15 olur, sonra 3'e düşer. 0 üzerinden bulunan uzun yol, 2 eklenince yerini çok daha kısa bir yola bırakır.

Dört tur bittiğinde köşegende negatif değer yoktur ve istisna fırlatılmaz. Yolu okumak için yine `onceki` kullanılır. *1 ⇝ 3*: `onceki[1][3] = 2`, `onceki[1][2] = 1`. Yol *1 → 2 → 3*, uzunluğu `d[1][3] = 3` 'tür. Sonuç matrisi `d = [[0, 3, 5, 6], [5, 0, 2, 3], [3, 6, 0, 1], [2, 5, 7, 0]]` 'dir.

Örnek 4.2.9

Kenarları *1 → 2* (4), *2 → 3* (1), *1 → 3* (7) ve *3 → 1* (−2) olan çizgenin son *D* matrisi bulunacak. Negatif döngü var mı?

1. ***k = 1.*** 1'e gelen kenar $3 \to 1$'dir. $3 \rightsquigarrow 2$: $-2 + 4 = 2$. $3 \rightsquigarrow 3$: $-2 + 7 = 5$, 0'dan büyük, değişmez.
2. ***k = 2.*** $1 \rightsquigarrow 3$: $4 + 1 = 5 < 7$. $3 \rightsquigarrow 3$: $2 + 1 = 3$, değişmez.
3. ***k = 3.*** $1 \rightsquigarrow 1$: $5 + (-2) = 3$, değişmez. $2 \rightsquigarrow 1$: $1 + (-2) = -1$. $2 \rightsquigarrow 2$: $1 + 2 = 3$, değişmez.
4. **Sonuç.** $D = [[0, 4, 5], [-1, 0, 1], [-2, 2, 0]]$. Köşegen tümüyle 0'dır; $1 \to 2 \to 3 \to 1$ döngüsünün ağırlığı $4 + 1 - 2 = 3$, $1 \to 3 \to 1$ döngüsününki $7 - 2 = 5$'tir. Negatif kenar vardır, negatif döngü yoktur.

Floyd-Warshall'ın yapısı en kısa yol dışında da kullanılır. Ağırlıklar yerine yalnız "kenar var mı?" bilgisi tutulursa, toplama yerine "ve", en küçük yerine "veya" yazılır. Ortaya çıkan algoritma, Warshall'ın asıl problemi olan **geçişli kapanışı** (*transitive closure*) hesaplar: *t[i][j]*, *i*'den *j*'ye bir yol olup olmadığını söyler.

```
boolean[][] kapanis(boolean[][] t) {
  int V = t.length;
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        t[i][j] = t[i][j] || (t[i][k] && t[k][j]);
  return t;
}
```

Kod yine $\Theta(V^3)$ sürer ve *k* yine en dışta durur. Kenarları $0 \to 1$, $1 \to 2$, $2 \to 0$ ve $3 \to 2$ olan dört düğümlü bir çizgede bütün satırlar `1110` çıkar. 0, 1 ve 2 aynı döngüdedir ve birbirine ulaşır. 3 de 2 üzerinden bu döngüye girer. 3'e ise hiçbir düğümden ulaşılamaz; bu yüzden son sütun baştan sona 0'dır.

4.2.5 A* arama

Dijkstra kaynaktan her yöne eşit hızla yayılır. Yalnız tek bir hedefe gidilecekse bu yayılmanın büyük kısmı boşa gider. İstanbul'dan Ankara'ya yol arayan bir Dijkstra, Edirne'ye ve Bursa'ya giden yolları da Ankara'yı bulana kadar inceler. A* (A yıldız) araması bu israfı azaltır. Her düğüm için hedefe ne kadar kaldığını tahmin eder ve aramayı hedefe doğru yöneltir.

Peter Hart, Nils Nilsson ve Bertram Raphael A*'ı 1968'de Stanford Araştırma Enstitüsü'nde geliştirdi. Amaçları, gezgin bir robotun yolunu planlamaktı. Bugün oyunlarda karakterlerin yol bulmasında, harita uygulamalarında ve robotikte en çok kullanılan arama yöntemlerinden biridir.

4.2.5.1 f = g + h

A*, iki aramanın güçlü yanlarını birleştirir. Dijkstra (yapay zekâ kitaplarında düzgün maliyetli arama diye de geçer) yalnız kaynaktan gelinen gerçek maliyete bakar. Bu yüzden doğrudur, ama hedefin nerede olduğunu bilmez. **Açgözlü en iyi ilk arama** (*greedy best-first search*) ise yalnız hedefe kalan tah-

mini maliyete bakar. Hedefe hızla koşar, ama gelinen yolun pahalı olup olmadığını önemsemez. A* her düğüm için üç değer kullanır:

- ***g(n)*, gerçek maliyet:** başlangıç düğümünden *n*'ye kadar bulunan yolun maliyetidir. Dijkstra'daki *d[n]* ile aynıdır.
- ***h(n)*, sezgisel tahmin:** *n*'den hedefe kalan maliyetin tahminidir. Haritada kuş uçuşu uzaklık, ızgarada Manhattan uzaklığı böyle tahminlerdir.
- ***f(n) = g(n) + h(n)*:** *n* üzerinden geçen bir çözümün tahmini toplam maliyetidir. A* her adımda *f* değeri en küçük olan düğümü genişletir.

Tanım 4.2.5

Sezgisel fonksiyon (*heuristic*), her *n* düğümüne hedefe kalan maliyetin bir tahmini olan $h(n) \geq 0$ sayısını atar. $h^*(n)$, *n*'den hedefe gerçek en kısa uzaklık olsun. Sezgisel tahminin kesin olması gerekmez; ama ne kadar isabetliyse arama o kadar az düğüm açar.

Seçim ölçütü	Algoritma	Davranış
yalnız g(n)	Dijkstra	her yöne eşit yayılır; doğru ama çok düğüm açar
yalnız h(n)	açgözlü en iyi ilk	hedefe koşar; hızlı ama en kısa yolu garanti etmez
g(n) + h(n)	A*	hedefe yönelir; h uygunsa en kısa yolu garanti eder

Tablo 4.2.19: Seçim ölçütüne göre arama davranışı.

Toplamın işe yarama nedeni şudur. Hedefe yakın görünen düğümlerin *h* değeri küçüktür ve bunlar öne geçer. Ama pahalı bir yoldan gelinmişse *g* büyüktür ve düğüm geri planda kalır. İki terim birbirini dengeler: biri "hedefe ne kadar kaldı?", öteki "buraya kaça geldik?" sorusunu cevaplar.

Örnek 4.2.10

Açık listede dört düğüm var; *(g, h)* değerleri: p (4, 7), q (6, 3), r (2, 10), u (8, 2). Hangi düğüm genişletilir?

1. **f değerleri.** $f(p) = 11, f(q) = 9, f(r) = 12, f(u) = 10$.
2. **Seçim.** En küçük *f* q'nundur; A* q'yu genişletir.
3. **Karşılaştırma.** Yalnız *g*'ye bakan Dijkstra r'yi (2), yalnız *h*'ye bakan açgözlü arama u'yu (2) seçerdi. A*'ın seçimi ikisinden de farklıdır.

4.2.5.2 Algoritmanın adımları

A* iki liste kullanır. **Açık liste** (*open list*), keşfedilmiş ama henüz genişletilmemiş düğümleri tutan bir öncelikli kuyruktur. **Kapalı liste** (*closed list*), genişletilmiş düğümleri tutar. Bir düğümü **genişletmek**, onu açık listeden çıkarıp komşularını incelemek demektir.

1. Başlangıç düğümünün g değeri 0 yapılır ve düğüm açık listeye eklenir. Diğer düğümlerin g değeri ∞'dur.
2. Açık listeden $f = g + h$ değeri en küçük düğüm çıkarılır ve kapalı listeye konur. Seçim ölçütü g ya da h tek başına değil, toplamlarıdır.
3. Çıkarılan düğüm hedefse çözüm bulunmuştur; yol önceki işaretçileriyle geri kurulur. Hedef açık listeye eklendiği anda değil, oradan çıkarıldığı anda kesinleşir.
4. Değilse her komşu v için $g' = g(u) + w(u, v)$ hesaplanır. g' daha küçükse g, f ve önceki değerleri güncellenir ve komşu açık listeye eklenir. Bu, Dijkstra'daki gevşetmenin aynısıdır.
5. İkinci adıma dönülür. Açık liste boşalırsa hedefe yol yoktur.

İkinci adımda seçim, açık listenin tamamından yapılır; yalnız son genişletilen düğümün komşularından yapılmaz. A* bu sayede bir kolda ilerlerken o kol pahalılaşırsa başka bir kola geçebilir. Çizge örneğinde tam olarak bu olur.

Sık yapılan hata

Hedef açık listeye eklendiği anda aramayı durdurmak. Eklendiği anda hedefin g değeri yalnız bir üst sınırdır; daha ucuz bir yol açık listede bekliyor olabilir. Dijkstra'da da bir düğümün uzaklığı kuyruğa girdiğinde değil, kuyruktan çıktığında kesinleşir. A* hedef için aynı kuralı uygular.

4.2.5.3 Sezgisel fonksiyonun özellikleri

A*'ın en kısa yolu bulup bulmayacağı sezgisele bağlıdır. İki özellik önemlidir.

Tanım 4.2.6

Her n için $h(n) \le h^*(n)$ ise h **kabul edilebilirdir** (*admissible*): gerçek kalan maliyeti hiçbir zaman aşmaz.

Her (u, v) kenarı için $h(u) \le w(u, v) + h(v)$ ve $h(hedef) = 0$ ise h **tutarlıdır** (*consistent*).

Teorem 4.2.4 (A*'ın en iyiliği)

h kabul edilebilirse, A* hedefi açık listeden çıkardığında bulduğu yol en kısa yoldur.

İspat. Hedef t, $g(t) > h^*(s)$ olan, yani en kısa olmayan bir yolla çıkarılmış olsun. P, s'den t'ye bir en kısa yol olsun. t çıkarılmadan önce P üzerindeki düğümlerden biri, n, açık listededir ve $g(n)$ değeri doğrudur; bu, s'nin ilk genişletmesinden başlayan basit bir tümevarımla gösterilir. h kabul edilebilir olduğu

için $f(n) = g(n) + h(n) \le g(n) + h^*(n) = h^*(s)$'dir. Hedefte $h(t) = 0$ olduğu için $f(t) = g(t) > h^*(s) \ge f(n)$ olur. t'nin seçildiği anda açık listede f değeri daha küçük olan n vardır. A* her zaman en küçük f'li düğümü seçtiğine göre bu mümkün değildir. Çelişki. □

Tutarlılık daha güçlü bir özelliktir. Tutarlı bir sezgisel her zaman kabul edilebilirdir. Tutarlı h ile bir düğüm kapalı listeye girdiğinde g değeri kesinleşir ve düğüm bir daha açılmaz. Bu, Dijkstra'daki kesinleşmenin tam karşılığıdır. Gerçekten de tutarlı bir h ile A*, kenar ağırlıkları $w(u, v) - h(u) + h(v) \ge 0$ olan bir çizgede çalışan Dijkstra'dır. Kabul edilebilir ama tutarsız bir sezgiselde ise bir düğüm sonradan daha ucuz bir yolla yeniden açılabilir.

Örnek 4.2.11

Zincir çizge $s \to a \to t$; $w(s, a) = 2$, $w(a, t) = 3$. İki sezgisel incelenecek: (1) $h(s) = 4$, $h(a) = 4$, $h(t) = 0$; (2) $h(s) = 4$, $h(a) = 1$, $h(t) = 0$.

1. **Gerçek kalanlar.** $h^*(s) = 5$, $h^*(a) = 3$, $h^*(t) = 0$.
2. **Birinci sezgisel.** $h(a) = 4 > 3 = h^*(a)$: gerçek kalanı aşar. Kabul edilebilir değildir.
3. **İkinci sezgisel, kabul edilebilirlik.** $4 \le 5$, $1 \le 3$, $0 \le 0$: kabul edilebilirdir.
4. **İkinci sezgisel, tutarlılık.** (s, a) kenarında $h(s) \le w(s, a) + h(a)$ sorulur: $4 \le 2 + 1 = 3$ yanlıştır. Sezgisel kabul edilebilir ama tutarlı değildir. s'den a'ya geçince tahmin 2 birimlik kenardan fazla, 3 birim birden düşer.

Izgara üzerindeki aramalarda yaygın sezgiseller hareket kuralına göre seçilir:

- **4 yönlü hareket** (yukarı, aşağı, sağ, sol; her adım 1): **Manhattan uzaklığı** $|\Delta x| + |\Delta y|$. Her adım bu toplamı en çok 1 azaltır; bu yüzden $h \le h^*$'dır.
- **8 yönlü hareket** (düz adım 10, çapraz adım 14): **oktil uzaklık** (*octile distance*) $14 \cdot \min(|\Delta x|, |\Delta y|) + 10 \cdot (\max - \min)$. Önce mümkün olduğu kadar çapraz, sonra düz gidilir. 14, $10 \cdot \sqrt{2} \approx 14{,}14$ değerinin tam sayıya yuvarlanmış hâlidir.
- **Serbest düzlem: Öklid uzaklığı** $\sqrt{(\Delta x^2 + \Delta y^2)}$, yani kuş uçuşu.

Manhattan uzaklığı 8 yönlü harekette kabul edilebilir değildir. Çapraz bir adım $|\Delta x| + |\Delta y|$ toplamını 2 azaltır, ama maliyeti yalnız 1'dir (ya da 1,4'tür). Örneğin $\Delta x = \Delta y = 3$ olan bir hedef için Manhattan uzaklığı 6'dır. Her adımı 1 olan 8 yönlü ızgarada gerçek kalan maliyet ise üç çapraz adımla yalnız 3'tür. Sezgisel gerçek değeri iki katına çıkarır.

Sezgiselin kalitesi iki uç durumla anlaşılır. $h = 0$ alınırsa $f = g$ olur ve A* tam olarak Dijkstra'ya dönüşür. Bu sezgisel kabul edilebilirdir, ama aramayı hiç yönlendirmez. $h = h^*$ (mükemmel tahmin) alınırsa en kısa yol üzerindeki her düğümün f değeri $h^*(s)$'ye eşit olur. Yol dışındaki düğümlerin f'si ise daha büyüktür. Eşitlikler uygun biçimde bozulursa yalnız yol üzerindeki düğümler açılır. $h > h^*$ olan, yani fazla tahmin eden bir sezgisel aramayı hızlandırabilir, ama en kısa yol garantisini kaldırır.

4.2.5.4 Karmaşıklık

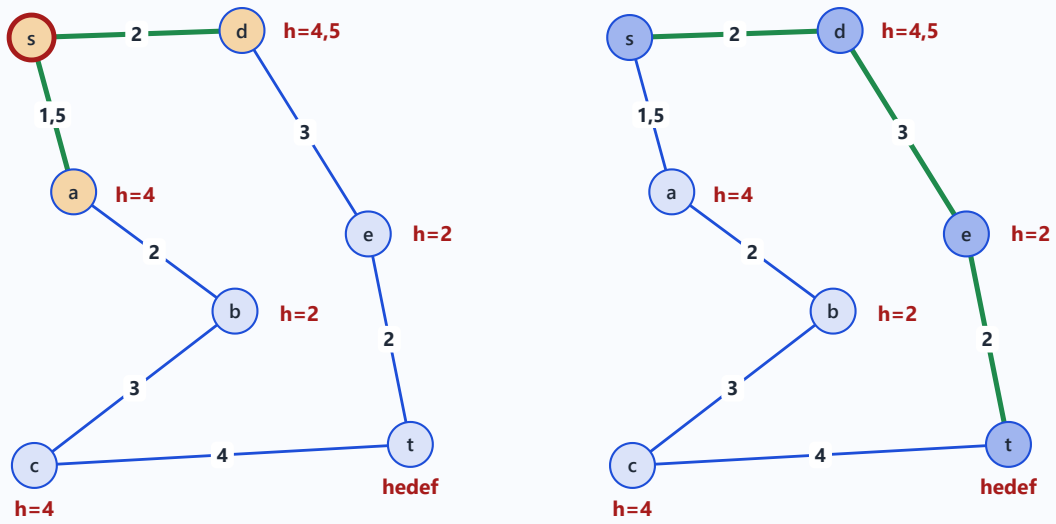
A*'ın maliyeti arama ağacı üzerinden anlatılır. b **dallanma faktörü**, yani bir düğümün ortalama komşu sayısı olsun. d ise hedefin derinliği, yani çözümdeki adım sayısı olsun. En kötü durumda arama, derinliği d'ye kadar olan bütün düğümleri açar. Zaman ve bellek bu yüzden $O(b^d)$'dir. Bu sınır işe yaramayan bir sezgiselin, örneğin $h = 0$'ın sınırıdır. Mükemmel bir tahminde arama yol uzunluğuyla doğrusaldır: yalnız yol üzerindeki d düğüm genişletilir.

Sonlu bir çizgede tutarlı bir h ile A*, Dijkstra'dan fazla iş yapmaz. Her düğüm en çok bir kez genişletilir ve toplam süre ikili heap ile $O((V + E) \log V)$'dir. Pratikte iyi bir sezgisel bu sınırın çok altında kalır, çünkü hedefin ters yönündeki düğümler hiç açılmaz.

A*'ın asıl zayıf noktası bellektir. Açık ve kapalı listeler aramayla birlikte büyür ve büyük haritalarda belleği doldurabilir. Bu sorunu çözmek için bellek sınırlı türevler geliştirilmiştir. IDA* (*iterative deepening A**), derinlik öncelikli aramayı artan bir f sınırıyla tekrarlar ve yalnız o anki yolu bellekte tutar. SMA* (*simplified memory-bounded A**) ise bellek dolunca en kötü görünen düğümleri listeden atar.

4.2.5.5 Örnek: çizgede A*

Örnek yönsüz bir çizgedir; başlangıç s, hedef t'dir. Kenarlar: s–a 1,5; s–d 2; a–b 2; b–c 3; c–t 4; d–e 3; e–t 2. Sezgisel değerler: $h(s) = 5$, $h(a) = 4$, $h(b) = 2$, $h(c) = 4$, $h(d) = 4{,}5$, $h(e) = 2$, $h(t) = 0$.



Şekil 4.2.18: A* örneği. Solda ilk genişletmeden sonra: s kapalı, a ve d açık. Sağda son durum: en kısa yol $s \to d \to e \to t$ (7). c hiç genişletilmez.

Adım	Genişletilen (f)	Açılan / güncellenen	Açık liste: düğüm (g, h, f)	Kapalı
1	s (5)	a: g = 1,5; d: g = 2	a (1,5; 4; 5,5), d (2; 4,5; 6,5)	s
2	a (5,5)	b: g = 3,5	b (3,5; 2; 5,5), d (2; 4,5; 6,5)	s, a

Adım	Genişletilen (f)	Açılan / güncellenen	Açık liste: düğüm (g, h, f)	Kapalı
3	b (5,5)	c: g = 6,5	d (2; 4,5; 6,5), c (6,5; 4; 10,5)	s, a, b
4	d (6,5)	e: g = 5	e (5; 2; 7), c (6,5; 4; 10,5)	s, a, b, d
5	e (7)	t: g = 7	t (7; 0; 7), c (6,5; 4; 10,5)	s, a, b, d, e
6	t (7)	hedef çıkarıldı, dur	c (6,5; 4; 10,5)	s, a, b, d, e, t

Tablo 4.2.20: A* örneğinde açık ve kapalı listeler. Açık listede f küçükten büyüğe sıralıdır.

Arama önce üst kola yönelir. a'nın *f* değeri 5,5 ile en küçüktür, ardından b de 5,5 ile öne çıkar. b genişletilince c açılır, ama c'nin *g* değeri 6,5'e çıkmıştır ve *f(c) = 10,5* olur. Bu kol pahalılaşmıştır. Açık listede en küçük *f* artık d'nindir (6,5) ve arama alt kola geçer. Bu geçiş, seçimin açık listenin tamamından yapılmasının sonucudur. d'den e açılır (*f = 7*), e'den t açılır (*g = 7*, *f = 7*).

t açık listeden çıkarıldığında arama durur. En kısa yol $s \to d \to e \to t$, maliyeti *2 + 3 + 2 = 7*'dir. Üst kol $s \to a \to b \to c \to t$ ise *1,5 + 2 + 3 + 4 = 10,5* tutar. c açılmış ama hiç genişletilmemiştir. Yedi düğümden altısı genişletilmiştir. Dijkstra aynı çizgede uzaklık sırasıyla s, a, d, b, e, c, t düğümlerini genişletirdi; c'nin uzaklığı (6,5) t'ninkinden (7) küçük olduğu için c'yi de genişletirdi.

4.2.5.6 Java kodu ve izleme

A*'ın Java kodu Dijkstra'nınkine çok benzer. Farklar iki yerdedir: kuyruk *g + h*'ye göre sıralanır ve hedef çıkarıldığında arama durur. `g`, `h` ve `ata` sınıf alanlarıdır; `g` ve `h` `double` dizileridir. `komsu[u]`, `u`'nun kenar listesidir ve her kenarın `v` (komşu) ile `w` (ağırlık) alanları vardır.

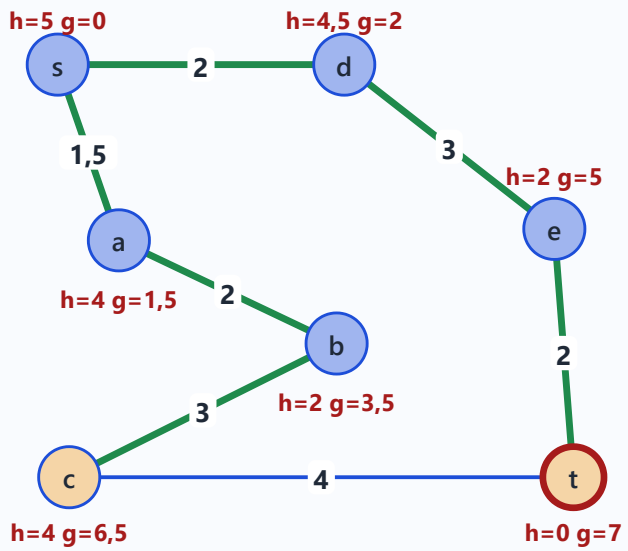
```
double aYildiz(int s, int t) {
  Arrays.fill(g, SONSUZ); g[s] = 0;
  Queue<Integer> acik = new PriorityQueue<>(
    Comparator.comparingDouble(v -> g[v] + h[v]));
  acik.add(s);
  while (!acik.isEmpty()) {
    int u = acik.poll();      // en küçük f
    if (u == t) return g[t];  // hedef
    for (Kenar k : komsu[u])
      if (g[u] + k.w < g[k.v]) {
        g[k.v] = g[u] + k.w; ata[k.v] = u;
        acik.remove(k.v); acik.add(k.v);
      }
  }
  return SONSUZ;
}
```

- **Satır 2:** Bütün *g* değerleri sonsuz, başlangıcınki 0 yapılır.
- **Satır 3–5:** Açık liste bir öncelikli kuyruktur. Karşılaştırıcı her `v` düğümünü `g[v] + h[v]` değerine göre sıralar. Başta yalnız `s` vardır.

- **Satır 7–8:** En küçük *f*'li düğüm çıkarılır. Hedefse `g[t]` döndürülür. Durma koşulu çıkarma anındadır.
- **Satır 9–13:** Her kenar gevşetilir. *g* küçülürse önceki düğüm `ata`'ya yazılır. Düğüm önce kuyruktan silinir, sonra yeniden eklenir.
- **Satır 15:** Kuyruk boşalırsa hedefe yol yoktur.

On ikinci satırdaki `remove` ve `add` çifti önemlidir. Karşılaştırıcı `g[v]` dizisini okur. Bir düğüm kuyruktayken `g[v]` değişirse heap'in sıralama düzeni bozulur, çünkü heap düğümün yerini eski değere göre belirlemiştir. Düğümü çıkarıp yeniden eklemek, onu yeni değerine göre doğru yere koyar. `PriorityQueue.remove(Object)` doğrusal zaman alır; büyük çizgelerde Dijkstra'daki gibi tembel silme tercih edilir.

Kod, A* örneğindeki çizgede çalıştırıldığında `poll()` sırası s (*f* = *5*), a (5,5), b (5,5), d (6,5), e (7), t (7) olur. Her adımdaki *g* güncellemeleri Tablo 4.2.20'deki değerlerle aynıdır. Altıncı çağrıda hedef çıkar ve `return g[t]` 7 döndürür. `ata` dizisi geriye okunursa $t \leftarrow e \leftarrow d \leftarrow s$, yani $s \rightarrow d \rightarrow e \rightarrow t$ elde edilir. c açık listede *f* = *10,5* ile kalır ve hiç çıkarılmaz.



Şekil 4.2.19: `aYildiz(s, t)` izlemesinin son durumu. Düğüm etiketlerinde *h* ve bulunan *g* yazılıdır; çerçeveli t çıkarılan hedeftir.

Örnek 4.2.12

Yönlü çizge: $s \rightarrow a$ (1), $a \rightarrow t$ (9), $s \rightarrow b$ (2), $b \rightarrow c$ (2), $c \rightarrow t$ (2). Sezgisel: $h(s) = 5$, $h(a) = 1$, $h(b) = 4$, $h(c) = 2$, $h(t) = 0$. Açgözlü en iyi ilk arama ile A* karşılaştırılacak.

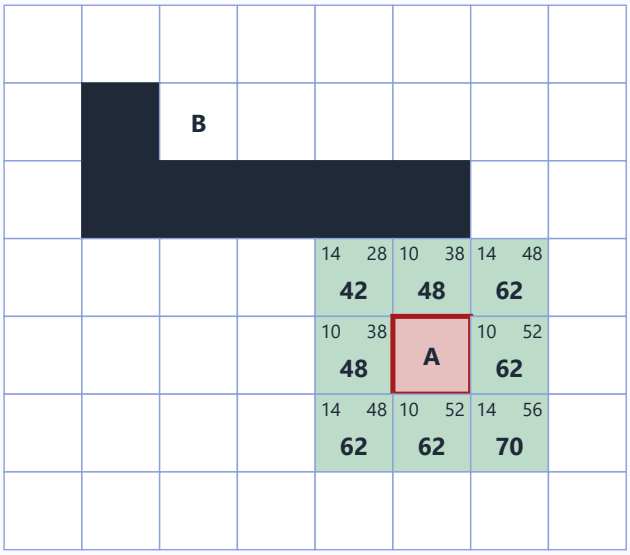
1. **Açgözlü arama.** s'den sonra a ($h = 1$), b'den ($h = 4$) daha cazip görünür. a'dan t açılır ($h = 0$) ve hemen seçilir. Yol $s \rightarrow a \rightarrow t$, maliyet 10.

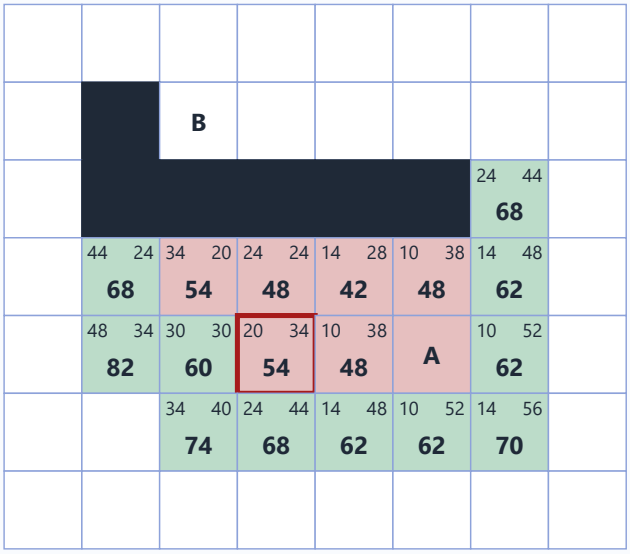
2. **A*.** *f(a) = 1 + 1 = 2*, *f(b) = 2 + 4 = 6*. a genişletilir; t, *g = 10*, *f = 10* ile açılır. Sonra b (6) ve c (*f = 4 + 2 = 6*) genişletilir; c'den t'nin *g* değeri 6'ya iner. t (*f = 6*) çıkarılır: yol *s → b → c → t*, maliyet 6.
3. **Sonuç.** Bu *h* kabul edilebilirdir, bu yüzden A* en kısa yolu bulur. Açgözlü arama hızlıdır, ama *g*'yi yok saydığı için pahalı yolu seçer.

4.2.5.7 Örnek: ızgarada duvarın çevresinden

A*'ın en tanıdık kullanımı ızgara haritalardır. Örnekte 8 sütun ve 7 satırdan oluşan bir ızgara vardır. Başlangıç A, hedef B'dir. Aralarında ters L biçiminde bir duvar durur. Hareket 8 yönlüdür: düz adım 10, çapraz adım 14 maliyetlidir. Sezgisel oktil uzaklıktır ve duvarları yok sayar. A'dan B'ye 3 sütun ve 3 satır fark vardır; *h(A) = 3 · 14 = 42*'dir.

Şekillerde her hücrede üç sayı yazılıdır: sol üstte *g*, sağ üstte *h*, ortada *f*.





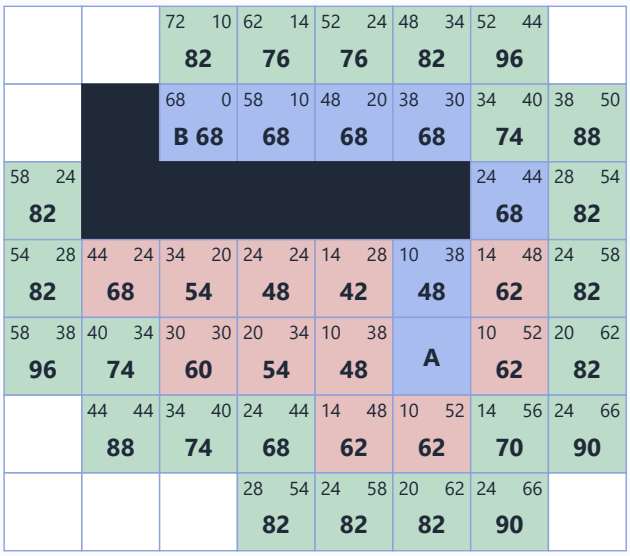


Şekil 4.2.20: Izgarada A*. Solda ilk genişletme: A'nın sekiz komşusu açılır; en küçük *f* 42'dir (doğrudan B'ye doğru, duvarın altındaki hücre). Sağda dördüncü genişletmeden sonra: duvarın altındaki hücreler *f = 42 … 54* ile önce denenir.

Arama önce duvarın hemen altına, B'nin doğrultusuna yönelir. Sezgisel duvarı görmediği için bu hücreler en umut verici görünür. Genişletilen hücrelerin *f* değerleri sırasıyla şöyledir:

> 42, 42, 48, 48, 48, 54, 54, 60, 62, 62, 62, 62, 68, 68, 68, 68, 68, 68

Değerler hiç azalmaz. Bu, tutarlı bir sezgiselin bir özelliğidir: genişletilen düğümlerin *f* değerleri azalmayan bir dizi oluşturur. Duvarın altındaki yol tıkalıdır; duvarın üstüne çıkmak için ya sağdan ya soldan dolaşmak gerekir. Dolaşma maliyeti tahminde yoktur ve *f* adım adım yükselir. On üçüncü genişletmede *f = 68*'e ulaşılır ve arama duvarın iki yanına yayılır. On yedinci genişletmede duvarın sağından dolaşan yol B'ye ulaşır; B açık listeye *f = 68* ile girer.



Şekil 4.2.21: B çıkarıldığında son durum. Mavi hücreler bulunan yoldur: A'dan bir adım yukarı, sağ üste çapraz, sol üste çapraz ve duvarın üstünden üç adım sola. Maliyet *2 · 14 + 4 · 10 = 68*.

B on sekizinci genişletmede çıkarılır ve arama durur. Bulunan yolun maliyeti 68'dir: iki çapraz adım (28) ve dört düz adım (40). On sekiz genişletmenin on biri kapalı (pembe) hücrelerdir, yedisi yol üzerindeki hücrelerdir. Duvarın solundan dolaşan en ucuz yol 96 tutar. A* bu yöndeki en soldaki sütunun hücrelerini açmış, ama hiçbirini genişletmemiştir. Izgaranın köşelerindeki birkaç hücre ise hiç açılmamıştır.

Örnekte çapraz adımlar duvarın köşesini sıyırabilir. Yolun iki çapraz adımı da bir duvar hücresinin köşesinden geçer. Birçok oyunda bu “köşe kesme” yasaktır: çapraz adım ancak iki yandaki düz komşu da boşsa yapılabilir. Böyle bir kural aramanın sonucunu değiştirebilir, ama A*'ın işleyişini değiştirmez.

Sık yapılan hata

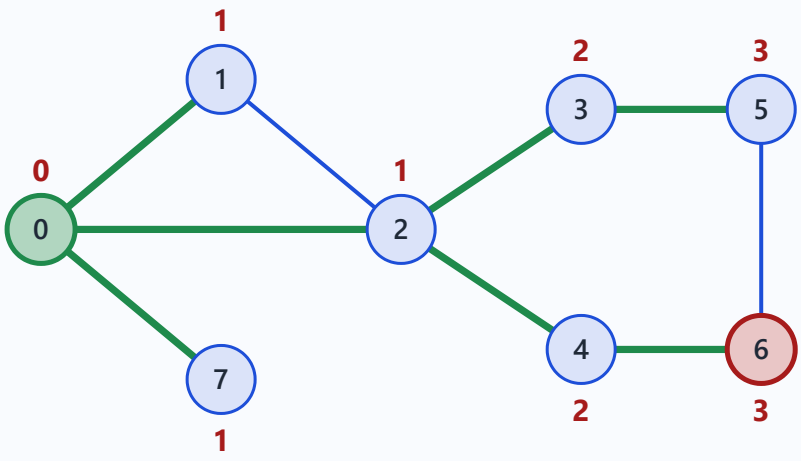
8 yönlü bir ızgarada Manhattan uzaklığını kullanmak. Sezgisel gerçek kalanı aşar ve A* bulduğu ilk yolu en kısa sanar. Arama hızlanmış gibi görünür, ama daha kısa bir çapraz yol kaçırılabilir. 8 yönde oktil, serbest düzlemde Öklid uzaklığı kullanılmalıdır.

4.2.6 Ağırlıksız çizgede en kısa yol: BFS

Bütün kenarların ağırlığı eşitse, örneğin hepsi 1 ise, bir yolun uzunluğu kenar sayısına eşittir. En kısa yol en az kenarlı yol olur. Bu özel durumda ağır algoritmalara gerek yoktur. **Genişlik öncelikli arama** (*breadth-first search*, BFS) problemi doğrusal zamanda çözer.

BFS düğümleri kaynağa uzaklık sırasıyla, katman katman ziyaret eder. Önce kaynağın komşuları (uzaklık 1), sonra onların ziyaret edilmemiş komşuları (uzaklık 2) gelir ve böyle sürer. Bir v düğümü ilk kez u'dan bulunduğunda $d[v] = d[u] + 1$ ve $\pi[v] = u$ yazılır. Düğüm ilk bulunduğu anda en kısa uzaklığına ulaşmıştır; sonra bulunan yollar daha uzun olamaz.

BFS'in Dijkstra'yla ilişkisi açıktır. Dijkstra öncelikli kuyrukla en küçük uzaklıklı düğümü seçer. Ağırlıklar eşitken düğümler kuyruğa uzaklık sırasıyla girer. İlk giren ilk çıkar (FIFO) ilkesiyle çalışan sıradan bir **kuyruk** bu sırayı zaten korur ve öncelikli kuyruğa gerek kalmaz. Kuyruk işlemleri $O(1)$ olduğu için toplam süre $O(V + E)$'dir; heap'in $\log V$ çarpanı ortadan kalkar.



Şekil 4.2.22: Ağırlıksız çizgede BFS. Sayılar BFS düzeyi, yani kaynak 0'dan kenar sayısıdır. Yeşil kenarlar BFS ağacıdır. 0'dan 6'ya en kısa yol $0 \to 2 \to 4 \to 6$'dır (3 kenar).

Şekildeki çizgede kenarlar 0–1, 0–2, 0–7, 1–2, 2–3, 2–4, 3–5, 4–6 ve 5–6'dır. 0'ın komşuları 1, 2 ve 7 birinci düzeydedir. 2'nin komşuları 3 ve 4 ikinci düzeydedir. 3'ün komşusu 5 ve 4'ün komşusu 6 üçüncü düzeydedir. 6'ya giden bir başka yol $0 \to 2 \to 3 \to 5 \to 6$'dır, ama dört kenarlıdır. BFS 6'yı ilk kez 4'ten bulur ve $\pi[6] = 4$ yazar; 5'ten gelen yol daha sonra görülür ve bir şey değiştirmez.

Bütün kenarların ağırlığı aynı sabit c ise de BFS yeterlidir. Uzaklıklar düzeylerin c katıdır. Örneğin bütün kenarları 5 olan aynı çizgede 6'nın uzaklığı $3 \cdot 5 = 15$'tir. Ağırlıklar farklıysa BFS yanılır; en az kenarlı yol en kısa yol olmayabilir.

Tek bir hedef varsa BFS'in bir türevi daha hızlıdır. **İki yönlü arama** (*bidirectional search*), BFS'i aynı anda hem kaynaktan hem hedeften başlatır. İki dalga ortada buluştuğunda arama durur. Dallanma faktörü b ve hedef derinliği d olan bir aramada tek yönlü BFS yaklaşık b^d düğüm açar. İki yönlü arama ise yaklaşık $2 \cdot b^{d/2}$ düğüm açar. $b = 10$ ve $d = 6$ için bu, bir milyon düğüm yerine iki bin düğüm demektir.

4.2.7 Karşılaştırma ve algoritma seçimi

Bölümdeki beş algoritma aynı temel işlemi, gevşetmeyi kullanır. Farkları, hangi problemi çözdüklerinde, hangi kısıtları kabul ettiklerinde ve ne kadar sürdüklerindedir. Tablo 4.2.21 bu farkları tek tabloda toplar.

Algoritma	Problem	Negatif kenar	Negatif döngü	Zaman	Yöntem
BFS	tek kaynak, ağırlıksız	—	—	$O(V + E)$	katman katman
Dijkstra	tek kaynak	hayır	hayır	$O(V^2)$ / $O((V + E) \log V)$	açgözlü
Bellman-Ford	tek kaynak	evet	tespit eder	$O(V \cdot E)$	tekrarlı gevşetme (DP)
Floyd-Warshall	tüm çiftler	evet	tespit eder	$\Theta(V^3)$	dinamik programlama
A*	tek çift	hayır	hayır	en kötü $O(b^d)$	sezgisel + açgözlü

Tablo 4.2.21: En kısa yol algoritmalarının karşılaştırması.

Seçim birkaç soruyla yapılır:

1. **Çizge ağırlıksız mı?** BFS yeterlidir.
2. **Ağırlıkların hepsi sıfır ya da pozitif mi?** Dijkstra kullanılır. Hedef tek ve iyi bir sezgisel varsa A* daha az düğüm açar.
3. **Negatif kenar var mı?** Bellman-Ford kullanılır; negatif döngü de böylece denetlenir.
4. **Tüm çiftler mi isteniyor ve *V* küçük mü?** Floyd-Warshall kullanılır.

Örnek 4.2.13

Dört durum için algoritma seçilecek.

1. **Sosyal ağda iki kişi arasındaki en az aracı sayısı.** Kenarlar ağırlıksızdır: BFS, $O(V + E)$.

2. **Haritada iki adres arasındaki en kısa güzergâh.** Uzunluklar pozitif, hedef tek ve kuş uçuşu uzaklık kabul edilebilir bir sezgiseldir: A*.
3. ***V = 1000*, *E = 5000*, bazı kenarlar negatif.** Dijkstra güvenilmez; Bellman-Ford kullanılır. Yaklaşık $V \cdot E = 1000 \cdot 5000 = 5 \cdot 10^6$ gevşetme denemesi yapılır.
4. **200 şehir arasındaki bütün mesafeler.** Tüm çiftler istenir ve V küçüktür: Floyd-Warshall, $200^3 = 8 \cdot 10^6$ işlem.

Örnek 4.2.14

Yönlü çizge: $A \to B$ 4, $A \to C$ 2, $C \to B$ 1, $B \to D$ 5, $C \to D$ 8, $C \to E$ 10, $D \to E$ 3; kaynak A. Dijkstra ve Bellman-Ford bu çizgede çalıştırılacak.

1. **Dijkstra.** A kesinleşir: B = 4, C = 2. C kesinleşir: B, *2 + 1 = 3* ile iyileşir; D = 10, E = 12. B kesinleşir: D, *3 + 5 = 8*'e iner. D kesinleşir: E, *8 + 3 = 11*'e iner. Son olarak E kesinleşir. Sıra A, C, B, D, E; uzaklıklar A: 0, B: 3, C: 2, D: 8, E: 11.
2. **Yol.** $\pi[E] = D$, $\pi[D] = B$, $\pi[B] = C$, $\pi[C] = A$. Yol $A \to C \to B \to D \to E$, uzunluk $2 + 1 + 5 + 3 = 11$.
3. **Bellman-Ford, alfabetik kenar sırası** ($A \to B$, $A \to C$, $B \to D$, $C \to B$, $C \to D$, $C \to E$, $D \to E$). Tur 1'de beş gevşetme olur: B = 4, C = 2, D = 9, B = 3, E = 12. Tur 2'de iki gevşetme olur: D = 8, E = 11. Tur 3 ve 4'te hiçbir şey değişmez; erken durma mümkündür. Sonuç Dijkstra'nınkiyle aynıdır.

4.2.8 Sık yapılan hatalar

Bölüm boyunca kutularda verilen hataların en önemlileri burada bir arada toplanmıştır.

- **Negatif kenarda Dijkstra.** Kesinleşmiş bir düğüme sonradan daha kısa yol açılabilir; sonuç hiçbir uyarı olmadan yanlış çıkar. Negatif kenarda Bellman-Ford gerekir. Ağırlıklara sabit eklemek de çözüm değildir; çok kenarlı yollar daha fazla ceza alır.
- **∞ + w taşması.** `Integer.MAX_VALUE + w` negatif bir sayıya döner ve "daha kısa" görünür. Bu yüzden toplamadan önce `d[u] != SONSUZ` denetlenir. Floyd-Warshall'da iki parçanın ikisi de denetlenmelidir.
- **Floyd-Warshall döngü sırası.** k ara düğüm döngüsü en dışta olmalıdır. `i`, `j`, `k` sırası bazı hücreleri ∞ ya da gereğinden büyük bırakır.
- **Tur sayısı.** Bellman-Ford'da $V - 1$ turun hepsi gerekebilir; iş tek turda bitmeyebilir. Denetim turu bu turlardan ayrıdır.
- **A*'da hedefte durmak.** Hedef açık listeye eklendiğinde değil, listeden çıkarıldığında durulur. Fazla tahmin eden bir h en kısa yolu kaçırabilir.
- **Yönsüz kenarda negatif ağırlık.** Yönsüz bir negatif kenar, $u \to v \to u$ yoluyla kendi başına negatif bir döngüdür. Yönsüz çizgede negatif ağırlık varsa en kısa yol tanımsızdır.

- **Ulaşılamayan hedefte yol yazdırmak.** Önceki dizisi kaynağa hiç ulaşmaz. Önce uzaklığın sonlu olduğu denetlenmelidir.

4.2.9 Özet

- Ağırlıklı bir çizgede yolun uzunluğu kenar ağırlıklarının toplamıdır. En kısa yol, en az kenarlı yol olmak zorunda değildir. $\delta(s, v)$, s'den v'ye en kısa yolun uzunluğudur.
- Üç problem türü vardır: tek kaynaklı (Dijkstra, Bellman-Ford, BFS), tüm çiftler (Floyd-Warshall) ve tek çift (A*).
- Bütün algoritmaların ortak işlemi gevşetmedir: $d[u] + w(u, v) < d[v]$ ise $d[v]$ ve $\pi[v]$ güncellenir. Tahmin yalnız azalır ve gerçek uzaklığın altına inmez.
- En kısa yolların her parçası da en kısadır (optimal alt yapı). Negatif döngü yoksa en kısa yol en çok $V - 1$ kenarlıdır. Ulaşılabilir bir negatif döngü varsa en kısa yol tanımsızdır.
- Dijkstra en küçük tahminli düğümü kesinleştirir ve komşularını gevşetir. Ağırlıklar negatif olmamalıdır. Dizi ile $O(V^2)$, ikili heap ile $O((V + E) \log V)$ sürer. Tembel silmeli kuyruk eski kayıtları ziyaret denetimiyle atlar.
- Bellman-Ford bütün kenarları $V - 1$ tur gevşetir. Negatif kenarla çalışır; V. turda hâlâ gevşeyen kenar negatif döngüyü gösterir. Süre $O(V \cdot E)$'dir; erken durmayla en iyi durum $O(E)$'dir.
- Floyd-Warshall $D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$ bağıntısını k en dışta olacak biçimde uygular. Süre $\Theta(V^3)$, bellek $\Theta(V^2)$'dir. $D[i][i] < 0$ negatif döngüyü gösterir.
- A* $f = g + h$ değeri en küçük düğümü genişletir. Kabul edilebilir h ile en kısa yolu bulur; tutarlı h ile her düğüm bir kez genişletilir. $h = 0$ alınırsa A* Dijkstra'ya dönüşür.
- Ağırlıksız çizgede BFS en az kenarlı yolu $O(V + E)$ sürede bulur.
- Yollar π (ya da Π) işaretçileriyle hedeften geriye okunur; işaretçiler birlikte en kısa yol ağacını oluşturur.

4.2.10 Kavram listesi

en kısa yol *shortest path* · uzaklık *distance* · ağırlıklı çizge *weighted graph* · tek kaynaklı en kısa yollar *single-source shortest paths* · tüm çiftler arası en kısa yollar *all-pairs shortest paths* · gevşetme *relaxation* · optimal alt yapı *optimal substructure* · negatif döngü *negative cycle* · Dijkstra algoritması *Dijkstra's algorithm* · öncelikli kuyruk *priority queue* · açgözlü algoritma *greedy algorithm* · Bellman-Ford algoritması *Bellman-Ford algorithm* · dinamik programlama *dynamic programming* · Floyd-Warshall algoritması *Floyd-Warshall algorithm* · A* algoritması *A* search* · sezgisel *heuristic* · kabul edilebilir sezgisel *admissible heuristic* · genişlik öncelikli arama *breadth-first search (BFS)*

4.2.11 Kendini Dene

1. Yönlü çizge: $s \to a$ 2, $s \to b$ 6, $a \to b$ 3, $a \to c$ 7, $b \to c$ 1, $c \to t$ 2, $b \to t$ 8. Kaynak s. Dijkstra düğümleri hangi sırayla kesinleştirir? Son uzaklıklar ve s'den t'ye yol nedir?

   Cevap

   Sıra s, a, b, c, t. a = 2; b önce 6 olur, a üzerinden $2 + 3 = 5$'e iner; c = $5 + 1 = 6$ (a üzerinden 9 olurdu); t = $6 + 2 = 8$ (b'den doğrudan 13 olurdu). Yol $s \to a \to b \to c \to t$, uzunluk 8.

2. Dizi ile seçim yapan Dijkstra ve ikili heap'li Dijkstra, $V = 10^5$ ve $E = 4 \cdot 10^5$ olan bir çizgede yaklaşık kaç işlem yapar? Bellman-Ford kaç gevşetme denemesi yapar?

   Cevap

   Dizi: $V^2 = 10^{10}$. Heap: $(V + E) \log_2 V \approx 5 \cdot 10^5 \cdot 16{,}6 \approx 8{,}3 \cdot 10^6$. Bellman-Ford: $V \cdot E = 4 \cdot 10^{10}$. Çizge seyrek olduğu için heap'li Dijkstra açık ara en hızlısıdır.

3. Zincir çizge $s \to a \to b \to c$, bütün ağırlıklar 1. Erken durmalı Bellman-Ford kenarları (a) *(b, c), (a, b), (s, a)* sırasıyla, (b) *(s, a), (a, b), (b, c)* sırasıyla incelerse kaç tur döner?

   Cevap

   (a) Her tur yalnız bir kenar ilerler: 1. turda a, 2. turda b, 3. turda c bulunur. $V - 1 = 3$ turun hepsi kullanılır. (b) 1. turda üçü birden bulunur; 2. turda hiçbir şey değişmez ve algoritma durur: 2 tur. Sonuç ikisinde de *[0, 1, 2, 3]*'tür.

4. Floyd-Warshall, kenarları $0 \to 1$ (4), $1 \to 2$ (2), $2 \to 0$ (1) olan çizgede çalıştırılıyor. $k = 0$ turundan sonra hangi hücre değişir? Son D matrisi nedir?

   Cevap

   $k = 0$: 0'a gelen kenar yalnız $2 \to 0$'dır; $D[2][1] = 1 + 4 = 5$. $k = 1$: $D[0][2] = 4 + 2 = 6$. $k = 2$: $D[1][0] = 2 + 1 = 3$. Son matris *[[0, 4, 6], [3, 0, 2], [1, 5, 0]]*. Köşegen 0, döngünün ağırlığı 7'dir.

5. Floyd-Warshall, kenarları $0 \to 1$ (1), $1 \to 2$ (−2), $2 \to 0$ (−1), $0 \to 2$ (4) olan çizgede çalıştırılıyor. Köşegende ne görülür? Kod ne yapar?

   Cevap

   $0 \to 1 \to 2 \to 0$ döngüsünün ağırlığı $1 - 2 - 1 = -2$'dir. Algoritmanın sonunda köşegende negatif değerler çıkar (yerinde güncellemeyle −2, −2 ve −4). `d[i][i] < 0` denetimi ilk satırda doğru olur ve `IllegalStateException("Negatif döngü")` fırlatılır. Matrisin diğer değerleri anlamsızdır.

6. Floyd-Warshall örneğinin son Π matrisinden $3 \rightsquigarrow 1$ yolu hangisidir? Yolun uzunluğu $D[3][1]$ ile uyuşur mu?

   Cevap

   3. satırda: $\Pi[3][1] = 2$, $\Pi[3][2] = 4$, $\Pi[3][4] = 3$. Yol $3 \to 4 \to 2 \to 1$, uzunluk $2 + 1 + 3 = 6 = D[3][1]$.

7. Zincir $s \to a \to t$, $w(s, a) = 2$, $w(a, t) = 5$. $h(s) = 6$, $h(a) = 5$, $h(t) = 0$ sezgiseli kabul edilebilir mi, tutarlı mı?

   Cevap

   Gerçek kalanlar $h^*(s) = 7$, $h^*(a) = 5$, $h^*(t) = 0$. $6 \le 7$, $5 \le 5$: kabul edilebilir. Tutarlılık: $h(s) = 6 \le 2 + 5 = 7$ ve $h(a) = 5 \le 5 + 0$; ikisi de sağlanır, sezgisel tutarlıdır.

8. Bütün kenar ağırlıkları 5 olan bir çizgede BFS kullanılabilir mi? BFS örneğindeki çizgede 6'nın uzaklığı ne olur?

   Cevap

   Kullanılabilir. Bütün ağırlıklar eşitken en kısa yol yine en az kenarlı yoldur; uzaklık, BFS düzeyinin 5 katıdır. 6 üçüncü düzeydedir: $3 \cdot 5 = 15$.

9. Dijkstra'nın Java kodunda karşılaştırıcı `(a, b) -> a[1] - b[1]` biçimindedir. Kuyruğa uzaklığı `Integer.MAX_VALUE` olan bir kayıt eklenseydi ne olabilirdi?

   Cevap

   Çıkarma taşabilirdi. Örneğin `Integer.MAX_VALUE - (-1)` negatif bir sayıya döner ve karşılaştırıcı yanlış sırayı bildirir. Nottaki kodda kuyruğa yalnız sonlu uzaklıklar eklendiği için sorun çıkmaz. Güvenli yazım `Integer.compare(a[1], b[1])` 'dir.

10. Dijkstra, her düğüme giden en kısa yolların sayısını da tutacak biçimde değiştiriliyor. Kenarları $s \to a$ 1, $s \to b$ 1, $a \to c$ 1, $b \to c$ 1, $c \to t$ 1, $s \to t$ 3, $a \to t$ 2 olan çizgede `sayi[t]` kaçtır?

    Cevap

    $d[t] = 3$'tür ve dört yol bu uzunluktadır: $s \to t$, $s \to a \to t$, $s \to a \to c \to t$, $s \to b \to c \to t$. Algoritma `sayi[c] = 2` bulur; t'ye s'den 1, a'dan 1, c'den 2 yol gelir: `sayi[t] = 4`. Kural: daha kısa yol bulununca sayı yeniden başlar (`sayi[v] = sayi[u]`), eşit uzunlukta yol bulunca eklenir (`sayi[v] += sayi[u]`).

Son sorudaki kural aşağıdaki kodla uygulanır. Kod seçimi dizi taramasıyla yapar ve Java ile denenmiştir.

```
long[] yolSayisi(int V, List<List<int[]>> komsu, int kaynak, int[] uzaklik) {
  long[] sayi = new long[V];
  boolean[] kesin = new boolean[V];
  Arrays.fill(uzaklik, SONSUZ);
  uzaklik[kaynak] = 0; sayi[kaynak] = 1;
  for (int tur = 0; tur < V; tur++) {
    int u = -1;
    for (int x = 0; x < V; x++)
      if (!kesin[x] && uzaklik[x] != SONSUZ && (u == -1 || uzaklik[x] < uzaklik[u])) u = x;
    if (u == -1) break;
    kesin[u] = true;
    for (int[] k : komsu.get(u)) {
      int v = k[0], w = k[1];
      if (uzaklik[u] + w < uzaklik[v]) {        // daha kısa yol: sayım yeniden başlar
        uzaklik[v] = uzaklik[u] + w;
        sayi[v] = sayi[u];
      } else if (uzaklik[u] + w == uzaklik[v]) { // eşit uzunlukta yeni yollar
        sayi[v] += sayi[u];
      }
    }
  }
  return sayi;
}
```

Kural, ağırlıkların pozitif olmasına dayanır. Bir düğüm kesinleştiğinde ona gelen bütün en kısa yollar daha küçük uzaklıklı, yani daha önce kesinleşmiş düğümlerden gelmiştir. Bu yüzden `sayi[u]`, `u` kesinleştiği anda tamdır ve komşulara doğru değer aktarılır.

4.2.12 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 22 ve 23: gevşetmenin özelliklerini, Dijkstra ve Bellman-Ford'un doğruluk kanıtlarını, Floyd-Warshall'ı ve seyrek çizgeler için Johnson algoritmasını ayrıntılı verir.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 4.4: en kısa yol algoritmalarının Java kodlarını, indeksli öncelikli kuyrukla Dijkstra'yı ve kuyruk tabanlı Bellman-Ford'u anlatır.
- S. Russell ve P. Norvig, *Artificial Intelligence: A Modern Approach* (4. baskı, 2020), Bölüm 3: A*, kabul edilebilir ve tutarlı sezgiseller, IDA* ve bellek sınırlı arama türevlerini yapay zekâ açısından ele alır.
- E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs” (1959): algoritmanın üç sayfalık özgün makalesidir; aynı makale minimum kapsayan ağaç problemini de çözer.

4.2.13 Kaynaklar

1. R. Bellman, “On a Routing Problem”, *Quarterly of Applied Mathematics* 16, 1958.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 22–23).
3. E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs”, *Numerische Mathematik* 1, 1959.
4. R. W. Floyd, “Algorithm 97: Shortest Path”, *Communications of the ACM* 5(6), 1962.
5. P. E. Hart, N. J. Nilsson, B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”, *IEEE Transactions on Systems Science and Cybernetics* 4(2), 1968.
6. S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4. baskı, Pearson, 2020.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Kesim 4.4: Shortest Paths).
8. S. Warshall, “A Theorem on Boolean Matrices”, *Journal of the ACM* 9(1), 1962.

KISIM II · BÖLÜM 4.3

Minimum Kapsayan Ağaç Algoritmaları

Bir şirket birkaç şehri elektrik hattıyla birbirine bağlamak istiyor. Her şehir çiftinin arasına hat çekmenin bir maliyeti vardır. Bütün şehirlerin birbirine ulaşması yeterlidir; iki şehrin doğrudan bağlı olması gerekmez. Şirket hangi hatları çekerse toplam maliyeti en küçük olur? Bu sorunun matematikteki adı minimum kapsayan ağaç problemidir. Not önce kapsayan ağacı, minimum kapsayan ağacı ve bütün algoritmaların dayandığı iki özelliği, kesim ve döngü özelliğini tanıtır. Ardından aynı problemi çözen dört açgözlü algoritma ele alınır. Kruskal kenarları küçükten büyüğe ekler, Prim ağacı tek düğümden büyütür. Borůvka bütün bileşenleri aynı anda büyütür, ters silme ise ağır kenarları siler. Kruskal'ın döngü denetimi için gereken ayrık kümeler yapısı da ayrıntılı işlenir. Her algoritma için Java kodu, adım adım izleme ve maliyet hesabı verilir; not, dört algoritmanın karşılaştırmasıyla biter.

ÖĞRENME ÇIKTILARI

Kavram

Kapsayan ağacı ve minimum kapsayan ağacı (MST) tanımlamak; ağacın $V - 1$ kenarlı olduğunu, kesim ve döngü özelliklerini açıklamak ve gerekçelendirmek.

Kruskal ve ayrık kümeler

Kenarları sıralayıp döngü oluşturmayanları seçmek; döngü denetimini ayrık kümelerle (bul / birleştir) yapmak; yol kısaltmanın ve rütbeye göre birleştirmenin etkisini izlemek.

Prim

Ağacı bir düğümden büyütmek; anahtar ve ebeveyn dizilerini hem dizi taramasıyla hem öncelikli kuyrukla adım adım izlemek; Prim ile Dijkstra arasındaki farkı açıklamak.

Borůvka, ters silme ve analiz

Bileşen tabanlı ve silme tabanlı yaklaşımları uygulamak; dört algoritmanın karmaşıklığını karşılaştırıp çizgenin yapısına göre uygun olanı seçmek.

4.3.1 Kapsayan ağaç ve minimum kapsayan ağaç

Çizgeler, nesneler ve aralarındaki bağlantılar üzerine kurulan problemlerin ortak dilidir. Bu notta ele alınan çizgeler **yönsüzdür**: u ile v arasındaki kenar iki yönde de kullanılabilir. Çizgeler ayrıca **ağırlıklıdır**: her kenarın bir sayısal değeri vardır ve bu değer kablonun uzunluğunu, yolun maliyetini ya da iki nokta arasındaki farkı gösterebilir. Düğüm sayısı V, kenar sayısı E ile gösterilir. Aynı harfler bağlama göre düğüm ve kenar kümesini de anlatır; $G = (V, E)$ yazımında olduğu gibi.

Ağaç kavramı ise ikili arama ağaçlarından ve heap'lerden tanıdıktır. Orada ağaç köklü bir veri yapısıdır. Çizge kuramında ağaç daha genel bir nesnedir: **bağlı ve döngüsüz** her yönsüz çizge bir ağaçtır. Kök yoktur, çocuk sayısı sınırı yoktur. Bir ağacın herhangi iki düğümü arasında tam olarak bir yol bulunur. Yol hiç olmasaydı çizge bağlı olmazdı; iki farklı yol olsaydı bu iki yol birlikte bir döngü oluştururdu.

4.3.1.1 Problemin tanımı

Bağlı bir çizgeden bazı kenarları silerek bütün düğümleri hâlâ birbirine bağlayan, ama hiç döngü içermeyen bir alt çizge elde edilebilir. Böyle bir alt çizge, çizgenin bütün düğümlerini “kapsar”.

Tanım 4.3.1

$G = (V, E)$ bağlı, yönsüz bir çizge olsun. G'nin bir **kapsayan ağacı** (*spanning tree*), G'nin *bütün* düğümlerini içeren, bağlı ve döngüsüz bir alt çizgesidir. Kapsayan ağaç yalnız G'de bulunan kenarları kullanır.

Kapsayan ağacın kaç kenarı olduğu, düğüm sayısından kesin olarak bellidir. Bu sonuç bütün algoritmaların durma koşulunda kullanılır.

Teorem 4.3.1 (Ağacın kenar sayısı)

V düğümlü her ağacın tam olarak $V - 1$ kenarı vardır. Ağaca yeni bir kenar eklemek tam olarak bir döngü oluşturur; ağaçtan bir kenar silmek ağacı iki parçaya böler.

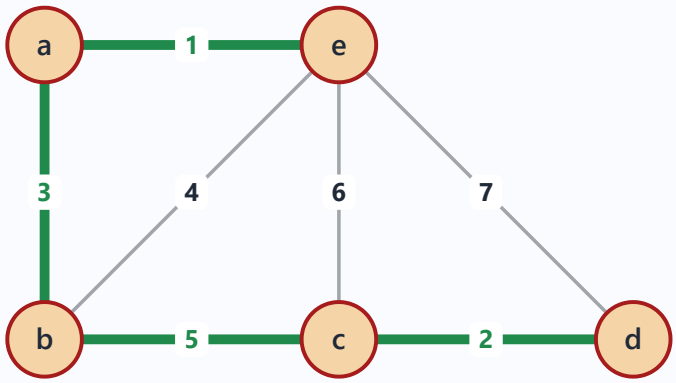
İspat. Başlangıçta kenarsız V düğüm olsun. Bu durumda V ayrı parça vardır. Ağacın kenarları tek tek eklensin. Bir kenar aynı parçanın iki düğümünü bağlasaydı, o iki düğüm arasında zaten bir yol bulunduğu için bir döngü oluşurdu; ağaç döngüsüz olduğundan bu olmaz. Demek ki her kenar iki ayrı parçayı birleştirir ve parça sayısını tam olarak bir azaltır. Sonunda ağaç bağlı olduğu için tek parça kalır. V parçadan bir parçaya inmek için $V - 1$ kenar gerekir. İkinci kısım için ağaçta u ile v arasındaki tek yol düşünülür: yeni (u, v) kenarı bu yolla birlikte bir döngü oluşturur. Bir kenar silindiğinde ise o kenarın iki ucu arasında başka yol kalmaz; ağaç iki parçaya ayrılır. □

Bir çizgenin genellikle birçok kapsayan ağacı vardır. Kenarlar ağırlıklı olduğunda bu ağaçların toplam ağırlıkları da farklıdır. Asıl ilgilendiğimiz, toplamı en küçük olanıdır.

Tanım 4.3.2

Kenarları $w(u, v)$ ile ağırlıklandırılmış bağlı, yönsüz bir çizgede bir kapsayan ağacın ağırlığı, kenar ağırlıklarının toplamıdır: $w(T) = \Sigma_{(u,v) \in T}\, w(u, v)$. Ağırlığı en küçük olan kapsayan ağaca **minimum kapsayan ağaç** (*minimum spanning tree*, MST) denir.

Tanımdaki "minimum" sözcüğü ağacın kendisini değil, toplam ağırlığı nitelendirir. Kenar sayısı her kapsayan ağaçta $V - 1$ olduğu için en az kenarlı ağaç diye bir arayış yoktur. Ağaçlar yalnız hangi kenarları seçtikleriyle ayrılır. Şekil 4.3.1 beş düğümlü küçük bir çizgeyi ve onun minimum kapsayan ağacını gösterir.



Şekil 4.3.1: *a, b, c, d, e* düğümlü ağırlıklı çizge. Kalın kenarlar ($ae = 1$, $cd = 2$, $ab = 3$, $bc = 5$) minimum kapsayan ağacı oluşturur: $w(T) = 1 + 2 + 3 + 5 = 11$.

Şekildeki çizgenin yedi kenarı vardır. Kapsayan ağaç beş düğümü bağlamak için dört kenar seçer; üç kenar dışarıda kalır. Kalın dört kenar bir ağaç oluşturur: *a*'dan *e*'ye ve *b*'ye, *b*'den *c*'ye, *c*'den *d*'ye gidilebilir ve hiçbir döngü yoktur. Dışarıda kalan $be = 4$, $ec = 6$ ve $ed = 7$ kenarlarının her biri ağaca eklenseydi bir döngü kapatırdı. Örneğin be eklenirse $b \to a \to e \to b$ döngüsü oluşur.

Sık yapılan hata

MST'yi "en hafif $V - 1$ kenar" sanmak. Şekildeki en hafif dört kenar $ae = 1$, $cd = 2$, $ab = 3$ ve $be = 4$'tür; toplamları 10'dur. Ama bu dört kenar bir ağaç değildir: *a, b, e* arasında döngü kurar ve $\{a, b, e\}$ ile $\{c, d\}$ birbirine bağlanmaz. Seçilen kenarlar hem döngüsüz hem bağlı olmalıdır; asıl zorluk da buradadır.

4.3.1.2 Kaç kapsayan ağaç vardır?

MST'yi bulmanın en kaba yolu, bütün kapsayan ağaçları tek tek kurup ağırlıklarını karşılaştırmaktır. Bu yöntemin neden işe yaramadığını kapsayan ağaç sayısı gösterir. Her düğüm çiftinin bir kenarla bağlandığı çizgeye **tam çizge** denir. Arthur Cayley 1889'da n düğümlü tam çizgenin tam olarak n^{n-2} kapsayan ağacı olduğunu gösterdi. Bu sonuç **Cayley formülü** diye bilinir.

n	Kapsayan ağaç sayısı
3	3
4	16
5	125
10	100.000.000
15	yaklaşık $1{,}95 \cdot 10^{15}$

Tablo 4.3.1: n düğümlü tam çizgedeki kapsayan ağaç sayısı, n^{n-2}.

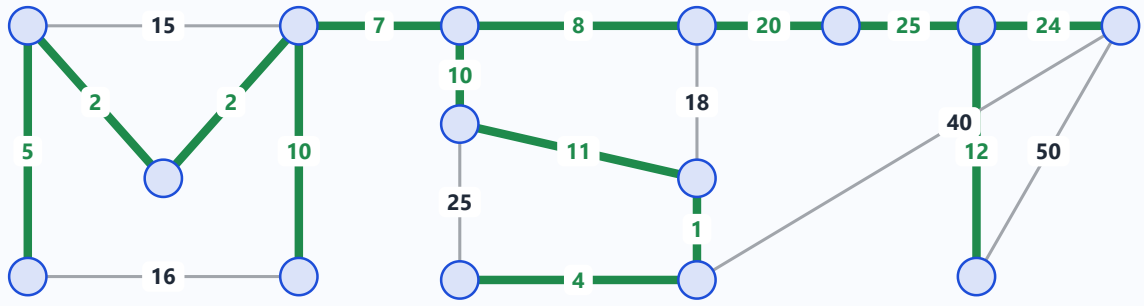
Örnek 4.3.1

Dört düğümlü tam çizgenin altı kenarı vardır. Kapsayan ağaç üç kenar seçer. Altı kenardan üç kenar $C(6, 3) = 20$ farklı biçimde seçilir. Bu seçimlerin dördü üçgendir: dört düğümden herhangi üçü bir döngü kurar ve dördüncü düğüm dışarıda kalır. Üçgen olmayan her üç kenarlı seçim ise döngüsüzdür ve dört düğümü bağlar. Kapsayan ağaç sayısı $20 - 4 = 16$'dır. Cayley formülü de $4^{4-2} = 16$ verir.

On beş düğümlü bir tam çizgede kapsayan ağaç sayısı katrilyon mertebesindedir. Bütün ağaçları denemek, küçük çizgelerde bile kullanışsızdır. Tam olmayan çizgelerde sayı daha azdır, ama yine hızla büyür: Şekil 4.3.1'deki yedi kenarlı çizgenin 21 kapsayan ağacı vardır. Bu ağaçların ağırlıkları 11 ile 21 arasında değişir ve 11 ağırlıklı yalnız bir ağaç vardır. Neyse ki MST'yi bulmak için ağaçları saymaya gerek yoktur. Problem, her adımda yerel olarak en iyi görünen seçimi yapan **açgözlü** (*greedy*) algoritmalarla doğrudan çözülür. Açgözlü yaklaşımın burada neden doğru sonuç verdiğini kesim ve döngü özellikleri açıklar.

4.3.1.3 Örnek: “MST” çizgesi

Daha büyük bir örnek, kapsayan ağacın hangi kenarları seçtiğini ve hangilerini dışarıda bıraktığını görmeye yardım eder. Şekil 4.3.2'deki çizge M, S ve T harflerini çizer. Çizgenin 15 düğümü ve 20 kenarı vardır.



Şekil 4.3.2: M, S ve T harflerini çizen 15 düğümlü, 20 kenarlı çizge. Kalın 14 kenar minimum kapsayan ağacı oluşturur; toplam ağırlık 141'dir. İnce kenarlar (15, 16, 18, 25, 40, 50) ağaca girmez.

Kalın kenar sayısı 14'tür ve bu sayı $V - 1 = 15 - 1$'e eşittir. Ağırlıkların toplamı şöyle bulunur:

> 2 + 2 + 5 + 10 + 7 + 8 + 10 + 11 + 4 + 1 + 20 + 25 + 24 + 12 = 141

Dışarıda kalan altı ince kenarın ortak bir özelliği vardır. Her biri kalın kenarlarla birlikte bir döngü kapatır ve o döngünün *en ağır* kenarıdır. M harfinin üstündeki 15'lik kenar, iki çapraz 2'lik kenarla bir üçgen kurar; üçgenin en ağırı 15'tir. M'nin altındaki 16'lık kenar, iki bacak (5 ve 10) ve iki çapraz (2 ve 2) ile döngü kurar. S harfinin sol alt kısmındaki 25'lik ince kenar ise 11, 1 ve 4 ağırlıklı kalın kenarlarla bir döngü kapatır. Her durumda ince kenarın yerine döngünün öteki kenarları bağlantıyı sağlar ve daha ucuzdur.

Ağırlığı yüksek bazı kenarlar ise ağaçtadır: 20, 24 ve S ile T arasındaki 25. Bu kenarlar hiçbir döngünün en ağırı değildir. Örneğin S'nin sağ altından T'nin sağ üst ucuna uzanan 40'lık kenar uzun bir döngü kapatır. Bu döngüde 20, 25 ve 24'lük kenarlar da vardır, ama döngünün en ağırı 40'tır. Aynı biçimde 50'lik kenar 24 ve 12 ile bir üçgen kurar. Çizgede bir de **köprü** vardır: M ile S'yi bağlayan 7'lik kenar silinirse çizge iki parçaya ayrılır. Bir köprü hiçbir döngüde yer almadığı için her kapsayan ağaçta, dolayısıyla her MST'de bulunmak zorundadır.

Sık yapılan hata

Ağır bir kenarın ağaçta bulunmasını “o düğüme ulaşmanın tek yolu bu kenardır” diye açıklamak. Şekildeki 20, 24 ve 25'lik kalın kenarlar köprü değildir; her biri silinse de çizge bağlı kalır. Ağaçta bulunmalarının nedeni, onları içeren her döngüde daha ağır bir kenarın (40 ya da 50) bulunmasıdır. Bir kenarın ağırlığı tek başına bir şey söylemez; önemli olan, kenarın döngülerdeki ve kesimlerdeki konumudur.

4.3.1.4 Kesim özelliği

Bütün MST algoritmaları iki basit gözleme dayanır. Birincisi düğümleri iki gruba ayırmakla ilgilidir.

Tanım 4.3.3

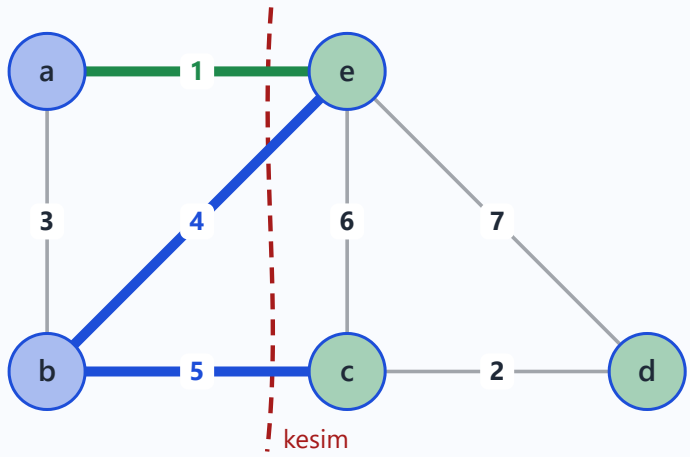
Bir çizgenin düğüm kümesini boş olmayan iki parçaya ayırmaya **kesim** (*cut*) denir ve *(S, V − S)* ile gösterilir. Bir ucu *S*'de, öbür ucu *V − S*'de olan kenara **kesimi geçen kenar** denir. Kesimi geçen kenarlar arasında ağırlığı en küçük olana kesimin **en hafif kenarı** denir.

Teorem 4.3.2 (Kesim özelliği)

Her kesimde, kesimi geçen en hafif kenar bir MST'ye aittir. Kesimi geçen en hafif kenar tek ise bu kenar *her* MST'de bulunur.

İspat. Kesimi geçen en hafif kenar $e = (u, v)$ olsun; $u \in S$, $v \in V - S$. T, e'yi içermeyen bir MST olsun. T bir kapsayan ağaç olduğu için u'dan v'ye T içinde bir yol vardır. Bu yol S'de başlar, $V - S$'de biter; demek ki en az bir yerde kesimi geçer. Geçtiği kenar f olsun. T'den f silinir ve e eklenirse yeni bir T' elde edilir. f silinince ağaç iki parçaya ayrılır: biri u'yu, öteki v'yi içerir. e bu iki parçayı yeniden bağlar. Böylece T' de $V - 1$ kenarlı, bağlı bir kapsayan ağaçtır. Ağırlığı $w(T') = w(T) - w(f) + w(e)$'dir. e kesimin en hafif kenarı olduğu için $w(e) \le w(f)$, dolayısıyla $w(T') \le w(T)$ olur. T minimum olduğundan T' de minimumdur ve e'yi içerir. e tek en hafif kenarsa $w(e) < w(f)$ olur ve $w(T') < w(T)$ çelişkisi doğar; yani e'yi içermeyen bir MST yoktur. □

İspattaki fikre **değiştirme argümanı** (*exchange argument*) denir. Açgözlü seçimi içermeyen bir en iyi çözüm alınır. Bu çözümün bir parçası değiştirilir ve çözüm, o seçimi içeren, en az onun kadar iyi bir çözüme dönüşür. Aynı fikir açgözlü algoritmaların çoğunun doğruluk ispatında kullanılır.



Şekil 4.3.3: *{a, b} | {c, d, e}* kesimi. Kesimi geçen kenarlar *ae = 1*, *be = 4* ve *bc = 5*'tir; en hafifi *ae = 1* güvenlidir ve MST'dedir.

Örnek 4.3.2

Şekil 4.3.1'deki çizgede birkaç kesim incelenebilir.

1. *{a, b} | {c, d, e}*: geçen kenarlar *ae = 1*, *be = 4*, *bc = 5*. En hafif *ae*'dir; MST'de bulunur.
2. *{d} | {a, b, c, e}*: geçen kenarlar *cd = 2* ve *ed = 7*. En hafif *cd*'dir; MST'de bulunur.
3. *{a, b, e} | {c, d}*: geçen kenarlar *bc = 5*, *ec = 6*, *ed = 7*. En hafif *bc*'dir. Bu kenar çizgenin hafif kenarlarından biri değildir, yine de MST'ye girer.

Kesim özelliği yalnız kesimi geçen kenarlar hakkında konuşur. *ab = 3* ilk kesimi geçmez; bu yüzden o kesim *ab* için bir şey söylemez. *ab*'nin MST'de olduğu başka bir kesimle gösterilir: *{b} | {a, c, d, e}* kesimini geçen kenarlar *ab = 3*, *be = 4*, *bc = 5*'tir ve en hafifi *ab*'dir.

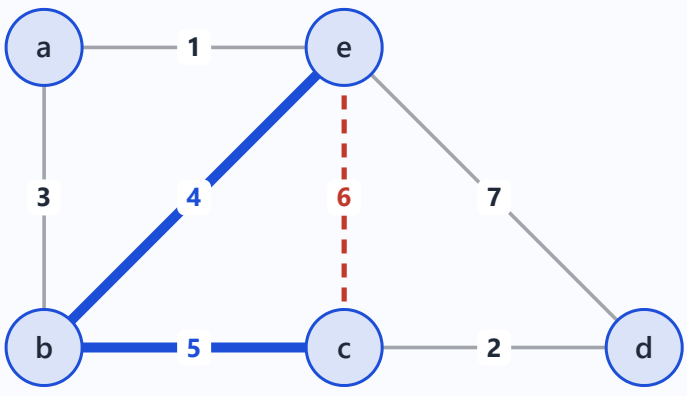
4.3.1.5 Döngü özelliği

İkinci gözlem, kesim özelliğinin bir bakıma tersidir. Kesim özelliği hangi kenarın kesinlikle seçilebileceğini söyler; döngü özelliği ise hangi kenarın kesinlikle dışarıda bırakılabileceğini söyler.

Teorem 4.3.3 (Döngü özelliği)

Çizgedeki herhangi bir döngünün en ağır kenarı tek ise bu kenar hiçbir MST'de yer almaz.

İspat. Bir *C* döngüsünün tek en ağır kenarı *f = (u, v)* olsun ve bir *T* MST'si *f*'yi içersin. *T*'den *f* silinirse ağaç iki parçaya ayrılır; *u* bir parçada, *v* öbür parçadadır. *C*'nin *f* dışındaki kenarları *u*'dan *v*'ye giden bir yol oluşturur. Bu yol bir parçadan öbürüne geçtiği için en az bir kenarı, örneğin *e*, iki parçayı birbirine bağlar. *T − f + e* yine bir kapsayan ağaçtır ve *w(e) < w(f)* olduğu için ağırlığı *T*'ninkinden küçüktür. Bu, *T*'nin minimum olmasıyla çelişir. □



Şekil 4.3.4: *b–c–e* döngüsünün kenarları 4, 5 ve 6'dır. En ağır kenar *ec = 6* hiçbir MST'de yer almaz.

Döngü özelliği küçük bir uyarı içerir: en ağır kenar *tek* olmalıdır. Bir döngüde en ağır ağırlık iki kenarda birden görülürse, bunlardan biri bir MST'de bulunabilir. Örneğin kenarları 1, 2 ve 2 olan bir üçgende 2'lik kenarlardan biri mutlaka seçilir. Böyle bir durumda özellik yalnız şunu söyler: en ağır kenar-

lardan birini içermeyen bir MST vardır.

İki özellik birlikte algoritmaların iskeletini verir. Tablo 4.3.2 hangi algoritmanın hangi özelliği kullandığını özetler.

Özellik	Ne söyler?	Kullanan algoritma
Kesim özelliği	Kesimi geçen en hafif kenar güvenle eklenir.	Kruskal, Prim, Borůvka
Döngü özelliği	Döngünün en ağır kenarı güvenle atılır.	Ters silme

Tablo 4.3.2: İki özellik ve onları kullanan algoritmalar.

4.3.1.6 MST ne zaman tektir?

Bir çizgenin birden fazla minimum kapsayan ağacı olabilir. Bunun için eşit ağırlıklı kenarlar gerekir. Kenarları 1, 2 ve 2 olan üçgende 1'lik kenar ile 2'lik kenarlardan herhangi biri bir MST oluşturur. İki farklı ağacın ağırlığı da 3'tür. Bütün ağırlıklar farklıysa böyle bir seçim serbestliği kalmaz.

Teorem 4.3.4 (MST'nin tekliği)

Bağlı bir çizgede bütün kenar ağırlıkları birbirinden farklıysa MST tektir.

İspat. T bir MST, e de T'nin herhangi bir kenarı olsun. T'den e silinince düğümler iki parçaya ayrılır ve bu iki parça bir kesim tanımlar. T'nin bu kesimi geçen tek kenarı e'dir. Kesimi geçen kenarlar arasında e'den hafif bir e' olsaydı $T - e + e'$ daha hafif bir kapsayan ağaç olurdu; bu olamaz. Ağırlıklar farklı olduğundan e bu kesimin *tek* en hafif kenarıdır. Kesim özelliğine göre e her MST'de bulunur. Bu, T'nin bütün kenarları için geçerlidir. Her MST T'nin $V - 1$ kenarını içerir ve kendisi de $V - 1$ kenarlıdır; demek ki her MST T'nin aynısıdır. □

Teoremin tersi doğru değildir: eşit ağırlıklı kenarlar bulunan bir çizgenin MST'si yine tek olabilir. Şekil 4.3.1'deki ağırlıklar zaten farklıdır; bu yüzden orada tek MST vardır. Eşit ağırlıklı kenarlar bulunduğunda farklı algoritmalar, hatta aynı algoritmanın farklı kenar sıraları farklı ağaçlar bulabilir. Bulunan ağaçların toplam ağırlığı ise her zaman aynıdır.

4.3.1.7 Açgözlü MST algoritmaları

Kesim özelliği genel bir yöntem önerir. Başta hiç kenar seçilmemiştir. Seçilen kenarlar her an bir MST'nin parçası olacak biçimde büyütülür. Her adımda, seçilmiş kenarların hiçbirinin geçmediği bir kesim bulunur ve o kesimin en hafif kenarı eklenir. Böyle bir kenara **güvenli kenar** denir: eklendiğinde seçilen kümenin yine bir MST'ye genişletilebileceği garanti edilir. $V - 1$ güvenli kenar eklenince ağaç tamamlanır.

Dört klasik algoritma bu genel yöntemin farklı uygulamalarıdır. Hepsi açgözlüdür: her adımda yerel olarak en iyi kenarı seçer ya da en kötüsünü atar ve bu karar hiçbir zaman geri alınmaz.

Algoritma	Fikir	Yardımcı yapı	Zaman
Kruskal (1956)	Kenarları küçükten büyüğe sırala, döngü oluşturmayanları ekle	ayrık kümeler	*O(E log E)*
Prim (1957)	Bir düğümden başla, ağacı en ucuz komşu kenarla büyüt	öncelikli kuyruk	*O((V + E) log V)*
Borůvka (1926)	Her bileşen kendi en ucuz kenarını aynı anda ekler	ayrık kümeler	*O(E log V)*
Ters silme	Kenarları büyükten küçüğe dene, bağlılığı bozmayanı sil	BFS/DFS ya da ayrık kümeler	*O(E · (V + E))*

Tablo 4.3.3: Açgözlü MST algoritmalarına genel bakış.

Dört algoritma da aynı ağırlıkta bir MST bulur. Ağırlıklar farklıysa, MST'nin tekliği nedeniyle aynı ağacı bulurlar. Aralarındaki fark kenarları hangi sırayla ele aldıklarında ve gereken veri yapısındadır. Kruskal ve Borůvka bir **orman**la, yani birden çok ağaçla çalışır. Prim her an tek bir ağaç tutar. Ters silme ise bütün çizgeden başlar ve onu inceltir. Tablodaki yıllar da ilginçtir: en eski algoritma Borůvka'nınkidir ve bilgisayarlardan önce, bir elektrik ağı problemi için yazılmıştır.

4.3.1.8 Uygulama alanları

MST, kuramsal bir merak değildir; birçok mühendislik probleminin ya kendisi ya da bir adımıdır.

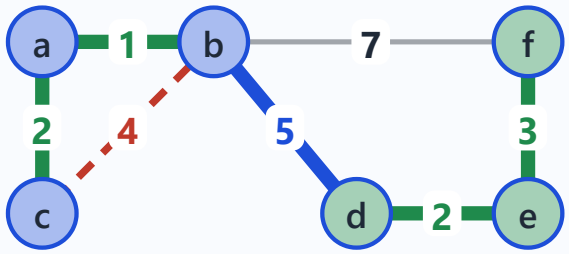
- **Ağ tasarımı.** Şehirleri, binaları ya da cihazları en az kabloyla bağlamak doğrudan bir MST problemidir. Elektrik, su, fiber optik ve yol ağları bu sınıfa girer. Borůvka'nın 1926 çalışmasının çıkış noktası da Moravya'nın elektrik ağıydı.
- **Kümeleme.** Noktalar düğüm, aralarındaki uzaklıklar ağırlık sayılır. MST'den en ağır $k - 1$ kenar silinirse k parça kalır ve her parça bir kümedir. Bu yönteme **tek bağlantılı kümeleme** (*single-linkage clustering*) denir. Dokuz düğümlü Kruskal örneğinde MST'nin en ağır iki kenarı silinince *{A, B}*, *{C, D, E, F, H}* ve *{G, I}* kümeleri oluşur.
- **Yaklaşık algoritmalar.** Metrik gezgin satıcı probleminde (TSP), MST'de derinlik öncelikli gezinip tekrar eden düğümleri atlayan tur, en iyi turun en fazla 2 katıdır. TSP'nin kesin çözümü çok pahalıdır; MST ise hızla bulunur.
- **Görüntü işleme.** Pikseller düğüm, komşu piksellerin renk farkı ağırlık sayılır. MST tabanlı yöntemler benzer renkteki bölgeleri bir araya toplayarak görüntüyü bölütler (*segmentation*).
- **Devre tasarımı.** Bir yonga üzerinde birbirine bağlanması gereken bacakları en kısa toplam iletkenle bağlamak, VLSI yönlendirmesinin ön adımıdır.
- **Başka çizge problemleri.** İki düğüm arasında, en ağır kenarı en hafif olan yola **darboğaz yol** denir. MST'de iki düğüm arasındaki yol her zaman böyle bir yoldur. Steiner ağacı problemi için de MST temel bir yaklaşık çözümdür.

4.3.2 Kruskal algoritması

Kruskal algoritması MST problemine akla gelen en doğal yanıtı verir: önce en ucuz kenarlar alınır, yeter ki döngü oluşmasın. Joseph Kruskal bu algoritmayı 1956'da, gezgin satıcı problemiyle ilgili kısa bir makalede yayımladı. Aynı makalede ters silme algoritmasının fikri de yer alır.

4.3.2.1 Fikir: ormanı birleştirmek

Başlangıçta her düğüm tek başına bir ağaçtır; çizge V ağaçlı bir **orman**dır. Kenarlar ağırlığa göre küçükten büyüğe sıralanır ve bu sırayla incelenir. İncelenen kenar iki *farklı* ağacın düğümlerini bağlıyorsa ağaca eklenir ve bu iki ağaç birleşir. Kenarın iki ucu zaten aynı ağaçtaysa, kenar eklendiğinde bir döngü oluşur; kenar atlanır. Her ekleme ağaç sayısını bir azaltır. $V - 1$ ekleme sonunda tek ağaç kalır ve algoritma biter.



yeşil: seçilen · kırmızı: döngü (atla)
mavi: sıradaki, iki ağacı birleştirir

Şekil 4.3.5: Kruskal'ın ara durumu: iki ağaçlı bir orman. Yeşil kenarlar seçilmiştir. Kırmızı kesikli 4'lük kenar aynı ağacın iki düğümünü bağlar ve döngü kurar; atlanır. Mavi 5'lik kenar iki ağacı birleştirir; eklenir.

Şekildeki ara durumda $\{a, b, c\}$ ve $\{d, e, f\}$ olmak üzere iki ağaç vardır. Sıradaki kenar $bc = 4$'tür. b ile c arasında $b \to a \to c$ yolu zaten vardır; bc eklenirse bu yol ile birlikte bir döngü oluşur. Ardından gelen $bd = 5$ ise iki ayrı ağacı bağlar ve eklenir. Bu noktada orman tek ağaç olur.

Algoritmanın kısa bir değerlendirmesi şöyledir. Kruskal, kenar sayısının düğüm sayısına yakın olduğu **seyrek** çizgelerde çok uygundur. Kodu basittir: bir kenar listesi ve bir ayrık kümeler yapısı yeterlidir. Çizge bağlı değilse algoritma her bileşen için ayrı bir ağaç, yani bir **minimum kapsayan orman** üretir. Zayıf yanları da vardır. Kenarları sıralamanın maliyeti $O(E \log E)$'dir. Kenar sayısının V^2'ye yaklaştığı **yoğun** çizgelerde Prim'in dizi taramalı sürümü daha hızlı olabilir. Döngü denetimi için ek bir yapı gerekir.

4.3.2.2 Adımlar ve sözde kod

Algoritmanın adımları şöyledir:

1. Çizgedeki bütün kenarlar ağırlıklarına göre artan sırada sıralanır.
2. Sıralı kenarlar arasından sıradaki en küçük ağırlıklı kenar seçilir.
3. Kenar farklı ağaçlara ait iki düğümü birleştiriyorsa ağaca eklenir ve iki ağaç birleştirilir.
4. Kenar aynı ağacın iki düğümünü bağlıyorsa döngü oluşturur ve atlanır.
5. *V − 1* kenar seçilene (ya da kenarlar bitene) kadar 2–4. adımlar tekrarlanır.

```
KRUSKAL(G):
  MST = [ ]
  ayrikKume.oluştur(V)
  kenarlar.sırala(artana göre)
  her bir (u, v, ağırlık) için kenarlar içinde:
    eğer ayrikKume.bul(u) ≠ ayrikKume.bul(v):
      MST.ekle((u, v, ağırlık))
      ayrikKume.birleştir(u, v)
  döndür MST
```

Sözde kodun tek gizemli yeri `ayrikKume.bul(u) ≠ ayrikKume.bul(v)` karşılaştırmasıdır. `bul`, bir düğümün hangi ağaçta olduğunu söyleyen bir temsilci döndürür. İki ucun temsilcisi farklıysa kenar iki ayrı ağacı bağlar. `birleştir` ise iki ağacı tek ağaçta toplar. “Bu iki düğüm aynı ağaçta mı?” sorusu ayrık kümeler yapısıyla neredeyse sabit zamanda yanıtlanır. Yapı bir sonraki kesimde ayrıntılı ele alınır.

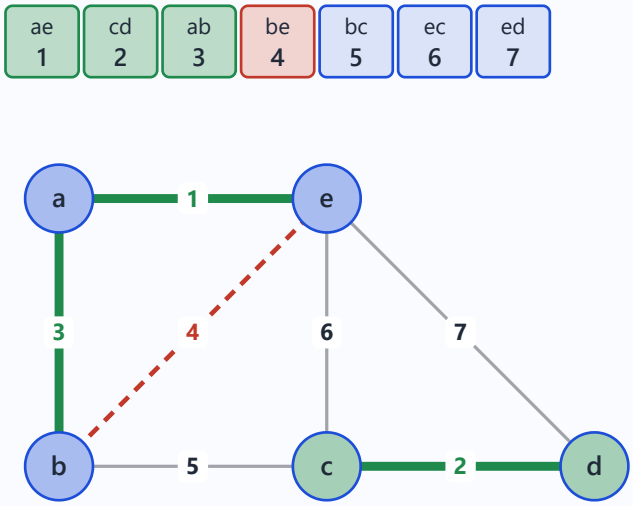
4.3.2.3 Adım adım örnek

Şekil 4.3.1'deki beş düğümlü çizgeye Kruskal uygulanır. Çizgenin kenarları *ab = 3*, *ae = 1*, *bc = 5*, *be = 4*, *cd = 2*, *ed = 7* ve *ec = 6*'dır.

Örnek 4.3.3

1. **Başlangıç.** Her düğüm ayrı bir ağaçtır: *{a} {b} {c} {d} {e}*. Seçilen kenar yoktur, *w(T) = 0*.
2. **Sıralama.** Kenarlar artan ağırlığa göre dizilir: *ae 1, cd 2, ab 3, be 4, bc 5, ec 6, ed 7*.
3. ***ae = 1.*** *a* ve *e* farklı ağaçlardadır; kenar eklenir. Ağaçlar *{a, e} {b} {c} {d}*, *w(T) = 1*.
4. ***cd = 2.*** *c* ve *d* farklı ağaçlardadır; eklenir. Ağaçlar *{a, e} {c, d} {b}*, *w(T) = 3*.
5. ***ab = 3.*** *a* ve *b* farklı ağaçlardadır; eklenir. Ağaçlar *{a, b, e} {c, d}*, *w(T) = 6*.
6. ***be = 4.*** *b* ve *e* zaten aynı ağaçtadır. Kenar eklenirse *b → a → e → b* döngüsü oluşur; atlanır.
7. ***bc = 5.*** *b* ve *c* farklı ağaçlardadır; eklenir. Tek ağaç kalır: *{a, b, c, d, e}*, *w(T) = 11*.

V − 1 = 4 kenar seçilmiştir; algoritma durur. *ec = 6* ve *ed = 7*'ye bakmaya gerek kalmaz. Sonuç Şekil 4.3.1'deki ağaçtır.



Şekil 4.3.6: *be = 4* adımı. Üstteki tablo sıralı kenarları gösterir: yeşiller seçilmiş, kırmızı reddedilmiş, maviler henüz incelenmemiştir. *b* ile *e* aynı ağaçta (mavi düğümler) olduğu için *be* döngü kurar.

İzlemede reddedilen kenar yalnız *be*'dir. Bu kenar Şekil 4.3.1'deki çizgenin dördüncü en hafif kenarıdır. Kruskal onu ağırlığı yüzünden değil, konumu yüzünden atlar: *b* ve *e* daha ucuz iki kenarla zaten birbirine bağlıdır.

Sık yapılan hata

Döngü denetimini "*u* ile *v* arasında seçilmiş bir kenar var mı?" sorusuyla yapmak. Örnekteki *be* adımında *b* ile *e* arasında doğrudan bir kenar seçilmemiştir; yine de ikisi *a* üzerinden bağlıdır ve *be* döngü kurar. Doğru soru "*u* ve *v* aynı bileşende mi?" sorusudur; aradaki yolun uzunluğu önemsizdir.

4.3.2.4 Kruskal neden doğrudur?

Teorem 4.3.5 (Kruskal'ın doğruluğu)

Bağlı bir çizgede Kruskal algoritmasının seçtiği kenarlar bir minimum kapsayan ağaç oluşturur.

İspat. Algoritmanın eklediği her kenarın güvenli olduğu gösterilir. Kruskal bir $e = (u, v)$ kenarını eklemeye karar verdiğinde *u* ve *v* farklı ağaçlardadır. *u*'nun ağacı S olsun ve $(S, V - S)$ kesimi düşünülsün. Seçilmiş kenarların hiçbiri bu kesimi geçmez, çünkü S bir ağacın tamamıdır. Kesimi geçen her kenar iki farklı ağacı bağlar; demek ki bu kenarlardan hiçbiri daha önce döngü gerekçesiyle atlanmamıştır. *e*'den hafif bir kesim kenarı olsaydı, sıralı listede *e*'den önce gelir ve o sırada da iki farklı ağacı bağla-

dığı için eklenirdi. Öyleyse e kesimin en hafif kenarıdır ve kesim özelliğine göre güvenlidir. Atlanan kenarlar ise döngü kurduğu için hiçbir ağaçta yer alamaz. Çizge bağlı olduğundan sonunda $V - 1$ kenar seçilir ve bunlar bir MST oluşturur. □

Ağırlıklar farklıysa sıralı kenar listesi tek türlüdür ve Kruskal her zaman aynı ağacı bulur. Eşit ağırlıklı kenarlar bulunduğunda sıralama onları farklı sıralarla dizebilir. Bu durumda sonuç ağaç değişebilir, ama ağırlığı değişmez: kanıt hangi sıralama seçilirse seçilsin geçerlidir.

4.3.3 Ayrık kümeler (union-find)

Kruskal'ın hızı, "iki düğüm aynı ağaçta mı?" sorusunun ne kadar çabuk yanıtlandığına bağlıdır. Her kenar için BFS ya da DFS ile yol aramak $O(V)$ sürer ve toplam maliyeti $O(E \cdot V)$'ye çıkarır. Ayrık kümeler yapısı aynı soruyu pratikte sabit zamanda yanıtlar.

4.3.3.1 Yapı ve işlemler

Tanım 4.3.4

Ayrık kümeler (*union-find, disjoint sets*) yapısı, kesişmeyen kümelerden oluşan bir koleksiyonu tutar ve üç işlem sunar:

- `oluştur(V)` : her öğe kendi tek öğeli kümesini oluşturur.
- `bul(x)` (*find*): x'in kümesinin **temsilcisini** döndürür. Aynı kümedeki bütün öğeler için aynı temsilci döner.
- `birleştir(x, y)` (*union*): x'in ve y'nin kümelerini tek kümede toplar.

Yapının en yaygın gösterimi bir ormandır. Her küme bir ağaçtır ve ağacın kökü kümenin temsilcisidir. Ağaç bellekte tek bir ebeveyn dizisinde (kodda `ata[]`) tutulur: `ata[i]`, i'nin ebeveynidir; kökte `ata[r] = r`'dir. Bu ağaçların, MST'nin kendisiyle bir ilgisi yoktur. Yalnız "kim hangi kümede?" bilgisini taşırlar ve biçimleri çizgenin kenarlarından farklı olabilir.

`oluştur(V)` her düğüm için `ata[i] = i` yazar. `bul(x)` ebeveyn bağlantılarını izleyerek köke kadar çıkar ve kökü döndürür. `birleştir(x, y)` iki kökü bulur ve birini öbürünün altına bağlar. Kruskal'da kullanım şöyledir: `bul(u) == bul(v)` ise u ve v aynı ağaçtadır ve kenar bir döngü kapatır. Sunumdaki Java kodu şöyledir:

```
int bul(int[] ata, int i) {
  if (ata[i] != i)
    ata[i] = bul(ata, ata[i]); // yol sıkıştırma
  return ata[i];
}
void birlestir(int[] ata, int x, int y) {
  ata[bul(ata, x)] = bul(ata, y);
}
```

`bul` özyinelemelidir. `ata[i] == i` ise *i* köktür ve kendisi döndürülür. Değilse ebeveynin kökü özyinelemeyle bulunur. Dönüşte bulunan kök `ata[i]`'ye yazılır. Bu atama, metodun asıl işi olan kökü bulmanın yanında bir yan etki yaratır: yol üzerindeki her düğüm doğrudan köke bağlanır. Bu iyileştirmeye **yol kısaltma** (*path compression*) denir; kodun yorumundaki "yol sıkıştırma" aynı işlemin öbür adıdır. `birlestir` ise tek satırdır: *x*'in kökünün ebeveyni, *y*'nin kökü yapılır. İki öğe zaten aynı kümedeyse iki `bul` aynı kökü döndürür ve atama kökü kendine bağlar; yapı değişmez.

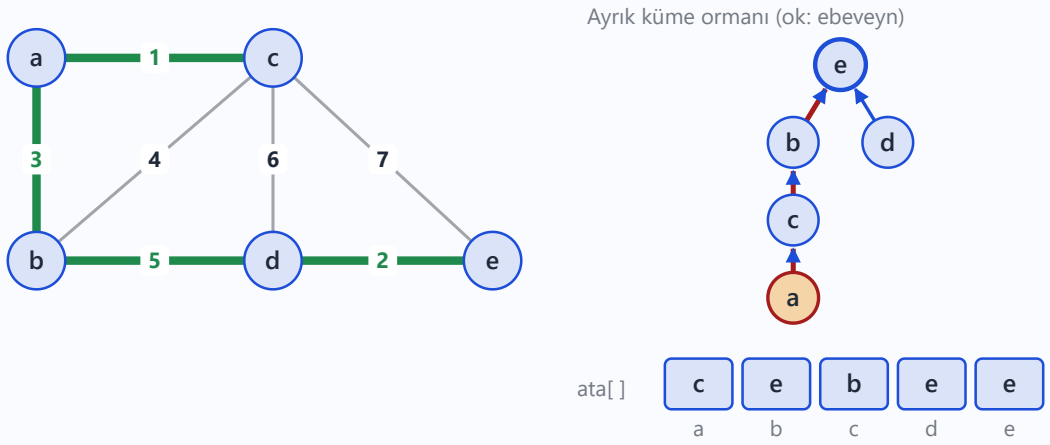
4.3.3.2 Kruskal'da döngü denetimi

Ayrık kümelerin Kruskal içinde nasıl çalıştığı, beş düğümlü bir çizge üzerinde izlenir. Kenarlar sıralıdır: *ac 1*, *de 2*, *ab 3*, *bc 4*, *bd 5*, *cd 6*, *ce 7*. Her kabul edilen *(u, v)* kenarı için `ata[bul(u)] = bul(v)` uygulanır.

Kenar	`bul(u)`	`bul(v)`	Karar	`ata`
başlangıç	—	—	—	a b c d e
ac = 1	a	c	ekle, `ata[a] = c`	c b c d e
de = 2	d	e	ekle, `ata[d] = e`	c b c e e
ab = 3	c	b	ekle, `ata[c] = b`	c b b e e
bc = 4	b	b	aynı kök: döngü, atla	c b b e e
bd = 5	b	e	ekle, `ata[b] = e`	c e b e e

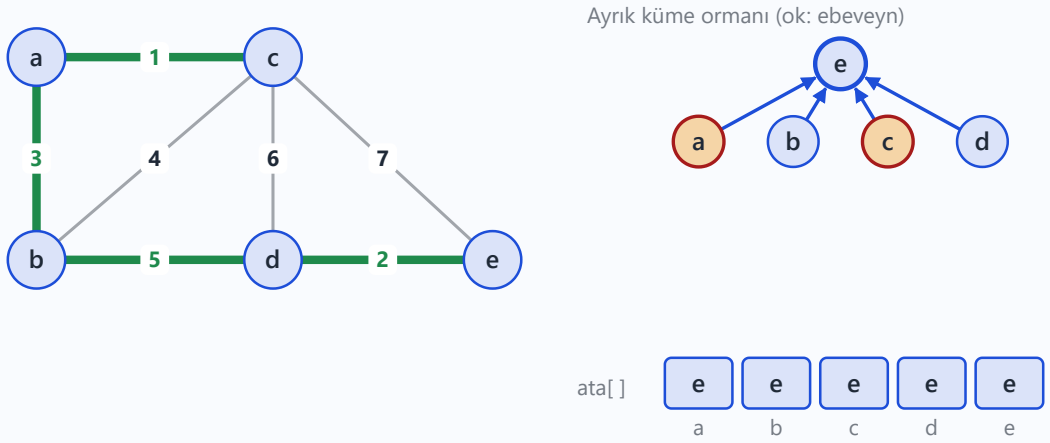
Tablo 4.3.4: Kruskal'da ayrık kümelerle döngü denetimi. Son sütun `ata` dizisini *a, b, c, d, e* sırasıyla verir.

ab adımında `bul(a)` önce *a*'nın ebeveyni *c*'ye çıkar. *c* köktür; kök *c* döner. `bul(b)` ise *b*'nin kendisini döndürür. Kökler farklı olduğu için kenar eklenir ve *c* kökü *b*'nin altına bağlanır. *bc* adımında `bul(b) = b` ve `bul(c)` de $c \to b$ yoluyla *b*'yi bulur. Kökler aynıdır; kenar döngü kapatır. Dört kenar seçildiğinde MST tamamlanır: $ac + de + ab + bd = 1 + 2 + 3 + 5 = 11$.



Şekil 4.3.7: MST tamamlandıktan sonra ayrık küme ormanı. Ebeveyn bağlantıları $a \to c \to b \to e$ zincirini kurmuştur; `bul(a)` köke 3 adımda ulaşır.

MST doğru bulunmuştur, ama ayrık küme ormanı uzun bir zincire dönüşmüştür. `bul(a)` çağrısı $a \to c \to b \to e$ yolunu izler. Bu durumda yol kısaltma devreye girer. `bul(a)` dönerken yol üzerindeki her düğümü doğrudan köke bağlar: `ata[a] = ata[c] = e` olur. b zaten köke bağlıdır. Sonraki aramalar tek adımda biter.



Şekil 4.3.8: `bul(a)` çağrısından sonra orman: yol üzerindeki a ve c doğrudan e köküne bağlanmıştır; `ata` dizisi $e\ e\ e\ e\ e$ olur.

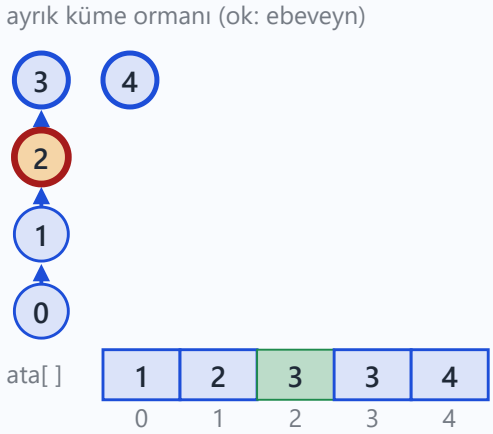
4.3.3.3 Kod izleme: bul ve birlestir

Yol kısaltmanın özyineleme içinde nasıl gerçekleştiği, *0, ..., 4* öğeli küçük bir örnekle adım adım izlenir. Çağrı sırası `birlestir(0, 1)`, `birlestir(1, 2)`, `birlestir(2, 3)`, `bul(0)` ve `birlestir(4, 0)`'dır.

Adım	Olay	`ata`
0	başlangıç, her öğe kök	[0, 1, 2, 3, 4]

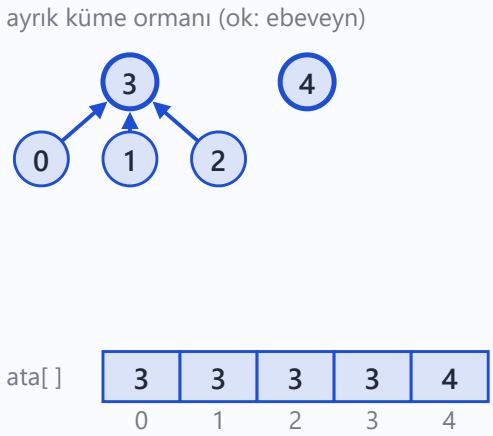
Adım	Olay	`ata`
1	`birlestir(0, 1)` : `bul(0) = 0`, `bul(1) = 1` → `ata[0] = 1`	[1, 1, 2, 3, 4]
2	`birlestir(1, 2)` : kökler 1 ve 2 → `ata[1] = 2`	[1, 2, 2, 3, 4]
3	`birlestir(2, 3)` : kökler 2 ve 3 → `ata[2] = 3`	[1, 2, 3, 3, 4]
4	`bul(0)` iner: 0 → 1 → 2 → 3; 3 köktür	[1, 2, 3, 3, 4]
5	dönüş: `ata[2] = 3` (değişmez)	[1, 2, 3, 3, 4]
6	dönüş: `ata[1] = 3`	[1, 3, 3, 3, 4]
7	dönüş: `ata[0] = 3` ; `bul(0) = 3`	[3, 3, 3, 3, 4]
8	`birlestir(4, 0)` : `bul(4) = 4`, `bul(0) = 3` (tek adım) → `ata[4] = 3`	[3, 3, 3, 3, 3]

Tablo 4.3.5: `bul` ve `birlestir` izlemesi. `ata` dizisi 0, 1, 2, 3, 4 indisleri için verilmiştir.



Şekil 4.3.9: Üç birleştirmeden sonra orman: kök kökün altına bağlandığı için *0 → 1 → 2 → 3* zinciri oluşmuştur.

İlk üç birleştirme kötü bir örüntü üretir. Her seferinde bir kök öbür kökün altına bağlanır ve zincir bir uzar. `bul(0)` artık üç adım sürer. Genel durumda n öğe bu biçimde zincirlenirse tek bir `bul` $n - 1$ adım sürer. `bul(0)` çağrısı özyinelemeyle köke iner: önce `bul(1)`, sonra `bul(2)`, sonra `bul(3)` çağrılır. `bul(3)` kökü bulur ve 3 döndürür. Özyineleme geri sarılırken her çerçeve kendi `ata[i]` değerine 3 yazar. Sonuçta 0, 1 ve 2 doğrudan 3'e bağlanır. Son birleştirmede `bul(0)` tek adımda biter ve 4 de 3'ün altına girer; bütün öğeler tek kümededir.



Şekil 4.3.10: `bul(0)` çağrısından sonra orman: yol üzerindeki 0, 1 ve 2 doğrudan kök 3'e bağlanmıştır.

Sık yapılan hata

`birlestir`'i `ata[x] = y` diye yazmak. Bu atama x'in kendisini y'ye bağlar; x kök değilse x eski ağacından koparılır, ağacın geri kalanı ise y'nin kümesine katılmaz. Örneğin `ata = [1, 1, 2]` iken `ata[0] = 2` yazılırsa 0 ile 2 birleşir, ama 1 dışarıda kalır. Her zaman *kökler* birbirine bağlanmalıdır: `ata[bul(x)] = bul(y)`.

4.3.3.4 Yol kısaltma ve rütbeye göre birleştirme

Ayrık kümelerin hızı ağaçların yüksekliğine bağlıdır. İki iyileştirme yüksekliği küçük tutar. Birincisi yukarıda görülen yol kısaltmadır. İkincisi birleştirme anında hangi kökün hangisinin altına gireceğini akıllıca seçer. Her kökte bir **rütbe** (*rank*) tutulur; rütbe, ağacın yüksekliği için bir üst sınırdır. Birleştirmede rütbesi küçük kök, rütbesi büyük kökün altına bağlanır. Rütbeler eşitse biri öbürünün altına girer ve yeni kökün rütbesi bir artar. Bu yönteme **rütbeye göre birleştirme** (*union by rank*) denir.

```
void birlestirRutbe(int[] ata, int[] rutbe, int x, int y) {
  int a = bul(ata, x), b = bul(ata, y);
  if (a == b) return;                    // zaten aynı küme
  if (rutbe[a] < rutbe[b]) ata[a] = b;   // alçak ağaç yüksek ağacın altına
  else if (rutbe[a] > rutbe[b]) ata[b] = a;
  else { ata[b] = a; rutbe[a]++; }       // eşitse biri altına, rütbe artar
}
```

Kod önce iki kökü bulur; kökler aynıysa iş yoktur. Rütbeler farklıysa alçak ağaç yüksek ağacın altına girer ve hiçbir rütbe değişmez: yüksek ağacın yüksekliği artmaz. Yalnız eşit rütbeli iki ağaç birleştiğinde yükseklik bir artabilir; bu yüzden yalnız o durumda rütbe artırılır. Önceki izlemedeki çağrı dizisi bu metotla tekrarlanırsa zincir oluşmaz. `birlestirRutbe(0, 1)` rütbeleri eşit bulur, 1'i 0'ın altına

bağlar ve 0'ın rütbesini 1 yapar. Sonraki birleştirmelerde 2, 3 ve 4 tek öğeli, rütbesi 0 olan ağaçlardır; hepsi rütbesi 1 olan 0'ın altına girer. Sonuçta `ata = [0, 0, 0, 0, 0]` olur ve ağacın yüksekliği 1'de kalır.

> **Teorem 4.3.6 (Rütbe ve küme boyu)**
>
> Rütbeye göre birleştirmede rütbesi r olan bir kökün ağacında en az 2^r öğe bulunur. Bu yüzden n öğeli yapıda her ağacın yüksekliği en fazla $\lfloor \log_2 n \rfloor$'dir ve bir `bul` işlemi $O(\log n)$ sürer.

İspat. Tümevarım kullanılır. Başlangıçta her ağaç tek öğelidir ve rütbesi 0'dır: $1 \geq 2^0$. Rütbesi farklı iki ağaç birleşince yeni kökün rütbesi değişmez ama ağacı büyür; eşitsizlik korunur. Rütbesi r olan iki ağaç birleşince yeni kökün rütbesi $r + 1$ olur. Her iki ağaçta en az 2^r öğe vardır; yeni ağaçta en az $2 \cdot 2^r = 2^{r+1}$ öğe bulunur. Toplam öğe sayısı n olduğu için $2^r \leq n$, yani $r \leq \log_2 n$'dir. Yükseklik rütbeyi aşmadığı için aynı sınır yükseklik için de geçerlidir. □

Sınır sıkıdır. 8 öğe ikişer ikişer, sonra dörder dörder, sonra sekizli olarak eşit rütbeli ağaçlar hâlinde birleştirilirse rütbe 3'e ulaşır; 1024 öğede aynı yöntem rütbeyi 10'a çıkarır. Yol kısaltma da eklendiğinde yapı çok daha hızlıdır. Robert Tarjan 1975'te iki iyileştirme birlikte kullanıldığında m işlemin toplam maliyetinin $O(m \cdot \alpha(n))$ olduğunu gösterdi. Burada α, çok hızlı büyüyen Ackermann fonksiyonunun tersidir. Ters Ackermann fonksiyonu o kadar yavaş büyür ki evrendeki atom sayısını aşmayan her pratik n için $\alpha(n) \leq 4$'tür. Bu yüzden işlem başına maliyet **amortize** olarak sabit kabul edilir. Amortize maliyet, uzun bir işlem dizisinde işlem başına düşen ortalama maliyettir. Tek bir `bul` yine de uzun sürebilir, ama bu uzun yürüyüş sonraki çağrıları hızlandırır.

> **Sık yapılan hata**
>
> Yol kısaltmanın tek bir `bul` çağrısını hızlandırdığını sanmak. Kısaltma yapan çağrının kendisi uzun yolu baştan sona yürür. Kazanç *sonraki* çağrılardadır: yol üzerindeki düğümler artık köke bir adım uzaklıktadır. Tek bir işlemin en kötü maliyeti ile amortize maliyet bu yüzden ayrı ayrı değerlendirilmelidir.

4.3.4 Kruskal kodu ve karmaşıklığı

4.3.4.1 Java kodu

Kruskal'ın Java uygulaması bir kenar sınıfına dayanır. Kenar iki uç düğümü ve ağırlığı tutar. `Collections.sort` ile sıralanabilmesi için `Comparable` arayüzünü uygular ve kenarları ağırlığa göre karşılaştırır:

```
class Kenar implements Comparable<Kenar> {
  int u, v, w;                  // uçlar ve ağırlık
  Kenar(int u, int v, int w) { this.u = u; this.v = v; this.w = w; }
  public int compareTo(Kenar o) {
    return Integer.compare(w, o.w);   // ağırlığa göre artan
  }
}
```

`Integer.compare(w, o.w)`, `w - o.w` çıkarmasının güvenli karşılığıdır. Çok büyük ya da negatif ağırlıklarda çıkarma taşabilir; `Integer.compare` taşmaz. Düğümler 1'den *V*'ye kadar numaralanır. Sunumdaki Kruskal kodu şöyledir:

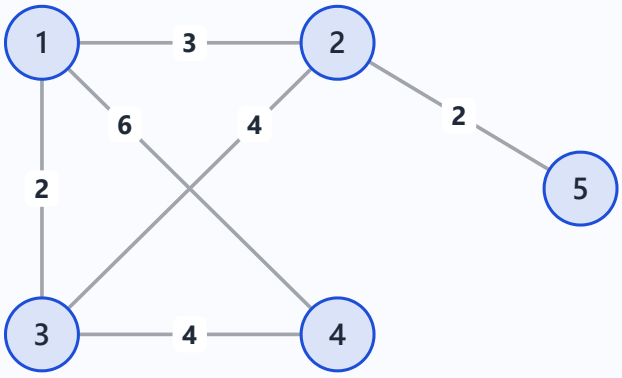
```
void kruskal(int V, List<Kenar> ke) {
  Collections.sort(ke);         // artan
  int[] ata = new int[V + 1];
  for (int i = 1; i <= V; i++) ata[i] = i;
  List<Kenar> agac = new ArrayList<>();
  int k = 0;
  while (agac.size() < V - 1) {
    Kenar e = ke.get(k++);
    int x = bul(ata, e.u);
    int y = bul(ata, e.v);
    if (x != y) {               // ayrı ağaç
      agac.add(e);
      ata[x] = y;               // birleştir
    }
  }
}
```

Kod satır satır şöyle çalışır. `Collections.sort(ke)` kenar listesini ağırlığa göre artan sıraya koyar. Java'nın liste sıralaması kararlıdır: eşit ağırlıklı kenarlar listedeki ilk sıralarını korur. `ata` dizisi *V + 1* uzunluktadır, çünkü düğümler 1'den başlar ve 0. hücre kullanılmaz. Döngü her düğümü kendi kümesinin kökü yapar. `agac` seçilen kenarları, `k` ise sıralı listede sıradaki kenarın indisini tutar.

`while (agac.size() < V - 1)` döngüsü *V − 1* kenar seçilene kadar sürer. Her turda sıradaki kenar alınır ve `k` bir artar. İki ucun kökü `x` ve `y` olarak bulunur. Kökler farklıysa kenar ağaca eklenir ve `ata[x] = y` ile iki küme birleşir. Burada `birlestir` çağrılmaz, çünkü kökler zaten bulunmuştur; atama doğrudan yapılır. Kökler aynıysa hiçbir şey yapılmaz ve döngü sıradaki kenara geçer.

4.3.4.2 Kod izleme: beş düğümlü çizge

Kod, düğümleri 1–5 olan bir çizge üzerinde izlenir. Kenarlar *(1,3) = 2*, *(2,5) = 2*, *(1,2) = 3*, *(2,3) = 4*, *(3,4) = 4* ve *(1,4) = 6*'dır.

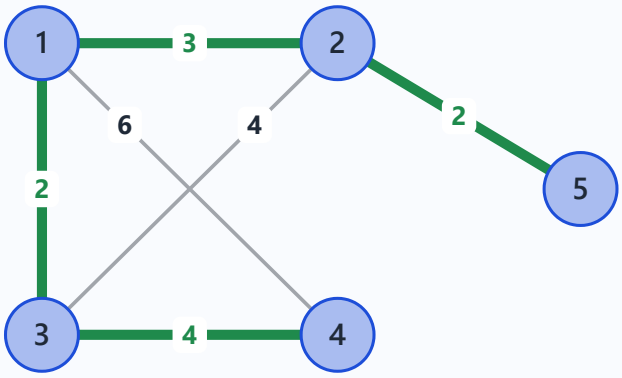


Şekil 4.3.11: Beş düğümlü çizge: *(1,3) = 2*, *(2,5) = 2*, *(1,2) = 3*, *(2,3) = 4*, *(3,4) = 4*, *(1,4) = 6*. Aynı çizge Prim izlemelerinde de kullanılır.

Kenar `e`	`x = bul(u)`	`y = bul(v)`	**İşlem**	`ata`	***w(T)***
—	—	—	ilkleme	[1, 2, 3, 4, 5]	0
(1,3) = 2	1	3	ekle, `ata[1] = 3`	[3, 2, 3, 4, 5]	2
(2,5) = 2	2	5	ekle, `ata[2] = 5`	[3, 5, 3, 4, 5]	4
(1,2) = 3	3	5	ekle, `ata[3] = 5`	[3, 5, 5, 4, 5]	7
(2,3) = 4	5	5	döngü, atla	[3, 5, 5, 4, 5]	7
(3,4) = 4	5	4	ekle, `ata[5] = 4`	[3, 5, 5, 4, 4]	11

Tablo 4.3.6: `kruskal(5, ke)` izlemesi. `ata` 1–5 indisleri için verilmiştir.

Üçüncü satırda `bul(1)` 1'in ebeveyni 3'e çıkar ve 3'ü döndürür; `bul(2)` 5'i döndürür. Kökler farklıdır ve `ata[3] = 5` ile iki ağaç birleşir. Dikkat edilirse değişen hücre 1'in ya da 2'nin hücresi değil, kök 3'ün hücresidir. Dördüncü satırda `bul(2)` ve `bul(3)` ikisi de 5'i bulur; *(2,3)* iki ayrı ağacı birleştirmediği için döngü kurar ve atlanır. Beşinci satırda 4 tek başınadır; *(3,4)* eklenir ve `ata[5] = 4` olur. `agac.size()` artık *4 = V − 1*'dir; döngü biter. *(1,4) = 6* hiç incelenmez. MST *(1,3)*, *(2,5)*, *(1,2)*, *(3,4)* kenarlarından oluşur ve *w(T) = 11*'dir.



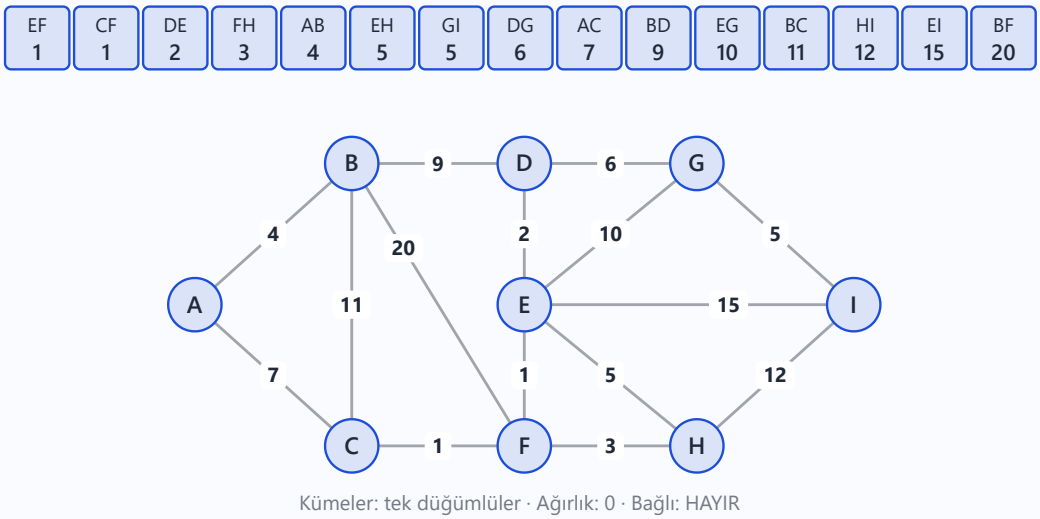
Şekil 4.3.12: Kruskal'ın sonucu: *(1,3)*, *(2,5)*, *(1,2)*, *(3,4)*; toplam ağırlık 11.

İzlemede eşit ağırlıklı iki çift vardır: *(1,3)* ile *(2,5)* ve *(2,3)* ile *(3,4)*. Kararlı sıralama onları listedeki sırada bırakır. *(3,4)* listede *(2,3)*'ten önce gelseydi önce *(3,4)* eklenir ve ağaç dört kenarla tamamlandığı için *(2,3)*'e hiç bakılmazdı. Bu çizgede sonuç değişmez. *(2,3)*'ün atlanmasının nedeni *2 → 1 → 3* yolunun daha ucuz kenarlarla zaten kurulmuş olmasıdır.

4.3.4.3 Dokuz düğümlü örnek

Daha büyük bir örnek olarak *A, ..., I* düğümlü, 15 kenarlı çizge ele alınır. Sıralı kenarlar şöyledir:

> EF 1, CF 1, DE 2, FH 3, AB 4, EH 5, GI 5, DG 6, AC 7, BD 9, EG 10, BC 11, HI 12, EI 15, BF 20

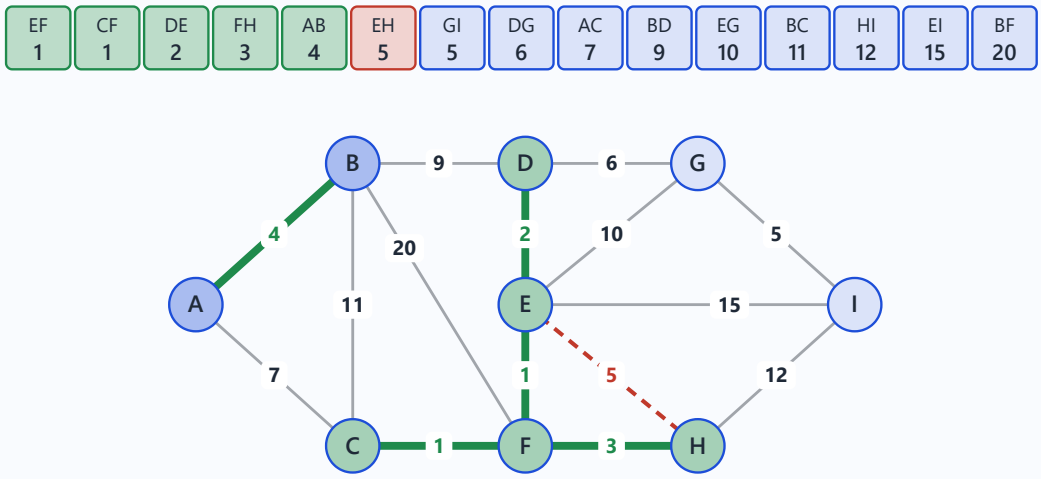


Şekil 4.3.13: Dokuz düğümlü çizge ve sıralı kenar tablosu. Henüz hiçbir kenar seçilmemiştir.

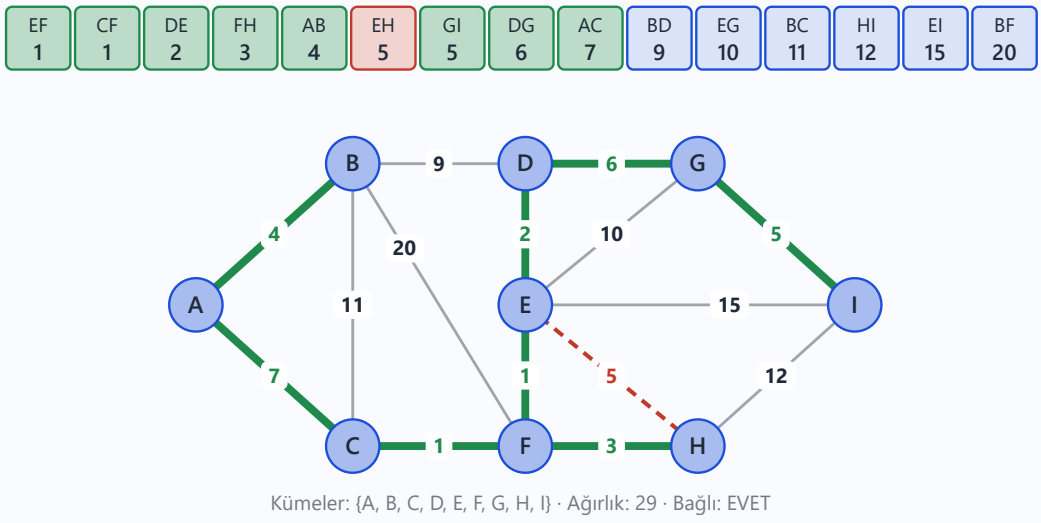
Kenar	Karar	Kümeler (tek düğümlüler hariç)	Toplam
EF 1	ekle	{E, F}	1
CF 1	ekle	{C, E, F}	2
DE 2	ekle	{C, D, E, F}	4
FH 3	ekle	{C, D, E, F, H}	7
AB 4	ekle	{C, D, E, F, H} {A, B}	11
EH 5	döngü (E–F–H)	değişmez	11
GI 5	ekle	{C, D, E, F, H} {A, B} {G, I}	16
DG 6	ekle	{C, D, E, F, G, H, I} {A, B}	22
AC 7	ekle	{A, B, C, D, E, F, G, H, I}	29

Tablo 4.3.7: Dokuz düğümlü çizgede Kruskal.

Dokuzuncu incelenen kenar *AC = 7*'dir ve sekizinci eklenen kenardır. *8 = V − 1* olduğu için algoritma burada durur; BD 9 ile başlayan son altı kenara bakılmaz. MST'nin ağırlığı 29'dur. Bu ağaç, uygulama alanlarında söz edilen kümeleme örneğinin de kaynağıdır. En ağır iki MST kenarı *AC = 7* ve *DG = 6*'dır. Bu iki kenar silinirse üç küme kalır: *{A, B}*, *{C, D, E, F, H}* ve *{G, I}*.



Şekil 4.3.14: *EH = 5* adımı. E ve H, *E–F–H* yolu üzerinden zaten aynı kümededir; kenar döngü kurar ve eklenmez.



Şekil 4.3.15: *AC = 7* eklendikten sonra: sekiz kenar seçilmiştir, çizge bağlıdır ve MST ağırlığı 29'dur.

4.3.4.4 Karmaşıklık

Kruskal'ın maliyeti işlemlerine ayrılarak hesaplanır. Tablo 4.3.8 her işlemin kaç kez yapıldığını ve maliyetini verir.

İşlem	Kaç kez	Maliyet
Kenarları ağırlığa göre sıralama	1	$O(E \log E)$
`oluştur(V)`	1	$O(V)$
`bul` (her kenar için 2)	$\leq 2E$	$O(E \cdot \alpha(V))$ yol kısaltma + rütbe ile; $O(E \log V)$ yalnız rütbe ile
`birleştir`	$V - 1$	

Tablo 4.3.8: Kruskal'ın karmaşıklık analizi.

Toplam $O(E \log E)$'dir ve bu maliyeti sıralama belirler. Bağlı bir çizgede $E \geq V - 1$ olduğu için $O(V)$'lik ilkleme de sıralamanın yanında küçük kalır. Çoğu kaynakta aynı sınır $O(E \log V)$ diye yazılır. İki yazım aynı mertebeyi gösterir. Yönsüz basit bir çizgede $E \leq V(V-1)/2 < V^2$'dir. Logaritma alınırsa $\log E < 2 \log V$ bulunur. Sabit çarpanlar mertebeyi değiştirmediği için $O(E \log E) = O(E \log V)$'dir. Örneğin $V = 1000$ düğümlü tam çizgede $E = 499.500$'dür; $\log_2 E \approx 18{,}9$, $2 \log_2 V \approx 19{,}9$'dur.

Kenarlar zaten sıralı geliyorsa ya da ağırlıklar küçük tam sayılarsa durum değişir. İkinci durumda saymalı sıralama kenarları doğrusal zamanda sıralar. Bu durumlarda toplam süre neredeyse doğrusaldır: $O(E \cdot \alpha(V))$. Ayrık kümelerde yalnız rütbeye göre birleştirme kullanılırsa her işlem en kötü $O(\log V)$ sürer; yol kısaltma da eklenince işlem başına amortize maliyet $O(\alpha(V))$'ye iner. Her iki durumda da toplamı sıralama belirler. Sunumdaki kod yalnız yol kısaltma yapar; rütbe kullanmaz. Bu kodun da pratikte çok hızlı olduğu bilinir.

Bellek gereksinimi kenar listesi ve ebeveyn dizisi kadardır: $O(V + E)$. Kenar listesi zaten girdinin kendisidir; ek bellek `ata` dizisi ve seçilen $V - 1$ kenardır.

4.3.4.5 Bağlı olmayan çizge

Sunumdaki kod, çizgenin bağlı olduğunu varsayar. Bağlı olmayan bir çizgede $V - 1$ kenar asla seçilemez. Liste bittiğinde döngü koşulu hâlâ doğrudur ve `ke.get(k++)` liste sınırının dışına çıkar.

Örnek 4.3.4

$V = 6$ ve kenarlar $(1,2) = 2$, $(2,3) = 3$, $(4,5) = 1$, $(5,6) = 4$, $(4,6) = 5$ olsun. Çizgenin iki bileşeni vardır: $\{1, 2, 3\}$ ve $\{4, 5, 6\}$. Sıralı liste $(4,5), (1,2), (2,3), (5,6), (4,6)$'dır. İlk dört kenar eklenir; $(4,6)$ döngü kurar ve atlanır. `agac.size() = 4 < 5` olduğu için döngü sürer ve `ke.get(5)` çağrısı `IndexOutOfBoundsException` fırlatır.

Çözüm, döngü koşuluna liste sınırını eklemektir. Sonda `agac.size() < V - 1` ise çizge bağlı değildir. Bu durumda seçilen kenarlar bir **minimum kapsayan orman** oluşturur: her bileşen için bir minimum kapsayan ağaç.

```
List<Kenar> kruskalOrman(int V, List<Kenar> ke) {
  Collections.sort(ke);
  int[] ata = new int[V + 1];
  for (int i = 1; i <= V; i++) ata[i] = i;
  List<Kenar> agac = new ArrayList<>();
  int k = 0;
  while (k < ke.size() && agac.size() < V - 1) {
    Kenar e = ke.get(k++);
    int x = bul(ata, e.u), y = bul(ata, e.v);
    if (x != y) { agac.add(e); ata[x] = y; }
  }
  return agac;                  // agac.size() < V - 1 ise orman
}
```

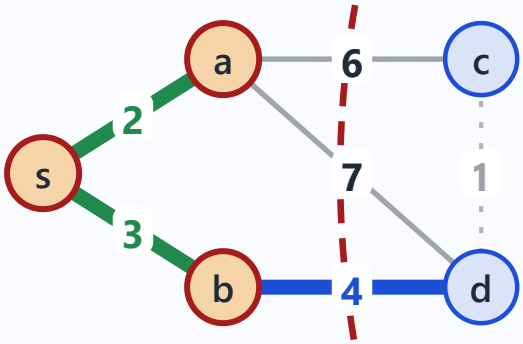
Örnekteki çizgede bu sürüm *(4,5), (1,2), (2,3), (5,6)* kenarlarını döndürür. Orman iki ağaçlıdır ve toplam ağırlığı *1 + 2 + 3 + 4 = 10*'dur. Kenar sayısı *V − 2*'dir. Genel olarak *c* bileşenli bir çizgenin kapsayan ormanında *V − c* kenar bulunur.

4.3.5 Prim algoritması

Kruskal bütün çizgeye yayılmış küçük ağaçlarla çalışır. Prim ise tek bir ağaçla başlar ve onu adım adım büyütür. Algoritma Robert C. Prim'in 1957 tarihli makalesiyle tanındı. Aynı yöntemi Vojtěch Jarník 1930'da bulmuştu; Edsger W. Dijkstra 1959'da yeniden keşfetti. Bu yüzden bazı kaynaklarda Jarník algoritması ya da Prim–Jarník algoritması adıyla geçer.

4.3.5.1 Fikir: tek ağacı büyütmek

Bir başlangıç düğümü seçilir; ağaç başta yalnız bu düğümden oluşur. Her adımda ağaçta olmayan düğümler arasından, ağaca *en ucuz kenarla* bağlanan düğüm seçilir. Bu düğüm ve onu bağlayan kenar ağaca eklenir. Tüm düğümler ağaca girince algoritma biter. Her an tek bir ağaç vardır; Kruskal'daki orman yoktur.



Şekil 4.3.16: Prim'in ara durumu. Ağaç *{s, a, b}*'dir. Kesikli çizgi ağaç ile geri kalan düğümler arasındaki kesimi gösterir. Kesimi geçen kenarlar 6, 7 ve 4'tür; en ucuzu 4 seçilir ve *d* ağaca girer.

Prim her adımda bir kesimin en hafif kenarını seçer: kesim, o ana kadar kurulan ağaç ile geri kalan düğümler arasındadır. Şekildeki *c–d = 1* kenarı çizgenin en hafif kenarıdır, ama kesimi geçmez; iki ucu da ağaçta değildir. Prim bu kenarı ancak *d* ağaca girdikten sonra kullanabilir. Kruskal ise aynı çizgede işe *c–d = 1* ile başlardı.

Algoritmanın değerlendirmesi şöyledir. Prim yoğun çizgelerde çok uygundur: komşuluk matrisi ile $O(V^2)$ sürede çalışır. Kenarları sıralamaya gerek yoktur. Komşuluk listesi ya da matrisiyle doğal biçimde çalışır. Zayıf yanı, başarımının kullanılan veri yapısına bağlı olmasıdır: dizi, ikili heap ve Fibonacci heap farklı süreler verir. Bağlı olmayan çizgede yalnız başlangıç düğümünün bileşenini kapsar.

4.3.5.2 Anahtar ve ebeveyn

"Ağaca en ucuz kenarla bağlanan düğüm" her adımda yeniden aranırsa algoritma yavaşlar. Prim bu bilgiyi her düğüm için iki dizide saklar ve ağaç büyüdükçe günceller.

Tanım 4.3.5

Prim algoritmasında ağaçta olmayan bir *v* düğümü için:

- `anahtar[v]` (*key*): *v*'yi şimdiye kadar kurulan ağaca bağlayan en ucuz kenarın ağırlığıdır. Başta ∞, kaynakta 0'dır.
- `ata[v]` (*parent*, π): bu en ucuz kenarın ağaçtaki ucudur. Algoritma bittiğinde *(ata[v], v)* kenarları MST'yi verir.

Ağaca yeni bir *u* düğümü girdiğinde yalnız *u*'nun komşularının anahtarı değişebilir; çünkü ağaca bağlanmanın yeni yolları yalnız *u* üzerinden açılır. Ağaçta olmayan her *v* komşusu için *w(u, v) < anahtar[v]* ise anahtar ve ebeveyn güncellenir: `anahtar[v] = w(u, v)`, `ata[v] = u`. Ardından anah-

tarı en küçük olan ağaç dışı düğüm seçilir. Bu düğümün anahtarı, kesimi geçen en hafif kenarın ağırlığıdır.

4.3.5.3 Sözde kod

Prim'in adımları şöyle sıralanabilir:

1. Başlangıç düğümü seçilir ve bu düğüme ait bütün kenarlar öncelikli kuyruğa eklenir.
2. Kuyruktan, ağaçta olmayan bir düğüme giden en küçük ağırlıklı kenar seçilir.
3. Seçilen kenar ile bağlantılı yeni düğüm ağaca eklenir.
4. Yeni düğüme bağlı kenarlar kuyruğa eklenir; bütün düğümler ağaca girene kadar 2–4. adımlar tekrarlanır.

Sunumdaki sözde kod iki parçadan oluşur. İlk parça dizileri ilkler, ikinci parça ana döngüdür:

```
PRIM(G, kaynak):
  MST = [ ]
  mesafeler = [ ]
  ata = [ ]
  ziyaretEdildi = [ ]
  her bir v için V içinde:
    mesafeler[v] = ∞
    ziyaretEdildi[v] = 0
  mesafeler[kaynak] = 0
  öncelikKuyruğu.ekle(kaynak, 0)
```

```
döngü öncelikKuyruğu boş değil iken:
  u = öncelikKuyruğu.çıkar()
  eğer ziyaretEdildi[u] == 0:
    ziyaretEdildi[u] = 1
    her bir (u, v, ağırlık) için G[u] içinde:
      eğer ziyaretEdildi[v] == 0 ve
           ağırlık < mesafeler[v]:
        mesafeler[v] = ağırlık
        ata[v] = u
        öncelikKuyruğu.ekle(v, ağırlık)
her bir v için V içinde (kaynak hariç):
  MST.ekle((ata[v], v, mesafeler[v]))
döndür MST
```

Sözde koddaki `mesafeler` dizisi, tanımdaki `anahtar` dizisidir. Ad yanıltıcı olabilir: değer bir uzaklık değil, tek bir kenarın ağırlığıdır. Kod bir düğümü kuyruğa birden çok kez ekleyebilir. Bir düğümün anahtarı her küçüldüğünde kuyruğa yeni bir kayıt girer; eski kayıt silinmez, kuyrukta kalır. Bu yüzden

bu sürüme **tembel** (*lazy*) Prim denir. Eski kayıt sonunda kuyruktan çıktığında `ziyaretEdildi[u]` denetimi onu atar. Kuyruk en fazla E kadar kayıt tutar; her kenar bir anahtar küçülmesine yol açabilir. Toplam süre $O(E \log E)$'dir.

> **Sık yapılan hata**
>
> Tembel sürümde `ziyaretEdildi` denetimini atlamak. Eski bir kayıt kuyruktan çıktığında düğüm zaten ağaçtadır. Denetim olmazsa düğüm ikinci kez "ağaca girer", komşuları yeniden incelenir ve ebeveyn dizisi bozulabilir. Her kuyruk çıkarımından sonra ilk iş, düğümün daha önce işlenip işlenmediğine bakmaktır.

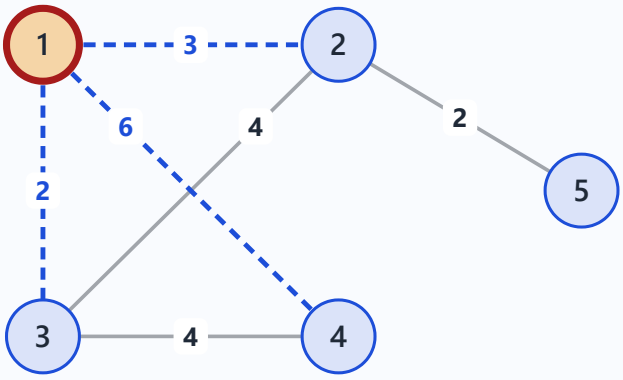
4.3.5.4 Adım adım: öncelikli kuyruk

Prim, Şekil 4.3.11'deki beş düğümlü çizgede 1'den başlatılır. Burada ders kitaplarındaki klasik gösterim kullanılır: kuyruk Q bütün düğümlerle başlar ve her düğüm anahtarıyla birlikte kuyrukta durur. Tablolardaki π, ebeveyndir; "/" henüz ebeveyn atanmadığını gösterir.

Adım	Çıkan düğüm	Eklenen kenar	Kuyruk Q (sonrası)	$w(T)$
0	—	—	1 (0, /), 2 (∞, /), 3 (∞, /), 4 (∞, /), 5 (∞, /)	0
1	1	—	3 (2, 1), 2 (3, 1), 4 (6, 1), 5 (∞, /)	0
2	3	*(1,3) = 2*	2 (3, 1), 4 (4, 3), 5 (∞, /)	2
3	2	*(1,2) = 3*	5 (2, 2), 4 (4, 3)	5
4	5	*(2,5) = 2*	4 (4, 3)	7
5	4	*(3,4) = 4*	boş	11

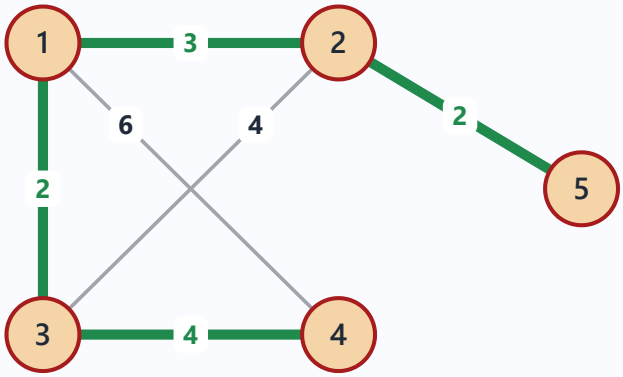
Tablo 4.3.9: Öncelikli kuyruklu Prim, kaynak 1. Kuyruk içeriği anahtara göre sıralıdır; her öğe *düğüm (anahtar, π)* biçimindedir.

Adımlar şöyle açıklanır. İlklemede yalnız kaynağın anahtarı 0'dır; bu yüzden ilk çıkan düğüm 1'dir. 1'in komşuları güncellenir: $key(3) = 2$, $key(2) = 3$, $key(4) = 6$. İkinci adımda en küçük anahtar 3'ündür; 3 çıkar ve *(1,3)* ağaca girer. 3'ün komşusu 4 için $(3,4) = 4 < 6$ olduğundan $key(4) = 4$, $\pi(4) = 3$ olur. Öbür komşu 2 için $(2,3) = 4 > key(2) = 3$ olduğundan değişiklik yoktur. Üçüncü adımda 2 çıkar ve *(1,2)* eklenir; 5'in anahtarı 2 olur. Dördüncü adımda 5 çıkar; ağaçta olmayan başka komşusu yoktur. Son adımda 4 çıkar ve *(3,4)* eklenir.



Şekil 4.3.17: 1 kuyruktan çıktıktan sonra durum. Kesikli mavi çizgiler π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantılardır: 3'e 2, 2'ye 3, 4'e 6.

Sonuç ağaç *(1,3), (1,2), (2,5), (3,4)* kenarlarından oluşur ve *w(T) = 11*'dir. Kruskal aynı çizgede aynı ağacı bulmuştu. Kenarların eklenme sırası ise farklıdır: Kruskal 2, 2, 3, 4 sırasıyla ekler; Prim 2, 3, 2, 4 sırasıyla. Prim'in 2'lik *(2,5)* kenarını 3'lük *(1,2)*'den sonra eklemesinin nedeni, 5'in ancak 2 ağaca girdikten sonra ağaca komşu olmasıdır.



Şekil 4.3.18: Prim'in son durumu: kuyruk boştur. MST *(1,3), (1,2), (2,5), (3,4)* kenarlarından oluşur; *w(T) = 11*.

4.3.5.5 Prim neden doğrudur?

Teorem 4.3.7 (Prim'in doğruluğu)

Bağlı bir çizgede Prim algoritmasının seçtiği *(ata[v], v)* kenarları bir minimum kapsayan ağaç oluşturur.

İspat. Bir adımın başında ağaçtaki düğümler *S* olsun ve *(S, V − S)* kesimi düşünülsün. Seçilmiş kenarların hepsinin iki ucu *S*'dedir; hiçbiri kesimi geçmez. `anahtar[v]`, *v*'den *S*'ye giden en ucuz kenarın ağırlığıdır. Bu değişmez her adımda korunur, çünkü ağaca yeni düğüm girdiğinde yalnız onun komşu-

larının değeri değişebilir ve güncelleme tam bunu yapar. Anahtarı en küçük olan u seçildiğinde $(ata[u], u)$ kesimi geçen en hafif kenardır. Kesim özelliğine göre bu kenar güvenlidir. Her adımda ağaca bir düğüm ve bir güvenli kenar eklenir; $V - 1$ adım sonunda bir MST elde edilir. □

İspattan çıkan bir sonuç da başlangıç düğümünün önemsiz olduğudur. Farklı bir düğümden başlayan Prim, kenarları farklı sırada ekler. Ağırlıklar farklıysa sonunda yine aynı ağacı bulur; eşit ağırlıklar varsa farklı ama aynı ağırlıkta bir ağaç bulabilir.

4.3.6 Prim kodu ve karmaşıklığı

4.3.6.1 Komşuluk matrisli sürüm

Prim'in en kısa uygulaması öncelikli kuyruk kullanmaz. Çizge bir komşuluk matrisiyle verilir: `G[u][v]`, u ile v arasındaki kenarın ağırlığıdır; kenar yoksa 0'dır. Anahtarı en küçük düğüm her adımda dizinin baştan sona taranmasıyla bulunur.

```
void prim(int[][] G, int kaynak) {
  int V = G.length;
  int[] anahtar = new int[V];
  int[] ata = new int[V];
  boolean[] agacta = new boolean[V];
  Arrays.fill(anahtar, Integer.MAX_VALUE);
  anahtar[kaynak] = 0;
  ata[kaynak] = -1;
  for (int s = 0; s < V - 1; s++) {
    int u = enKucukAnahtar(anahtar, agacta);
    agacta[u] = true;
    for (int v = 0; v < V; v++)
      if (G[u][v] != 0 && !agacta[v]
          && G[u][v] < anahtar[v]) {
        anahtar[v] = G[u][v];
        ata[v] = u;
      }
  }
}
```

Kod satır satır şöyle çalışır. `anahtar` dizisinin bütün hücrelerine `Integer.MAX_VALUE` yazılır; bu değer ∞ yerine kullanılır. Kaynağın anahtarı 0, ebeveyni -1'dir; -1 "ebeveyn yok" anlamına gelir. `agacta` dizisi hangi düğümlerin ağaca girdiğini tutar.

Dış döngü $V - 1$ kez döner. Her turda `enKucukAnahtar` ağaçta olmayan düğümler arasında anahtarı en küçük olanı bulur ve bu düğüm ağaca alınır. İç döngü bu düğümün bütün komşularını gezer. Üç koşul birlikte aranır: `G[u][v] != 0` kenarın var olduğunu, `!agacta[v]` komşunun henüz ağaçta olmadığını, `G[u][v] < anahtar[v]` ise yeni kenarın eski en iyiden ucuz olduğunu söyler. Üçü de doğruysa anahtar ve ebeveyn güncellenir.

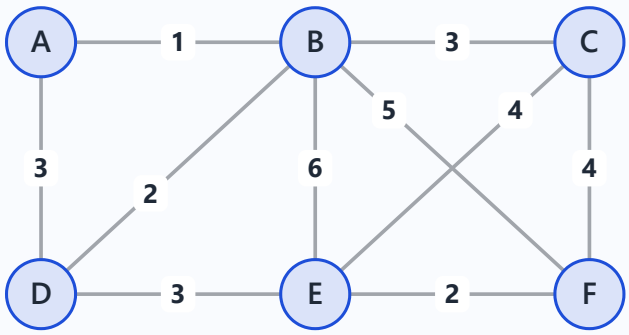
Dış döngünün *V* değil *V − 1* kez dönmesi bilinçli bir seçimdir. *V − 1* düğüm ağaca girdiğinde son düğümün anahtarı ve ebeveyni artık değişemez: onu ağaca bağlayan bütün kenarların öbür ucu ağaçtadır ve hepsi incelenmiştir. Son düğüm `agacta` dizisinde işaretlenmez, ama `ata` değeri doğrudur. Sunumdaki kod `enKucukAnahtar` yardımcısını kullanır. Bu yardımcı şöyle yazılabilir:

```
int enKucukAnahtar(int[] anahtar, boolean[] agacta) {
  int enKucuk = Integer.MAX_VALUE, indis = -1;
  for (int v = 0; v < anahtar.length; v++)
    if (!agacta[v] && anahtar[v] < enKucuk) {
      enKucuk = anahtar[v];
      indis = v;
    }
  return indis;
}
```

Yardımcı, eşitlikte küçük indisli düğümü seçer, çünkü karşılaştırma `<` ile yapılır: sonra gelen eşit anahtar, bulunan en küçüğü değiştirmez. Bağlı olmayan bir çizgede ağaç dışındaki bütün anahtarlar ∞ kalabilir; o zaman metot *−1* döndürür ve `prim` içinde `agacta[-1]` erişimi hata verir. Kod bağlı çizge varsayar.

4.3.6.2 Kod izleme: A–F çizgesi

Kod altı düğümlü bir çizgede izlenir. Düğümler A–F'dir ve indisleri 0–5'tir. Kaynak B'dir (indis 1). Kenarlar *AB = 1*, *BC = 3*, *AD = 3*, *BD = 2*, *BE = 6*, *BF = 5*, *CE = 4*, *CF = 4*, *DE = 3* ve *EF = 2*'dir.



Şekil 4.3.19: A–F çizgesi. Kaynak B'dir; başta bütün anahtarlar ∞, `anahtar[B] = 0` 'dır.

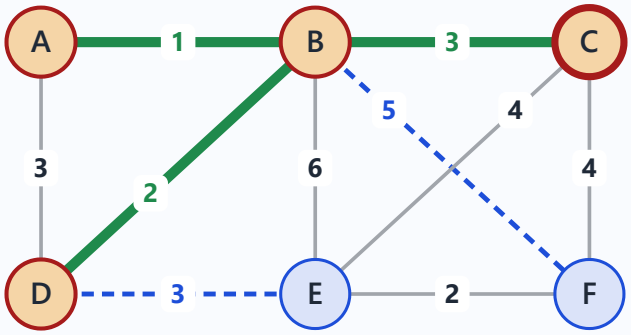
Tur `s`	`u`	Güncellenen komşular	`anahtar`	*w(T)*
—	—	ilkleme	∞, 0, ∞, ∞, ∞, ∞	0
0	B (0)	A → 1, C → 3, D → 2, E → 6, F → 5 (ebeveyn hepsinde B)	1, 0, 3, 2, 6, 5	0
1	A (1)	yok: *AD = 3 > anahtar[D] = 2*	1, 0, 3, 2, 6, 5	1
2	D (2)	E → 3 (ebeveyn D)	1, 0, 3, 2, 3, 5	3

Tur `s`	`u`	Güncellenen komşular	`anahtar`	*w(T)*
3	C (3)	F → 4 (ebeveyn C); *CE = 4 > 3*, E değişmez	1, 0, 3, 2, 3, 4	6
4	E (3)	F → 2 (ebeveyn E)	1, 0, 3, 2, 3, 2	9
—	(F)	döngü biter; F'nin ebeveyni E	1, 0, 3, 2, 3, 2	11

Tablo 4.3.10: `prim(G, 1)` izlemesi. Anahtar dizisi A, B, C, D, E, F sırasıyla verilmiştir; ∞ başlangıç değeridir.

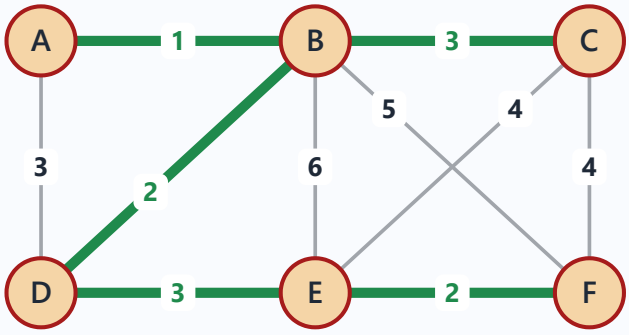
İlk turda `enKucukAnahtar` B'yi döndürür, çünkü yalnız B'nin anahtarı sonludur. B'nin beş komşusu da güncellenir. İkinci turda en küçük anahtar A'nındır (1); *(B, A)* ağaca girer. A'nın ağaç dışı tek komşusu D'dir, ama *AD = 3*, D'nin mevcut anahtarı 2'den büyüktür. Üçüncü turda D (2) girer ve E'nin anahtarı 6'dan 3'e iner: E'ye D üzerinden bağlanmak B üzerinden bağlanmaktan ucuzdur.

Dördüncü turda C ve E'nin anahtarları eşittir (3). `enKucukAnahtar` küçük indisli C'yi seçer. C'den F'ye 4'lük kenar, F'nin mevcut anahtarı 5'ten ucuzdur; F'nin ebeveyni C olur. Beşinci turda E (3) girer ve *EF = 2* ile F'nin anahtarı 2'ye iner. Dış döngü beş tur döner ve biter. F ağaca resmen alınmaz, ama ebeveyni E'dir. MST *(ata[v], v)* kenarlarından okunur: *BA, BC, BD, DE, EF*. Toplam *1 + 3 + 2 + 3 + 2 = 11*'dir.



Şekil 4.3.20: Dördüncü tur: C ve E'nin anahtarları eşittir (3); küçük indisli C ağaca girer. Mavi kesikli kenarlar, henüz ağaçta olmayan düğümlerin mevcut en ucuz bağlantılarıdır.

Eşitlikte E seçilseydi ne olurdu? E girer, F'nin anahtarı 2 olur; sonra C (3) ve F (2) yarışır ve F seçilir. Sonunda C de girer. Seçilen kenarlar yine *BA, BD, DE, EF, BC* olurdu; yalnız sıra değişirdi. Bu çizgede eşit ağırlıklı kenarlar bulunmasına rağmen MST tektir.



Şekil 4.3.21: Prim'in sonucu: AB, BD, BC, DE, EF; toplam 11.

4.3.6.3 Tembel öncelikli kuyruk sürümü

Seyrek çizgelerde dizinin her turda baştan sona taranması gereksiz bir maliyettir. Java'nın `PriorityQueue` sınıfı en küçük öğeyi *O(log n)* sürede çıkarır. Sunumdaki tembel sürüm, sözde koddaki yöntemi Java ile yazar. Çizge komşuluk listesiyle verilir: `komsu.get(u)`, *u*'dan çıkan kenarların listesidir. `anahtar`, `ata`, `agacta` ve `komsu` sınıf alanlarıdır.

```
void primPQ(int kaynak) {
  Arrays.fill(anahtar, SONSUZ);
  anahtar[kaynak] = 0;
  PriorityQueue<int[]> pq =
      new PriorityQueue<>((a, b) -> a[1] - b[1]);
  pq.add(new int[] { kaynak, 0 });
  while (!pq.isEmpty()) {
    int u = pq.poll()[0];
    if (agacta[u]) continue;  // eski
    agacta[u] = true;
    for (Kenar k : komsu.get(u))
      if (!agacta[k.v] && k.w < anahtar[k.v]) {
        anahtar[k.v] = k.w; ata[k.v] = u;
        pq.add(new int[] { k.v, k.w });
      }
  }
}
```

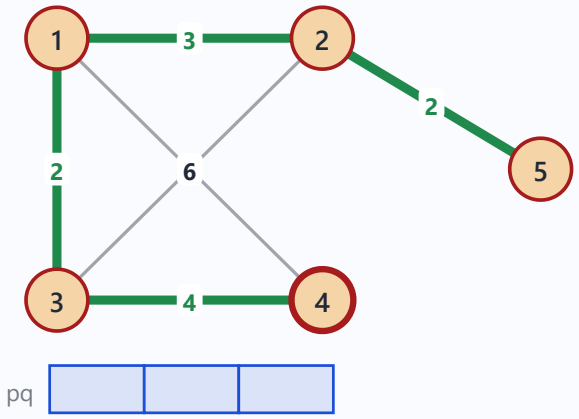
Kuyruktaki her kayıt iki öğeli bir dizidir: *{düğüm, anahtar}*. Karşılaştırıcı `a[1] - b[1]` kayıtları anahtara göre sıralar. Ağırlıklar küçük olduğu sürece çıkarma güvenlidir; çok büyük ağırlıklarda `Integer.compare(a[1], b[1])` yazmak daha doğrudur. Döngü her turda en küçük kaydı çıkarır. Düğüm zaten ağaçtaysa kayıt eskidir ve `continue` ile atlanır. Değilse düğüm ağaca girer ve komşuları incelenir. Anahtarı küçülen her komşu için kuyruğa yeni bir kayıt eklenir. Eski kayıt silinmez; silme işi, kaydın kuyruktan çıkacağı ana ertelenir. Bu ertelemeye **tembel silme** denir.

Kod beş düğümlü çizgede, kaynak 1 ile izlenir. Tablo 4.3.11'deki kuyruk sütunu, kaydı çıkarmadan önceki içeriği anahtara göre sıralı verir.

Kuyruk (önce)	Çıkan	İşlem	`anahtar`
(1,0)	(1,0)	1 ağaca girer; 2 → 3, 3 → 2, 4 → 6	0, 3, 2, 6, ∞
(3,2) (2,3) (4,6)	(3,2)	3 girer (kenar 1–3); 4 → 4; 2 için 4 > 3, değişmez	0, 3, 2, 4, ∞
(2,3) (4,4) (4,6)	(2,3)	2 girer (kenar 1–2); 5 → 2	0, 3, 2, 4, 2
(5,2) (4,4) (4,6)	(5,2)	5 girer (kenar 2–5); güncelleme yok	0, 3, 2, 4, 2
(4,4) (4,6)	(4,4)	4 girer (kenar 3–4); güncelleme yok	0, 3, 2, 4, 2
(4,6)	(4,6)	4 zaten ağaçta: eski kayıt, `continue`	0, 3, 2, 4, 2

Tablo 4.3.11: `primPQ(1)` izlemesi. Kayıtlar *(düğüm, anahtar)* biçimindedir; `anahtar` 1–5 indisleri için verilmiştir.

İkinci satırda 4'ün anahtarı 6'dan 4'e iner ve kuyruğa *(4,4)* kaydı eklenir. Eski *(4,6)* kaydı kuyrukta kalır. 4 ağaca *(4,4)* kaydıyla girer. *(4,6)* en son çıkar ve atlanır. Kuyruk boşalınca MST kenarları *(ata[v], v)* olarak okunur: 1–3 (2), 1–2 (3), 2–5 (2) ve 3–4 (4); toplam 11.



Şekil 4.3.22: Son adım: eski *(4,6)* kaydı çıkarıldı ve atlandı; kuyruk boştur. Ağaç kenarları 1–3, 1–2, 2–5, 3–4'tür.

Tembel sürümün karşılığı, anahtarı yerinde küçülten **istekli** (*eager*) sürümdür. İstekli sürüm her düğüm için kuyrukta tek kayıt tutar ve anahtar küçüldüğünde o kaydı günceller. Bu işleme **anahtar azaltma** (*decrease-key*) denir. Java'nın `PriorityQueue` sınıfı bu işlemi doğrudan sunmaz; bu yüzden Java uygulamalarında tembel sürüm yaygındır. Tembel sürüm kuyrukta V yerine en fazla E kayıt tutar, ama $\log E \le 2 \log V$ olduğu için mertebe değişmez.

4.3.6.4 Dokuz düğümlü örnek

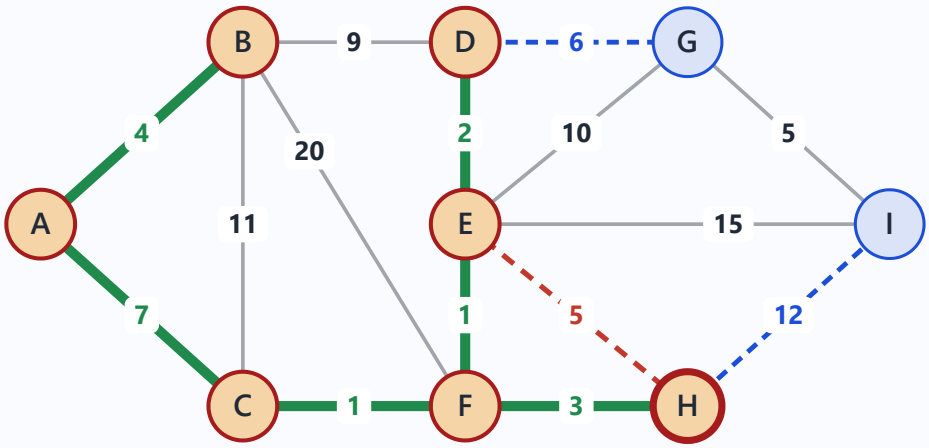
Kruskal'ın dokuz düğümlü örneği Prim ile A'dan başlanarak çözülür. Her satır, o adımda ağaca giren düğümden sonraki anahtar ve ebeveyn değerlerini verir.

Giren	A	B	C	D	E	F	G	H	I	Toplam
A	0/–	4/A	7/A	∞	∞	∞	∞	∞	∞	0
B	0/–	4/A	7/A	9/B	∞	20/B	∞	∞	∞	4

Giren	A	B	C	D	E	F	G	H	I	Toplam
C	0/–	4/A	7/A	9/B	∞	1/C	∞	∞	∞	11
F	0/–	4/A	7/A	9/B	1/F	1/C	∞	3/F	∞	12
E	0/–	4/A	7/A	2/E	1/F	1/C	10/E	3/F	15/E	13
D	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	15/E	15
H	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	12/H	18
G	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	5/G	24
I	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	5/G	29

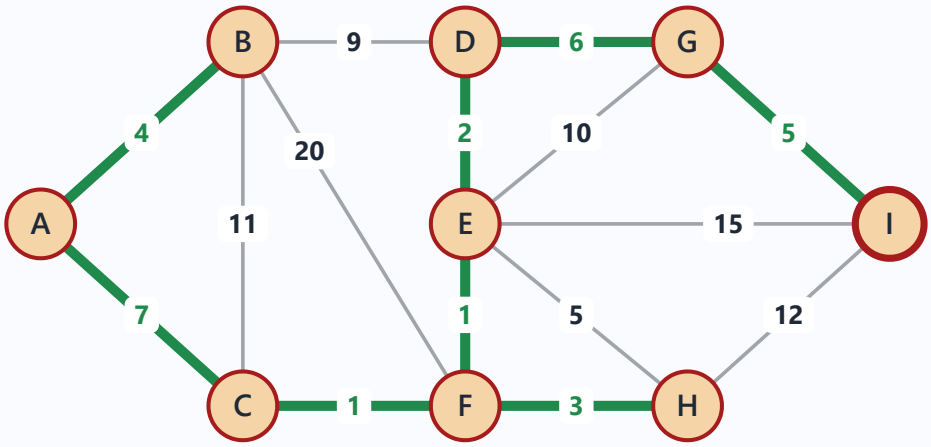
Tablo 4.3.12: Dokuz düğümlü çizgede Prim (A'dan). Her hücrede *anahtar / ebeveyn* yazılıdır; "–" ebeveyn atanmadığını gösterir.

Tablo birkaç şeyi açıkça gösterir. F'nin anahtarı önce B üzerinden 20 olur, C girince 1'e iner. D'nin anahtarı B üzerinden 9'dur, E girince 2'ye iner. I'nın anahtarı üç kez değişir: E üzerinden 15, H üzerinden 12, G üzerinden 5. Bir düğümün anahtarı yalnız küçülür; büyümesi mümkün değildir. H ağaca girdiğinde *EH = 5* kenarı artık kullanılamaz, çünkü E ve H ikisi de ağaçtadır. Kruskal aynı kenarı döngü gerekçesiyle atlamıştı.



Şekil 4.3.23: H ağaca girdikten sonra: toplam 18. Kırmızı kesikli *EH = 5* artık döngü kurar; mavi kesikli kenarlar G ve I için mevcut en ucuz bağlantılardır.

Sonuç ağırlığı 29'dur ve Kruskal'ın bulduğu ağaçla aynıdır. İki algoritmanın kenar ekleme sıraları ise belirgin biçimde farklıdır. Kruskal ağırlıkları *1, 1, 2, 3, 4, 5, 6, 7* sırasıyla ekler. Prim *4, 7, 1, 1, 2, 3, 6, 5* sırasıyla ekler: önce A'nın çevresindeki pahalı kenarları alır, ucuz kenarlara ancak ağaç onlara ulaşınca erişir.



Şekil 4.3.24: I ağaca girer: bütün düğümler ağaçtadır ve MST ağırlığı 29'dur.

4.3.6.5 Karmaşıklık

Prim'in süresi iki işlemin maliyetine bağlıdır: anahtarı en küçük düğümü çıkarmak ve bir komşunun anahtarını azaltmak. Birinci işlem V kez yapılır. İkinci işlem en fazla E kez yapılır, çünkü her kenar iki ucundan biri ağaca girdiğinde bir kez incelenir. Yönsüz çizgede her kenar iki komşuluk listesinde görünür; incelemelerin sayısı $2E$'yi aşmaz.

Uygulama	En küçüğü çıkar	Anahtar azalt	Toplam
Komşuluk matrisi + dizi taraması	$V \times O(V)$	$O(1)$	$O(V^2)$
Komşuluk listesi + ikili heap	$V \times O(\log V)$	$E \times O(\log V)$	$O((V + E) \log V)$
Tembel sürüm: kuyrukta kenar kopyaları	$\le E$ ekleme ve çıkarma, her biri $O(\log E)$		$O(E \log E)$
Fibonacci heap	$O(\log V)$ amortize	$O(1)$ amortize	$O(E + V \log V)$

Tablo 4.3.13: Prim'in farklı uygulamalarının maliyeti.

Matris sürümünde dış döngü $V - 1$ kez döner. Her turda `enKucukAnahtar` V hücreyi tarar ve iç döngü V komşuyu gezer. Toplam $O(V^2)$'dir ve bu süre kenar sayısından bağımsızdır. Heap sürümünde her çıkarma ve her anahtar azaltma $O(\log V)$ sürer; toplam $O((V + E) \log V)$ olur. Bağlı çizgede $E \ge V - 1$ olduğu için bu ifade çoğu zaman $O(E \log V)$ diye kısaltılır.

Hangi sürümün daha iyi olduğu çizgenin yoğunluğuna bağlıdır. **Yoğun** bir çizgede ($E \approx V^2$) matris sürümü $O(V^2)$ ile en iyisidir; heap sürümü $O(V^2 \log V)$'ye çıkar. **Seyrek** bir çizgede ($E \approx V$) ise heap sürümü $O(V \log V)$ sürer; matris sürümü gereksiz yere $O(V^2)$ harcar.

Örnek 4.3.5

$V = 1000$ düğümlü bir çizgede iki sürüm karşılaştırılsın. Matris sürümü yaklaşık $V^2 = 10^6$ adım atar. Heap sürümü için $\log_2 1000 \approx 10$ alınır.

1. $E = 5000$ (seyrek): $E \log V \approx 5000 \cdot 10 = 5 \cdot 10^4$. Heap sürümü yaklaşık 20 kat az iş yapar.
2. $E = 500.000$ (tam çizgeye yakın): $E \log V \approx 5 \cdot 10^6$. Matris sürümü yaklaşık 5 kat az iş yapar.

Sınır kabaca $E \log V \approx V^2$ noktasındadır; burada $E \approx 10^5$'tir. Sabit çarpanlar sınırı biraz kaydırır, ama yön değişmez.

4.3.6.6 Prim ile Dijkstra arasındaki fark

Prim'in kodu, en kısa yol için kullanılan Dijkstra algoritmasının koduna çok benzer. İkisi de bir kaynaktan başlar, öncelikli kuyruktan en küçük anahtarlı düğümü çıkarır ve komşularını günceller. Tek fark anahtarın anlamıdır.

	Prim (MST)	**Dijkstra (en kısa yol)**
`anahtar[v]`	*v*'yi ağaca bağlayan en ucuz tek kenar	kaynaktan *v*'ye toplam yol uzunluğu
Güncelleme	*w(u, v) < anahtar[v]*	*uzaklik[u] + w(u, v) < uzaklik[v]*
Sonuç	toplam ağırlığı en küçük ağaç	en kısa yollar ağacı
Kaynak	sonucu etkilemez (aynı ağırlık)	sonucu belirler
Negatif ağırlık	sorun değil	doğru çalışmaz

Tablo 4.3.14: Prim ile Dijkstra'nın karşılaştırması.

Örnek 4.3.6

A–F çizgesinde B'den Dijkstra çalıştırılsın. Uzaklıklar $A = 1$, $C = 3$, $D = 2$, $E = 5$ (B → D → E) ve $F = 5$'tir. F'ye doğrudan $BF = 5$ kenarıyla gidilir; B → D → E → F yolu $2 + 3 + 2 = 7$ tutar. En kısa yollar ağacı *BA, BC, BD, DE, BF* kenarlarından oluşur ve toplam ağırlığı $1 + 3 + 2 + 3 + 5 = 14$'tür.

Prim ise F'yi E'ye bağlar: F'yi ağaca bağlamanın en ucuz yolu $EF = 2$'dir. Prim'in ağacı 11 ağırlıklıdır. İki ağaç da bütün düğümleri kapsar. Biri toplam ağırlığı, öbürü kaynaktan uzaklıkları küçültür.

Beş düğümlü çizgede de benzer bir fark görülür. 1'den 4'e en kısa yol uzunluğu 6'dır ve bu uzaklık hem doğrudan *(1,4)* kenarıyla hem $1 \to 3 \to 4$ yoluyla elde edilir. Bu yüzden en kısa yollar ağacı $(1,4) = 6$ kenarını kullanabilir. MST onun yerine $(3,4) = 4$ kenarını seçer.

Sık yapılan hata

Prim'in güncelleme koşulunu `anahtar[u] + G[u][v] < anahtar[v]` diye yazmak. Bu satır Prim'i Dijkstra'ya dönüştürür ve sonuç bir en kısa yollar ağacı olur. A–F çizgesinde bu hata F'yi B'ye bağlar ve ağırlığı 11 yerine 14 olan bir ağaç verir. Prim'de anahtar her zaman tek bir kenarın ağırlığıdır; toplam yazılmaz.

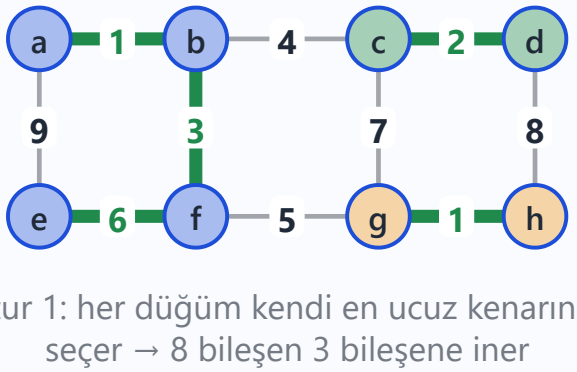
4.3.7 Borůvka algoritması

Minimum kapsayan ağacı bulan en eski algoritma Otakar Borůvka'nın 1926 tarihli çalışmasıdır. Borůvka, Moravya bölgesinin elektrik ağını en ucuz biçimde kurma problemiyle uğraşıyordu. Yöntem sonradan birkaç kez yeniden bulundu; Georges Sollin'in 1960'lardaki çalışması nedeniyle **Sollin algoritması** diye de anılır.

4.3.7.1 Fikir: bütün bileşenler aynı anda

Başlangıçta her düğüm bağımsız bir bileşendir. Algoritma turlar hâlinde ilerler. Her turda her bileşen, kendisini dışarıya bağlayan en ucuz kenarı seçer. Seçilen kenarların hepsi birden eklenir ve bağladıkları bileşenler birleşir. Tek bileşen kalana kadar turlar sürer. Bağlı bir çizgede algoritma $V - 1$ kenar üretir.

Bir bileşenin en ucuz dış kenarı, o bileşen ile geri kalan düğümler arasındaki kesimin en hafif kenarıdır. Kesim özelliğine göre bu kenar güvenlidir. Kruskal ve Prim her adımda tek bir kesim seçer; Borůvka ise her turda bileşen sayısı kadar kesimi aynı anda kullanır.



Şekil 4.3.25: Borůvka'nın ilk turu. Her düğüm kendi en ucuz kenarını seçer: $ab = 1$, $cd = 2$, $bf = 3$, $ef = 6$, $gh = 1$. Beş kenar aynı anda eklenir; sekiz bileşen üç bileşene iner.

Şekildeki çizgede e'nin en ucuz kenarı $ef = 6$'dır, çünkü öbür kenarı $ae = 9$'dur. f ise $bf = 3$'ü seçer. Bir kenar iki ucundan da seçilebilir; ab hem a'nın hem b'nin seçimidir ama bir kez eklenir. İlk tur sonunda bileşenler $\{a, b, e, f\}$, $\{c, d\}$ ve $\{g, h\}$'dir. İkinci turda $bc = 4$ ve $fg = 5$ eklenir ve ağaç tamamlanır; toplam ağırlık 22 olur.

Algoritmanın en önemli üstünlüğü bileşenlerin seçimlerinin birbirinden bağımsız olmasıdır. Her bileşenin en ucuz kenarı ayrı bir işlemcide bulunabilir. Bu yüzden Borůvka paralel ve dağıtık hesaplamaya çok uygundur. Tur sayısı da küçüktür: en fazla $\lceil \log_2 V \rceil$. Zayıf yanları da vardır. Kodlaması Kruskal ve Prim'den daha karmaşıktır. Eşit ağırlıklı kenarlar bulunduğunda tutarlı bir eşitlik bozucu gerekir; yoksa döngü oluşabilir.

4.3.7.2 Adımlar ve sözde kod

1. Başlangıçta her düğüm tek başına bir bileşen (küme) sayılır.
2. Her bileşen için, onu başka bir bileşene bağlayan en küçük ağırlıklı kenar bulunur.
3. Bulunan kenarların hepsi eklenir; bağladıkları bileşenler birleştirilir. Aynı kenar iki kez eklenmez.
4. Tek bileşen kalana kadar 2. ve 3. adımlar tekrarlanır.

```
BORUVKA(G):
  MST = [ ]
  bileşenSayısı = |V|
  döngü bileşenSayısı > 1 iken:
    enUcuz = [ ]  // bileşen başına
    her bir e = (u, v, ağırlık) için E içinde:
      cu = ayrikKume.bul(u)
      cv = ayrikKume.bul(v)
      eğer cu ≠ cv:
        enUcuz[cu] = min(enUcuz[cu], e)
        enUcuz[cv] = min(enUcuz[cv], e)
    her bir (u, v, ağırlık) için enUcuz içinde:
      eğer ayrikKume.bul(u) ≠ ayrikKume.bul(v):
        MST.ekle((u, v, ağırlık))
        ayrikKume.birleştir(u, v)
        bileşenSayısı -= 1
  döndür MST
```

Borůvka kenarları sıralamaz. Her turda bütün kenarları bir kez tarar ve her bileşen için o ana kadar görülen en küçük dış kenarı tutar. Bileşenler ayrık kümeler yapısıyla izlenir: bir kenarın iki ucunun `bul` sonuçları aynıysa kenar bileşenin içindedir ve hesaba katılmaz. İkinci döngüde seçilen kenarlar eklenirken yeniden `bul` sorulur. Bunun nedeni, iki bileşenin aynı kenarı seçmiş olabilmesidir. İlk kez eklendiğinde iki bileşen birleşir; ikinci kez sırası geldiğinde iki ucu artık aynı bileşendedir ve kenar atlanır.

4.3.7.3 Altı düğümlü örnek

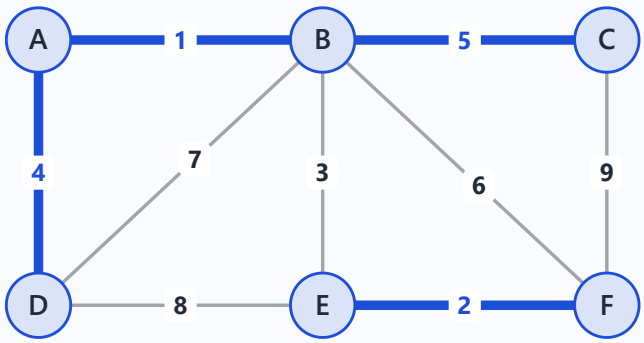
Algoritma A–F düğümlü, dokuz kenarlı bir çizgede çalıştırılır. Kenarlar $AB = 1$, $BC = 5$, $AD = 4$, $BE = 3$, $BD = 7$, $BF = 6$, $CF = 9$, $DE = 8$ ve $EF = 2$'dir.

Bileşen	Dış kenarlar	En ucuz
{A}	AB 1, AD 4	AB 1
{B}	AB 1, BC 5, BE 3, BD 7, BF 6	AB 1
{C}	BC 5, CF 9	BC 5
{D}	AD 4, BD 7, DE 8	AD 4
{E}	BE 3, DE 8, EF 2	EF 2

Bileşen	Dış kenarlar	En ucuz
{F}	BF 6, CF 9, EF 2	EF 2

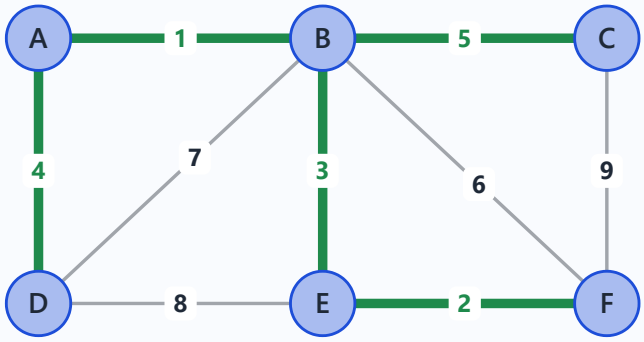
Tablo 4.3.15: Altı düğümlü çizgede Borůvka, tur 1. Her bileşen ve kendi en ucuz dış kenarı.

AB'yi ve EF'yi ikişer bileşen seçer, ama her biri bir kez eklenir. Tur 1'de dört farklı kenar eklenir: AB, BC, AD ve EF. İki bileşen kalır: *{A, B, C, D}* ve *{E, F}*. Toplam ağırlık *1 + 5 + 4 + 2 = 12*'dir.



Şekil 4.3.26: Tur 1: her bileşenin seçtiği en ucuz dış kenar maviyle gösterilmiştir. Dört farklı kenar seçilir.

Tur 2'de iki bileşen arasındaki kenarlar *BE = 3*, *BF = 6*, *CF = 9* ve *DE = 8*'dir. Bileşenlerin ikisi de en ucuz olanı, *BE = 3*'ü seçer. BE eklenir ve tek bileşen kalır. MST *AB, BC, AD, EF, BE* kenarlarından oluşur ve ağırlığı *12 + 3 = 15*'tir.



Şekil 4.3.27: Tur 2 sonu: BE eklenmiştir ve tek bileşen kalmıştır. MST ağırlığı 15'tir.

Kruskal aynı çizgede kenarları *1, 2, 3, 4, 5* sırasıyla tek tek ekler: AB, EF, BE, AD, BC. Borůvka'da ise ilk turda dört kenar aynı anda eklenir. Bulunan ağaç aynıdır.

4.3.7.4 On iki düğümlü örnek

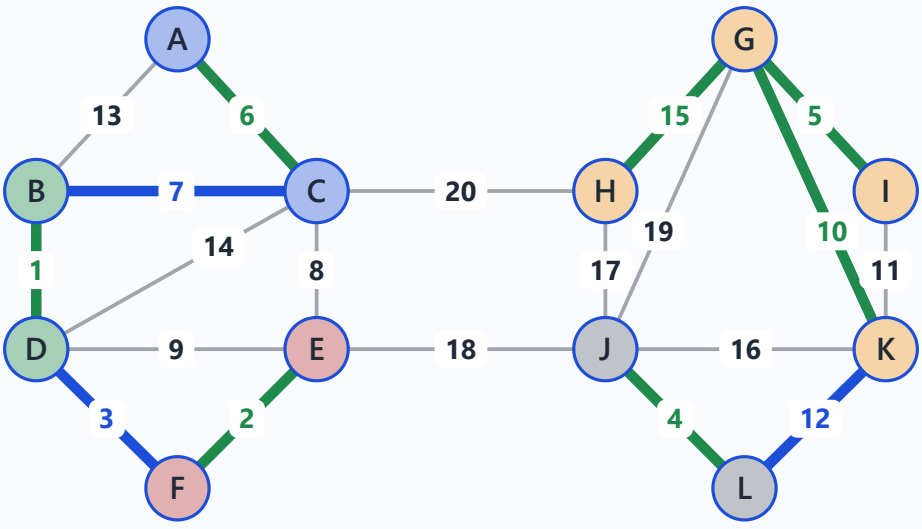
Borůvka'nın gücü büyük çizgelerde daha açık görülür. A–L düğümlü, 20 kenarlı bir çizge ele alınır. Kenarlar şunlardır:

> AB 13, AC 6, BC 7, BD 1, CD 14, CE 8, CH 20, DE 9, DF 3, EF 2, EJ 18, GH 15, GJ 19, GK 10, GI 5, HJ 17, IK 11, JK 16, JL 4, KL 12

Tur	Bileşenlerin en ucuz kenarları	Eklenen farklı kenarlar	Bileşen sayısı	Toplam
1	A: AC 6, B: BD 1, C: AC 6, D: BD 1, E: EF 2, F: EF 2, G: GI 5, H: GH 15, I: GI 5, J: JL 4, K: GK 10, L: JL 4	AC, BD, EF, GI, GH, JL, GK	12 → 5	43
2	{A, C}: BC 7, {B, D}: DF 3, {E, F}: DF 3, {G, H, I, K}: KL 12, {J, L}: KL 12	BC, DF, KL	5 → 2	65
3	{A, …, F}: EJ 18, {G, …, L}: EJ 18	EJ	2 → 1	83

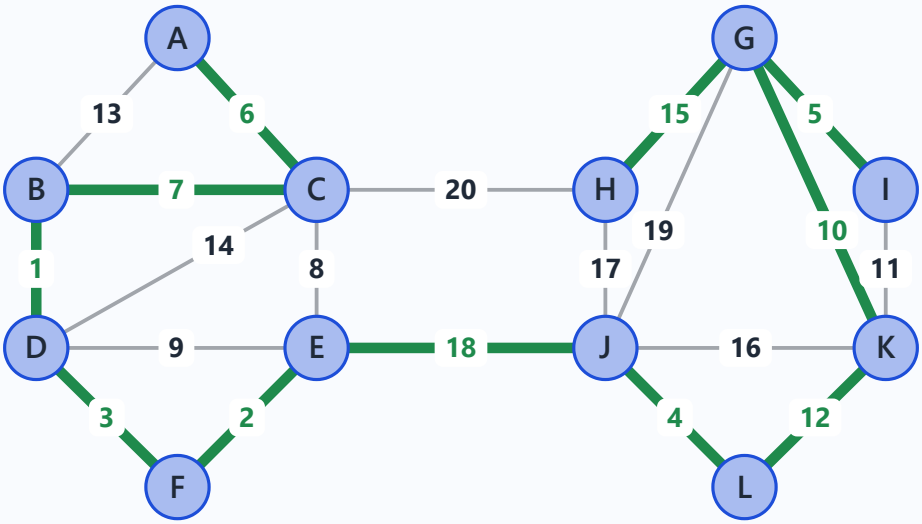
Tablo 4.3.16: On iki düğümlü çizgede Borůvka turları.

İlk turda yedi farklı kenar eklenir ve bileşen sayısı 12'den 5'e iner. H'nin seçimi dikkat çekicidir: H'nin bütün kenarları pahalıdır (15, 17, 20) ve en ucuzu GH 15'tir. Bu kenar ağır olmasına rağmen güvenlidir, çünkü *{H}* kesiminin en hafif kenarıdır. K da GK 10'u seçer; IK 11 ve KL 12 daha pahalıdır. G, H, I ve K böylece tek bileşende toplanır.



Şekil 4.3.28: Tur 2: beş bileşen (renklerle) ve her birinin en ucuz dış kenarı (mavi). Yeşil kalın kenarlar tur 1'de eklenmiştir.

İkinci turda *{A, C}* bileşeni BC 7'yi seçer. *{B, D}* ve *{E, F}* bileşenleri aynı DF 3 kenarını seçer. Sağ taraftaki iki bileşen de KL 12'yi seçer. Üç farklı kenar eklenir ve iki bileşen kalır. Üçüncü turda iki yarıyı bağlayan yalnız iki kenar vardır: CH 20 ve EJ 18. Daha ucuz olan EJ 18'dir ve iki bileşen de onu seçer. MST üç turda tamamlanır ve ağırlığı 83'tür.



Şekil 4.3.29: Tur 3 sonu: EJ eklenmiştir; on iki düğüm tek bileşendedir. MST ağırlığı 83'tür.

Kruskal bu çizgede 11 kenarı 11 ayrı adımda ekler. Borůvka aynı ağacı üç turda kurar: 7 + 3 + 1 kenar. Her tur bütün kenarları bir kez tarar; toplam iş *3 · 20 = 60* kenar incelemesidir.

4.3.7.5 Java kodu ve izleme

Sunumdaki Java kodu, ana döngünün gövdesini verir. `kenarlar` kenar listesi, `ata` ayrık kümelerin ebeveyn dizisi, `agac` seçilen kenarlar, `bilesenSayisi` ise kalan bileşen sayısıdır ve başta *V*'dir.

```
while (bilesenSayisi > 1) {
  Kenar[] enUcuz = new Kenar[V];
  for (Kenar e : kenarlar) {        // 1) en ucuzlar
    int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
    if (k1 == k2) continue;         // aynı bileşen
    if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
      enUcuz[k1] = e;
    if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
      enUcuz[k2] = e;
  }
  for (Kenar e : enUcuz) {          // 2) hepsini ekle
    if (e == null) continue;
    int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
    if (k1 == k2) continue;         // zaten eklendi
    agac.add(e);
    ata[k1] = k2;                   // birleştir
    bilesenSayisi--;
  }
}
```

Her turda `enUcuz` dizisi yeniden oluşturulur. Dizinin indisi bir bileşenin köküdür; hücre o bileşenin şimdiye kadar bulunan en ucuz dış kenarını tutar. Birinci döngü bütün kenarları tarar. Kenarın iki ucunun kökü aynıysa kenar bileşen içindedir ve `continue` ile geçilir. Değilse kenar iki kökün de adayıdır: her kök için hücre boşsa ya da yeni kenar daha ucuzsa hücreye yazılır.

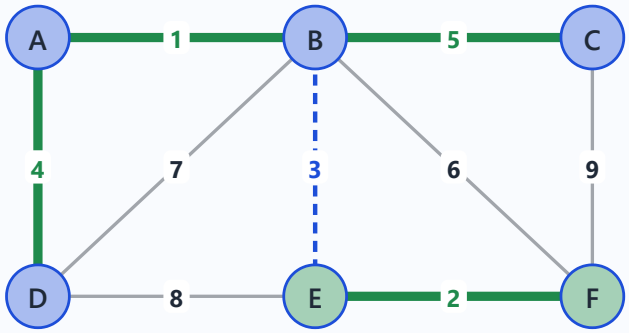
İkinci döngü `enUcuz` dizisini gezer. Boş hücreler kök olmayan düğümlere aittir ve atlanır. Her aday kenar için kökler yeniden bulunur. Kökler aynıysa kenar bu turda zaten eklenmiştir ya da iki bileşen başka bir kenarla birleşmiştir; kenar atlanır. Değilse kenar ağaca girer, `ata[k1] = k2` ile bileşenler birleşir ve bileşen sayısı bir azalır.

Kod, altı düğümlü örnekte izlenir. Düğümler A–F, indisler 0–5'tir. Kenarlar listede şu sırayla durur: AB, BC, AD, BE, BD, BF, CF, DE, EF.

Aşama	**Olay**	`bileseNSayisi`
Tur 1, tarama	AB(1): `enUcuz[A] = enUcuz[B] = AB`. BC(5): `enUcuz[C] = BC`. AD(4): `enUcuz[D] = AD`. BE(3): `enUcuz[E] = BE`. BD(7): değişiklik yok. BF(6): `enUcuz[F] = BF`. CF(9), DE(8): değişiklik yok. EF(2): `enUcuz[E] = enUcuz[F] = EF`.	6
Tur 1, ekleme	`enUcuz[A] = AB`: kökler A, B → ekle, `ata[A] = B`	5
	`enUcuz[B] = AB`: iki uç aynı bileşende → atla	5
	`enUcuz[C] = BC`: kökler B, C → ekle, `ata[B] = C`	4
	`enUcuz[D] = AD`: kökler C, D → ekle, `ata[C] = D`	3
	`enUcuz[E] = EF`: kökler E, F → ekle, `ata[E] = F`	2
	`enUcuz[F] = EF`: aynı bileşen → atla	2
Tur 2, tarama	AB, BC, AD, BD, EF bileşen içinde kalır. Bileşenler arası BE(3), BF(6), CF(9), DE(8) içinden en ucuz BE: `enUcuz[D] = enUcuz[F] = BE`.	2
Tur 2, ekleme	`enUcuz[D] = BE`: kökler D, F → ekle, `ata[D] = F`	1
	`enUcuz[F] = BE`: aynı bileşen → atla; döngü biter	1

Tablo 4.3.17: Borůvka kodunun izlemesi (altı düğümlü örnek).

Tur 1'in taramasında E'nin adayı önce BE(3) olur, sonra EF(2) gelince değişir. F'nin adayı da önce BF(6), sonra EF(2) olur. Ekleme döngüsünde AD eklenirken `bul(A)`, $A \to B \to C$ yolunu izler ve C'yi bulur. Kökler C ve D olduğu için `ata[C] = D` yazılır. Tur 2'de bileşen kökleri D ve F'dir; bu yüzden aday kenar `enUcuz[D]` ve `enUcuz[F]` hücrelerine yazılır. Sonuç ağırlığı 15'tir.



Şekil 4.3.30: Tur 2'nin taraması: iki bileşen (renkler) arasındaki en ucuz kenar BE(3) kesikli maviyle gösterilmiştir.

4.3.7.6 Karmaşıklık ve eşitlik bozucu

Teorem 4.3.8 (Borůvka'nın tur sayısı)

V düğümlü bağlı bir çizgede Borůvka en fazla $\lceil \log_2 V \rceil$ tur çalışır ve toplam süresi $O(E \log V)$'dir.

İspat. Bir turun başında $c > 1$ bileşen olsun. Çizge bağlı olduğu için her bileşenin en az bir dış kenarı vardır ve her bileşen bir kenar seçer. Seçilen kenar bileşeni en az bir başka bileşenle birleştirir. Tur sonunda oluşan her yeni bileşen en az iki eski bileşeni içerir. Bu yüzden yeni bileşen sayısı en fazla $\lfloor c/2 \rfloor$'dir. k turdan sonra en fazla $V/2^k$ bileşen kalır; $k \geq \log_2 V$ olunca bu sayı 1'e iner. Her tur bütün kenarları bir kez tarar ve her kenar için iki `bul` çağrısı yapar. Ayrık kümelerin neredeyse sabit maliyeti ihmal edilirse bir tur $O(E)$ sürer. Toplam $O(E \log V)$'dir. □

İspattaki $\lfloor c/2 \rfloor$ gözlemi aslında biraz daha sıkı bir sınır verir. $V = 1000$ için bileşen sayısı en kötü durumda 1000, 500, 250, 125, 62, 31, 15, 7, 3, 1 diye iner; 9 tur yeter. $\lceil \log_2 1000 \rceil = 10$ ise güvenli bir üst sınırdır. Uygulamada tur sayısı çoğu zaman daha da küçüktür, çünkü bir turda ikiden fazla bileşen birleşebilir. On iki düğümlü örnek 3 turda bitmiştir; $\lceil \log_2 12 \rceil = 4$'tür. Borůvka kenarları sıralamaz; ağırlıkları yalnız karşılaştırır.

Eşit ağırlıklar Borůvka'da özel bir sorun yaratır. Örnek olarak kenarlarının hepsi 1 olan bir *a, b, c* üçgeni alınsın. Eşitlik keyfî çözülürse *a* *ab*'yi, *b* *bc*'yi, *c* de *ca*'yı seçebilir. Üç kenar aynı anda eklenirse bir döngü oluşur. Bunu önlemek için tutarlı bir **eşitlik bozucu** gerekir: ağırlıklar eşitse kenarlar örneğin listedeki indislerine göre karşılaştırılır. Böylece kenarların hepsi kesin bir sıraya girer ve döngü oluşamaz. Sunumdaki kodda `e.w < enUcuz[k1].w` karşılaştırması katı küçüktür; eşitlikte listede önce gelen kenar korunur. Bütün bileşenler aynı listeyi aynı sırayla taradığı için bu seçim tutarlıdır. Kodun ikinci döngüsündeki `bul` denetimi de ayrıca bir güvenlik sağlar: zaten bağlanmış iki ucu olan kenar eklenmez.

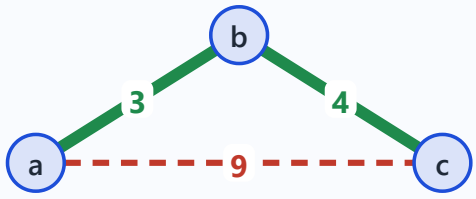
Sık yapılan hata

Bir turda seçilen kenarları, aynı turun içinde bileşenler güncellendikçe yeniden hesaplamak. Borůvka'da bir turun bütün seçimleri tur başındaki bileşenlere göre yapılır; ekleme ondan sonra gelir. Seçim ile ekleme iç içe yazılırsa algoritma Kruskal ile Prim arasında belirsiz bir şeye dönüşür ve tur sayısı sınırı geçerliliğini yitirir.

4.3.8 Ters silme algoritması

4.3.8.1 Fikir: Kruskal'ın aynası

Ters silme algoritması (*reverse-delete*) da açgözlüdür ve Kruskal'ın 1956 makalesinde önerilmiştir. Kruskal boş çizgeden başlayıp küçük kenarları eklerken, ters silme bütün çizgeden başlar ve büyük kenarları silmeye çalışır. Kenarlar ağırlıklarına göre azalan sırada sıralanır. En ağır kenardan başlanarak çizgenin bağlılığını bozmayan her kenar silinir. Silinmesi çizgeyi koparan kenar yerinde bırakılır.



Şekil 4.3.31: Ters silmenin temel adımı: a–b–c döngüsündeki en ağır kenar 9 silinir; a ile c hâlâ b üzerinden bağlıdır.

Algoritma döngü özelliğine dayanır. Bir kenar silindiğinde çizge bağlı kalıyorsa, kenarın iki ucu arasında başka bir yol vardır; kenar bir döngünün üzerindedir. Kenarlar azalan sırada işlendiği için bu döngünün kalan kenarları en fazla onun kadar ağırdır. Daha ağır kenarlar ise zaten ya silinmiş ya da köprü olduğu için kalmıştır. Bu yüzden silinen kenar, kalan çizgedeki bir döngünün en ağır kenarıdır ve döngü özelliğine göre bir MST'ye gerekmez. Silinemeyen kenar ise o anda bir köprüdür ve her kapsayan ağaçta bulunmak zorundadır.

Ters silmenin daha çok kuramsal ve öğretici bir değeri vardır. Her silme için bir bağlılık testi gerekir ve bu test pahalıdır; pratikte algoritma genellikle yavaştır.

4.3.8.2 Adımlar, sözde kod ve karmaşıklık

1. Bütün kenarlar ağırlığına göre azalan şekilde sıralanır.
2. Sıradaki en ağır kenar silinir.
3. Çizge artık bağlı değilse silinen kenar geri eklenir.

4. Bütün kenarlar işlenene kadar 2. ve 3. adımlar tekrarlanır.

```
TERS_SİLME(G):
  MST = tüm kenarlar
  kenarlar.sırala(azalana göre)
  her bir (u, v, ağırlık) için kenarlar içinde:
    MST.sil((u, v, ağırlık))
    ayrikKume.sıfırla()
    her bir (x, y) için MST içinde:
      ayrikKume.birleştir(x, y)
    kök = ayrikKume.bul(V[0])
    eğer bir v için ayrikKume.bul(v) ≠ kök:
      MST.ekle((u, v, ağırlık))  // geri koy
  döndür MST
```

Sözde kod bağlılığı ayrık kümelerle sınar. Her silmeden sonra yapı sıfırlanır ve kalan bütün kenarlar için `birleştir` çağrılır. Sonra ilk düğümün kökü bulunur. Kökü bundan farklı bir düğüm varsa çizge kopmuştur ve silinen kenar bir kez geri konur. Geri konan kenar, döngünün başında silinen *(u, v, ağırlık)* kenarıdır; koşul satırındaki *v* ise yalnız "bir düğüm" anlamında kullanılmıştır ve iki ad birbirine karıştırılmamalıdır.

Karmaşıklık hesabı basittir. Her kenar için bir bağlılık testi yapılır. Test BFS ya da DFS ile $O(V + E)$ sürer; ayrık kümelerle yeniden kurma da aynı mertebededir. E test toplam $O(E \cdot (V + E))$ eder. Sıralamanın $O(E \log E)$ maliyeti bunun yanında küçük kalır. $V = 1000$, $E = 5000$ olan bir çizgede bu, yaklaşık $3 \cdot 10^7$ adım demektir; Kruskal aynı çizgede yaklaşık $6 \cdot 10^4$ adımla yetinir.

4.3.8.3 Yedi düğümlü örnek

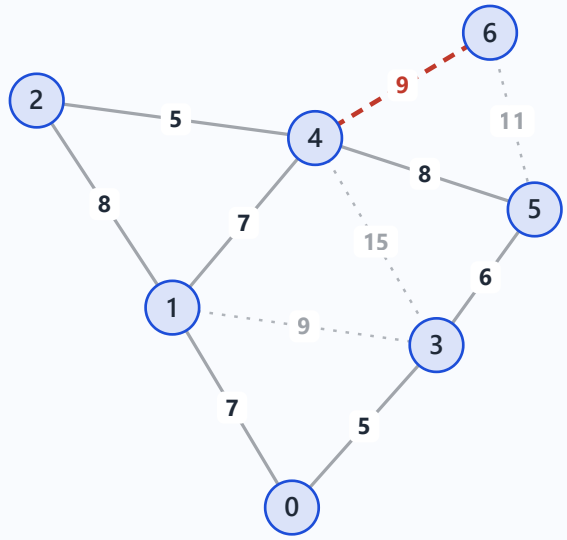
Ters silme, düğümleri 0–6 olan, 11 kenarlı bir çizgede çalıştırılır. Kenarların toplam ağırlığı 90'dır. Azalan sıra şöyledir:

> 3–4 15, 5–6 11, 1–3 9, 4–6 9, 1–2 8, 4–5 8, 1–4 7, 0–1 7, 3–5 6, 2–4 5, 0–3 5

Kenar	*w*	Silinirse çizge bağlı mı?	Sonuç
3–4	15	evet	silindi
5–6	11	evet	silindi
1–3	9	evet	silindi
4–6	9	hayır: 6 yalnız kalır	kaldı
1–2	8	evet	silindi
4–5	8	evet	silindi
1–4, 0–1, 3–5, 2–4, 0–3	7, 7, 6, 5, 5	hayır: her biri köprü	kaldı

Tablo 4.3.18: Yedi düğümlü çizgede ters silme.

İlk üç silme döngü kenarlarını kaldırır. 3–4 = 15 silindiğinde 3 ile 4, *3 → 0 → 1 → 4* yoluyla bağlı kalır. 4–6 = 9 sırası geldiğinde 5–6 zaten silinmiştir; 6'nın tek kenarı 4–6'dır. Bu kenar silinirse 6 yalnız kalır, bu yüzden geri konur. 1–2 = 8 ve 4–5 = 8 silinebilir, çünkü 2'ye 2–4 ile, 5'e 3–5 ile ulaşılır. Bu noktada kalan altı kenar bir ağaç oluşturur ve her biri köprüdür; hiçbiri silinemez. Kalan kenar sayısı *6 = V − 1*'dir ve MST ağırlığı *9 + 7 + 7 + 6 + 5 + 5 = 39*'dur.



Şekil 4.3.32: 4–6 = 9 adımı. Gri noktalı kenarlar silinmiştir. 5–6 silindiği için 6'nın tek bağlantısı 4–6'dır (kırmızı kesikli); kenar silinirse 6 kopar, bu yüzden geri konur.

Kruskal aynı çizgede önce 5, 5, 6, 7, 7 ağırlıklı kenarları ekler. Ardından 8'lik iki kenarı ve 9'luk 1–3'ü döngü gerekçesiyle atlar. Son olarak 4–6 = 9'u ekler. İki algoritma aynı ağaca ters yönlerden ulaşır: Kruskal neyin ekleneceğine, ters silme neyin kalacağına karar verir.

4.3.8.4 Java kodu ve izleme

```
void tersSil(int V, List<Kenar> ke) {
  ke.sort((a, b) -> b.w - a.w); // azalan
  int i = 0;
  while (i < ke.size()) {
    Kenar e = ke.remove(i);     // geçici sil
    if (!bagliMi(V, ke)) {
      ke.add(i, e);             // geri koy
      i++;
    }
  }
}
```

`ke.sort((a, b) -> b.w - a.w)` kenarları azalan sıraya koyar; karşılaştırıcının `b - a` biçiminde yazılması sırayı tersine çevirir. `i`, listede incelenecek sıradaki kenarın indisidir. Döngünün her turunda kenar `ke.remove(i)` ile geçici olarak silinir. Silmeden sonra liste bir kayar ve sıradaki kenar yine `i` indisine gelir; bu yüzden silme başarılıysa `i` artırılmaz. Kenar silinince çizge kopuyorsa `ke.add(i, e)` kenarı eski yerine koyar ve `i` bir artar. Döngü `i` liste boyuna ulaşınca biter. Bu anda listede kalan kenarlar MST'dir. Kodun kullandığı `bagliMi` metodu BFS ile yazılabilir:

```
boolean bagliMi(int V, List<Kenar> ke) {
  List<List<Integer>> komsu = new ArrayList<>();
  for (int i = 0; i < V; i++) komsu.add(new ArrayList<>());
  for (Kenar e : ke) { komsu.get(e.u).add(e.v); komsu.get(e.v).add(e.u); }
  boolean[] goruldu = new boolean[V];
  ArrayDeque<Integer> kuyruk = new ArrayDeque<>();
  kuyruk.add(0); goruldu[0] = true;
  int sayac = 1;
  while (!kuyruk.isEmpty()) {
    int x = kuyruk.poll();
    for (int y : komsu.get(x))
      if (!goruldu[y]) { goruldu[y] = true; sayac++; kuyruk.add(y); }
  }
  return sayac == V;            // her düğüme ulaşıldı mı?
}
```

Metot önce kenar listesinden komşuluk listesini kurar, sonra 0. düğümden BFS başlatır ve ulaşılan düğümleri sayar. Sayı *V*'ye eşitse çizge bağlıdır. Her çağrı *O(V + E)* sürer; çağrı sayısı *E* olduğu için toplam maliyet yukarıda hesaplanan *O(E · (V + E))*'dir.

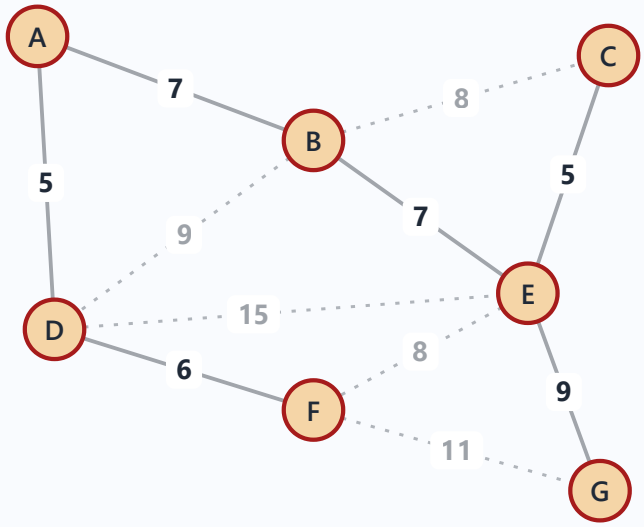
Kod, aynı yedi düğümlü çizgenin harfli sürümünde izlenir: 0–6 düğümleri A–G olarak adlandırılır. Kenar listesi alfabetik sırayla verilir: AB 7, AD 5, BC 8, BD 9, BE 7, CE 5, DE 15, DF 6, EF 8, EG 9, FG 11. Sıralama kararlıdır; eşit ağırlıklı kenarlar bu sırayı korur.

Kenar	`bagliMi`	İşlem	`i` (sonra)
DE 15	true	silinmiş kalır	0
FG 11	true	silinmiş kalır	0
BD 9	true	silinmiş kalır	0
EG 9	false: G yalnız kalır	geri konur	1
BC 8	true	silinmiş kalır	1
EF 8	true	silinmiş kalır	1
AB 7	false: {A, D, F} kopar	geri konur	2
BE 7	false: {C, E, G} kopar	geri konur	3
DF 6	false: {F} kopar	geri konur	4
AD 5	false: {D, F} kopar	geri konur	5

Kenar	`bagliMi`	İşlem	`i` (sonra)
CE 5	false: {C} kopar	geri konur	6

Tablo 4.3.19: `tersSil(7, ke)` izlemesi.

EG adımında FG zaten silinmiştir ve G'nin tek bağlantısı EG'dir. AB adımına gelindiğinde A'nın kenarları AB ve AD, D'nin kalan kenarları AD ve DF, F'nin kalan kenarı ise yalnız DF'dir. AB silinirse A, D ve F öbür düğümlerden kopar. Son satırdan sonra `i = 6` liste boyuna eşittir ve döngü biter. Kalan ağaç AB, AD, BE, CE, DF ve EG kenarlarından oluşur; ağırlığı 39'dur.



Şekil 4.3.33: Ters silmenin sonucu: AB, AD, BE, CE, DF, EG; toplam 39. Gri noktalı kenarlar silinmiştir.

Sık yapılan hata

Silme başarılı olduğunda da `i` 'yi artırmak. `ke.remove(i)` listeyi bir sola kaydırır; sıradaki kenar artık `i` indisindedir. `i` artırılırsa her başarılı silmeden sonra bir kenar hiç incelenmeden atlanır ve sonuç bir ağaç değil, gereksiz kenarlar içeren bir alt çizge olabilir.

4.3.9 Dört algoritmanın karşılaştırması

Dört algoritma aynı problemi çözer ve aynı kuramsal temele dayanır, ama çalışma biçimleri ve maliyetleri farklıdır. Tablo 4.3.20 farkları bir arada gösterir.

	Kruskal	Prim	Borůvka	Ters silme
Bakış	kenar odaklı	düğüm odaklı	bileşen odaklı	kenar odaklı (silme)
Ara durum	orman	tek ağaç	orman	bağlı çizge

	Kruskal	Prim	Borůvka	Ters silme
Dayandığı özellik	kesim	kesim	kesim	döngü
Veri yapısı	sıralı kenarlar + ayrık kümeler	öncelikli kuyruk / dizi	ayrık kümeler	BFS/DFS
Zaman	$O(E \log E)$	$O(V^2)$ ya da $O((V + E) \log V)$	$O(E \log V)$	$O(E \cdot (V + E))$
En uygun	seyrek çizge	yoğun çizge	paralel / dağıtık	öğretim, kuram
Bağlı olmayan çizge	orman verir	bir bileşen	orman verir	bileşenleri korur

Tablo 4.3.20: Dört MST algoritmasının karşılaştırması.

Tablonun satırları birkaç pratik soruya karşılık gelir. Çizge bir kenar listesi olarak geliyor ve seyrekse Kruskal en kolay ve en hızlı seçimdir. Çizge bir komşuluk matrisiyle veriliyor ve yoğunsa, dizi taramalı Prim $O(V^2)$ ile en iyi sonucu verir; sıralama ya da heap gerekmez. Seyrek çizge komşuluk listesiyle verildiğinde heap'li Prim de Kruskal ile aynı mertebededir. İş birden çok işlemciye dağıtılacaksa Borůvka'nın bağımsız seçimleri büyük kolaylık sağlar. Ters silme ise üretim kodunda pek kullanılmaz; döngü özelliğinin algoritmik bir karşılığı olarak öğreticidir.

Bağlı olmayan çizgelerde davranış farkı da önemlidir. Kruskal ve Borůvka her bileşeni ayrı ayrı tamamlar ve bir minimum kapsayan orman döndürür. Prim yalnız başlangıç düğümünün bileşenini kapsar; ormanı bulmak için kapsanmayan her bileşenden yeniden başlatılmalıdır. Ters silme hiçbir bileşeni koparmadığı için bileşenleri olduğu gibi korur; bağlılık testinin bileşen sayısını koruyacak biçimde yazılması yeter.

Dokuz düğümlü çizge farkı somut olarak gösterir. Kruskal ve Prim aynı ağacı (29) bulur. Kruskal kenarları *1, 1, 2, 3, 4, 5, 6, 7* sırasıyla, Prim ise *4, 7, 1, 1, 2, 3, 6, 5* sırasıyla ekler. Kruskal'ın ekleme sırası ağırlık sırasıdır; Prim'in sırası ise ağacın büyüme yönünü izler.

Bu dört algoritma tarihin sonu değildir. Karger, Klein ve Tarjan 1995'te beklenen süresi doğrusal olan rastgele bir algoritma verdi. Bernard Chazelle 2000'de ters Ackermann fonksiyonuyla neredeyse doğrusal süreli belirlenimci bir algoritma yayımladı. Belirlenimci, kesin doğrusal süreli bir MST algoritmasının var olup olmadığı hâlâ açık bir sorudur.

4.3.10 Sık yapılan hatalar

Bu kesim, notun içine dağılmış uyarıları ve sunumdaki hata listesini bir arada toplar.

Sık yapılan hata

MST'yi en kısa yollar ağacı sanmak. MST toplam ağırlığı küçültür; iki düğüm arasındaki MST yolu en kısa yol olmak zorunda değildir. A–F çizgesinde MST'deki B–F yolu $B \to D \to E \to$

F'dir ve uzunluğu 7'dir; doğrudan *BF* kenarı 5'tir.

Sık yapılan hata

Döngüyü komşulukla denetlemek. Kruskal'da "*u* ile *v* doğrudan bağlı mı?" sorusu yetmez. Sorulması gereken, iki ucun aynı bileşende olup olmadığıdır: `bul(u) == bul(v)`.

Sık yapılan hata

Prim'de uzaklık toplamak. `anahtar[u] + w` yazmak Dijkstra'dır. Prim'de anahtar yalnız tek bir kenarın ağırlığıdır.

Sık yapılan hata

Negatif ağırlıktan korkmak. MST algoritmaları negatif ağırlıklarla da doğru çalışır, çünkü yalnız ağırlıkları karşılaştırır. Kesim ve döngü özelliklerinin ispatı da ağırlıkların işaretine bakmaz. Negatif ağırlık sorunu Dijkstra'ya özgüdür. Hatta bütün ağırlıklara aynı sabit eklenirse MST değişmez. Kenarların sırası korunur. Her kapsayan ağaçta $V - 1$ kenar olduğu için bütün ağaçların toplamı aynı miktarda artar.

Sık yapılan hata

Bağlı olmayan çizgeyi unutmak. Bağlı olmayan çizgenin MST'si yoktur; minimum kapsayan ormanı vardır. `while (agac.size() < V - 1)` koşulu kenarlar bittiğinde liste sınırını aşar. Koşula `k < ke.size()` eklenmelidir.

Sık yapılan hata

Yönlü çizgeye uygulamak. Bu algoritmalar yönsüz çizgeler içindir. Yönlü çizgede bir kökten bütün düğümlere ulaşan en ucuz yapıya **minimum kapsayan dallanma** denir ve bu problem Chu–Liu/Edmonds algoritmasıyla çözülür. Kruskal ya da Prim yönlü kenarlara olduğu gibi uygulanırsa yanlış sonuç verir.

4.3.11 Özet

- **Kapsayan ağaç ve MST:** bağlı, yönsüz bir çizgenin bütün düğümlerini içeren, bağlı ve döngüsüz alt çizgesi kapsayan ağaçtır ve tam $V - 1$ kenarlıdır. Toplam ağırlığı en küçük olana minimum kapsayan ağaç denir. Tam çizgede n^{n-2} kapsayan ağaç vardır; bu yüzden hepsini denemek yerine aç-

gözlü algoritmalar kullanılır.

- **Kesim özelliği:** her kesimin en hafif kenarı bir MST'ye aittir; kesimin en hafif kenarı tekse her MST'dedir. Kruskal, Prim ve Borůvka bu özelliğe dayanır.
- **Döngü özelliği:** bir döngünün tek en ağır kenarı hiçbir MST'de yer almaz. Ters silme bu özelliğe dayanır. Bütün ağırlıklar farklıysa MST tektir.
- **Kruskal:** kenarlar artan sırada incelenir; `bul(u) ≠ bul(v)` ise kenar eklenir ve iki ağaç birleşir. Süre $O(E \log E) = O(E \log V)$'dir ve sıralama baskındır.
- **Ayrık kümeler:** `ata[]` ormanında her küme bir ağaçtır, kök temsilcidir. Yol kısaltma ve rütbeye göre birleştirme birlikte işlem başına amortize $O(\alpha(V))$ verir; α pratikte 4'ü geçmez. Yalnız rütbe ile her işlem $O(\log V)$'dir.
- **Prim:** tek bir ağaç büyütülür; `anahtar[v]`, v'yi ağaca bağlayan en ucuz tek kenarın ağırlığıdır. Dizi taramasıyla $O(V^2)$, ikili heap ile $O((V + E) \log V)$, tembel sürümle $O(E \log E)$, Fibonacci heap ile $O(E + V \log V)$ sürer.
- **Prim ve Dijkstra:** kodlar neredeyse aynıdır; Prim tek kenar ağırlığını, Dijkstra kaynaktan toplam uzaklığı anahtar yapar. Prim'in sonucu kaynaktan bağımsız ağırlıktadır ve negatif ağırlıkla da doğrudur.
- **Borůvka:** her turda her bileşen en ucuz dış kenarını seçer ve bu kenarlar aynı anda eklenir. Bileşen sayısı her turda en az yarıya iner; en fazla $\lceil \log_2 V \rceil$ tur, toplam $O(E \log V)$. Eşit ağırlıklarda tutarlı bir eşitlik bozucu gerekir.
- **Ters silme:** kenarlar azalan sırada silinir; çizge koparsa kenar geri konur. Her silmede bağlılık testi yapıldığı için süre $O(E \cdot (V + E))$'dir.
- **Seçim:** seyrek çizge ve kenar listesi → Kruskal; yoğun çizge ve komşuluk matrisi → dizi taramalı Prim; paralel hesaplama → Borůvka; bağlı olmayan çizgede orman → Kruskal ya da Borůvka.

4.3.12 Kavram listesi

kapsayan ağaç *spanning tree* · minimum kapsayan ağaç *minimum spanning tree (MST)* · minimum kapsayan orman *minimum spanning forest* · bileşen *connected component* · kesim *cut* · kesim özelliği *cut property* · döngü özelliği *cycle property* · köprü *bridge* · açgözlü algoritma *greedy algorithm* · değiştirme argümanı *exchange argument* · Kruskal algoritması *Kruskal's algorithm* · ayrık kümeler *union-find, disjoint sets* · yol kısaltma *path compression* · rütbeye göre birleştirme *union by rank* · amortize analiz *amortized analysis* · Prim algoritması *Prim's algorithm* · öncelikli kuyruk *priority queue* · ebeveyn *parent* · komşuluk matrisi *adjacency matrix* · komşuluk listesi *adjacency list* · Dijkstra algoritması *Dijkstra's algorithm* · Borůvka algoritması *Borůvka's algorithm* · ters silme algoritması *reverse-delete algorithm* · darboğaz *bottleneck*

4.3.13 Kendini Dene

Sorular 1–6, aynı *K* çizgesini kullanır. *K*'nin düğümleri A–F, kenarları *AB = 7*, *AC = 3*, *AD = 6*, *BC = 4*, *BE = 2*, *CD = 5*, *CE = 8*, *CF = 9*, *DF = 1* ve *EF = 10*'dur.

1. *K*'de *{A, C} | {B, D, E, F}* kesimini geçen kenarlar hangileridir? Kesim özelliği hangi kenar için ne söyler?

   Cevap

   Kesimi geçen kenarlar AB 7, AD 6, BC 4, CD 5, CE 8 ve CF 9'dur. AC 3 kesimi geçmez, çünkü iki ucu da aynı taraftadır. En hafif geçen kenar BC 4'tür ve tek en hafif olduğu için her MST'de bulunur.

2. Kruskal *K*'de hangi kenarları hangi sırayla ekler? Kaç kenar reddedilir, hangi kenarlar hiç incelenmez? *w(T)* kaçtır?

   Cevap

   Sıralı liste DF 1, BE 2, AC 3, BC 4, CD 5, AD 6, AB 7, CE 8, CF 9, EF 10'dur. İlk beş kenarın hepsi iki ayrı ağacı bağlar ve eklenir; hiçbir kenar reddedilmez. Beşinci ekleme *V − 1 = 5*'e ulaştığı için AD, AB, CE, CF ve EF hiç incelenmez. *w(T) = 1 + 2 + 3 + 4 + 5 = 15*.

3. Prim *K*'de A'dan çalıştırılırsa düğümler hangi sırayla ağaca girer? C ağaca girdikten sonra anahtar dizisi nedir?

   Cevap

   Sıra A (0), C (3), B (4), E (2), D (5), F (1)'dir. A'dan sonra anahtarlar B = 7, C = 3, D = 6, E = F = ∞ olur. C girince B, BC ile 4'e; D, CD ile 5'e iner; E = 8 ve F = 9 olur. Dizi A, …, F için *0, 4, 3, 5, 8, 9*'dur. Ebeveynler B → C, D → C, E → B, F → D olarak sonuçlanır; toplam yine 15'tir.

4. Borůvka *K*'de kaç turda biter? Her turda hangi kenarlar eklenir?

   Cevap

   Tur 1'de A ve C aynı AC 3'ü, B ve E aynı BE 2'yi, D ve F aynı DF 1'i seçer. Üç farklı kenar eklenir ve *{A, C}*, *{B, E}*, *{D, F}* bileşenleri oluşur. Tur 2'de *{A, C}* ve *{B, E}* BC 4'ü, *{D, F}* CD 5'i seçer. İki kenar eklenir ve tek bileşen kalır: 2 tur, ağırlık 15.

5. Ters silme *K*'de hangi kenarları siler? İlk "kalır" kararı hangi kenarda verilir ve neden?

Cevap

Silinenler EF 10, CF 9, CE 8, AB 7 ve AD 6'dır. İlk "kalır" kararı CD 5'te verilir: AD ve CF silindikten sonra D ile F'yi geri kalan düğümlere bağlayan tek kenar CD'dir. Ardından BC, AC, BE ve DF de köprü oldukları için kalır.

6. *K*'de A'dan Dijkstra çalıştırılsa en kısa yollar ağacı hangi kenarlardan oluşur ve ağırlığı kaçtır? Bu ağacın Prim'in A'dan bulduğu ağaçtan farkı nedir?

Cevap

Uzaklıklar B = 7, C = 3, D = 6, E = 9 (A → B → E), F = 7 (A → D → F)'dir. Ağaç AB, AC, AD, BE, DF kenarlarından oluşur ve ağırlığı *7 + 3 + 6 + 2 + 1 = 19*'dur. Prim'in ağacı (AC, BC, CD, BE, DF) 15 ağırlıklıdır. Dijkstra B'ye doğrudan AB 7 ile gider; A → C → B yolu da 7'dir ve daha kısa değildir. Prim ise B'yi C'ye 4'lük kenarla bağlar.

7. 0–6 öğeli bir ayrık kümeler yapısında sunumdaki yol kısaltmalı `bul` ve `birlestir` kullanılıyor. Sırasıyla `birlestir(0, 1)`, `(2, 3)`, `(0, 2)`, `(4, 5)`, `(6, 4)`, `(5, 0)` çağrılıyor. Son `ata` dizisi nedir? Ardından `bul(6)` neyi değiştirir?

Cevap

Adımlar: `ata[0] = 1`; `ata[2] = 3`; `bul(0) = 1`, `bul(2) = 3` → `ata[1] = 3`; `ata[4] = 5`; `bul(6) = 6`, `bul(4) = 5` → `ata[6] = 5`. Son çağrıda `bul(5) = 5`; `bul(0)`, 0 → 1 → 3 yolunu izler ve 0'ı doğrudan 3'e bağlar. Ardından `ata[5] = 3` yazılır. Dizi `[3, 3, 3, 3, 5, 3, 5]` olur. `bul(6)`, 6 → 5 → 3 yolunu izler ve `ata[6] = 3` yapar: `[3, 3, 3, 3, 5, 3, 3]`. 4 bu yolda olmadığı için 5'e bağlı kalır.

8. *K*'nin *en büyük* kapsayan ağacı nasıl bulunur ve ağırlığı kaçtır?

Cevap

Kruskal kenarları azalan sırayla işler (ya da bütün ağırlıklar $-w$ yapılıp olağan Kruskal çalıştırılır). Sıra EF 10, CF 9, CE 8, AB 7, AD 6, CD 5'tir. EF, CF eklenir; CE döngü kurar (C–F–E); AB, AD eklenir; son olarak CD eklenir. Ağaç EF, CF, AB, AD, CD'dir ve ağırlığı *10 + 9 + 7 + 6 + 5 = 37*'dir.

9. 50 düğümlü, 3 bileşenli bir çizgede minimum kapsayan ormanda kaç kenar vardır? Sunumdaki

Kruskal kodu bu çizgede ne yapar?

Cevap

Her bileşen için *(bileşen boyu − 1)* kenar gerekir; toplam *50 − 3 = 47* kenardır. Sunumdaki kod 49 kenar bekler. 47 kenar ekledikten sonra liste biter ve `ke.get(k++)` `IndexOutOfBoundsException` fırlatır.

10. $V = 10^4$ düğümlü bir çizgede Prim'in iki sürümü karşılaştırılıyor: dizi taramalı $O(V^2)$ ve ikili heap'li $O((V + E) \log V)$. $E = 5 \cdot 10^4$ ve $E = 5 \cdot 10^7$ için hangisi daha az iş yapar?

Cevap

$V^2 = 10^8$, $\log_2 10^4 \approx 13{,}3$'tür. $E = 5 \cdot 10^4$ için heap sürümü yaklaşık $6 \cdot 10^4 \cdot 13{,}3 \approx 8 \cdot 10^5$ adım atar; yüz kattan fazla daha iyidir. $E = 5 \cdot 10^7$ (tam çizgenin aşağı yukarı tamamı) için heap sürümü yaklaşık $6{,}6 \cdot 10^8$ adım atar; dizi sürümü 6–7 kat daha iyidir.

4.3.14 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 19 ve 21: ayrık kümelerin ters Ackermann analizini ve genel MST yöntemini, güvenli kenar kavramıyla birlikte ispatlarıyla verir.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.5 ve 4.3: ayrık kümeleri, tembel ve istekli Prim'i ve Kruskal'ı Java koduyla ve deneysel ölçümlerle anlatır.
- R. L. Graham ve P. Hell, "On the History of the Minimum Spanning Tree Problem", *Annals of the History of Computing* 7(1), 1985: Borůvka, Jarník, Kruskal ve Prim'in çalışmalarını ve problemin erken tarihini karşılaştırmalı olarak inceler.
- R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm", *Journal of the ACM* 22(2), 1975: yol kısaltma ve rütbeye göre birleştirmenin ters Ackermann sınırını ilk kez kanıtlayan makale.

4.3.15 Kaynaklar

1. O. Borůvka, "O jistém problému minimálním", *Práce Moravské přírodovědecké společnosti* 3, 1926.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 19: ayrık kümeler, Bölüm 21: MST).
3. R. L. Graham, P. Hell, "On the History of the Minimum Spanning Tree Problem", *Annals of the History of Computing* 7(1), 43–57, 1985.
4. J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proceedings of the American Mathematical Society* 7(1), 1956.

5. R. C. Prim, “Shortest Connection Networks and Some Generalizations”, *Bell System Technical Journal* 36(6), 1957.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 4.3: Minimum Spanning Trees).
7. R. E. Tarjan, “Efficiency of a Good But Not Linear Set Union Algorithm”, *Journal of the ACM* 22(2), 1975.

KISIM II · BÖLÜM 4.4

Ağ Akışı

Bir şehrin su şebekesi, bir ülkenin karayolu ağı ya da bir veri merkezinin bağlantıları aynı soruyu sorar. Bir noktadan ötekine, hatların taşıma sınırlarını aşmadan en fazla ne kadar şey gönderilebilir? Bu soru çizgeler üzerinde **maksimum akış problemi** olarak yazılır. Not, önce akış ağını, akışın kısıtlarını ve akışın değerini tanımlar. Ardından bütün algoritmaların ortak aracı olan artık ağ, artış yolu ve kesim kavramlarını kurar. Maksimum akış minimum kesim teoremi, bulunan bir akışın neden en iyi olduğunu kanıtlar. Ford-Fulkerson yöntemi, Edmonds-Karp, Dinic ve itme–yeniden etiketleme algoritmaları aynı örnek ağlar üzerinde adım adım çalıştırılır. Her birinin kodu satır satır izlenir ve süre sınırının nereden geldiği gösterilir. Not, algoritmaların karşılaştırılması ve iki parçalı eşleştirme gibi uygulamalarla biter.

ÖĞRENME ÇIKTILARI

Modelleme
Bir problemi kaynak, hedef ve kapasiteli kenarlardan oluşan bir akış ağı olarak kurmak; akışın kapasite ve korunum kısıtlarını yazmak.

Artık ağ ve kesim
Artık ağı, artış yolunu ve darboğazı hesaplamak; s–t kesiminin kapasitesini bulmak ve maksimum akış minimum kesim teoremini uygulamak.

Algoritmalar
Ford-Fulkerson, Edmonds-Karp, Dinic ve itme–yeniden etiketleme algoritmalarını bir örnek ağ üzerinde adım adım çalıştırmak.

Analiz
$O(E \cdot |f^*|)$, $O(V \cdot E^2)$ ve $O(V^2 \cdot E)$ sınırlarının nereden geldiğini açıklamak; algoritmaları karşılaştırıp uygun olanı seçmek.

4.4.1 Akış ağları

Çizgelerde şimdiye kadar sorulan sorular çoğunlukla yollarla ilgiliydi: iki düğüm arasında yol var mı, en kısa yol hangisi, bütün düğümleri en ucuza bağlayan ağaç hangisi? Ağ akışı başka bir şey sorar. Bir yol bulmak yetmez; aynı anda pek çok yol kullanılır ve bu yollar ortak kenarları paylaşır. Her kenarın bir taşıma sınırı vardır. Soru, bütün yolların birlikte en fazla ne kadar taşıyabileceğidir.

Bu soru gündelik hayatta sık karşımıza çıkar. Boru hattında su, yol ağında trafik, bilgisayar ağında veri paketleri birer akıştır. Her boru en fazla kendi kapasitesi kadar su taşır. Kavşaklarda su birikmez: bir kavşağa ne kadar su girerse o kadarı çıkar. Bu iki gözlem, akış ağı modelinin iki temel kısıtına dönüşür.

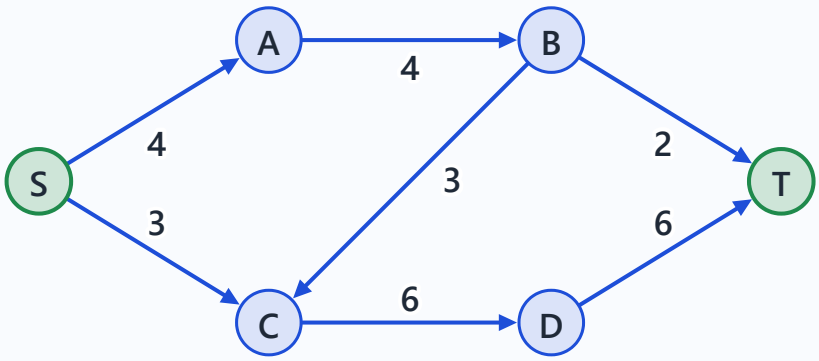
4.4.1.1 Maksimum akış problemi

Model yönlü bir çizgedir. Çizgede iki özel düğüm bulunur. Akışın çıktığı düğüme **kaynak** (*source*), vardığı düğüme **hedef** (*sink*) denir. Kaynak s, hedef t ile gösterilir. Her kenarın bir **kapasitesi** (*capacity*) vardır: o kenardan birim zamanda geçebilecek en fazla miktar.

Tanım 4.4.1

Akış ağı (*flow network*), $G = (V, E)$ yönlü çizgesi, ayrı iki düğüm $s, t \in V$ ve her (u, v) kenarına negatif olmayan bir $c(u, v) \geq 0$ kapasitesi atayan fonksiyondan oluşur. Çizgede olmayan bir (u, v) çifti için $c(u, v) = 0$ kabul edilir. **Maksimum akış problemi** (*maximum flow problem*), kapasiteleri aşmadan kaynaktan hedefe gönderilebilecek en büyük akış miktarını bulmaktır.

Notun büyük bölümünde aynı küçük ağ kullanılacak. Şekil 4.4.1 bu ağı gösterir. Altı düğüm ve yedi kenar vardır; kenar etiketleri kapasitelerdir. Kaynak S, hedef T'dir.



Şekil 4.4.1: Örnek akış ağı. Kenar etiketleri kapasitedir: $S \to A$ 4, $S \to C$ 3, $A \to B$ 4, $B \to C$ 3, $B \to T$ 2, $C \to D$ 6, $D \to T$ 6. Bu

ağda maksimum akış 7'dir.

Maksimum akışın 7 olduğunu iki yönden görmek gerekir. Birincisi, 7 birim gönderilebilir; bunu gösteren bir akış birazdan verilecek. İkincisi, 7'den fazlası gönderilemez. Bu ikinci iddia kaynağa bakarak hemen anlaşılır. S'den çıkan iki kenarın kapasiteleri toplamı $4 + 3 = 7$'dir. Kaynaktan çıkan her birim bu iki kenardan birini kullanmak zorundadır. Öyleyse hedefe 7'den fazlası ulaşamaz. Bu basit gözlem, ilerideki *kesim* kavramının ilk örneğidir.

Aynı akıl yürütme hedef tarafında da yapılabilir. T'ye giren kenarların kapasiteleri $2 + 6 = 8$'dir. Bu da bir üst sınırdır, ama 7'den gevşektir. İki sınırın küçüğü geçerlidir. Her zaman doğru bir üst sınır şudur: maksimum akış, kaynaktan çıkan kapasitelerin toplamını ve hedefe giren kapasitelerin toplamını aşamaz. Bu sınır her zaman sıkı değildir; darboğaz ağın ortasında da olabilir.

4.4.1.2 Akış ve kısıtları

Akış, her kenara o kenardan geçen miktarı atar. Bu atamanın anlamlı olması için iki kural sağlanmalıdır.

Tanım 4.4.2

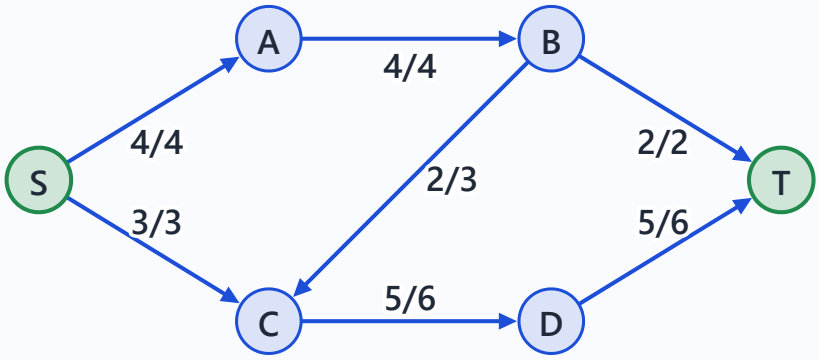
Akış (*flow*), her (u, v) kenarına bir $f(u, v)$ sayısı atayan ve şu iki kısıtı sağlayan fonksiyondur:

- **Kapasite kısıtı:** her kenar için $0 \le f(u, v) \le c(u, v)$.
- **Akış korunumu** (*flow conservation*): s ve t dışındaki her v düğümü için $\Sigma_u f(u, v) = \Sigma_w f(v, w)$, yani giren akış çıkan akışa eşittir.

Akışın **değeri** $|f| = \Sigma_v f(s, v) - \Sigma_v f(v, s)$'dir: kaynaktan çıkan net akış.

Kapasite kısıtı borunun taşıma sınırını anlatır. Akış negatif olamaz; boru ters yönde çalışmaz. Kapasiteyi de aşamaz. Korunum kısıtı ise ara düğümlerin depo olmadığını söyler. Bir ara düğüme giren her birim bir yerden çıkmalıdır. Kaynak ve hedef bu kuraldan muaftır: kaynak akışı üretir, hedef tüketir.

Değer tanımındaki çıkarma terimi ilk bakışta gereksiz görünebilir. Örnek ağımızda kaynağa giren kenar yoktur, dolayısıyla bu terim sıfırdır. Ama genel bir ağda kaynağa geri dönen kenarlar olabilir. Kaynaktan 5 birim çıkıp 1 birimi geri dönüyorsa, ağa gerçekte 4 birim gönderilmiştir. Bu yüzden değer, kaynaktan çıkan *net* akıştır. Korunum sayesinde bu net akış, hedefe giren net akışa eşittir: ara düğümlerin hiçbiri akış biriktirmediği için kaynaktan çıkan her şey eninde sonunda hedefe varır.



Şekil 4.4.2: Aynı ağda değeri 7 olan bir akış. Etiketler *akış/kapasite* biçimindedir. *C*'ye *3 + 2 = 5* birim girer ve 5 birim çıkar; *|f| = 4 + 3 = 7*'dir.

Örnek 4.4.1

Şekil 4.4.2'deki akışın geçerli olduğunu denetleyelim.

Kapasite kısıtı. Kenarlar tek tek: $4 \le 4$, $3 \le 3$, $4 \le 4$, $2 \le 3$, $2 \le 2$, $5 \le 6$, $5 \le 6$. Hiçbir kenar sınırını aşmıyor.

Korunum. Dört ara düğüm vardır.

- *A*: giren $f(S, A) = 4$, çıkan $f(A, B) = 4$.
- *B*: giren $f(A, B) = 4$, çıkan $f(B, C) + f(B, T) = 2 + 2 = 4$.
- *C*: giren $f(S, C) + f(B, C) = 3 + 2 = 5$, çıkan $f(C, D) = 5$.
- *D*: giren $f(C, D) = 5$, çıkan $f(D, T) = 5$.

Değer. $|f| = f(S, A) + f(S, C) = 4 + 3 = 7$. Hedef tarafından sağlama: $f(B, T) + f(D, T) = 2 + 5 = 7$. İki sayı eşittir.

Bu akışta bazı kenarlar sınırına kadar doludur. $f(u, v) = c(u, v)$ olan bir kenara **doymuş** (*saturated*) kenar denir; üzerinden daha fazla akış gönderilemez. Şekildeki akışta $S \to A$, $S \to C$, $A \to B$ ve $B \to T$ doymuştur. Kaynaktan çıkan iki kenarın da doymuş olması, akışın neden artırılamayacağını açıklar. Doymuş kenarlar, maksimum akışın neden daha büyük olamayacağını anlatan kesimlerin parçasıdır.

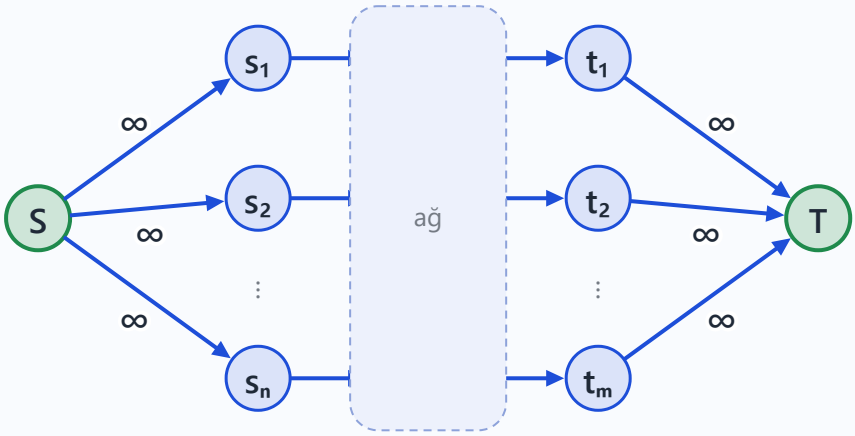
Sık yapılan hata

Akışın değerini bütün kenarlardaki akışların toplamı sanmak. Şekildeki akışta kenar akışlarının toplamı *4 + 3 + 4 + 2 + 2 + 5 + 5 = 25*'tir, ama akışın değeri 7'dir. Aynı birim akış yol boyunca birçok kenardan geçer ve her kenarda yeniden sayılır. Değer yalnızca kaynağın (ya da hede-

fin) kenarlarından okunur.

4.4.1.3 Birden çok kaynak ve hedef

Gerçek problemlerde tek bir kaynak ve tek bir hedef nadiren bulunur. Birkaç fabrika birkaç depoya mal gönderir; birkaç sunucu birkaç kullanıcı grubuna veri akıtır. Bu durumu ayrı bir algoritma gerektirmeden çözmenin yolu küçük bir çizge dönüşümüdür.



Şekil 4.4.3: Süper kaynak ve süper hedef. Yeni S düğümü sonsuz kapasiteli kenarlarla bütün s_i kaynaklarına, bütün t_j hedefleri de sonsuz kapasiteli kenarlarla yeni T düğümüne bağlanır.

Ağa iki yeni düğüm eklenir: **süper kaynak** S ve **süper hedef** T. Her eski kaynak s_i için bir $S \to s_i$ kenarı, her eski hedef t_j için bir $t_j \to T$ kenarı konur. Bu kenarlara sınırsız (∞) kapasite verilir. Böylece yeni kenarlar hiçbir zaman darboğaz olmaz; akışı yalnızca eski ağın kenarları sınırlar. Yeni ağda S'den T'ye maksimum akış, eski ağda bütün kaynaklardan bütün hedeflere gönderilebilecek toplam akışa eşittir. Uygulamada ∞ yerine, eski ağın bütün kapasitelerinin toplamından büyük herhangi bir sayı yeterlidir.

Dönüşüm başka sınırlamalara da uyarlanabilir. Örneğin s_1 fabrikası en fazla 50 birim üretebiliyorsa $S \to s_1$ kenarına ∞ yerine 50 kapasitesi verilir. Aynı düşünce, düğümlerin de kapasitesi olduğu problemlerde kullanılır: bir düğüm iki parçaya bölünür ve iki parça arasına düğümün kapasitesi kadar bir kenar konur. Bu yöntem uygulamalar arasında yeniden ele alınacak.

4.4.2 Artık ağ ve artış yolu

Maksimum akışı bulmanın akla gelen ilk yolu açgözlü bir yöntemdir. Kaynaktan hedefe dolu olmayan kenarlardan geçen bir yol bulunur, yol boyunca gönderilebilecek kadar akış gönderilir ve bu iş yol kalmayana kadar sürdürülür. Bu yöntem bazen doğru sonucu verir, bazen vermez. Sorun, ilk kararların

geri alınamamasıdır. Kötü seçilmiş bir yol, kenarları başka yolların daha iyi kullanabileceği bir biçimde doldurabilir. Çözüm, gönderilmiş akışı geri alma imkânını modele katmaktır. Bunu yapan yapı artık ağdır.

4.4.2.1 Artık ağ

Tanım 4.4.3

f akışı altında bir kenarın **artık kapasitesi** (*residual capacity*) iki biçimde tanımlanır. Özgün ağdaki her (u, v) kenarı için:

- **İleri kenar:** $c_f(u, v) = c(u, v) - f(u, v)$. Kenardan hâlâ gönderilebilecek miktardır.
- **Ters kenar:** $c_f(v, u) = f(u, v)$. Gönderilmiş akışın geri alınabilecek miktarıdır.

Artık ağ (*residual network*) G_f, yalnızca $c_f > 0$ olan kenarlardan oluşur.

Artık ağ, mevcut akış altında hâlâ yapılabilecek değişiklikleri gösterir. İleri kenar "bu yönde daha fazla gönderebilirim" der. Ters kenar ise "bu yönde gönderdiğimin bir kısmını geri çekebilirim" der. Şekil 4.4.4 bunu tek bir kenarda gösterir. Kapasitesi 5 olan $u \to v$ kenarında 3 birim akış vardır. Artık ağda bu kenar iki kenara dönüşür: kalan 2 birim için $u \to v$ ileri kenarı ve geri alınabilecek 3 birim için $v \to u$ ters kenarı.



Şekil 4.4.4: (a) Akış ağında bir kenar: akış 3, kapasite 5. (b) Artık ağdaki karşılığı: $5 - 3 = 2$ birimlik ileri kenar (düz ok) ve 3 birimlik ters kenar (kesikli ok).

İki uç durum da öğreticidir. Akış sıfırsa ters kenarın kapasitesi sıfır olur ve ters kenar artık ağa girmez; artık ağ, özgün ağın aynısıdır. Kenar doymuşsa ileri kenarın kapasitesi sıfır olur ve artık ağda yalnızca ters kenar kalır. Her özgün kenar artık ağda en fazla iki kenar üretir. Bu yüzden artık ağın kenar sayısı en fazla $2E$'dir.

Örnek 4.4.2

Şekil 4.4.2'deki akışın artık ağını kuralım. Her kenar için iki sayı hesaplanır: ileri artık kapasite $c - f$ ve ters artık kapasite f.

- $S \to A$ $(4/4)$: ileri 0, ters $A \to S$ 4.
- $S \to C$ $(3/3)$: ileri 0, ters $C \to S$ 3.
- $A \to B$ $(4/4)$: ileri 0, ters $B \to A$ 4.
- $B \to C$ $(2/3)$: ileri $B \to C$ 1, ters $C \to B$ 2.
- $B \to T$ $(2/2)$: ileri 0, ters $T \to B$ 2.
- $C \to D$ $(5/6)$: ileri $C \to D$ 1, ters $D \to C$ 5.
- $D \to T$ $(5/6)$: ileri $D \to T$ 1, ters $T \to D$ 5.

Sıfır olanlar atılınca artık ağda on kenar kalır. S'den çıkan artık kenar yoktur; S'nin yalnızca gelen ters kenarları vardır. Bu yüzden artık ağda S'den T'ye yol yoktur. Bu gözlemin anlamı birazdan açıklanacak: akış maksimumdur.

4.4.2.2 Ters kenarın anlamı

Ters kenar fiziksel bir boru değildir. $v \to u$ yönünde gerçek bir bağlantı olması gerekmez. Ters kenar bir muhasebe kaydıdır: daha önce u'dan v'ye gönderilmiş akışın iptal edilebileceğini gösterir. Ters kenar boyunca δ birim "göndermek", $f(u, v)$'yi δ kadar azaltmak demektir.

Bu iptal neden işe yarar? Bir artış yolu $v \to u$ ters kenarını kullanıyorsa, yol v'ye gelen akışı u'dan gelen eski akışın yerine koyar. u'dan v'ye giden eski akış artık serbest kalır ve yolun devamında u'dan başka bir yöne gider. Sonuçta akış yeniden düzenlenir: bazı birimler eski yollarını bırakıp yenilerine geçer. Korunum her düğümde sağlanmaya devam eder. Ters kenarlar sayesinde algoritma ilk yanlış seçimlerini düzeltebilir. Bu düzeltmenin somut bir örneği Ford-Fulkerson'ın ikinci örneğinde görülecek.

Sık yapılan hata

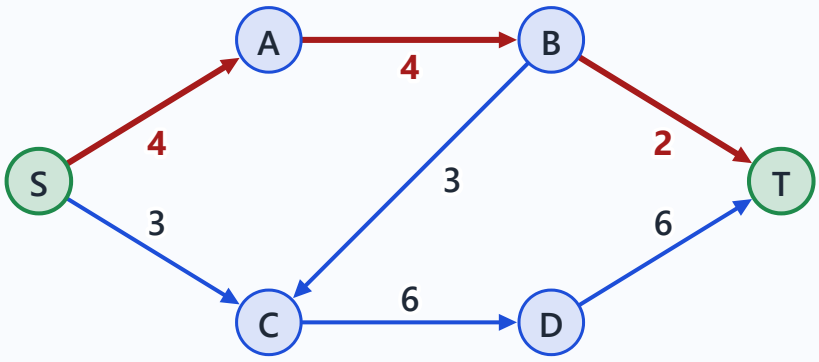
Ters kenarı, ağa eklenmiş gerçek bir kenar ya da "geriye akış göndermek" olarak yorumlamak. Ters kenar yönünde akış artmaz; ileri kenardaki akış azalır: $f(u, v) \mathrel{-}= \delta$. Bu yüzden son akışta hiçbir kenarın akışı negatif olmaz ve ters kenarlar özgün ağa hiçbir kapasite eklemez.

4.4.2.3 Artış yolu ve darboğaz

Tanım 4.4.4

Artış yolu (*augmenting path*), artık ağ G_f'de kaynaktan hedefe giden yönlü bir yoldur. Yolun **darboğazı** (*bottleneck*), yoldaki kenarların artık kapasitelerinin en küçüğüdür: $c_f(p) = \min \{ c_f(u, v) : (u, v) \in p \}$.

Artış yolundaki her kenarın artık kapasitesi pozitiftir. Yani yol ya kapasitesinden az akış taşıyan ileri kenarlardan ya da üzerinde akış bulunan kenarların ters yönlerinden geçer. Yol boyunca gönderilebilecek en fazla akış, yolun en dar kenarı kadardır. Zincirin gücü en zayıf halkası kadardır sözünün akıştaki karşılığı budur.



Şekil 4.4.5: Başlangıç artık ağında $S \to A \to B \to T$ artış yolu (vurgulu). Artık kapasiteler 4, 4 ve 2'dir; darboğaz $\min(4, 4, 2) = 2$'dir ve $B \to T$ kenarından kaynaklanır.

Akış başlangıçta her yerde sıfırsa artık ağ özgün ağın aynısıdır ve artış yolu sıradan bir yönlü yoldur. Artış yolları genellikle derinlik öncelikli arama (DFS, *depth-first search*) ya da genişlik öncelikli arama (BFS, *breadth-first search*) ile bulunur. Hangi aramanın kullanıldığı sonuçtaki akış değerini değiştirmez, ama algoritmanın kaç adım süreceğini çok değiştirir. Bu fark Edmonds-Karp algoritmasının çıkış noktasıdır.

4.4.2.4 Akışı bir yol boyunca artırmak

Bir artış yolu p ve darboğazı $\delta = c_f(p)$ bulunduğunda akış şöyle güncellenir. Yoldaki her (u, v) kenarı için:

- (u, v) özgün ağın bir kenarıysa (ileri kenar) $f(u, v)$, δ kadar artırılır.
- (u, v) bir ters kenarsa, yani özgün kenar (v, u) ise $f(v, u)$, δ kadar azaltılır.

Teorem 4.4.1 (Artırma)

f bir akış, *p* artık ağda bir artış yolu ve $\delta = c_f(p)$ olsun. Yukarıdaki güncellemeyle elde edilen *f′* yine bir akıştır ve $|f'| = |f| + \delta$'dır.

İspat. **Kapasite.** İleri kenarda yeni akış $f(u, v) + \delta \le f(u, v) + c_f(u, v) = c(u, v)$ olur. Ters kenarda yeni akış $f(v, u) - \delta \ge f(v, u) - c_f(u, v) = 0$ olur. Her iki durumda da akış $[0, c]$ aralığında kalır.

Korunum. Yol üzerindeki bir ara düğüm *v*'yi düşünelim. Yol *v*'ye bir kenarla girer, bir kenarla çıkar. Giren kenar ileri kenarsa *v*'ye giren akış δ artar; ters kenarsa *v*'den çıkan akış δ azalır. Her iki durumda *v*'nin "giren − çıkan" farkı $+\delta$ değişir. Çıkan kenar için benzer düşünceyle fark $-\delta$ değişir. Toplam değişim sıfırdır; korunum bozulmaz.

Değer. Yol basit olduğu için kaynağı yalnızca bir kez, ilk kenarında kullanır. Bu kenar kaynaktan çıkar. İleri kenarsa kaynaktan çıkan akış δ artar; ters kenarsa kaynağa giren akış δ azalır. İki durumda da net çıkış δ artar. □

Bu teorem bütün artış yolu tabanlı algoritmaların temelidir. Her artışta akışın değeri kesin olarak büyür ve akış geçerliliğini korur. Geriye iki soru kalır. Artış yolu kalmadığında akış gerçekten maksimum mudur? Kaç artış gerekir? Birinci sorunun cevabı kesimlerle verilir. İkinci sorunun cevabı, yolun nasıl seçildiğine bağlıdır.

Güncellemenin iki önemli sonucu daha vardır. Darboğaz kenarının artık kapasitesi tam olarak δ'dır; güncellemeden sonra bu artık kapasite sıfıra iner. Yani her artışta yoldaki en az bir kenar artık ağdan çıkar. Ayrıca yoldaki her kenarın karşı yönünde δ birimlik artık kapasite doğar. Artık ağ böylece her artışta biraz değişir: bazı kenarlar kaybolur, bazı ters kenarlar ortaya çıkar.

4.4.3 Kesimler

Bir akışın maksimum olduğunu kanıtlamak için daha büyük bir akışın olamayacağını göstermek gerekir. Bütün akışları tek tek denemek olanaksızdır. Kesim kavramı bunun yerine tek bir sayıyla üst sınır verir. Kaynağı içeren tarafla hedefi içeren taraf arasındaki kenarlar, akışın geçmek zorunda olduğu bir "sınır kapısı" gibidir. Kapının genişliği, geçebilecek akışı sınırlar.

4.4.3.1 s–t kesimi ve kapasitesi

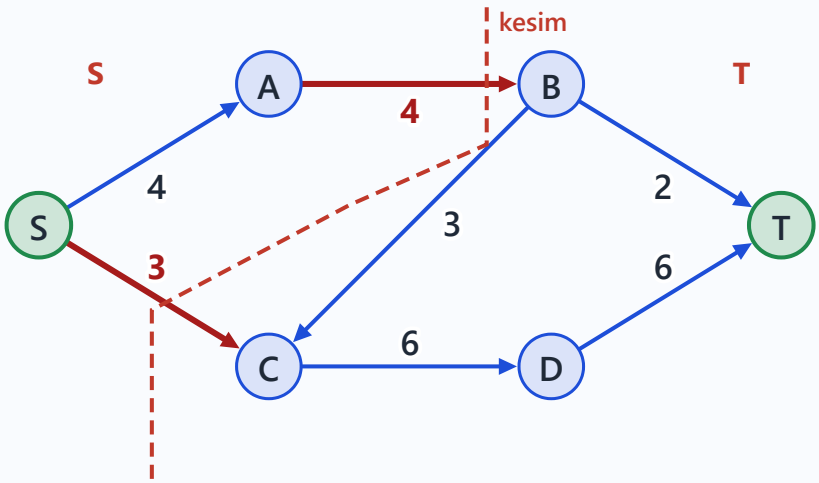
Tanım 4.4.5

s–t kesimi (*s–t cut*) *(S, T)*, düğüm kümesi *V*'yi $s \in S$ ve $t \in T$ olacak biçimde ayrık iki kümeye bölen ayrımdır. Kesimin **kapasitesi**, *S*'den *T*'ye giden kenarların kapasiteleri toplamıdır:

$$c(S, T) = \Sigma_{u \in S, v \in T}\, c(u, v)$$

Kapasitesi en küçük olan kesime **minimum kesim** (*minimum cut*) denir.

Tanımda kesim bir kenar kümesi değil, bir düğüm ayrımıdır. Kesilen kenarlar bu ayrımdan çıkar: bir ucu *S*'de, öbür ucu *T*'de olan kenarlar. Bu kenarlar ağdan silinirse kaynaktan hedefe yönlü yol kalmaz. Kapasite hesabında yalnızca *S*'den *T*'ye giden kenarlar sayılır. *T*'den *S*'ye giden kenarlar kapasiteye eklenmez, çünkü akışın hedefe ulaşmasını engelleyen onlar değildir.



Şekil 4.4.6: *S = {S, A}* ve *T = {B, C, D, T}* kesimi. Kesik çizgiyi soldan sağa geçen kenarlar *A→B* (4) ve *S→C* (3)'tür: *c(S, T) = 4 + 3 = 7*.

Şekil 4.4.6'daki kesimde *B→C* kenarı da kesik çizgiyi geçer, ama *T* tarafından *T* tarafına gider; iki ucu da *T*'dedir. Bu yüzden hesaba girmez. Benzer biçimde *S→A* kenarının iki ucu da *S* tarafındadır.

Örnek 4.4.3

Örnek ağda dört ara düğüm vardır: *A, B, C, D*. Her biri *S* ya da *T* tarafına konabildiği için toplam $2^4 = 16$ s–t kesimi vardır. Birkaçını hesaplayalım.

- *S = {S}*: kesilen kenarlar *S→A* ve *S→C*; *4 + 3 = 7*.
- *S = {S, A, B}*: kesilen kenarlar *S→C* (3), *B→C* (3), *B→T* (2); toplam 8.

- $S = \{S, C\}$: $S \to A$ (4) ve $C \to D$ (6); toplam 10. $B \to C$ kenarı T'den S'ye gider ve sayılmaz.
- $S = \{S, B, D\}$: $S \to A$ (4), $S \to C$ (3), $B \to C$ (3), $B \to T$ (2), $D \to T$ (6); toplam 18. $A \to B$ ve $C \to D$, T'den S'ye gider.

Bütün kesimler hesaplandığında en küçük kapasite 7'dir ve iki kesim bu değeri alır: $\{S\}$ ve $\{S, A\}$. Minimum kesim tek olmak zorunda değildir.

S tarafı	S'den T'ye kenarlar	$c(S, T)$
$\{S\}$	S→A 4, S→C 3	7
$\{S, A\}$	A→B 4, S→C 3	7
$\{S, A, B\}$	S→C 3, B→C 3, B→T 2	8
$\{S, A, B, C, D\}$	B→T 2, D→T 6	8
$\{S, C\}$	S→A 4, C→D 6	10
$\{S, B, D\}$	S→A 4, S→C 3, B→C 3, B→T 2, D→T 6	18

Tablo 4.4.1: Örnek ağda bazı s–t kesimleri. On altı kesimin en küçüğü 7'dir.

Sık yapılan hata

Minimum kesimi en az kenarı kesen ayrım sanmak. Minimum kesim kenar *sayısını* değil, kapasite *toplamını* en küçük yapar. Kapasitesi 100 olan tek bir kenarı kesmek, kapasitesi 1 olan üç kenarı kesmekten pahalıdır. Kenar sayısı ancak bütün kapasiteler 1 olduğunda kapasiteyle aynı şeyi ölçer.

4.4.3.2 Kesimden geçen net akış

Bir kesimden geçen **net akış**, S'den T'ye giden kenarlardaki akışların toplamından T'den S'ye giden kenarlardaki akışların toplamının çıkarılmasıyla bulunur:

$$f(S, T) = \Sigma_{u \in S, v \in T} f(u, v) - \Sigma_{u \in S, v \in T} f(v, u)$$

Kapasitenin aksine net akışta geri yöndeki kenarlar da hesaba girer, çünkü geri dönen akış karşıya geçen akıştan düşülmelidir.

Teorem 4.4.2 (Kesim ve akış)

Her f akışı ve her (S, T) s–t kesimi için $f(S, T) = |f|$'dir. Bu yüzden $|f| \le c(S, T)$'dir.

İspat. S'deki her düğüm için "çıkan − giren" farkını yazalım ve bu farkları toplayalım. Kaynak için fark, tanım gereği $|f|$'dir. S'deki diğer düğümler ara düğümdür (hedef T'dedir); korunum nedeniyle farkları sıfırdır. Öyleyse toplam $|f|$'dir.

Aynı toplamı kenarlar üzerinden de hesaplayabiliriz. İki ucu da S'de olan bir kenar, bir düğümün çıkanında ve başka bir düğümün gireninde birer kez görünür; katkıları birbirini götürür. Geriye yalnızca kesimi geçen kenarlar kalır. S'den T'ye gidenler toplamda artı, T'den S'ye gidenler eksi işaretle görünür. Bu tam olarak $f(S, T)$'dir. Dolayısıyla $f(S, T) = |f|$.

Eşitsizlik için ileri kenarlarda $f(u, v) \le c(u, v)$, geri kenarlarda $f(v, u) \ge 0$ kullanılır: $|f| = f(S, T) \le \Sigma\, c(u, v) - 0 = c(S, T)$. □

Teoremin söylediği şey sezgiye uygundur. Kaynaktan çıkan bütün akış hedefe varmak için kesimi bir yerde geçmelidir. Geri dönen akış olursa, o da en az bir kez daha ileri geçmek zorundadır. Sonunda kesimden geçen net miktar, ağa gönderilen miktara eşittir. Şekil 4.4.2'deki akışı $\{S, A\}$ kesiminde sınayalım: ileri kenarlar $A \to B$ (4) ve $S \to C$ (3), geri kenar yok; net akış 7'dir. $\{S, C\}$ kesiminde ileri kenarlar $S \to A$ (4) ve $C \to D$ (5), geri kenar $B \to C$ (2); net akış $4 + 5 - 2 = 7$'dir. İki kesim farklı, net akış aynıdır.

Teoremin ikinci yarısı, her kesimin her akış için bir üst sınır olduğunu söyler. Herhangi bir akışın değeri, herhangi bir kesimin kapasitesini aşamaz. Öyleyse maksimum akış da minimum kesimin kapasitesini aşamaz. Bu bir eşitsizliktir. Asıl sürpriz, bu eşitsizliğin her zaman eşitlik olmasıdır.

4.4.3.3 Maksimum akış minimum kesim teoremi

Teorem 4.4.3 (Maksimum akış minimum kesim)

Bir f akışı için aşağıdaki üç ifade denktir (Ford ve Fulkerson, 1956):

1. f bir maksimum akıştır.
2. Artık ağ G_f'de hiçbir artış yolu yoktur.
3. Bir (S, T) kesimi için $|f| = c(S, T)$'dir.

Dolayısıyla maksimum akışın değeri, minimum kesimin kapasitesine eşittir.

İspat. **(1) ⇒ (2).** Artık ağda bir artış yolu olsaydı, Teorem 4.4.1'e göre akış darboğaz kadar artırılabilirdi. Bu, f'nin maksimum olmasıyla çelişir.

(2) ⇒ (3). S, artık ağda kaynaktan ulaşılabilen düğümlerin kümesi olsun; $T = V - S$. Kaynak S'dedir. Artış yolu olmadığı için hedef S'de değildir; (S, T) bir s–t kesimidir. $u \in S$, $v \in T$ olan bir (u, v) kenarı için $f(u, v) = c(u, v)$ olmalıdır; aksi hâlde $c_f(u, v) > 0$ olur ve v de ulaşılabilir olurdu. Aynı nedenle T'den S'ye giden bir (v, u) kenarında $f(v, u) = 0$ olmalıdır; aksi hâlde $u \to v$ ters kenarı artık ağda bulunurdu. Teorem 4.4.2'den $|f| = f(S, T) = \Sigma\, c(u, v) - 0 = c(S, T)$.

(3) ⇒ (1). Her akış her kesimin kapasitesini aşamaz. *|f| = c(S, T)* ise hiçbir akış *c(S, T)*'den, dolayısıyla *|f|*'den büyük olamaz. □

Teoremin üç ayrı yararı vardır. Birincisi, artış yolu tabanlı algoritmaların doğruluğunu kanıtlar: yol kalmadığında durmak güvenlidir. İkincisi, bir akışın maksimum olduğunu kolayca denetlenebilen bir *sertifika* verir. Değeri bir kesimin kapasitesine eşit olan akış, başka hiçbir hesap yapmadan maksimumdur. Üçüncüsü, minimum kesimi bulmanın yolunu gösterir: son artık ağda kaynaktan ulaşılabilen düğümler *S* tarafını oluşturur.

Örnek ağımızda bulunan akışın değeri 7'dir ve *{S}* kesiminin kapasitesi de 7'dir. Teoremin üçüncü maddesi sağlanır; akış maksimumdur. Örnek 4.4.2'de kurulan artık ağda kaynaktan hiçbir yere gidilemiyordu. Ulaşılabilen küme yalnızca *{S}*'dir ve ispattaki kesim tam olarak budur.

4.4.3.4 Minimum kesimi bulmak

Teoremin ispatı aynı zamanda bir algoritmadır. Maksimum akış bulunduktan sonra son artık ağda kaynaktan bir arama yapılır. Ziyaret edilen düğümler *S* tarafıdır. Bir ucu ziyaret edilmiş, öbür ucu edilmemiş özgün kenarlar minimum kesimin kenarlarıdır. Aşağıdaki Java metodu bu işi yapar. Metot, artık kapasiteleri tutan `artik` matrisini ve özgün kapasitelerin bir kopyasını kullanır.

```
List<int[]> minKesim(int kaynak, int[][] kapasite) {
  boolean[] ziyaret = new boolean[V];
  Queue<Integer> kuyruk = new LinkedList<>();
  kuyruk.add(kaynak);
  ziyaret[kaynak] = true;
  while (!kuyruk.isEmpty()) {          // son artık ağda BFS
    int u = kuyruk.poll();
    for (int v = 0; v < V; v++)
      if (!ziyaret[v] && artik[u][v] > 0) {
        ziyaret[v] = true;
        kuyruk.add(v);
      }
  }
  List<int[]> kesim = new ArrayList<>();
  for (int u = 0; u < V; u++)          // S'den T'ye giden özgün kenarlar
    for (int v = 0; v < V; v++)
      if (ziyaret[u] && !ziyaret[v] && kapasite[u][v] > 0)
        kesim.add(new int[]{u, v});
  return kesim;
}
```

Metodun ilk yarısı sıradan bir BFS'dir. Tek fark, kenarın varlığının `artik[u][v] > 0` ile sınanmasıdır: arama yalnızca artık ağda gezinir. İkinci yarı bütün düğüm çiftlerini tarar ve `ziyaret[u]` doğru, `ziyaret[v]` yanlış olan çiftlerden özgün ağda kenar bulunanları listeye ekler. Kapasite koşulu önemlidir. Ters kenarlar artık matrisinde yer kaplar, ama özgün ağda yoktur; kesim kenarı olamazlar.

Komşuluk matrisiyle BFS $O(V^2)$, çift taraması da $O(V^2)$ sürer. Toplam süre $O(V^2)$'dir; bu, maksimum akışı bulmanın maliyetinin yanında önemsizdir. Ek bellek `ziyaret` dizisi, kuyruk ve sonuç listesi için $O(V + k)$'dir; k kesimdeki kenar sayısıdır. Metot, ileride Ford-Fulkerson'ın ikinci örneğinin son artık ağında çalıştırıldığında $S \to 1$ (10) ve $3 \to 4$ (9) kenarlarını döndürür. Toplam 19, o ağdaki maksimum akışa eşittir.

4.4.3.5 Ağ akış algoritmalarına genel bakış

Maksimum akışı bulan algoritmaların hepsi aynı amaca, aynı kavramlarla ulaşır. Farkları, akışı hangi sırayla ve hangi parçalar hâlinde büyüttükleridir. Tablo 4.4.2 notta ele alınan dört algoritmayı özetler.

Algoritma	Yaklaşım	Süre	Not
Ford-Fulkerson (1956)	herhangi bir artış yolu (çoğunlukla DFS)	$O(E \cdot \lvert f^* \rvert)$	basit; süre akış değerine bağlı
Edmonds-Karp (1972)	en kısa artış yolu (BFS)	$O(V \cdot E^2)$	Ford-Fulkerson'ın polinom zamanlı hâli
Dinic (1970)	düzey çizgesi + engelleyici akış	$O(V^2 \cdot E)$	Edmonds-Karp'tan hızlı; birim kapasitede çok verimli
İtme–yeniden etiketleme (1988)	ön akış + yükseklikler, yerel işlemler	$O(V^2 \cdot E)$	yol aramaz; FIFO sürümü $O(V^3)$

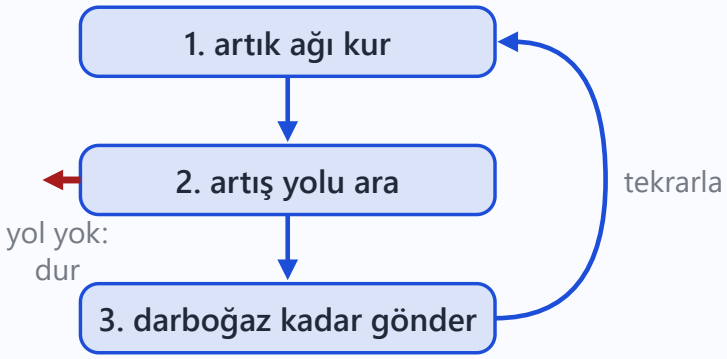
Tablo 4.4.2: Ağ akış algoritmaları. $\lvert f^* \rvert$ maksimum akışın değeridir; Ford-Fulkerson'ın süresi tam sayı kapasiteler için geçerlidir.

İlk üç algoritma artış yolu düşüncesine dayanır ve her adımda geçerli bir akış tutar. Dördüncüsü farklıdır: ara adımlarda korunumu geçici olarak bozar ve en sonda düzeltir. Tablodaki yıllar da ilginç bir ayrıntı taşır. Dinic'in algoritması Edmonds-Karp'tan önce, 1970'te Sovyetler Birliği'nde yayımlanmıştır; iki çalışma birbirinden bağımsızdır.

4.4.4 Ford-Fulkerson yöntemi

4.4.4.1 Yöntemin fikri

Ford-Fulkerson yöntemi L. R. Ford Jr. ve D. R. Fulkerson'ın 1956'da yayımladığı çalışmaya dayanır. Kapasiteli yönlü bir çizgede iki düğüm arasındaki maksimum akışı bulur. Fikir şimdiye kadar kurulan kavramların doğal sonucudur: akış sıfırdan başlar, artık ağda bir artış yolu bulundukça akış o yol boyunca darboğaz kadar artırılır. Yol kalmadığında döngü durur; maksimum akış minimum kesim teoremi sonucun en iyi olduğunu garanti eder.



Şekil 4.4.7: Ford-Fulkerson döngüsü: artık ağı kur, artış yolu ara, darboğaz kadar gönder ve tekrarla. Yol bulunamazsa döngü biter.

Ford-Fulkerson'a çoğu kaynakta “algoritma” yerine **yöntem** denir. Bunun nedeni, artış yolunun nasıl bulunacağını belirtmemesidir. Herhangi bir yol kabul edilir. Yolu DFS ile arayan sürüm genellikle Ford-Fulkerson adıyla anılır; BFS ile arayan sürümün ayrı bir adı vardır: Edmonds-Karp. Yol seçiminin sonuca değil, süreye etki ettiği bu iki sürümün karşılaştırmasında açıkça görülecek.

4.4.4.2 Algoritma adımları

1. **Başlangıç.** Bütün kenarlarda $f(u, v) = 0$. Akış sıfır olduğu için her kenarın artık kapasitesi kendi kapasitesidir; artık ağ özgün ağın aynısıdır.
2. **Yol arama.** Artık ağda kaynaktan hedefe bir artış yolu p aranır. Yalnızca $c_f > 0$ olan kenarlar kullanılır.
3. **Darboğaz.** $\delta = \min c_f(u, v)$, $(u, v) \in p$ hesaplanır. Yol boyunca gönderilebilecek en fazla akış budur.
4. **Artırma.** Yol boyunca δ kadar akış gönderilir: ileri kenarlarda $f(u, v) \mathrel{+}= \delta$, ters kenarlarda $f(v, u) \mathrel{-}= \delta$; $|f| \mathrel{+}= \delta$. Yoldaki en az bir kenar doyar.
5. **Tekrar.** Artış yolu kalmayana kadar 2–4. adımlar tekrarlanır. Döngü bittiğinde $|f|$ maksimumdur.

Uygulamada akış ve artık kapasite ayrı ayrı tutulmak zorunda değildir. Yalnızca artık kapasiteler saklanabilir. Akış eklenirken ileri kenarın artık kapasitesi δ azalır, ters kenarınki δ artar. Bir kenarın akışı gerektiğinde $f(u, v) = c(u, v) - c_f(u, v)$ ile geri hesaplanır. Sunumdaki Java kodları bu yolu izler.

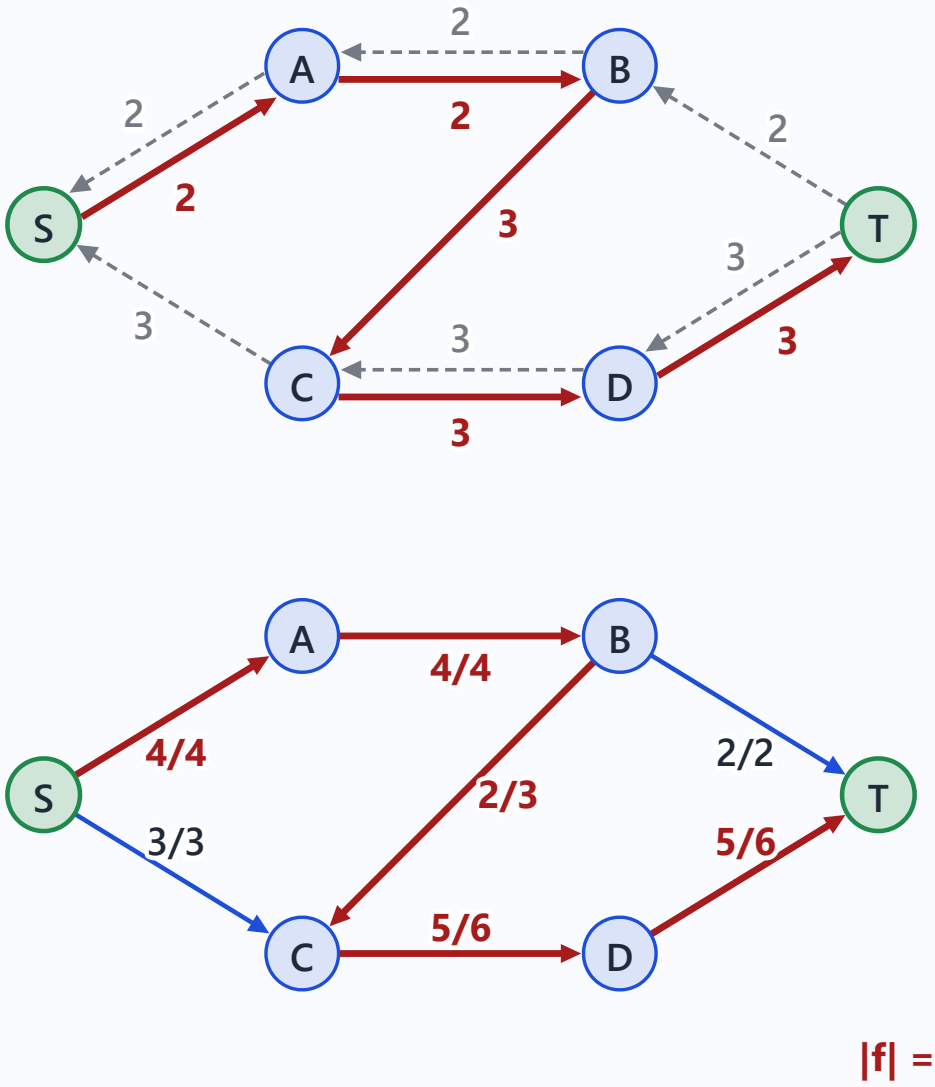
4.4.4.3 Örnek 1: adım adım

Yöntemi Şekil 4.4.1'deki ağda çalıştıralım. Başlangıçta bütün akışlar sıfırdır ve artık ağ özgün ağla aynıdır. Yolların bu örnekteki sırası, her seferinde kenar sayısı en az olan yolu seçen bir aramanın vereceği sıradır. Ford-Fulkerson herhangi bir yolu kabul ettiği için bu sıra geçerlidir.

Birinci artış. $p = S \to A \to B \to T$. Artık kapasiteler 4, 4 ve 2'dir; darboğaz 2 birimdir ve $B \to T$'den gelir. Yol boyunca 2 birim gönderilir: $|f| = 2$. $B \to T$ doyar. Artık ağda $S \to A$ ve $A \to B$'de 2'şer birim kalır ve karşı yönlerinde 2'şer birimlik ters kenarlar doğar.

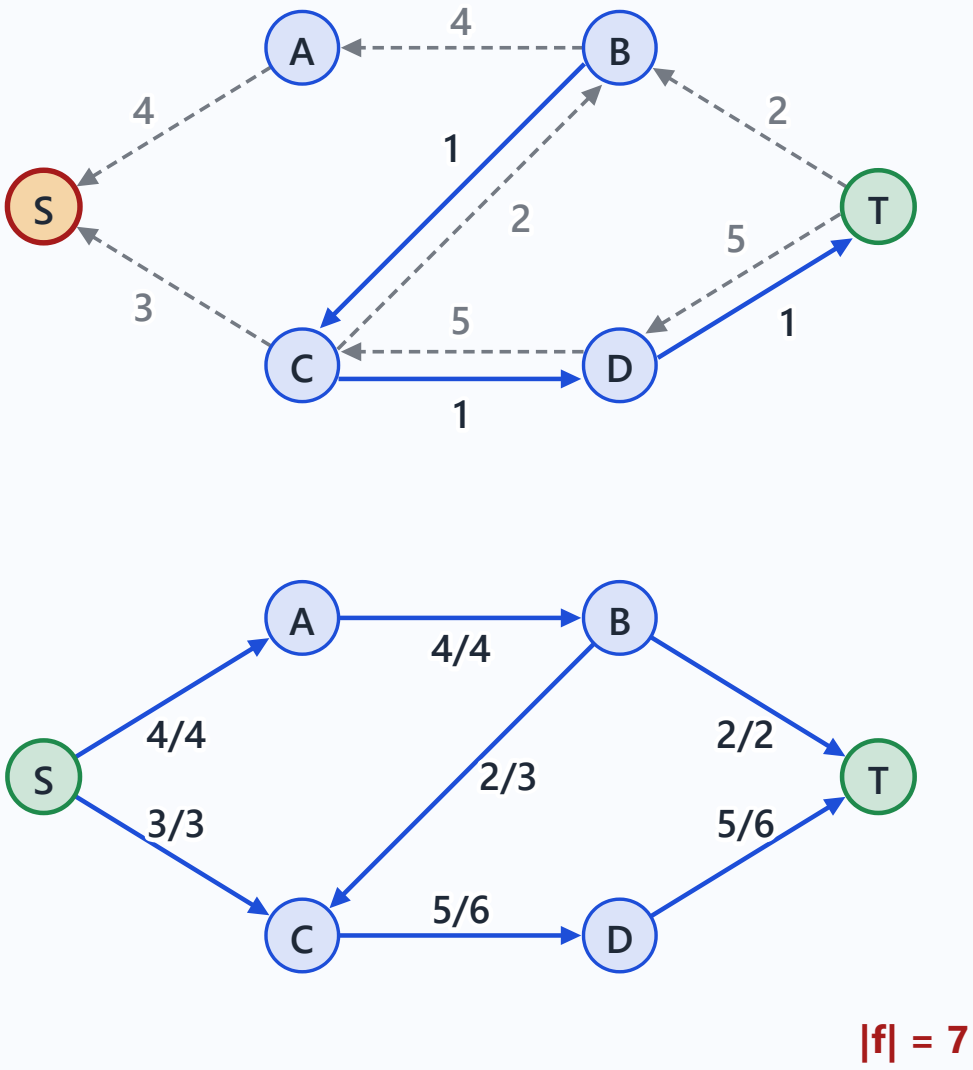
İkinci artış. $p = S \to C \to D \to T$. Artık kapasiteler 3, 6 ve 6'dır; darboğaz 3'tür. $|f| = 2 + 3 = 5$. $S{\to}C$ doyar. $C{\to}D$ ve $D{\to}T$'de 3'er birim kalır.

Üçüncü artış. S'den çıkan tek artık kenar $S{\to}A$'dır (2 birim). A'dan B'ye 2 birim gidilebilir. $B{\to}T$ doymuştur, ama $B{\to}C$ boştur (3 birim). $C{\to}D$ ve $D{\to}T$'de 3'er birim vardır. Yol $p = S \to A \to B \to C \to D \to T$, darboğaz $\min(2, 2, 3, 3, 3) = 2$'dir. $|f| = 7$.



Şekil 4.4.8: Üçüncü artış. (a) Artık ağda $S \to A \to B \to C \to D \to T$ yolu (vurgulu); kesikli oklar önceki iki artışın ters kenarlarıdır. (b) Yol boyunca 2 birim eklendikten sonraki akış; $|f| = 7$.

Üçüncü artıştan sonra S'den çıkan iki kenar da doymuştur. Artık ağda kaynaktan hiçbir yere gidilemez; döngü biter.



Şekil 4.4.9: (a) Son artık ağ: S'den çıkan artık kenar yoktur, yalnızca $A \to S$ ve $C \to S$ ters kenarları kalmıştır. (b) Maksimum akış: $|f| = 7$. Min kesim $(\{S\}, \text{diğerleri})$, kapasitesi $4 + 3 = 7$.

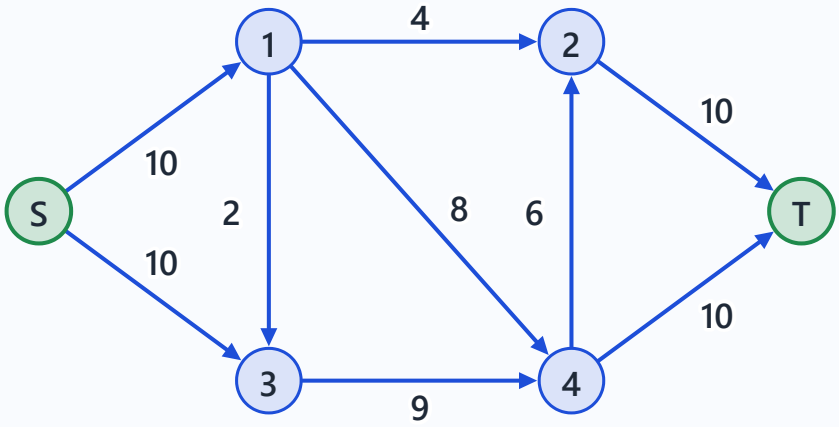
Artış	Yol	Artık kapasiteler	Darboğaz	$\|f\|$	Doyan kenar
1	S→A→B→T	4, 4, 2	2	2	B→T
2	S→C→D→T	3, 6, 6	3	5	S→C
3	S→A→B→C→D→T	2, 2, 3, 3, 3	2	7	S→A, A→B

Tablo 4.4.3: Örnek 1'deki artışlar.

Son akış Şekil 4.4.2'deki akışın aynısıdır: $B \to C$'de 2, $C \to D$'de 5 birim. Bu bir rastlantıdır. Maksimum akışın değeri tektir, kenarlara dağılımı ise tek olmak zorunda değildir. Aynı ağda $B \to T$'de 1, $B \to C$'de 3, $C \to D$ ve $D \to T$'de 6'şar birim taşıyan akış da geçerlidir ve değeri yine 7'dir. Değişmeyen tek şey, kaynaktan çıkan iki kenarın doymuş olmasıdır; minimum kesim $\{S\}$ bunu zorunlu kılar.

4.4.4.4 Örnek 2: ters kenarın kullanılması

İlk örnekte hiçbir artış yolu ters kenar kullanmadı. Ters kenarın neden gerekli olduğunu görmek için biraz daha büyük bir ağ gerekir. Şekil 4.4.10'daki ağda kaynak *S*, hedef *T*'dir; ara düğümler 1'den 4'e kadar numaralıdır.

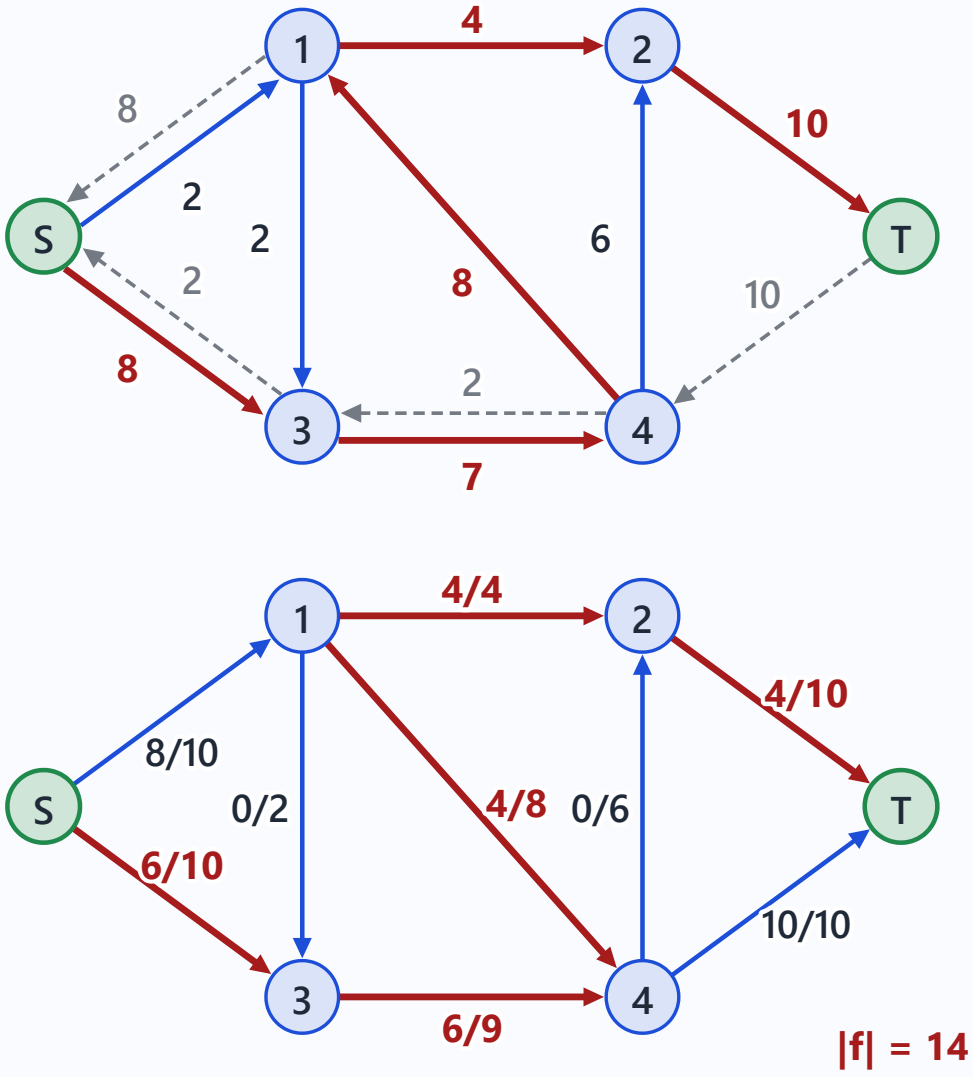


Şekil 4.4.10: İkinci örnek ağ. Kapasiteler: *S→1* 10, *S→3* 10, *1→2* 4, *1→3* 2, *1→4* 8, *3→4* 9, *4→2* 6, *2→T* 10, *4→T* 10.

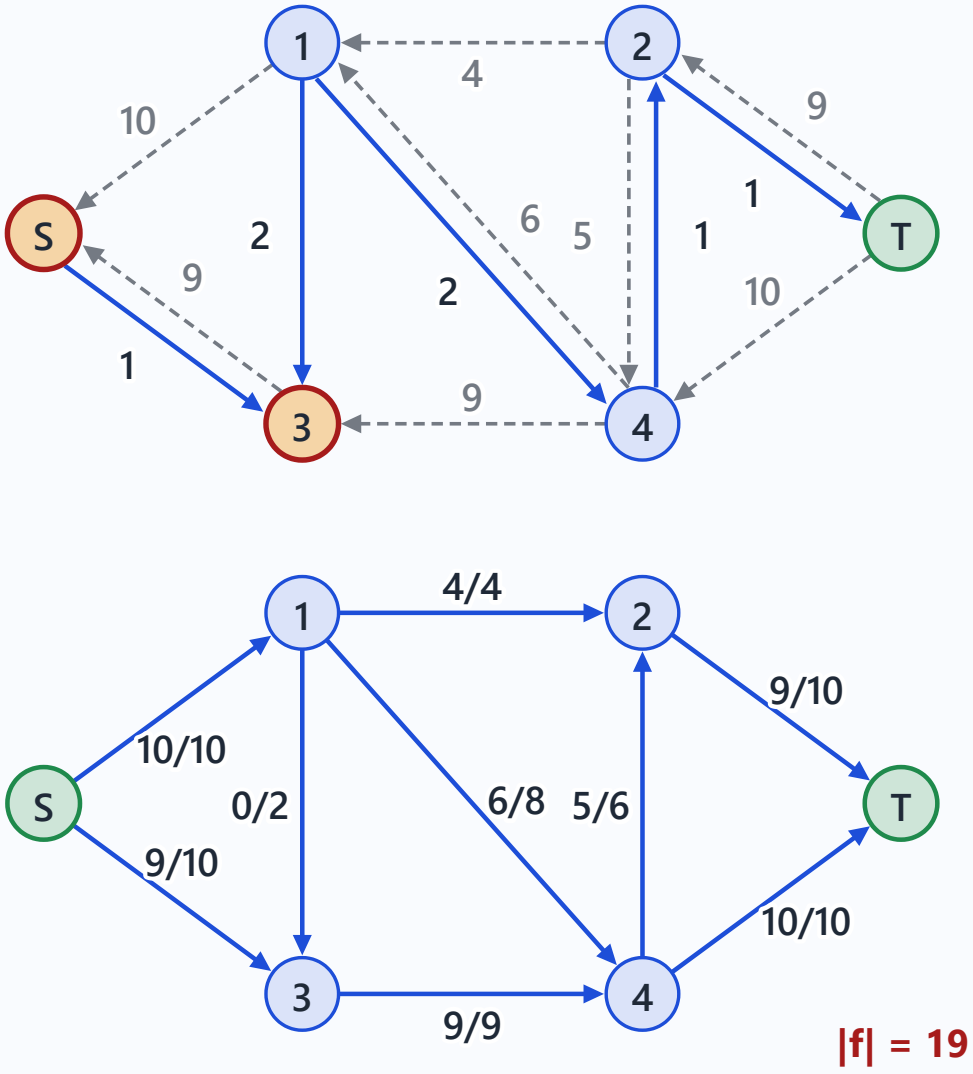
Bu kez yolların sırası bilerek kötü seçilmiştir: ilk yol, sonradan düzeltilmesi gereken bir karar verir.

1. $p = S \to 1 \to 4 \to T$: artık kapasiteler 10, 8, 10; darboğaz $1 \to 4$ (8). $|f| = 8$.
2. $p = S \to 3 \to 4 \to T$: $4 \to T$'de yalnızca $10 - 8 = 2$ birim kalmıştır. Darboğaz 2; $|f| = 10$. $4 \to T$ doyar.
3. $p = S \to 3 \to 4 \to 1 \to 2 \to T$: $3 \to 4$'te $9 - 2 = 7$, $4 \to 1$ ters kenarında 8 (birinci adımda $1 \to 4$'e gönderilen akış), $1 \to 2$'de 4, $2 \to T$'de 10 birim vardır. Darboğaz 4; $|f| = 14$.
4. $p = S \to 1 \to 4 \to 2 \to T$: $S \to 1$'de 2 birim kalmıştı. $1 \to 4$'te artık $8 - 4 = 4$ birim boşluk vardır. Darboğaz 2; $|f| = 16$.
5. $p = S \to 3 \to 4 \to 2 \to T$: darboğaz $3 \to 4$'tür ($9 - 6 = 3$ kaldı). $|f| = 19$.

Üçüncü adım örneğin kalbidir. $4 \to 1$ ters kenarı kullanıldığında $1 \to 4$'teki akış 8'den 4'e düşer. Bunun anlamı şudur: daha önce 1'den 4'e giden 4 birim, artık 4'e 3'ten gelir. 1'de serbest kalan 4 birim ise $1 \to 2$ üzerinden hedefe yönlenir. Hiçbir düğümde korunum bozulmaz. 4'e giren toplam akış ve 1'den çıkan toplam akış değişmez; yalnızca yolları değişir.



Şekil 4.4.11: Üçüncü artış. (a) Artık ağda $S \to 3 \to 4 \to 1 \to 2 \to T$ yolu; $4 \to 1$ bir ters kenardır. (b) Sonraki akış: $1 \to 4$'teki akış 8'den 4'e indi, $1 \to 2$ 4 birimle doldu; $|f| = 14$.



Şekil 4.4.12: (a) Son artık ağ: S'den yalnızca 3'e ulaşılır (vurgulu düğümler). (b) Maksimum akış 19. Min kesim *({S, 3}, diğerleri)*: $S{\to}1$ (10) + $3{\to}4$ (9) = 19.

Son artık ağda $S \to 1$ doymuştur; $S \to 3$'te 1 birim kalmıştır, ama 3'ten çıkan tek kenar olan $3 \to 4$ doymuştur. Kaynaktan ulaşılabilen küme $\{S, 3\}$'tür. Bu kümeden çıkan özgün kenarlar $S \to 1$ ve $3 \to 4$'tür; ikisi de doludur. $1 \to 3$ kenarı T tarafından S tarafına gider ve üzerinde akış yoktur. Bu, ispattaki koşulların tam bir örneğidir: ileri kenarlar dolu, geri kenarlar boş.

4.4.4.5 Ters kenarlar neden gerekli?

Aynı ağda ters kenarları hiç kullanmayan açgözlü bir yöntem düşünelim. Bu yöntem yalnızca dolu olmayan ileri kenarlardan geçen yollar arar.

Örnek 4.4.4

İlk seçim yine $S \to 1 \to 4 \to T$ olsun ve 8 birim gönderilsin. Ardından:

- $S \to 3 \to 4 \to T$: 2 birim ($4 \to T$ doyar). Toplam 10.
- $S \to 1 \to 2 \to T$: $S \to 1$'de 2 birim kaldığı için 2 birim. Toplam 12.
- $S \to 3 \to 4 \to 2 \to T$: $3 \to 4$'te 7, $4 \to 2$'de 6, $2 \to T$'de 8 birim var; 6 birim. Toplam 18.

Şimdi $S \to 1$ doludur. S'den yalnızca 3'e, oradan 4'e gidilebilir. 4'ün iki çıkışı da ($4 \to 2$ ve $4 \to T$) doludur. Yöntem takılır ve 18'de durur. Oysa maksimum akış 19'dur.

Kayıp tek bir birimdir, ama ilke önemlidir. İlk yol $1 \to 4$ kenarına 8 birim yükledi. Bu akışın bir kısmı aslında $1 \to 2$ üzerinden gitmeliydi, böylece $3 \to 4$ kenarına yer açılırdı. İleri kenarlarla çalışan yöntem bu kararı geri alamaz. Ters kenarlı artık ağda ise $4 \to 1$ kenarı bu kararın bir kısmını geri almayı sağlar: 1'den gelen akış 2'ye, 3'ten gelen akış 4'e yönlenir. Sonuç 19'dur ve min kesime eşittir. Bunun genel bir kuralı vardır: ters kenarlı artık ağda yollar hangi sırayla seçilirse seçilsin, yöntem maksimuma ulaşır.

Sık yapılan hata

Ters kenarları unutmak. Yalnızca ileri kenarlarla aramak açgözlü bir algoritmadır ve maksimumu kaçırabilir (örnekte 18 < 19). Kodda ters kenarlar çoğunlukla tek bir satırdır: `artik[v][u] += akis`. Bu satır silinirse program yine çalışır ve bir sayı döndürür; sorun ancak sonuç doğrulandığında fark edilir.

4.4.4.6 Sözde kod

Yöntemin sözde kodu aşağıdadır. Burada akış değerleri ayrı bir `akış` tablosunda tutulur ve artık kapasite gerektiğinde `kapasite - akış` ile hesaplanır.

```
FORD_FULKERSON(G, kaynak, hedef):
  akış[u, v] = 0, her bir (u, v) için E içinde     // başlangıçta 0
  maksimumAkış = 0
  döngü artırıcı yol var iken:
    yol = DFS(G, kaynak, hedef, akış)            // c − akış > 0 kenarlarla
    delta = ∞                                    // yoldaki en küçük kalan kapasite
    v = hedef
    döngü v ≠ kaynak iken:
      u = yol.ata[v]
      delta = min(delta, kapasite[u, v] − akış[u, v])
      v = u
    v = hedef                                    // yoldaki akışları güncelle
    döngü v ≠ kaynak iken:
      u = yol.ata[v]
      akış[u, v] += delta
      akış[v, u] −= delta                        // ters kenar
      v = u
    maksimumAkış += delta
  döndür maksimumAkış, akış
```

Dış döngü bir artış yolu bulunduğu sürece döner. `DFS` çağrısı artık kapasitesi pozitif kenarlarla bir yol arar ve yolu, her düğümün bir önceki düğümünü tutan ebeveyn dizisiyle (kodda `ata[]`) bildirir. Yolun kendisi bir liste olarak saklanmaz; hedeften kaynağa doğru `ata` izlenerek geri okunur.

İlk iç döngü darboğazı bulur. `v` hedeften başlar; her adımda `u = yol.ata[v]`, yani `v`'ye gelinen düğümdür. `delta`, o ana kadar görülen en küçük artık kapasiteyle sınırlanır. Döngü kaynağa ulaşınca biter. İkinci iç döngü aynı yolu bir kez daha yürür ve akışı günceller: ileri yönde `delta` eklenir, ters yönde çıkarılır. Ters yöndeki çıkarma, akışı ters simetrik bir tabloda tutmanın sonucudur: `akış[v, u] = −akış[u, v]`. Bu gösterimde ters kenarın artık kapasitesi `kapasite[v, u] − akış[v, u] = 0 − (−akış[u, v]) = akış[u, v]` olur; tanımdaki değerin aynısı.

Bu ters simetrik gösterim kodu sadeleştirir. Ayrı bir “ters kenar mı?” sorusu sormaya gerek kalmaz; ileri ve ters kenarlar aynı formülle işlenir. Bir ayrıntı vardır: iki düğüm arasında iki yönde de özgün kenar varsa (örneğin hem $1 \to 2$ hem $2 \to 1$), iki kenarın akışları aynı hücrede birbirini götürebilir. Matris gösteriminde bu durum sorun çıkarmaz, çünkü yalnızca net akış önemlidir.

4.4.4.7 Java: DFS ile artış yolu

Sunumdaki Java uygulaması artık kapasiteleri `artik[u][v]` matrisinde tutar. Başlangıçta bu matris kapasitelerle doldurulur; ters kenarların artık kapasitesi sıfırdır. Artış yolunu bulan DFS şöyledir:

```
boolean dfs(int u, int hedef,
            boolean[] ziyaret, int[] ata) {
  ziyaret[u] = true;
  if (u == hedef) return true;
  for (int v = 0; v < V; v++) {
    if (!ziyaret[v] && artik[u][v] > 0) {
      ata[v] = u;
      if (dfs(v, hedef, ziyaret, ata))
        return true;
    }
  }
  return false;   // bu daldan hedefe yol yok
}
```

Metot dört parametre alır: şu anki düğüm `u`, hedef, ziyaret işaretleri ve yolu kaydeden `ata` dizisi. İlk satır `u`'yu ziyaret edilmiş olarak işaretler. Böylece aynı düğüme bir daha girilmez ve arama döngüye düşmez. `u` hedefse yol bulunmuştur ve `true` döner.

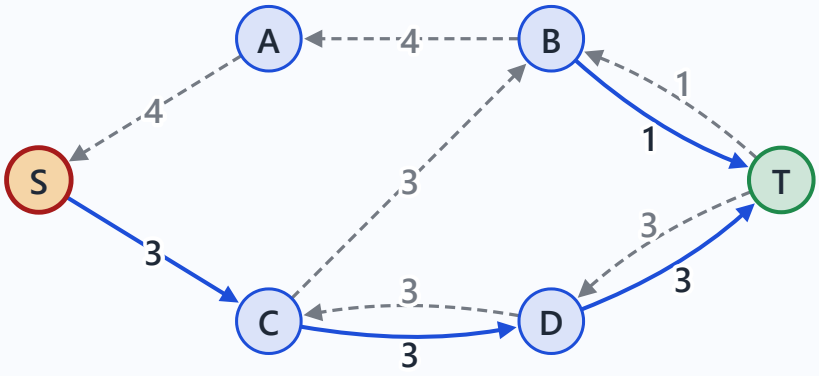
Döngü bütün düğümleri sırayla dener; komşuluk matrisinde "komşu" demek, `artik[u][v] > 0` olan `v` demektir. Ziyaret edilmemiş ve artık kapasitesi pozitif bir komşu bulununca `ata[v] = u` yazılır ve arama `v`'den sürer. Özyinelemeli çağrı `true` dönerse bu `true` hemen yukarı iletilir; döngünün geri kalanı denenmez. Bütün komşular denenip hedefe varılamazsa metot `false` döner: bu daldan hedefe yol yoktur.

Ziyaret işaretleri geri dönüşte silinmez. Bu önemlidir. Bir düğümden hedefe yol yoksa, o düğüme başka bir yoldan gelmek de sonucu değiştirmez; düğümü bir daha denemek boşa iştir. Bu sayede her düğüm en fazla bir kez ziyaret edilir. Komşuluk matrisinde her ziyaret V hücreyi tarar; bir DFS $O(V^2)$ sürer. Komşuluk listesiyle aynı arama $O(V + E)$ sürer.

Akış eklenirken `artik[u][v] -= d` ve `artik[v][u] += d` yapılır. Ters kenar ayrıca oluşturulmaz; ikinci satır onu kendiliğinden doğurur. DFS bir sonraki çağrıda bu hücreyi pozitif görür ve ters kenarı sıradan bir kenar gibi kullanır. Kodun ters kenarları "bildiği" tek yer bu atamadır.

4.4.4.8 Kod izleme: ölü uç ve ters kenar

Kodu Şekil 4.4.1'deki ağda çalıştıralım. Düğüm sırası *S, A, B, C, D, T* (indisler 0'dan 5'e) olsun; DFS komşuları bu sırayla dener. Bu sıra Örnek 1'deki yollardan farklı yollar üretir. İlk DFS $S \to A \to B \to C \to D \to T$ yolunu bulur (darboğaz 3). İkinci DFS'te $B \to C$ doymuştur ve $S \to A \to B \to T$ yolu 1 birim taşır. İki artıştan sonra $|f| = 4$'tür. Üçüncü aramayı ayrıntılı izleyelim.

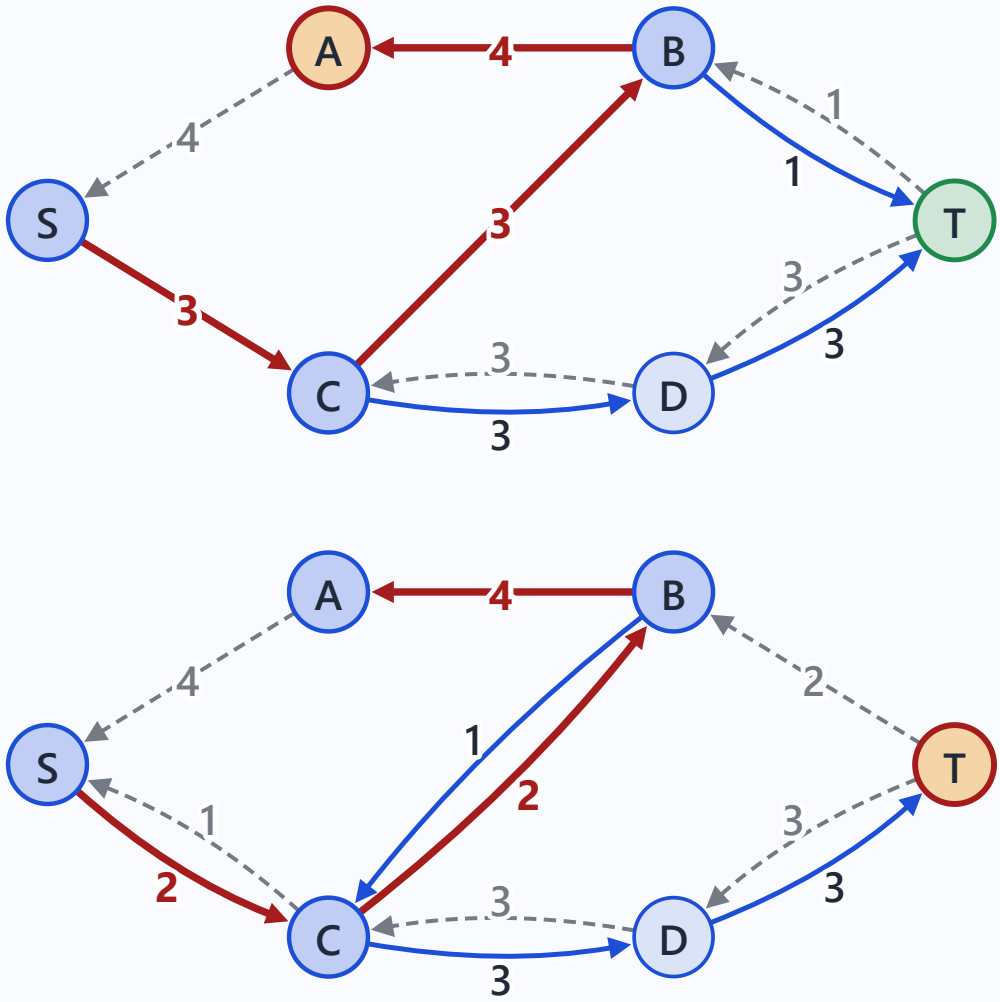


Şekil 4.4.13: İki artıştan sonraki artık ağ ($|f| = 4$). Kesikli oklar ters kenarlardır. $S \to A$ ve $A \to B$ doymuştur; $C \to B$ ters kenarı 3, $B \to A$ ters kenarı 4 birimdir.

Adım	Çağrı	Olay	`ata`
1	`dfs(S)`	$S \to A$: artık 0, atlanır; $S \to B$: kenar yok. $S \to C$: artık 3 > 0, C ziyaret edilmemiş.	`ata[C] = S`
2	`dfs(C)`	$C \to S$: S ziyaretli; $C \to A$: 0. $C \to B$: artık 3 > 0 (ters kenar).	`ata[B] = C`
3	`dfs(B)`	$B \to S$: 0. $B \to A$: artık 4 > 0 (ters kenar).	`ata[A] = B`
4	`dfs(A)`	$A \to S$: artık 4, ama S ziyaretli. Diğerleri 0. Ölü uç: `false`.	—
5	`dfs(B)` sürer	$B \to C$: C ziyaretli; $B \to D$: kenar yok. $B \to T$: artık 1 > 0.	`ata[T] = B`
6	`dfs(T)`	$u = T$ hedeftir: `true`. Her çağrı `true` 'yu yukarı iletir.	—

Tablo 4.4.4: Üçüncü artış yolu için `dfs(S, T, ziyaret, ata)` çağrılarının izi.

Dördüncü adımdaki ölü uç DFS'in karakteristik davranışıdır. A'ya bir ters kenarla gelinmiştir, ama A'dan hedefe giden bir artık yol yoktur. A'nın tek pozitif artık kenarı kaynağa geri döner, kaynak da zaten ziyaret edilmiştir. Arama A'yı terk eder ve B'de kaldığı yerden devam eder. `ata[A] = B` kaydı silinmez, ama zararsızdır: yol hedeften geriye okunurken A'ya hiç uğranmaz.



Şekil 4.4.14: (a) *A*'da ölü uç: $S \to C \to B \to A$ ile inildi, *A*'dan ilerlenemiyor. (b) Bulunan yol $S \to C \to B \to T$ ile 1 birim eklendikten sonraki artık ağ; $|f| = 5$.

Yol hedeften geriye okunur: `ata[T] = B`, `ata[B] = C`, `ata[C] = S`. Yol $S \to C \to B \to T$'dir. Darboğaz $min(3, 3, 1) = 1$'dir. Bu artış $C \to B$ ters kenarını kullanır: $B \to C$'deki akış 3'ten 2'ye iner. Anlamı şudur: *B*'den *C*'ye giden akışın bir birimi artık $B \to T$ üzerinden doğrudan hedefe gider; *C*'nin eksilen girişini kaynaktan gelen yeni birim karşılar.

Dördüncü DFS $S \to C \to D \to T$ yolunu bulur ve 2 birim ekler. $|f| = 7$ olur. Beşinci DFS kaynaktan hiçbir yere gidemez ve `false` döner. Bu sürüm 4 artış yapar, Örnek 1'deki sıra ise 3 artış yapmıştı. İki sürümün son akışı aynıdır: Şekil 4.4.2'deki akış.

4.4.4.9 Karmaşıklık ve tam sayılılık

Ford-Fulkerson'ın çalışma süresi ağın yapısına ve kapasitelere bağlıdır. Her tur iki parçadan oluşur. Bir artış yolu bulmak (DFS ya da BFS) ve akışı güncellemek $O(E)$ sürer; komşuluk listesi varsayılır. Asıl soru tur sayısıdır.

Teorem 4.4.4 (Tam sayılılık)

Bütün kapasiteler tam sayı ise Ford-Fulkerson yönteminin her adımında akış ve artık kapasiteler tam sayıdır. Yöntem en fazla $|f^*|$ artış yapar ve $O(E \cdot |f^*|)$ sürede biter. Bulunan maksimum akış tam sayı değerlidir.

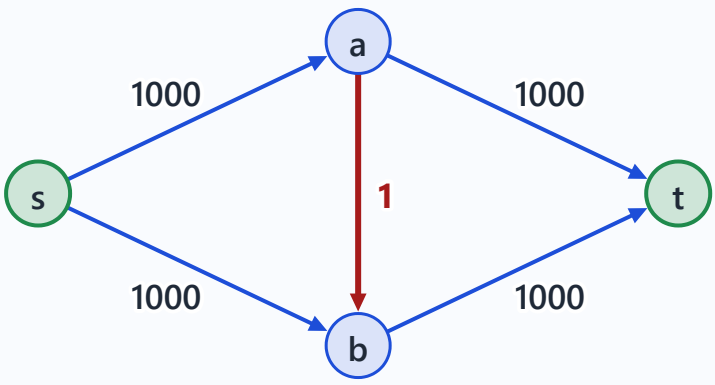
İspat. Tümevarımla: başlangıçta akış sıfırdır ve artık kapasiteler kapasitelere eşittir; hepsi tam sayıdır. Artık kapasiteler tam sayıysa darboğaz da tam sayıdır ve en az 1'dir. Güncelleme tam sayılara tam sayı ekler ya da çıkarır; tam sayılık korunur. Her artış akışın değerini en az 1 artırır ve değer $|f^*|$'yi aşamaz. Öyleyse en fazla $|f^*|$ artış yapılır; her biri $O(E)$ sürer. □

Tam sayılılık özelliğinin önemli bir uygulaması vardır. Eşleştirme gibi problemlerde her kenar ya kullanılır ya kullanılmaz; yarım kenar anlamsızdır. Kapasiteler 1 olarak kurulursa teorem, maksimum akışın her kenarda 0 ya da 1 olduğunu garanti eder. Böylece akış doğrudan bir seçim olarak okunabilir.

Sınırın bir kusuru vardır: $O(E \cdot |f^*|)$ polinom değildir. Bir algoritmanın süresi girdinin *uzunluğuyla*, yani girdiyi yazmak için gereken bit sayısıyla ölçülür. Kapasite 10^9 yaklaşık 30 bitle yazılır, ama $|f^*|$ bu sayının kendisi kadar büyüyebilir. Bu tür sınırlara **sözde polinom** sınır denir. Kapasiteler küçük tam sayılarsa sorun yoktur. Büyük kapasitelerde ise yöntem, yolu kötü seçerse çok yavaşlayabilir.

Kapasiteler irrasyonel sayılar olabilirse durum daha da kötüdür. Bilinen örneklerde kötü seçilmiş yollarla artışlar giderek küçülür ve yöntem hiç sonlanmaz; hatta akış değeri maksimum akışa da yakınsamaz. Bilgisayarlarda kapasiteler sonlu duyarlıkla saklandığı için bu durum pratikte görülmez. Ama yöntemin “herhangi bir yol” özgürlüğünün bedeli olarak akılda tutulmalıdır.

4.4.4.10 Kötü yol seçimi



Şekil 4.4.15: Kötü durum ağı. Dört dış kenarın kapasitesi 1000, ortadaki $a \to b$ kenarınınki 1'dir. Maksimum akış 2000'dir.

Şekil 4.4.15'teki ağda maksimum akış 2000'dir: $s \to a \to t$ ve $s \to b \to t$ yollarının her biri 1000 birim taşır. İki artış yeterlidir. Şimdi kötü bir yol seçimini düşünelim. İlk yol $s \to a \to b \to t$ olsun. Ortadaki kenar yüzünden darboğaz 1'dir. Bu artış $b \to a$ ters kenarını doğurur. İkinci yol $s \to b \to a \to t$ olsun; ters kenar yüzünden darboğaz yine 1'dir ve $a \to b$ yeniden boşalır. Bu iki yol sırayla seçilmeye devam edilirse her artış yalnızca 1 birim ekler. Maksimuma ulaşmak için 2000 artış gerekir.

Kapasiteler 10^9 olsaydı aynı seçim $2 \cdot 10^9$ artış gerektirirdi. Oysa ağ yalnızca dört düğüm ve beş kenarlıdır. DFS bu kötü seçimi her zaman yapmaz. Örneğin komşuları indis sırasıyla deneyen DFS bu ağda 4 artışla biter: önce $s \to a \to b \to t$ (1), sonra $s \to a \to t$ (999), $s \to b \to a \to t$ (1) ve $s \to b \to t$ (999). Ama DFS'in kötü seçimi yapmasını engelleyen hiçbir kural yoktur. Bu yüzden süre sınırı $O(E \cdot |f^*|)$ olarak kalır.

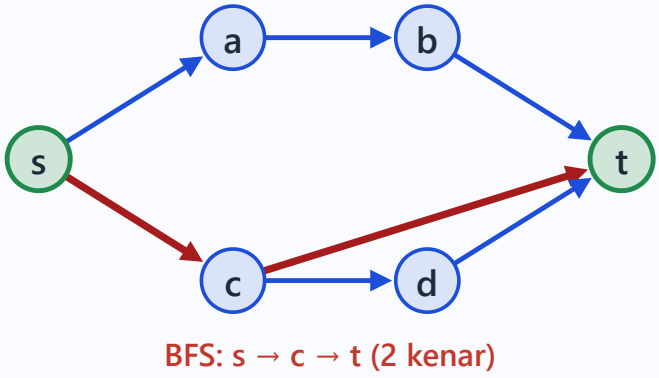
Edmonds-Karp, yani BFS ile çalışan sürüm, en kısa yolları seçer. Bu ağda en kısa yollar iki kenarlıdır: $s \to a \to t$ ve $s \to b \to t$. Her biri 1000 birim taşır; 2 artış yeter. Üç kenarlı ortadaki yol hiç seçilmez. BFS'in kapasitelerin büyüklüğüne bağımlılığı nasıl kaldırdığı bir sonraki kesimin konusudur.

> **Sık yapılan hata**
>
> Ford-Fulkerson'ı polinom zamanlı sanmak. $O(E \cdot |f^*|)$ kapasitelerin değerine bağlıdır; aynı yapıdaki ağda kapasiteler büyüdükçe süre de büyüyebilir. Edmonds-Karp ve Dinic bu bağımlılığı kaldırır.

4.4.5 Edmonds-Karp algoritması

4.4.5.1 Fikir: en kısa artış yolu

Edmonds-Karp algoritması, Ford-Fulkerson yönteminin bir türevidir. Jack Edmonds ve Richard Karp 1972'de yayımlamıştır; Dinic aynı düşünceye 1970'te bağımsız olarak ulaşmıştır. Algoritma yöntemin tek serbest kararını sabitler: artış yolu genişlik öncelikli aramayla bulunur. BFS kaynaktan katman katman ilerlediği için hedefe ilk ulaştığı yol, kenar sayısı en az olan yoldur. Edmonds-Karp her turda bu en kısa artış yolunu seçer.



Şekil 4.4.16: BFS, kenar sayısı en az olan artış yolunu seçer. Burada $s \to c \to t$ iki kenarlıdır; üç kenarlı yollar ($s \to a \to b \to t, s \to c \to d \to t$) o tur seçilmez.

Burada "kısa" kenar sayısı anlamındadır, kapasite ya da ağırlık anlamında değil. Kenarların kapasitesi yol uzunluğunu etkilemez. Bu seçim ilk bakışta keyfî görünebilir. Ama kötü yol seçimi örneğindeki sorunun kökü, uzun ve dar bir yolun tekrar tekrar seçilmesiydi. En kısa yolları seçmek bu tür kısır döngüleri engeller. Algoritmanın adımları şöyledir:

1. BFS ile artık ağda kaynaktan hedefe en kısa artış yolu bulunur.
2. Yol boyunca darboğaz kadar akış gönderilir, ters kenarlar güncellenir.
3. BFS hedefe ulaşamayana kadar bu iki adım tekrarlanır.

BFS'in ağ akışında iki farklı kullanımı olduğunu baştan belirtmek yararlıdır. Edmonds-Karp BFS'i doğrudan artış yolu bulmak için kullanır ve her BFS'ten sonra tek bir yol işler. Dinic ise BFS'i hedefin hâlâ ulaşılabilir olup olmadığını anlamak ve bir düzey çizgesi kurmak için kullanır; o çizgede birçok yolu birden işler.

4.4.5.2 Karmaşıklık: $O(V \cdot E^2)$

Her tur bir BFS ve bir yol güncellemesinden oluşur: *O(E)*. Edmonds-Karp'ın asıl katkısı, artış sayısının kapasitelerden bağımsız bir sınırı olmasıdır. Bu sınırın gerekçesi iki gözleme dayanır.

Artık ağda kaynaktan bir *v* düğümüne olan en kısa yolun kenar sayısına *d(v)* diyelim. **Birinci gözlem:** Edmonds-Karp çalıştıkça hiçbir düğümün *d(v)* değeri azalmaz. Bir artış, yoldaki bazı kenarları artık ağdan çıkarır ve bazı ters kenarları ekler. Eklenen her ters kenar, en kısa yol üzerindeki bir *(u, v)* kenarının tersidir; yani *d(v) = d(u) + 1* olan bir çiftte *v*'den *u*'ya gider. Böyle bir kenar uzaklığı bir düğüm geriye götürür ve hiçbir yolu kısaltamaz.

İkinci gözlem: Bir artışta darboğazı belirleyen kenara **kritik kenar** denir; artıştan sonra artık ağdan çıkar. Her artışta en az bir kenar kritiktir. *(u, v)* kenarı kritik olduğunda *d(v) = d(u) + 1*'dir. Aynı kenarın yeniden artık ağa girmesi için bir artış yolunun *(v, u)* ters kenarını kullanması gerekir. O anda *d'(u) = d'(v) + 1*'dir. Uzaklıklar azalmadığı için *d'(u) = d'(v) + 1 ≥ d(v) + 1 = d(u) + 2* olur. Demek ki *u*'nun uzaklığı, kenar iki kez kritik olana kadar en az 2 artar.

Teorem 4.4.5 (Edmonds-Karp)

Edmonds-Karp algoritması en fazla $O(V \cdot E)$ artış yapar ve $O(V \cdot E^2)$ sürede biter. Süre kapasitelerin değerine bağlı değildir.

İspat. Bir kenar her kritik oluşunda kuyruğundaki düğümün uzaklığı en az 2 artar. Uzaklık 0'dan başlar ve ulaşılabilen bir düğüm için en fazla $V - 1$ olabilir; düğüm ulaşılamaz olursa kenar bir daha kullanılmaz. Öyleyse bir kenar en fazla yaklaşık $V/2$ kez kritik olur. Artık ağda en fazla $2E$ kenar vardır. Her artışta en az bir kenar kritik olduğundan artış sayısı $O(V \cdot E)$'dir. Her artış $O(E)$ sürdüğünden toplam süre $O(V \cdot E) \cdot O(E) = O(V \cdot E^2)$'dir. □

Sınırın anlamı kötü durum ağında açıkça görülür. Kapasiteler 1000 de olsa 10^9 da olsa Edmonds-Karp 2 artışla biter. Süre artık $|f^*|$'ye bağlı değildir; algoritma polinom zamanlıdır. Bedeli, her artışta tam bir BFS yapılmasıdır. Sınır kötümserdir: pratikte artış sayısı $V \cdot E$'nin çok altında kalır. $V = 6$, $E = 9$ olan ağlarda sınır yaklaşık 27 artışa izin verir; bu notun örneklerinde artış sayısı 3 ile 4 arasındadır.

4.4.5.3 Java: BFS ile artış yolu

Edmonds-Karp'ın Ford-Fulkerson'dan tek farkı yol arayan fonksiyondur; ana döngü aynıdır. BFS sürümü şöyledir:

```
boolean bfs(int kaynak, int hedef, int[] ata) {
  boolean[] ziyaret = new boolean[V];
  Queue<Integer> kuyruk = new LinkedList<>();
  kuyruk.add(kaynak);
  ziyaret[kaynak] = true;
  ata[kaynak] = -1;
  while (!kuyruk.isEmpty()) {
    int u = kuyruk.poll();
    for (int v = 0; v < V; v++)
      if (!ziyaret[v] && artik[u][v] > 0) {
        kuyruk.add(v);
        ata[v] = u;
        ziyaret[v] = true;
      }
  }
  return ziyaret[hedef];
}
```

Metot her çağrıda yeni bir `ziyaret` dizisi oluşturur. Kuyruk kaynakla başlar; kaynak ziyaret edilmiş sayılır ve `ata[kaynak] = -1` yazılır. Bu −1 değeri, yolu geri okurken kaynağa varıldığını belirten bir işarettir. Döngüde kuyruğun başındaki düğüm `u` çıkarılır. Ardından bütün `v`'ler sırayla denenir. Ziyaret edilmemiş ve `artik[u][v] > 0` olan her `v` kuyruğa eklenir, ebeveyni `u` olarak kaydedilir ve hemen ziyaret edilmiş işaretlenir.

İşaretin kuyruğa eklerken konması önemlidir. Düğüm kuyruktan çıkarken işaretlenseydi, aynı düğüm birkaç farklı ebeveynden kuyruğa girebilirdi. `ata` değeri de son ekleyen ebeveyne göre değişirdi ve BFS ağacı bozulurdu. Erken işaretleme, her düğümün tam bir kez ve kaynağa en yakın ebeveyninden kuyruğa girmesini sağlar.

Metot `ziyaret[hedef]` değerini döndürür: hedefe ulaşıldıysa bir artış yolu vardır. `ata` dizisi BFS ağacını tutar; hedeften kaynağa doğru izlenince en az kenarlı yol elde edilir. Bu kod hedefe ulaştığında durmaz, kuyruk boşalana kadar sürer. Sonuç değişmez, çünkü hedefin ebeveyni ilk ulaşıldığında yazılmıştır; erken durmak yalnızca biraz zaman kazandırır. Komşuluk matrisiyle bir BFS $O(V^2)$, komşuluk listesiyle $O(V + E)$ sürer.

Ana döngü şöyledir:

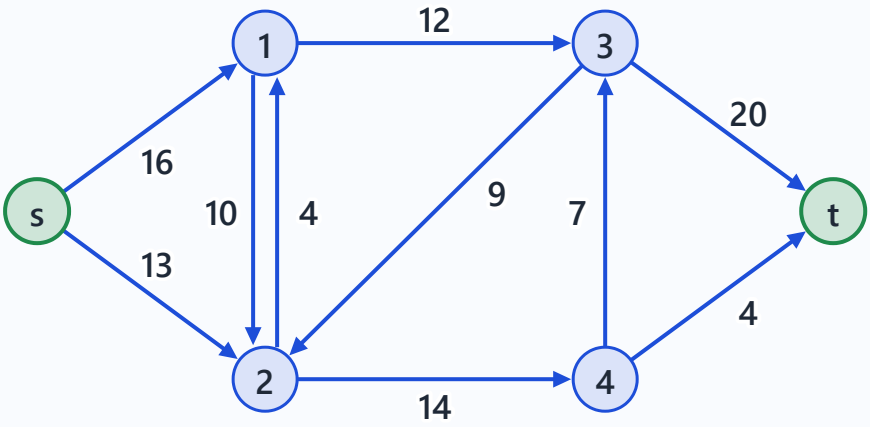
```
int edmondsKarp(int s, int t) {
  int maksAkis = 0;
  int[] ata = new int[V];
  while (bfs(s, t, ata)) {
    int akis = Integer.MAX_VALUE;
    for (int v = t; v != s; v = ata[v])
      akis = Math.min(akis, artik[ata[v]][v]);
    for (int v = t; v != s; v = ata[v]) {
      int u = ata[v];
      artik[u][v] -= akis;  // ileri
      artik[v][u] += akis;  // ters
    }
    maksAkis += akis;
  }
  return maksAkis;
}
```

`while (bfs(s, t, ata))` koşulu her turda yeni bir BFS çalıştırır. İlk `for` döngüsü hedeften kaynağa doğru yürür. `ata[v]` yolun bir önceki düğümüdür; `artik[ata[v]][v]` o kenarın artık kapasitesidir. `akis` bu değerlerin en küçüğüne iner. Başlangıç değeri `Integer.MAX_VALUE` 'dir, böylece ilk karşılaştırma her zaman gerçek bir kenarın değerini alır. İkinci döngü aynı yolu bir kez daha yürür: ileri artık kapasiteden `akis` çıkarılır, ters artık kapasiteye `akis` eklenir. Tur sonunda `maksAkis` büyür. BFS hedefe ulaşamadığında döngü biter ve sonuç döndürülür.

Bellek kullanımı `artik` matrisi için $O(V^2)$, `ata`, `ziyaret` ve kuyruk için $O(V)$'dir. Matris seyrek ağlarda savurgandır. Büyük seyrek ağlarda komşuluk listesi ve her kenarın ters kenarına bir bağlantı tutulur; Dinic'in Java kodu bu gösterimi kullanır.

4.4.5.4 Kod izleme: bfs(s, t, ata)

BFS'i Şekil 4.4.17'deki ağda izleyelim. Bu ağ Cormen ve arkadaşlarının kitabındaki klasik örnektir. Düğümler *s, 1, 2, 3, 4, t* sırasıyla 0'dan 5'e numaralıdır; döngü komşuları bu sırayla dener. Başlangıçta bütün akışlar sıfırdır, artık kapasite kapasiteye eşittir.

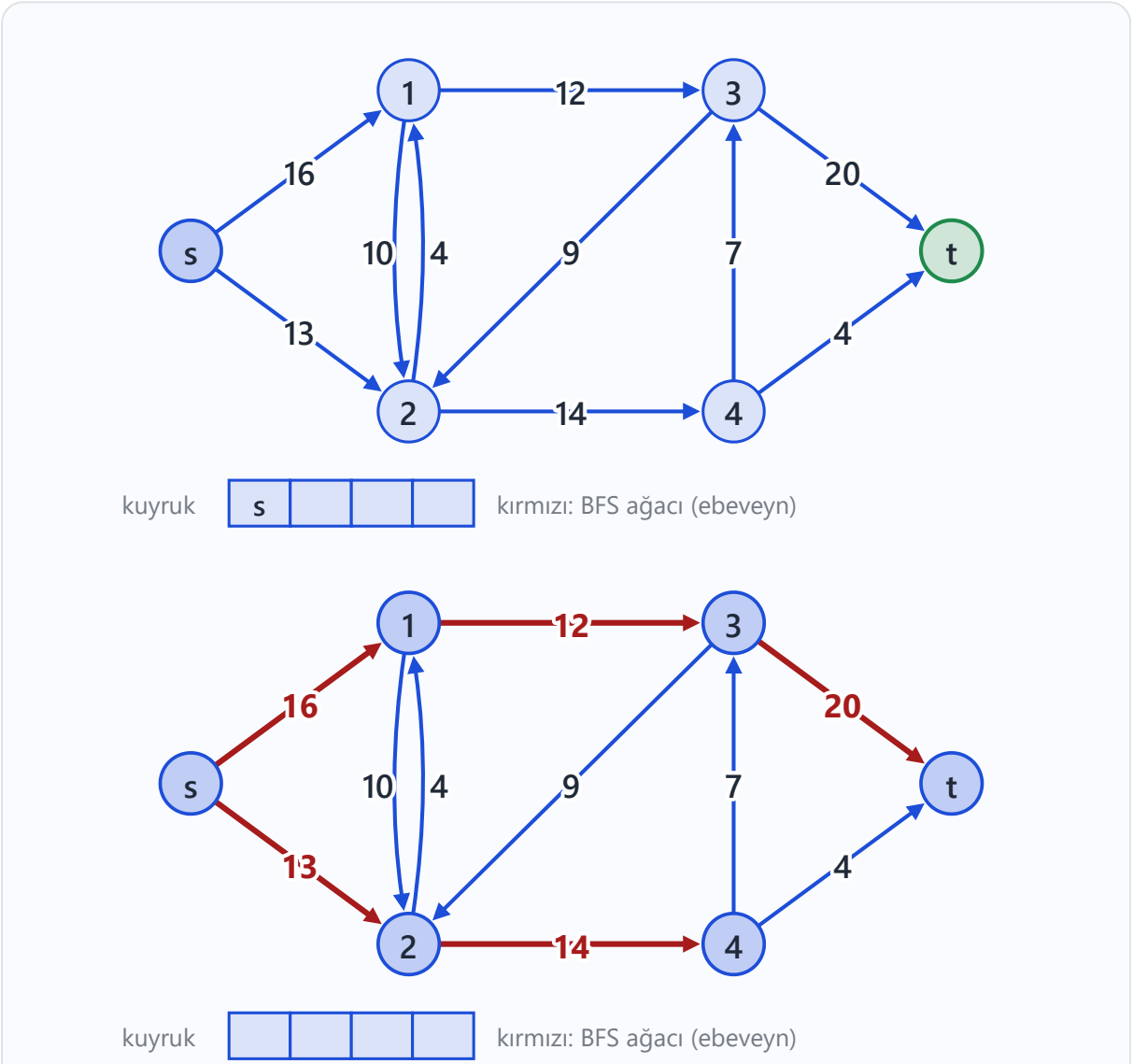


Şekil 4.4.17: İzleme ağı. Kapasiteler: *s→1* 16, *s→2* 13, *1→2* 10, *2→1* 4, *1→3* 12, *3→2* 9, *2→4* 14, *4→3* 7, *3→t* 20, *4→t* 4.

Çıkan `u`	Eklenen komşular (artık)	Atlananlar	Kuyruk
—	başlangıç: `ziyaret[s] = true`, `ata[s] = -1`	—	[s]
s	1 (16), 2 (13); ebeveyn s	—	[1, 2]
1	3 (12); ebeveyn 1	2 ziyaretli	[2, 3]
2	4 (14); ebeveyn 2	1 ziyaretli	[3, 4]
3	t (20); ebeveyn 3	2 ziyaretli	[4, t]
4	—	3 ve t ziyaretli	[t]
t	—	pozitif artık çıkış kenarı yok	[ ]

Tablo 4.4.5: İlk `bfs(s, t, ata)` çağrısı. Her satır bir düğümün kuyruktan çıkışıdır.

Hedef, 3 kuyruktan çıktığında ziyaret edilir. Kod yine de kuyruk boşalana kadar sürer. Döngü bittiğinde `ata` dizisi *[−1, s, s, 1, 2, 3]*'tür. Hedeften geriye okununca yol *s → 1 → 3 → t* çıkar: yalnızca 3 kenar. Bu ağda 3'ten kısa bir s–t yolu yoktur; BFS en az kenarlı artış yolunu bulmuştur. Darboğaz *min(16, 12, 20) = 12*'dir.



Şekil 4.4.18: BFS'in başı ve sonu. (a) Kuyrukta yalnızca s var. (b) Kuyruk boş; kırmızı kenarlar BFS ağacıdır (`ata` ilişkisi). Hedefe giden dal $s \to 1 \to 3 \to t$'dir.

4.4.5.5 Kod izleme: edmondsKarp(s, t)

Ana döngünün tamamı Tablo 4.4.6'da görülür. Her tur için bulunan yol, darboğaz ve güncellenen artık kapasiteler verilmiştir.

Tur	BFS yolu	Artık kapasiteler	`akis`	Güncellenen hücreler	`maksAkis`
1	s→1→3→t (3 kenar)	16, 12, 20	12	s→1: 4, 1→3: 0, 3→t: 8; ters: 1→s 12, 3→1 12, t→3 12	12
2	s→2→4→t (3 kenar)	13, 14, 4	4	s→2: 9, 2→4: 10, 4→t: 0; ters: 2→s 4, 4→2 4, t→4 4	16
3	s→2→4→3→t (4 kenar)	9, 10, 7, 8	7	s→2: 2, 2→4: 3, 4→3: 0, 3→t: 1; ters artar	23

Tur	BFS yolu	Artık kapasiteler	`akis`	Güncellenen hücreler	`maksAkis`
4	yok	—	—	BFS t'ye ulaşamaz	23

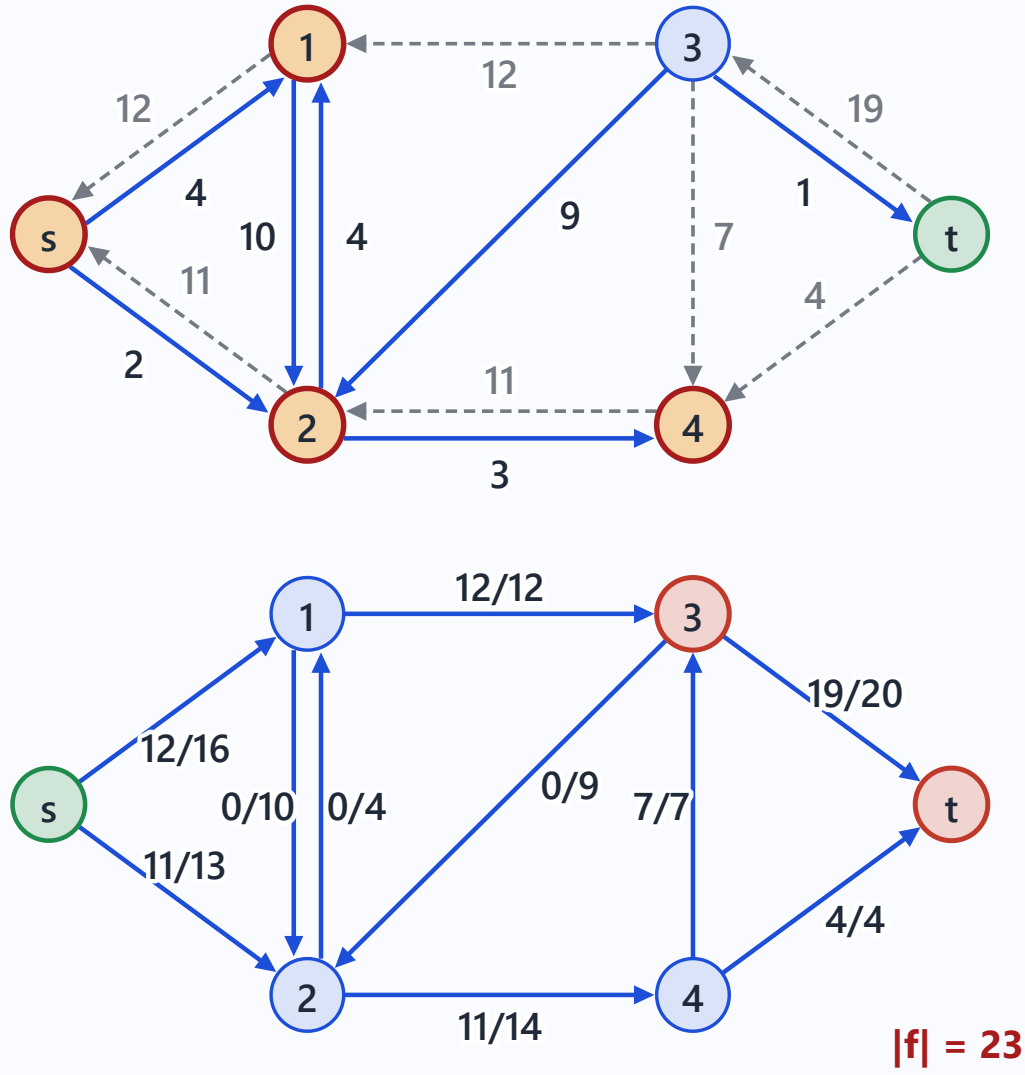
Tablo 4.4.6: `edmondsKarp(s, t)` izlemesi.

Yol uzunlukları 3, 3 ve 4'tür; azalmaz. Bu, Edmonds-Karp'ın karmaşıklık ispatındaki birinci gözlemin küçük bir örneğidir. İkinci turdan sonra $4 \to t$ doymuştur. Üçüncü turda 4'ten hedefe doğrudan gidilemez; BFS 4'ten 3'e geçen ve bir kenar daha uzun olan yolu bulur. Darboğaz $4 \to 3$'ün kapasitesi olan 7'dir.

Dördüncü BFS'te kaynaktan 1'e (artık 4) ve 2'ye (artık 2) gidilir. 1'den 3'e giden kenar doymuştur. 2'den 4'e 3 birim gidilebilir, ama 4'ün iki çıkışı ($4 \to 3$ ve $4 \to t$) doludur. 3'e ve hedefe ulaşılamaz; ulaşılabilen küme $\{s, 1, 2, 4\}$'tür. Metot 23 döndürür. Algoritma 3 artış ve 4 BFS yapmıştır.

4.4.5.6 Örnek: son akış ve minimum kesim

Aynı ağda akışın neden 23'te durduğunu kesimlerle doğrulayalım. Algoritma başlamadan önce bile basit bir üst sınır vardır. Kaynaktan çıkan kapasite $16 + 13 = 29$, hedefe giren kapasite $20 + 4 = 24$'tür. Küçüğü alınırsa $|f^*| \le 24$. Bulunan değer 23'tür; bu sınırın bir altında. Darboğaz ne kaynakta ne hedeftedir; ağın ortasındadır.



Şekil 4.4.19: (a) Son artık ağ; kaynaktan ulaşılan düğümler *s, 1, 2, 4* vurgulu. (b) Son akış ve minimum kesim: *T* tarafı *{3, t}* (kırmızı). Kesim kapasitesi *12 + 7 + 4 = 23*.

Örnek 4.4.5

Son artık ağda kaynaktan ulaşılabilen küme $S = \{s, 1, 2, 4\}$, kalan küme $T = \{3, t\}$'dir. S'den T'ye giden özgün kenarlar:

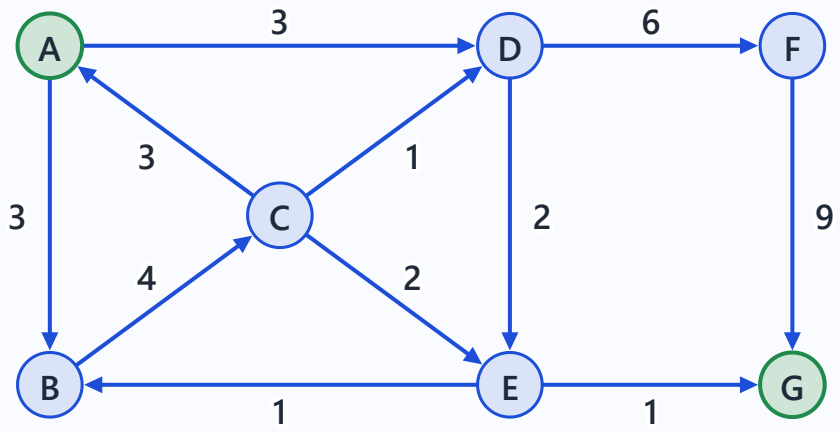
- $1 \to 3$: kapasite 12, akış 12 (dolu).
- $4 \to 3$: kapasite 7, akış 7 (dolu).
- $4 \to t$: kapasite 4, akış 4 (dolu).

T'den S'ye giden tek kenar $3 \to 2$'dir ve akışı 0'dır. Kesim kapasitesi $12 + 7 + 4 = 23$'tür ve akışın değerine eşittir. Hedef tarafından sağlama: t'ye giren akış $19 + 4 = 23$. Akış maksimumdur. Kesim kenarlarından birinin kapasitesi artırılmadıkça akış büyütülemez.

Son akışın kenarlara dağılımına bakmak da öğreticidir. $1 \to 2$ ve $2 \to 1$ kenarlarında akış yoktur; bu iki kenar bu çözümde hiç kullanılmamıştır. $s \to 1$'de 12, $s \to 2$'de 11 birim vardır. Kaynak kenarlarında boş kapasite kalmasına rağmen akış artırılamaz, çünkü kesim kenarları doludur.

4.4.5.7 Örnek 2: A–G ağı

İkinci örnek, BFS'in uzun yollar bulmak zorunda kaldığı ve ters kenar kullandığı bir ağdır. Kaynak A, hedef G'dir. BFS komşuları alfabetik sırayla dener.



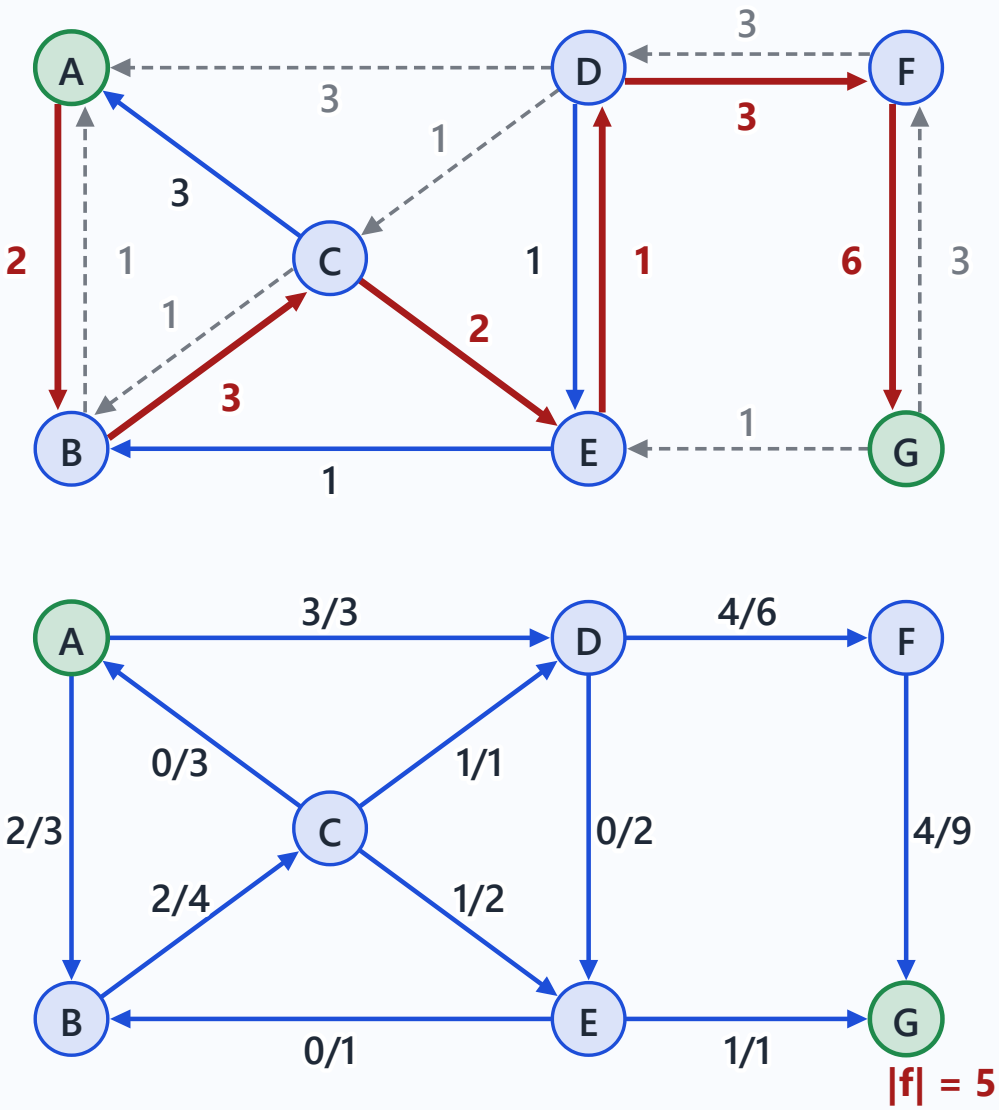
Şekil 4.4.20: A–G ağı. Kapasiteler: $A \to B$ 3, $A \to D$ 3, $B \to C$ 4, $C \to A$ 3, $C \to D$ 1, $C \to E$ 2, $D \to E$ 2, $D \to F$ 6, $E \to B$ 1, $E \to G$ 1, $F \to G$ 9.

Tur	Yol	Kenar	Darboğaz	Akış
1	A→D→E→G	3	E→G (1)	1
2	A→D→F→G	3	A→D (kalan 2)	3
3	A→B→C→D→F→G	5	C→D (1)	4
4	A→B→C→E→D→F→G	6	E→D ters kenarı (1)	5

Tablo 4.4.7: A–G ağında Edmonds-Karp'ın bulduğu artış yolları.

İlk turda BFS A'dan B ve D'yi, B'den C'yi, D'den E ve F'yi kuyruğa koyar. E, F'den önce kuyruktan çıkar ve G'yi ilk o bulur. Yol $A \to D \to E \to G$'dir; $E \to G$'nin kapasitesi 1 olduğu için yalnızca 1 birim taşır. İkinci turda $E \to G$ doludur. BFS $A \to D \to F \to G$'yi bulur; $A \to D$'de 2 birim kalmıştır. Akış 3 olur ve $A \to D$ doyar.

Üçüncü turda kaynaktan yalnızca B'ye çıkılabilir. Yol beş kenara uzar: $A \to B \to C \to D \to F \to G$. Darboğaz $C \to D$'dir (1). Dördüncü tur en ilginç olanıdır. $C \to D$ doymuştur; C'den E'ye gidilir. E'den hedefe doğrudan yol yoktur, ama ilk turda $D \to E$'ye 1 birim gönderilmişti. Artık ağda $E \to D$ ters kenarı vardır ve yol onu kullanır: $A \to B \to C \to E \to D \to F \to G$.



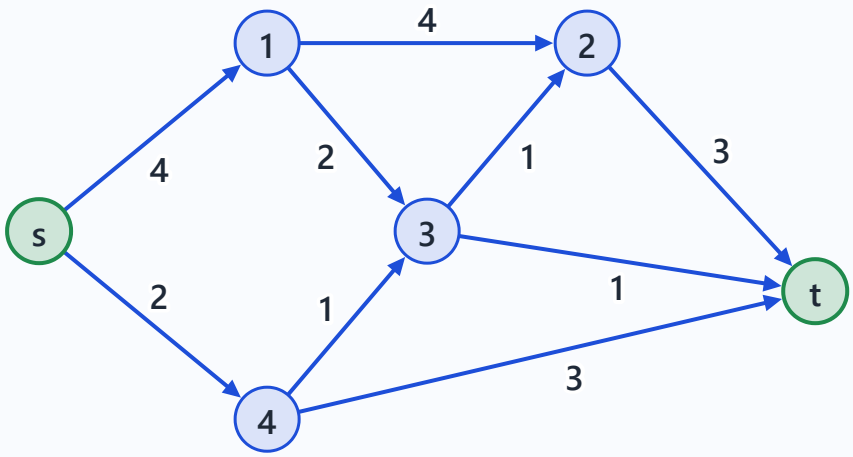
Şekil 4.4.21: (a) Dördüncü turun artış yolu; $E \to D$ bir ters kenardır. (b) Son akış: $|f| = 5$. $D \to E$'deki akış 1'den 0'a inmiştir.

Ters kenarın etkisi şudur: ilk turda D'den E'ye geçip $E \to G$ ile hedefe giden birim artık C'den gelir. D'de serbest kalan birim $D \to F \to G$ üzerinden hedefe gider. Son akışta $D \to E$ kenarı boştur. Yol uzunlukları 3, 3, 5 ve 6'dır; yine azalmaz.

Beşinci BFS hedefe ulaşamaz. Kaynaktan A, B, C, E düğümlerine ulaşılır. Bu kümeden çıkan özgün kenarlar $A \to D$ (3), $C \to D$ (1) ve $E \to G$ (1)'dir. Kesim kapasitesi $3 + 1 + 1 = 5 = |f^*|$'dır. $D \to E$ kenarı kesimi ters yönde geçer ve son akışta boştur; ispattaki koşul yine sağlanır.

4.4.5.8 Örnek 3: artık ağ üzerinde

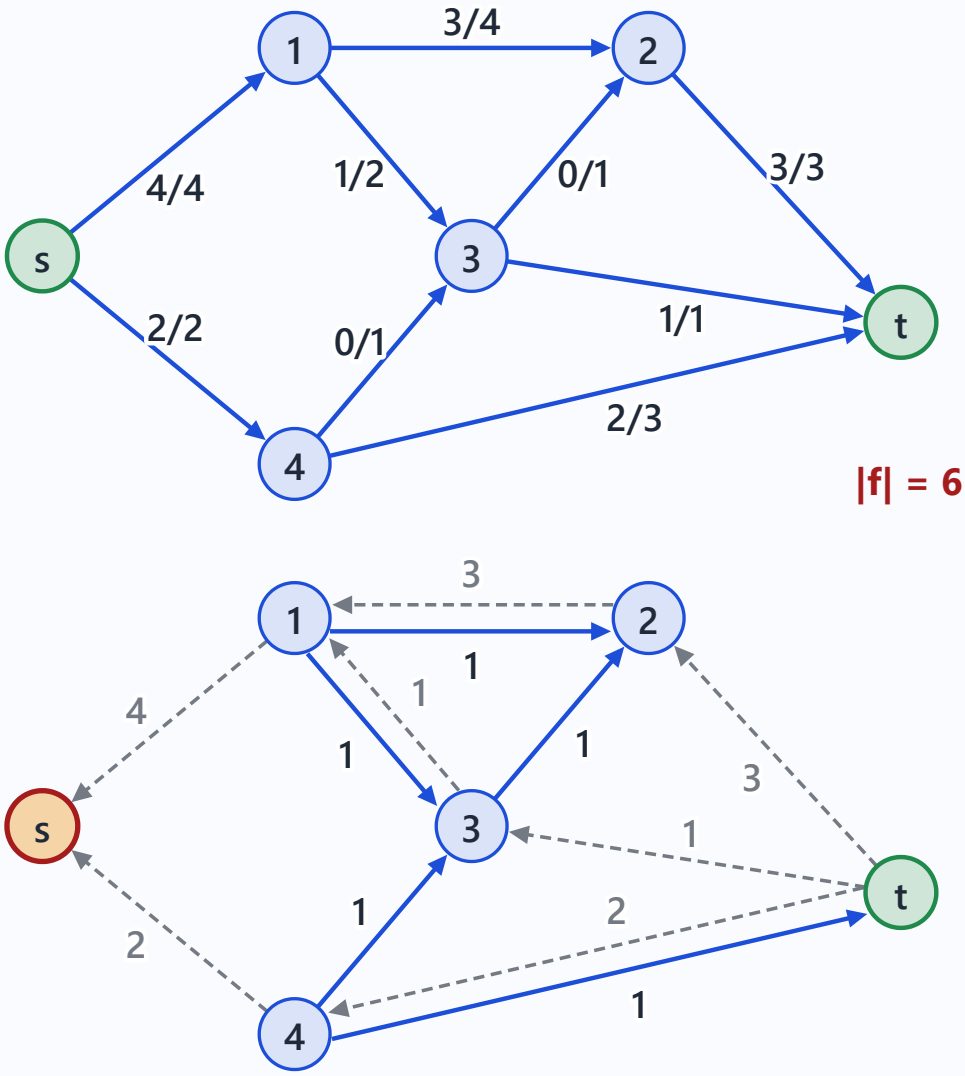
Üçüncü örnekte algoritma doğrudan artık ağ üzerinde izlenir. Her artışın artık ağı nasıl değiştirdiği adım adım görülür.



Şekil 4.4.22: Üçüncü örnek ağ. Kapasiteler: *s*→*1* 4, *s*→*4* 2, *1*→*2* 4, *1*→*3* 2, *3*→*2* 1, *2*→*t* 3, *3*→*t* 1, *4*→*3* 1, *4*→*t* 3.

Başlangıçta artık ağ kapasitelerle aynıdır. BFS önce iki kenarlı $s \to 4 \to t$ yolunu bulur. Darboğaz $s \to 4$'tür (2); 2 birim itilir. $s \to 4$'ün artık kapasitesi 0'a iner, $4 \to s$ ters kenarı 2 olur. $4 \to t$'de 1 birim kalır.

Artık iki kenarlı yol yoktur. BFS üç kenarlı $s \to 1 \to 2 \to t$ yolunu bulur; darboğaz $2 \to t$'dir (3). 3 birim itilir. $s \to 1$'de 1, $1 \to 2$'de 1 birim kalır; $2 \to t$ doyar. Sonraki BFS'te 2'den hedefe gidilemez. $s \to 1 \to 3 \to t$ yolu bulunur; $s \to 1$'de 1 birim kaldığı için 1 birim itilir.



Şekil 4.4.23: (a) Son akış (akış/kapasite). (b) Son artık ağ: *s*'ten çıkan iki kenar da doymuştur, kaynaktan hiçbir yere gidilemez.

Kaynaktan çıkan kenarlar doymuştur; artış yolu kalmamıştır. Maksimum akış *|f*| = 2 + 3 + 1 = 6*'dır: 3 birim *s → 1 → 2 → t*, 1 birim *s → 1 → 3 → t*, 2 birim *s → 4 → t* üzerinden gider. Minimum kesim *({s}, diğerleri)*'dir ve kapasitesi *4 + 2 = 6*'dır. Bu ağda *{s, 1, 2, 3}* kümesi de 6 kapasiteli bir kesim verir; minimum kesim yine tek değildir.

Bu örnekte hiçbir yol ters kenar kullanmamıştır. Bazı kenarlar da hiç dolmamıştır: *3 → 2* ve *4 → 3* boş kalır, *4 → t*'de 1 birim boşluk vardır. Maksimum akış her kenarı doldurmaz; yalnızca bir minimum kesimin kenarlarını doldurur.

4.4.6 Dinic algoritması

Edmonds-Karp her BFS'ten sonra tek bir yol işler. Oysa BFS'in bulduğu bilgi tek bir yoldan çok daha zengindir: kaynaktan her düğüme olan uzaklığı verir. Bu uzaklıklar kullanılarak aynı uzunluktaki bütün en kısa yollar tek bir aşamada doyurulabilir. Yefim Dinitz 1970'te yayımladığı algoritmada bu fikri kullanmıştır; algoritma Batı literatüründe Dinic adıyla tanınır.

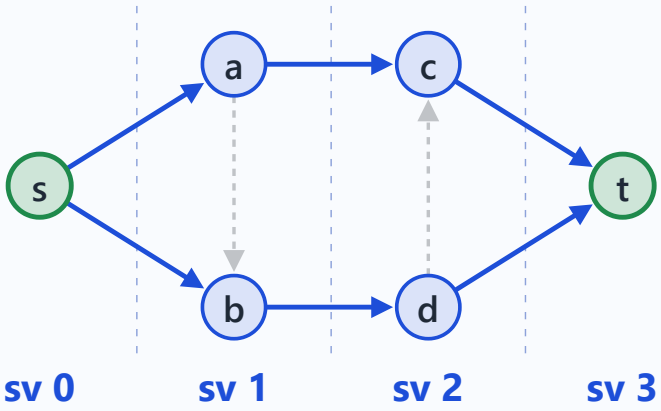
4.4.6.1 Düzey çizgesi ve engelleyici akış

Tanım 4.4.6

Artık ağda kaynaktan BFS yapılarak her v düğümünün **düzeyi** bulunur: *düzey(v)*, kaynaktan v'ye en kısa artık yolun kenar sayısıdır. **Düzey çizgesi** (*level graph*), artık ağın yalnızca *düzey(v) = düzey(u) + 1* koşulunu sağlayan (u, v) kenarlarından oluşan alt çizgesidir.

Düzey çizgesinde bir akış, her s–t yolunda en az bir doymuş kenar bırakıyorsa bu akışa **engelleyici akış** (*blocking flow*) denir.

Düzey çizgesinde her kenar bir düzey ileri gider. Aynı düzeydeki düğümler arasındaki kenarlar ve geri giden kenarlar atılır. Bu yüzden düzey çizgesindeki her s–t yolu, artık ağdaki en kısa s–t yollarından biridir. Üstelik düzey çizgesi döngüsüzdür: düzey her adımda artar, aynı düğüme geri dönülemez.



Şekil 4.4.24: Düzey çizgesi. Düğümler kaynağa uzaklıklarına göre sütunlara dizilir (sv 0, 1, 2, 3). Yalnızca bir sonraki sütuna giden kenarlar tutulur; aynı sütundaki $a \to b$ ve $d \to c$ kenarları (soluk) kullanılmaz.

Engelleyici akış maksimum akış değildir. Yalnızca düzey çizgesindeki yolların hepsini tıkar; düzey çizgesinde artık s–t yolu kalmaz. Ama artık ağın tamamında, düzey atlamayan kenarlar ve yeni doğan ters kenarlar üzerinden daha uzun artış yolları kalabilir. Dinic bu yüzden fazlar hâlinde çalışır: her fazda düzeyleri yeniden hesaplar ve yeni bir engelleyici akış bulur.

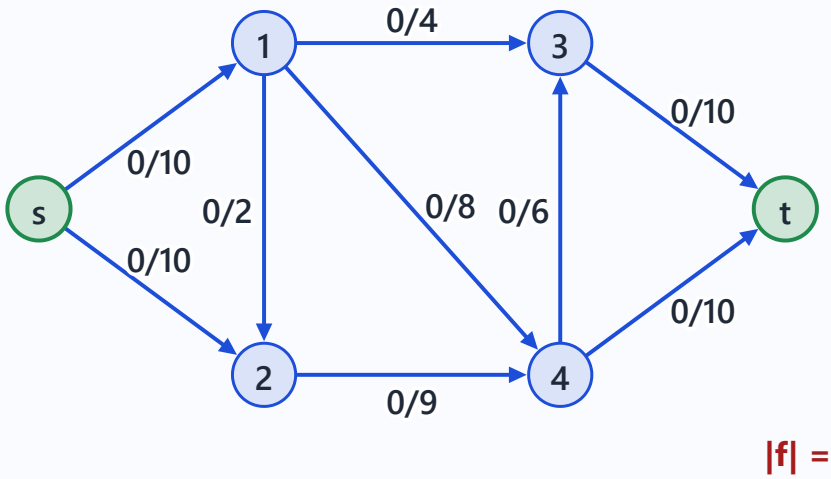
4.4.6.2 Algoritma adımları

1. **Başlangıç.** Bütün kenarların akışı 0'dır; artık ağ özgün ağla aynıdır.
2. **BFS.** Kaynaktan BFS ile her düğümün düzeyi belirlenir. BFS yalnızca artık kapasitesi pozitif kenarları kullanır. Hedefin düzeyi yoksa, yani hedefe ulaşılamıyorsa, akış maksimumdur ve algoritma durur.
3. **DFS.** Her adımda bir düzey ileri giden artış yolları DFS ile aranır. Yalnızca *düzey(v) = düzey(u) + 1* olan kenarlar izlenir. Çıkmaz olduğu anlaşılan kenarlar o faz boyunca bir daha denenmez.
4. **Güncelleme.** Bulunan her yoldaki en küçük artık kapasite toplam akışa eklenir; kenarların ve ters kenarlarının artık kapasiteleri güncellenir. Yol kalmayınca bu fazın engelleyici akışı bulunmuş olur.
5. **Yeni faz.** 2. adıma dönülür.

Her fazda kaynaktan hedefe olan uzaklık en az 1 artar. Uzaklık en fazla $V - 1$ olabileceğinden en fazla $V - 1$ faz vardır. Bu, algoritmanın süre sınırındaki ilk çarpandır.

4.4.6.3 Örnek: fazlar

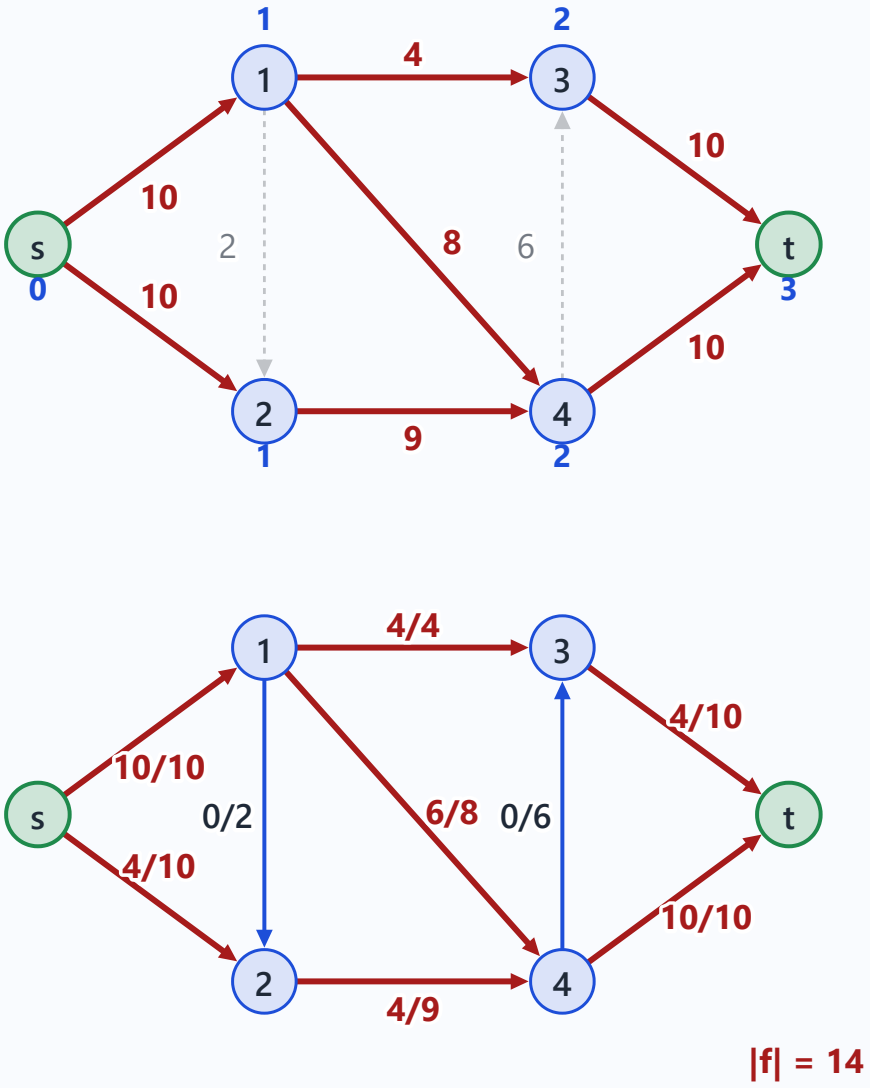
Algoritmayı Şekil 4.4.25'teki ağda çalıştıralım. Bu ağ, Ford-Fulkerson'ın ikinci örneğindeki ağın aynısıdır; yalnızca 2 ve 3 numaralı düğümlerin adları yer değiştirmiştir. Dolayısıyla maksimum akışın 19 olduğunu biliyoruz.



Şekil 4.4.25: Dinic örnek ağı (akış/kapasite); başlangıçta bütün akışlar 0. Kapasiteler: *s→1* 10, *s→2* 10, *1→2* 2, *1→3* 4, *1→4* 8, *2→4* 9, *4→3* 6, *3→t* 10, *4→t* 10.

Faz 1, BFS. Düzeyler: *s = 0*; *1, 2 = 1*; *3, 4 = 2*; *t = 3*. *1 → 2* kenarı iki düzey-1 düğümünü, *4 → 3* kenarı iki düzey-2 düğümünü bağlar. İkisi de düzey atlamaz ve bu fazda kullanılmaz. Kaynaktan hedefe uzaklık 3'tür.

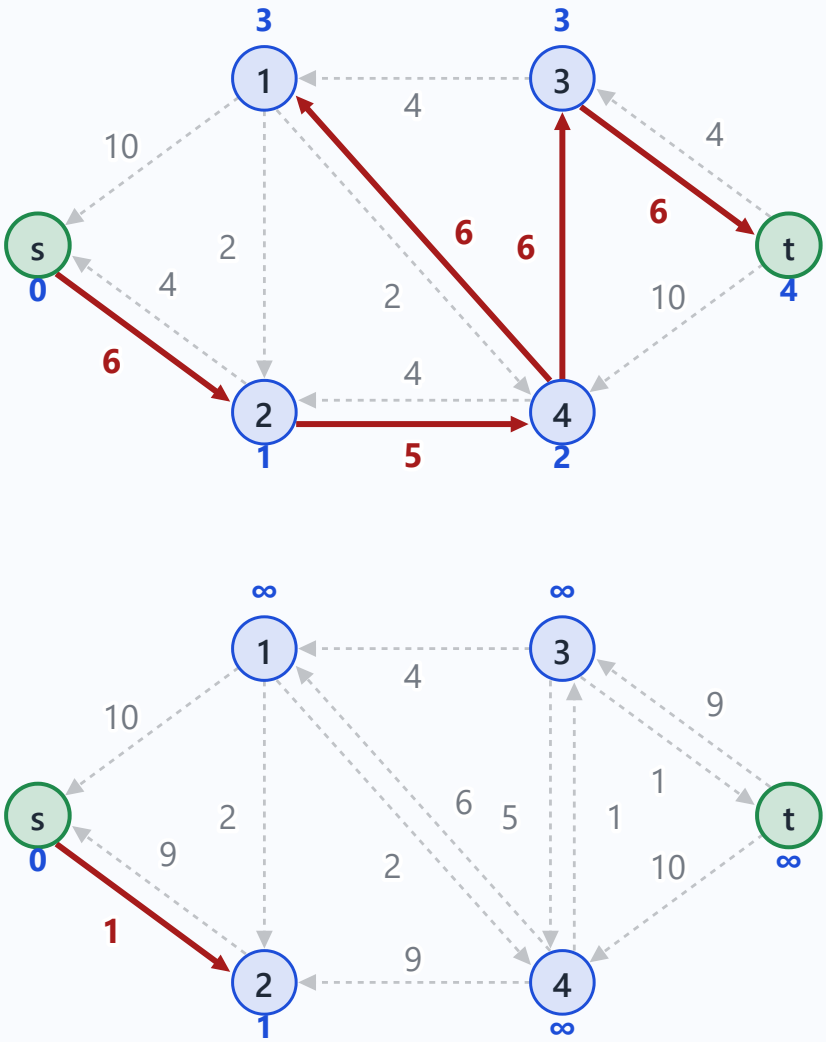
Faz 1, engelleyici akış. DFS düzey çizgesinde üç yol bulur: *{s, 1, 3, t}* 4 birim (*1 → 3*'ün kapasitesi), *{s, 1, 4, t}* 6 birim (*s → 1*'de kalan), *{s, 2, 4, t}* 4 birim (*4 → t*'de kalan). Toplam *|f| = 14*. Bu fazdan sonra düzey çizgesinde s–t yolu kalmaz: *s → 1*, *1 → 3* ve *4 → t* doymuştur.



Şekil 4.4.26: Birinci faz. (a) Düzey çizgesi; düğümlerin yanındaki mavi sayılar düzeylerdir, soluk kenarlar düzey atlamaz. (b) Engelleyici akıştan sonraki akış; *|f| = 14*.

Faz 2, BFS. Artık ağda kaynaktan yalnızca 2'ye çıkılabilir (*s → 2*'de 6 birim kaldı). 2'den 4'e 5 birim gidilir. 4'ten iki yöne gidilebilir: *4 → 3* kenarı ve birinci fazda *1 → 4*'e gönderilen akışın ters kenarı *4 → 1*. 3'ten de hedefe gidilir. Düzeyler: *s = 0*; *2 = 1*; *4 = 2*; *1, 3 = 3*; *t = 4*. Kaynaktan hedefe uzaklık 3'ten 4'e çıkmıştır.

Faz 2, engelleyici akış. DFS önce 4'ten 1'e iner. 1'in düzey 4'teki bir düğüme kenarı yoktur; *4 → 1* kenarı çıkmaz sokaktır. DFS bu kenarı bırakır ve *4 → 3 → t* ile devam eder. Yol *{s, 2, 4, 3, t}*, darboğaz *2 → 4*'tür (*9 − 4 = 5*). *|f| = 14 + 5 = 19*.



Şekil 4.4.27: (a) İkinci fazın düzey çizgesi; kesikli kenarlar birinci fazın ters kenarlarıdır. (b) Üçüncü BFS: kaynaktan yalnızca 2'ye ulaşılır, hedefin düzeyi yoktur (∞).

Faz 3, BFS. Artık ağda kaynaktan yalnızca 2'ye ulaşılır ($s \to 2$'de 1 birim kaldı); $2 \to 4$ doymuştur. Hedefin düzeyi yoktur, algoritma biter. Minimum kesim *({s, 2}, diğerleri)*'dir: $s \to 1$ (10) + $2 \to 4$ (9) = 19.

Sonuç: maksimum akış 19'dur ve yalnızca 2 faz, yani $2 + 1 = 3$ BFS yeterli olmuştur. Aynı ağda Ford-Fulkerson'ın ikinci örneği 5 artış yolu kullanmıştı. Edmonds-Karp bu ağda 4 artış ve 5 BFS yapar. Dinic'in kazancı, birinci fazda uzunluğu 3 olan üç yolu tek bir BFS'in bilgisiyle işlemesinden gelir.

4.4.6.4 Sözde kod

Dinic'in ana döngüsü ve BFS'si aşağıdadır.

```
DINIC(G, kaynak, hedef):
  akış = [ ]       // her kenar için 0
  maksimumAkış = 0
  döngü BFS(G, kaynak, hedef, akış) iken:
    // düzey çizgesi kuruldu
    döngü engelleyici akış var iken:
      delta = DFS(G, kaynak, hedef,
                  akış, seviye, ∞)
      maksimumAkış += delta
  döndür maksimumAkış

BFS(G, kaynak, hedef, akış):
  seviye[v] = -1, her v için
  seviye[kaynak] = 0
  kuyruk.ekle(kaynak)
  döngü kuyruk boş değil iken:
    u = kuyruk.çıkar()
    her bir (u, v) için G[u] içinde:
      eğer seviye[v] < 0 ve
           kapasite[u,v] - akış[u,v] > 0:
        seviye[v] = seviye[u] + 1
        kuyruk.ekle(v)
  döndür seviye[hedef] ≥ 0
```

Dış döngü BFS hedefe ulaştığı sürece döner; her tur bir fazdır. İç döngü, engelleyici akış tamamlanana kadar DFS'i tekrar tekrar çağırır. Her DFS bir yol bulur ve o yolun akışını döndürür; 0 dönünce faz biter. BFS, Edmonds-Karp'taki BFS'e çok benzer. Fark şudur: ebeveyn yerine düzey kaydedilir. Ulaşılmamış düğümlerin düzeyi −1'dir. BFS, hedefin düzeyi negatif değilse `true` döner.

```
DFS(G, u, hedef, akış, seviye, lim):
  eğer u == hedef: döndür lim
  her bir (u, v) için G[u] içinde:
    kalan = kapasite[u,v] - akış[u,v]
    eğer seviye[v] == seviye[u] + 1
         ve kalan > 0:
      delta = DFS(G, v, hedef, akış,
                  seviye, min(lim, kalan))
      eğer delta > 0:
        akış[u, v] += delta
        akış[v, u] -= delta   // ters
        döndür delta
  döndür 0
```

DFS'in `lim` parametresi, kaynaktan `u`'ya kadar gelen yolun darboğazıdır. Kaynakta ∞ ile başlar ve her kenarda `min(lim, kalan)` ile küçülür. Hedefe ulaşınca bu değer döndürülür; bu yolun darboğazıdır. Dönüşte her kenar akışı `delta` kadar artırır, ters kenarınki azalır. Düzey koşulu (`seviye[v] == seviye[u] + 1`) aramanın yalnızca düzey çizgesinde ilerlemesini sağlar.

4.4.6.5 Java: düzeyli DFS ve kaldığı yer dizisi

Sözde koddaki DFS her çağrıda bütün kenarları baştan dener. Bir kenarın çıkmaz olduğu anlaşılsa bile sonraki çağrıda yeniden denenir. Bu, fazın süresini gereksiz uzatır. Java uygulaması bunu küçük bir diziyle önler:

```
int dfs(int u, int t, int akis, int[] basla) {
  if (u == t) return akis;
  for (; basla[u] < komsu[u].size(); basla[u]++) {
    Kenar k = komsu[u].get(basla[u]);
    if (seviye[k.hedef] == seviye[u] + 1 && k.akis < k.kapasite) {
      int d = dfs(k.hedef, t, Math.min(akis, k.kapasite - k.akis), basla);
      if (d > 0) {
        k.akis += d;                                  // ileri kenar
        komsu[k.hedef].get(k.tersIndis).akis -= d;    // ters kenar
        return d;
      }
    }
  }
  return 0;   // u'dan t'ye düzeyli yol kalmadı
}
```

Kenarlar komşuluk listesinde tutulur. Her `Kenar` nesnesi hedef düğümü, kapasiteyi, akışı ve `tersIndis` alanını taşır. `tersIndis`, ters kenarın hedef düğümün listesindeki yeridir. Böylece `komsu[k.hedef].get(k.tersIndis)` ifadesi ters kenara $O(1)$'de ulaşır. Bir kenar eklenirken iki nesne birlikte oluşturulur: kapasiteli ileri kenar ve kapasitesi 0 olan ters kenar.

`basla[u]`, `u`'da denenecek sıradaki kenarın indisidir; İngilizce kaynaklarda bu yapıya *current arc* denir. `for` döngüsü `basla[u]`'dan başlar ve sayaç doğrudan bu dizide artırılır. Bir kenardan hedefe akış ulaşmazsa (`d == 0`), döngü `basla[u]++` ile sonraki kenara geçer ve bu kenar o faz boyunca bir daha denenmez. Akış ulaşırsa metot `basla[u]`'yu artırmadan döner. Kenarda hâlâ yer olabilir; bir sonraki çağrı aynı kenardan başlar.

Koşuldaki `k.akis < k.kapasite`, artık kapasitenin pozitif olduğunu söyler. Ters kenarlar için kapasite 0, akış ise negatiftir: $-f$. Koşul $-f < 0$, yani $f > 0$ olur ve artık kapasite $0 - (-f) = f$ çıkar. İleri ve ters kenarlar aynı satırla işlenir. Her fazın başında `basla` dizisi sıfırlanır ve düzeyler BFS ile yeniden hesaplanır.

> **Sık yapılan hata**
>
> `basla[u]`'yu her durumda artırmak. Akış ulaştığında kenar doymamış olabilir; sayaç artırılırsa kenarın kalan kapasitesi o fazda kullanılmaz. Algoritma yine doğru sonucu verir, ama engelleyici akış eksik kalır ve faz sayısı artar. Tersine, sayaç hiç kullanılmazsa çıkmaz kenarlar tekrar tekrar denenir ve faz başına $O(V \cdot E)$ sınırı kaybolur.

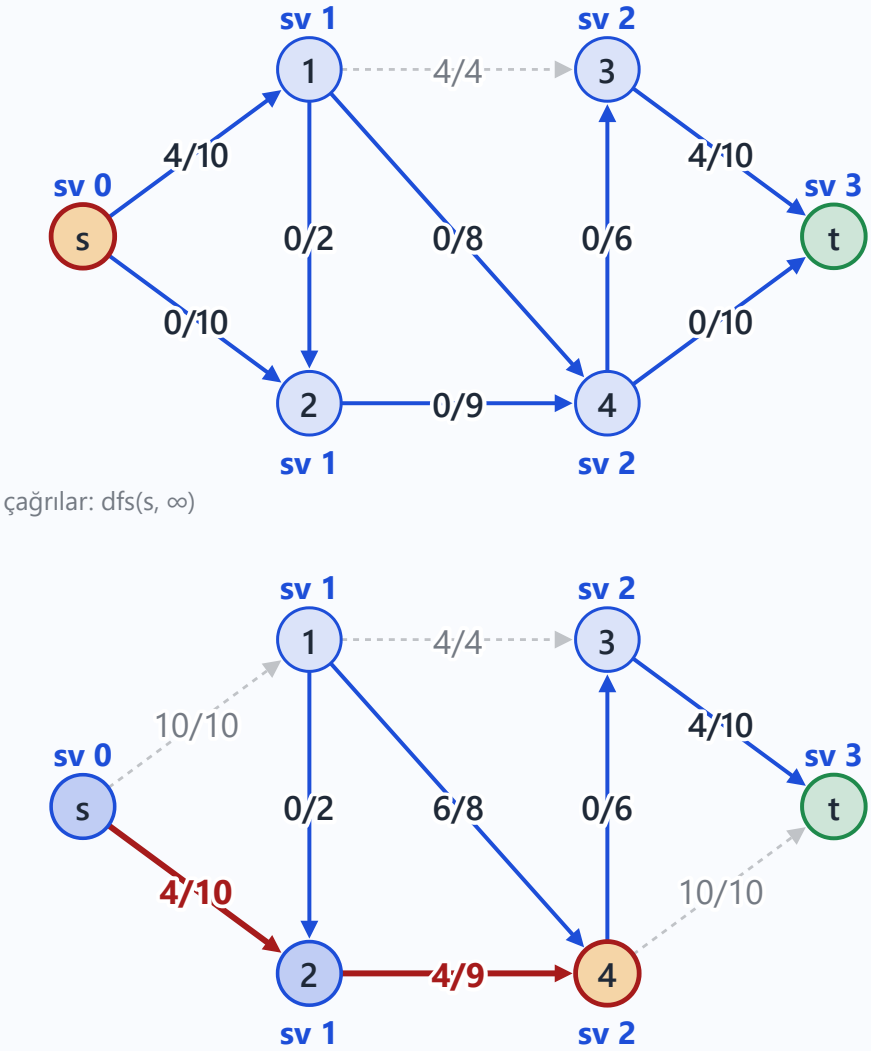
4.4.6.6 Kod izleme: birinci fazda engelleyici akış

Birinci fazdaki DFS çağrılarını izleyelim. Komşuluk listelerinde kenarlar eklenme sırasındadır: 1'in listesinde ters *s* kenarından sonra *1 → 2*, *1 → 3*, *1 → 4*; 4'ün listesinde iki ters kenardan sonra *4 → 3* ve *4 → t* gelir. Ters kenarlar bu fazda düzey koşulunu sağlamaz.

Çağrı	İnilen yol (kalan)	Olay	Dönen `d`	*|f|*
1	s→1 (10), 1→3 (4), 3→t (10)	*1 → 2* aynı düzeyde, `basla[1]++`. Akış *min(10, 4, 10) = 4* ile sınırlanır.	4	4
2	s→1 (6), 1→4 (8), 4→t (10)	*1 → 3* doymuş, `basla[1]++`. 4'te *4 → 3* aynı düzeyde, `basla[4]++`.	6	10
3	s→2 (10), 2→4 (9), 4→t (4)	*s → 1* doymuş, `basla[s]++`. *4 → t*'de 4 birim kaldı.	4	14
4	s→2 (6), 2→4 (5)	4'te *4 → t* doymuş, `basla[4]++`; liste bitti: 0. 2'de `basla[2]++`: 0. *s*'de `basla[s]++`: 0.	0	14

Tablo 4.4.8: Birinci fazda `dfs(s, t, ∞, basla)` çağrıları.

Dördüncü çağrı engelleyici akışın tamamlandığını gösterir. Bu çağrıda 4'ten düzeyli yol kalmamıştır: *4 → 3* aynı düzeydedir, *4 → t* doymuştur. 0 değeri yukarı doğru iletilirken her düğüm sayacını ilerletir. Sonunda `basla[s]` listenin sonuna ulaşır ve faz *|f| = 14* ile biter.



Şekil 4.4.28: Düzeyli DFS'ten iki an. (a) Birinci yolun akışı geri dönerken her kenara 4 eklenir; $|f| = 4$. (b) Dördüncü çağrı: $s \to 2 \to 4$ ile inildi, 4'ten düzeyli yol kalmadı.

Tabloda `basla` sayaçlarının toplam artış sayısı küçüktür: her kenar faz boyunca en fazla bir kez "çıkmaz" ilan edilir. Kaldığı yer dizisinin değeri buradadır. Çıkmaz bir kenara bir daha bakılmadığı için bir fazdaki bütün DFS çağrılarının toplam işi sınırlı kalır.

4.4.6.7 Karmaşıklık: $O(V^2 \cdot E)$

Teorem 4.4.6 (Dinic)

Dinic algoritması en fazla $V - 1$ faz yapar. Kaldığı yer dizisiyle her faz $O(V \cdot E)$ sürer. Toplam süre $O(V^2 \cdot E)$'dir.

İspat. **Faz sayısı.** Bir fazın sonunda düzey çizgesindeki her s–t yolunda doymuş bir kenar vardır. Sonraki fazın artık ağında yeni kenarlar yalnızca ters kenarlardır ve bunlar düzey çizgesinde bir düzey geri gider. Uzunluğu eskisiyle aynı olan bir s–t yolu yalnızca düzey çizgesinin kenarlarından oluşabilirdi; bu yolların hepsi tıkanmıştır. Öyleyse kaynaktan hedefe uzaklık kesin olarak artar. Uzaklık 1 ile $V - 1$ arasında olduğundan en fazla $V - 1$ faz vardır.

Faz süresi. BFS $O(V + E)$ sürer. DFS'lerin işi iki parçaya ayrılır. Başarılı her DFS en fazla $V - 1$ kenar uzunluğunda bir yol izler ve yolda en az bir kenarı doyurur. Doyan kenar o fazda bir daha kullanılmaz; bu yüzden en fazla E başarılı DFS vardır ve toplam maliyetleri $O(V \cdot E)$'dir. Başarısız denemelerde sayaç bir adım ilerler; her kenar için en fazla bir kez, toplam $O(E)$. Faz başına $O(V \cdot E)$.

Toplam: $O(V) \cdot O(V \cdot E) = O(V^2 \cdot E)$. □

Dinic'in sınırı Edmonds-Karp'ınkiyle karşılaştırılınca kazanç açıktır. $E \geq V$ olan ağlarda $V^2 \cdot E \leq V \cdot E^2$'dir; ağ yoğunlaştıkça fark büyür. $V = 1000$, $E = 10.000$ için $V \cdot E^2$ değeri 10^{11}, $V^2 \cdot E$ değeri 10^{10}'dur.

Özel ağlarda sınır daha da iyidir. Bütün kapasiteler 1 ise faz sayısı $O(\sqrt{E})$ ile sınırlanır ve her faz $O(E)$ sürer: toplam $O(E \cdot \sqrt{E})$. İki parçalı eşleştirme gibi, her düğümden en fazla bir birim akışın geçtiği birim ağlarda süre $O(E \cdot \sqrt{V})$'ye iner. Bu, Hopcroft-Karp algoritmasının sınırıdır; Hopcroft-Karp aslında Dinic'in eşleştirmeye uyarlanmış bir biçimidir.

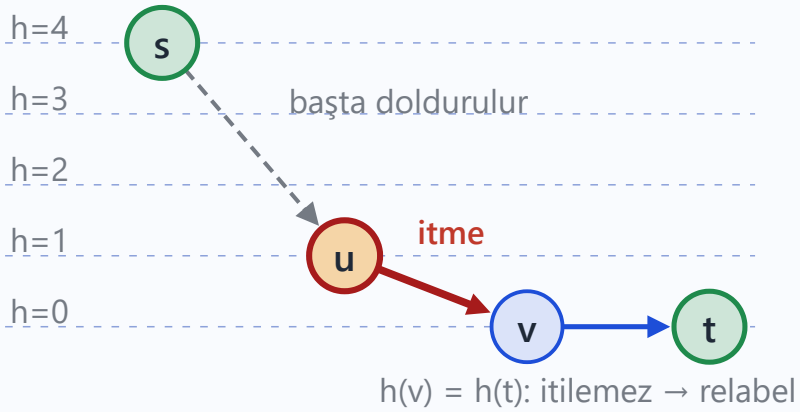
Pratikte Dinic, yarışma programlamada en çok kullanılan maksimum akış algoritmasıdır. Kodu kısadır ve genellikle sınırının çok altında çalışır. Faz sayısı çoğu ağda V'den çok küçüktür; bu notun örneğinde 2 faz yetmiştir.

4.4.7 İtme–yeniden etiketleme

4.4.7.1 Fikir: suyu yokuş aşağı akıtmak

Şimdiye kadarki algoritmaların hepsi bütünsel çalışır: her adımda kaynaktan hedefe tam bir yol arar. A. Goldberg ve R. Tarjan'ın 1988'de yayımladığı **itme–yeniden etiketleme** (*push–relabel*) algoritması bambaşka bir yol izler. Artış yolu aramaz. Her işlem tek bir düğüm ve onun komşularıyla ilgilidir; algoritma bu yerel işlemlerin tekrarıyla maksimum akışa ulaşır.

Algoritmayı bir su sistemi gibi düşünmek kolaylık sağlar. Başlangıçta kaynaktan çıkan bütün borular sonuna kadar doldurulur. Su, kaynağın komşularında birikir. Her düğümün bir **yüksekliği** (*height*) vardır ve su yalnızca yüksekten bir alçağa doğru akar. Birikmiş suyunu hiçbir komşusuna akıtamayan bir düğüm yükseltilir; buna **yeniden etiketleme** (*relabel*) denir. Suyun bir komşuya akıtılmasına ise **itme** (*push*) denir. Hedefe ulaşamayan fazla su, yükseklik kaynağın yüksekliğini geçince kaynağa geri döner. Hiçbir düğümde birikmiş su kalmayınca hedefe varmış su maksimum akıştır.



Şekil 4.4.29: İtme–yeniden etiketlemenin fikri. Kaynak $h = 4$ yüksekliğindedir ve kenarları başta doldurulur. u ($h = 1$) fazlasını bir alçak komşuya (v, $h = 0$) iter. v ile t aynı yükseklikte olduğundan v hedefe itemez; önce yeniden etiketlenmesi gerekir.

4.4.7.2 Ön akış ve aşırı akış

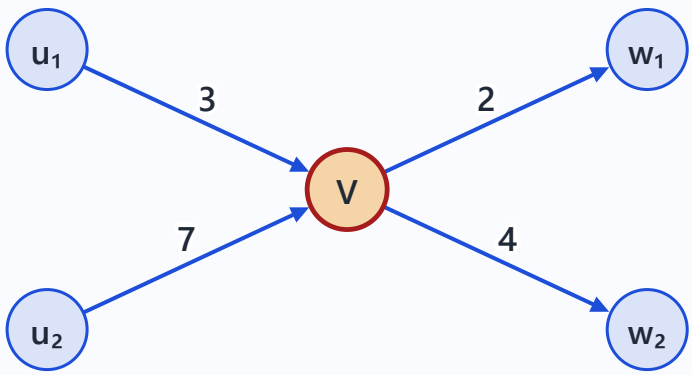
Algoritmanın ara adımlarındaki yapı bir akış değildir; korunum kısıtı bozulur. Bir düğüme giren su çıkandan fazla olabilir. Bu gevşek yapıya ön akış denir.

Tanım 4.4.7

Ön akış (*preflow*), kapasite kısıtına uyan ve korunum yerine şu koşulu sağlayan bir f fonksiyonudur: kaynak dışındaki her v için

$$e(v) = \Sigma_u f(u, v) - \Sigma_w f(v, w) \geq 0$$

$e(v)$'ye v'nin **aşırı akışı** (*excess flow*) denir: giren toplam akış eksi çıkan toplam akış. $e(v) > 0$ olan, kaynak ve hedef dışındaki bir düğüm **aktif düğümdür**.



Şekil 4.4.30: V düğümüne u_1'den 3, u_2'den 7 birim girer; V'den w_1'e 2, w_2'ye 4 birim çıkar. Aşırı akış

$e(V) = (3 + 7) - (2 + 4) = 4$'tür.

Ön akışta korunum yalnızca bir yönde gevşetilmiştir: bir düğüm akış biriktirebilir, ama elinde olmayan akışı veremez. Aşırı akış hiçbir zaman negatif olmaz. Korunum, bütün ara düğümlerde $e(v) = 0$ olması demektir. Öyleyse aktif düğüm kalmadığında ön akış gerçek bir akışa dönüşür. Algoritma bu ana kadar çalışır.

4.4.7.3 Yükseklik fonksiyonu ve başlangıç

Algoritma her düğüme bir $h(v)$ tam sayısı atar. Başlangıç şöyledir:

- Bütün kenarların akışı 0'dır.
- $h(s) = |V|$, yani düğüm sayısı; diğer bütün düğümlerde $h = 0$'dır.
- Kaynaktan çıkan her kenar doldurulur: $f(s, v) = c(s, v)$ ve $e(v) = c(s, v)$.

Algoritma boyunca iki değişmez korunur. Birincisi, $h(s) = |V|$ ve $h(t) = 0$ hiç değişmez; kaynak ve hedef etiketlenmez. İkincisi, artık ağdaki her (u, v) kenarı için $h(u) \le h(v) + 1$'dir. Bu ikinci değişmez, bir kenar boyunca yüksekliğin en fazla 1 düşebileceğini söyler. Yokuşlar dik olamaz.

İkinci değişmezin önemli bir sonucu vardır: $h(u)$, u'dan hedefe artık ağ uzaklığının bir alt sınırıdır. u'dan hedefe k kenarlı bir artık yol varsa, her kenarda yükseklik en fazla 1 düşer; hedefin yüksekliği 0 olduğundan $h(u) \le k$ olur.

Teorem 4.4.7 (s–t yolu yok)

Değişmezler sağlandığı sürece artık ağda kaynaktan hedefe yol yoktur. Bu yüzden algoritma aktif düğüm kalmadığında durursa elde edilen akış maksimumdur.

İspat. Artık ağda kaynaktan hedefe bir yol olsaydı, basit bir yol olarak en fazla $|V| - 1$ kenarı olurdu. Az önceki gözlemle $h(s) \le |V| - 1$ gerekirdi. Ama $h(s) = |V|$'dir. Çelişki. Aktif düğüm kalmadığında ön akış bir akıştır ve artık ağında artış yolu yoktur; maksimum akış minimum kesim teoremine göre maksimumdur. □

Bu bakış açısı algoritmanın Ford-Fulkerson ailesine göre tersine işlediğini gösterir. Ford-Fulkerson her an geçerli bir akış tutar ve artış yolu kalmayana kadar çalışır. İtme–yeniden etiketleme ise her an “artış yolu yok” koşulunu tutar ve ön akış geçerli bir akış olana kadar çalışır. Başlangıç değeri $h(s) = |V|$ bu koşulun kurulmasını sağlar. Kaynaktan çıkan kenarlar doldurulduğu için artık ağda kaynaktan çıkan kenar kalmaz; değişmez başlangıçta da doğrudur.

4.4.7.4 İtme ve yeniden etiketleme işlemleri

İtme: push(u, v). Koşul üç parçalıdır: $e(u) > 0$; $c_f(u, v) > 0$, yani kenarda kapasite ile akış arasında fark vardır; ve $h(u) = h(v) + 1$. Bu koşullarda $\delta = min(e(u), c_f(u, v))$ kadar akış itilir: $f(u, v)$ artar (ya da ters kenarsa $f(v, u)$ azalır), $e(u) -= \delta$, $e(v) += \delta$. Kenar bu itmeyle doyarsa **doyuran itme**, doymazsa **doyurmayan itme** denir. Doyurmayan itmede $\delta = e(u)$'dur ve u'nun fazlası tamamen boşalır.

Yeniden etiketleme: relabel(u). Koşul: $e(u) > 0$, ama artık ağdaki her (u, v) kenarı için $h(u) \le h(v)$; yani hiçbir komşuya itilemiyor. İşlem $h(u) = 1 + min\ h(v)$'dir; en küçük, artık ağdaki (u, v) kenarları üzerinden alınır. Bu yeni değerle en az bir itme mümkün olur: en alçak komşu tam bir basamak aşağıdadır.

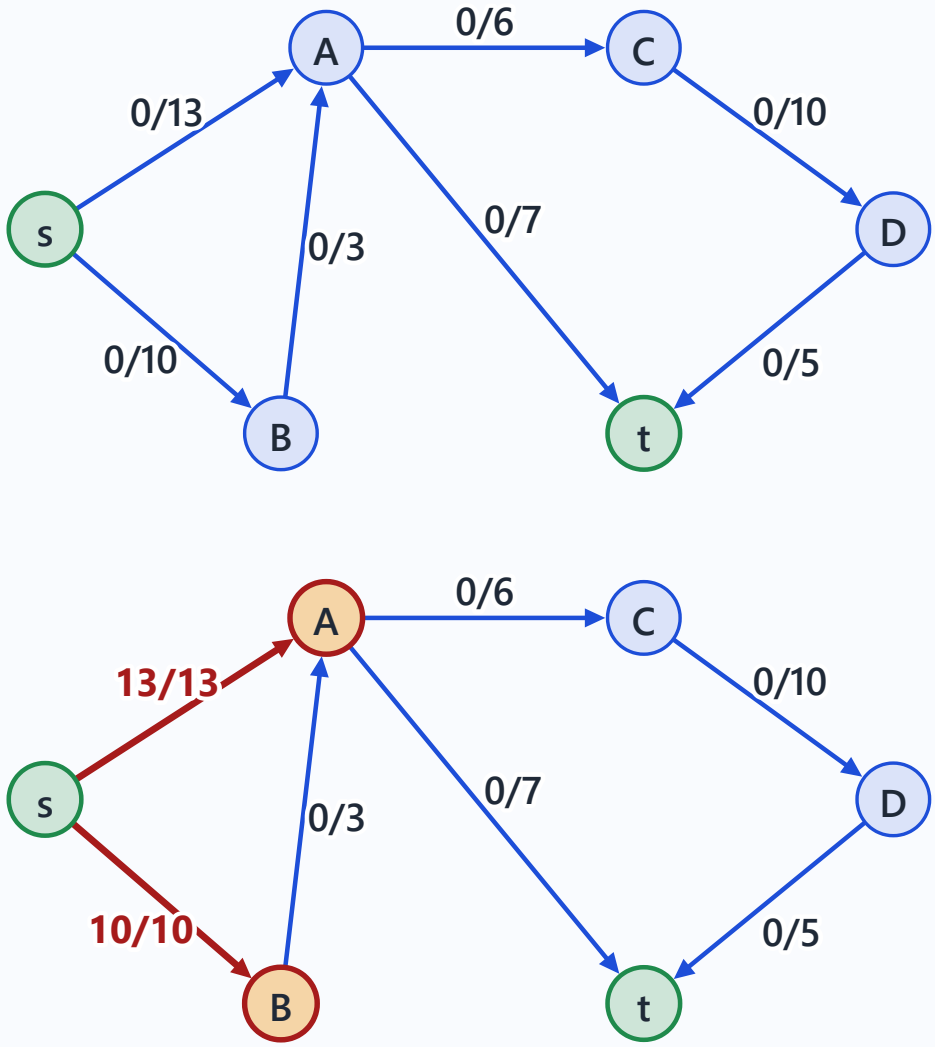
Her iki işlem de değişmezleri korur. İtme yalnızca $h(u) = h(v) + 1$ olan bir kenarda yapılır. Yeni doğabilecek (v, u) ters kenarında $h(v) = h(u) - 1 \le h(u) + 1$ olur; değişmez sağlanır. Yeniden etiketlemede $h(u)$, artık komşularının en küçüğünden tam bir fazla seçilir; u'dan çıkan kenarlarda değişmez korunur. u'ya giren kenarlarda ise $h(u)$'nun artması koşulu yalnızca gevşetir.

Akış yüksekten alçağa itilir ve yükseklikler yalnızca artar. Aşırı akışı olan bir düğümün artık ağda her zaman kaynağa giden bir yolu vardır: bu akış kaynaktan bir yoldan gelmiştir ve o yolun ters kenarları artık ağdadır. Bu yüzden yeniden etiketlemedeki *min* her zaman tanımlıdır. Hedefe ulaşamayan fazla, yüksekliği kaynağın yüksekliği olan $|V|$'yi aşınca kaynağa geri itilir.

> **Sık yapılan hata**
>
> İtme yönünü ya da başlangıç yüksekliğini karıştırmak. Akış yalnızca $h(u) = h(v) + 1$ iken, yüksekten alçağa itilir. Eşit yükseklikteki bir komşuya itmek değişmezi bozar. Başlangıçta $h(s) = |V|$ olmalıdır; kaynak 0'da bırakılırsa fazla kaynağa dönemez ve “artış yolu yok” garantisi kaybolur.

4.4.7.5 Örnek: adım adım

Şekil 4.4.31'deki ağda altı düğüm vardır: $|V| = 6$. Aktif düğümler istenen sırada seçilebilir; örnekteki sıra işlemleri açıkça göstermek için seçilmiştir.



Şekil 4.4.31: (a) İtme–yeniden etiketleme örnek ağı (akış/kapasite): $s \to A$ 13, $s \to B$ 10, $B \to A$ 3, $A \to C$ 6, $A \to t$ 7, $C \to D$ 10, $D \to t$ 5. (b) Başlangıç: $h(s) = 6$, kaynağın iki kenarı dolduruldu; $e(A) = 13$, $e(B) = 10$.

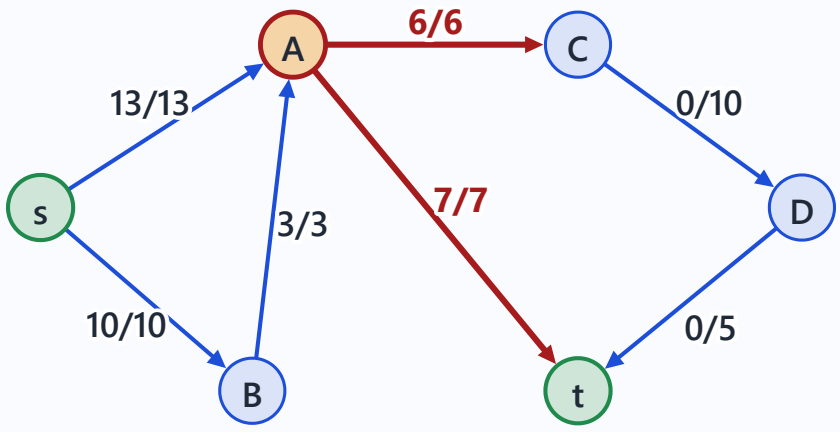
Adım	İşlem	A (h, e)	B (h, e)	C (h, e)	D (h, e)	e(t)
0	başlangıç: $s \to A$, $s \to B$ doldurulur	0, 13	0, 10	0, 0	0, 0	0
1	relabel(B): $1 + \min(h(A) = 0, h(s) = 6)$	0, 13	1, 10	0, 0	0, 0	0
2	push(B, A): $\delta = \min(10, 3) = 3$	0, 16	1, 7	0, 0	0, 0	0
3	relabel(A): $1 + \min(0, 0, 1, 6)$	1, 16	1, 7	0, 0	0, 0	0
4	push(A, C) 6, push(A, t) 7 (ikisi de doyar)	1, 3	1, 7	0, 6	0, 0	7
5	relabel(C): $1 + h(D)$	1, 3	1, 7	1, 6	0, 0	7
6	relabel(A): komşular B (1), s (6)	2, 3	1, 7	1, 6	0, 0	7
7	push(A, B) 3 ($B \to A$ akışı iptal)	2, 0	1, 10	1, 6	0, 0	7
8	relabel(B) → 3, push(B, A) 3	2, 3	3, 7	1, 6	0, 0	7

Adım	İşlem	A (h, e)	B (h, e)	C (h, e)	D (h, e)	e(t)
9	A ile B arasında sırayla etiketleme ve itme: *h(A) = 4*, *h(B) = 5*, *h(A) = 6*, *h(B) = 7*; B, 3 birimi A'ya, 7 birimi kaynağa iter	6, 3	7, 0	1, 6	0, 0	7
10	relabel(A): *1 + min(6, 7) = 7*; push(A, s) 3	7, 0	7, 0	1, 6	0, 0	7
11	push(C, D) 6	7, 0	7, 0	1, 0	0, 6	7
12	relabel(D) → 1, push(D, t) 5	7, 0	7, 0	1, 0	1, 1	12
13	relabel(D) → 2, push(D, C) 1	7, 0	7, 0	1, 1	2, 0	12
14	C ile D arasında sırayla etiketleme ve itme: C 3, 5, 7, 8; D 4, 6, 8; C, 1 birimi A'ya iter	7, 1	7, 0	8, 0	8, 0	12
15	push(A, s) 1	7, 0	7, 0	8, 0	8, 0	12

Tablo 4.4.9: İtme–yeniden etiketleme izlemesi. Her satırdan sonraki yükseklikler (h) ve aşırı akışlar (e); $h(s) = 6$ ve $h(t) = 0$ değişmez.

İlk adımlar işlemlerin koşullarını gösterir. Birinci adımda B aktiftir, ama artık komşularından A aynı yüksekliktedir (0); kaynak ise çok yüksektedir. B itemez ve yeniden etiketlenir. Ardından A'ya bir basamak yukarıdan bakar ve $B \to A$ kenarının kapasitesi kadar, 3 birim iter. Üçüncü adımda A'nın komşuları C (0), t (0), B (1; $B \to A$ akışının ters kenarı) ve s (6; $s \to A$ akışının ters kenarı)'dir. En küçük 0'dır, $h(A) = 1$ olur.

Dördüncü adımda A fazlasını hedefe ve C'ye dağıtır. Hedefin kenarı 7'de, C'nin kenarı 6'da doyar. A'da 3 birim kalır. Bu 3 birimin hedefe ulaşma yolu yoktur: A'nın iki çıkışı da doludur. Sonraki adımların çoğu, bu fazlanın kaynağa geri dönmesidir.

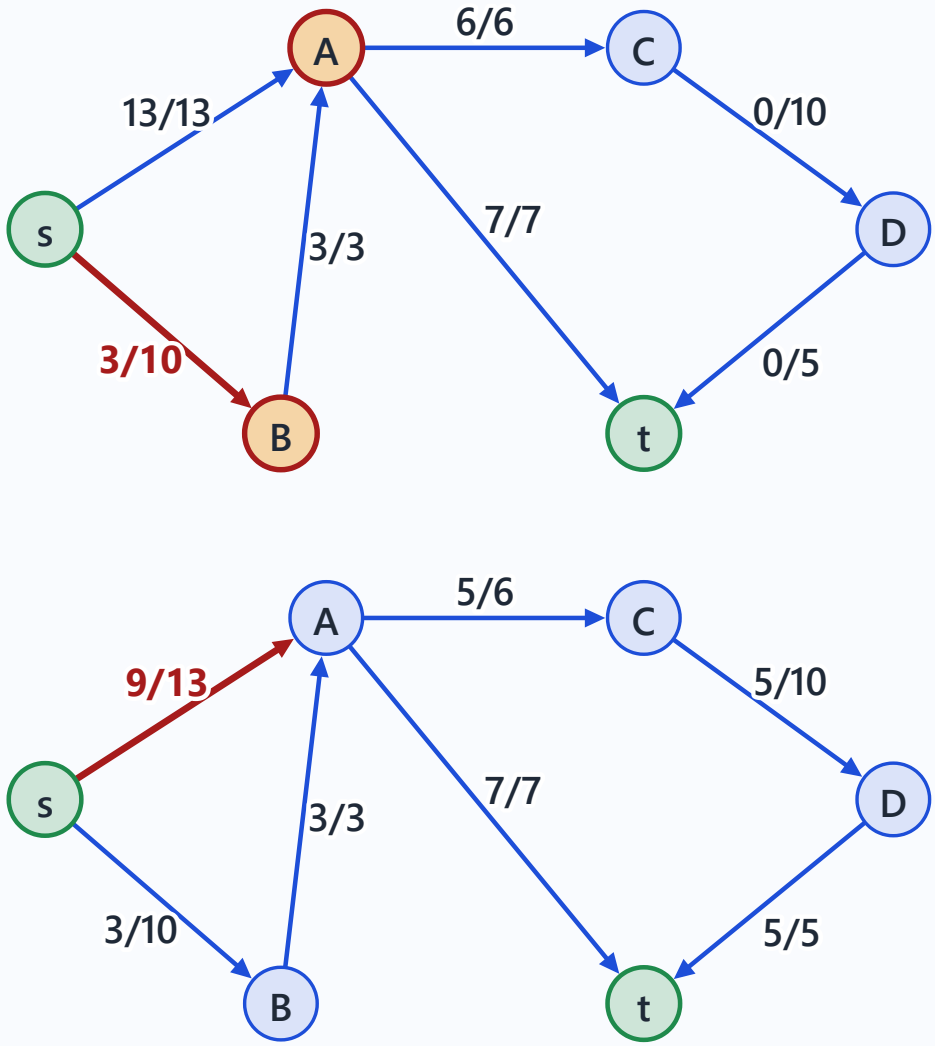


Şekil 4.4.32: Dördüncü adım: A, C'ye 6 ve hedefe 7 birim iter. İki kenar da doyar; $e(A) = 3$, $e(C) = 6$, hedefte 7.

Altıncı adımdan dokuzuncu adıma kadar olan bölüm algoritmanın en sık eleştirilen yönünü gösterir. A ile B fazlayı birbirine iter; her seferinde biri bir basamak yükselir. Bu "top oynama", ikisinin yüksekliği kaynağın yüksekliği olan 6'yı geçene kadar sürer. $h(B) = 7$ olunca B kaynaktan yüksektir. B'nin 10

birimlik fazlasından 3'ü A'ya, 7'si kaynağa gider ($s \to B$ akışı 10'dan 3'e iner). Onuncu adımda A da 7'ye yükselir ve kendi 3 birimini kaynağa iter.

Sonraki adımlarda aynı şey C ile D arasında olur. C'nin 6 birimi D'ye gider; D bunun 5'ini hedefe iter, çünkü $D \to t$'nin kapasitesi 5'tir. Kalan 1 birim C ile D arasında gidip gelir. Yükseklikler A'nın yüksekliği olan 7'yi geçince C, bu birimi A'ya iter ($A \to C$ akışı 6'dan 5'e iner); A da onu kaynağa iter.



Şekil 4.4.33: (a) Dokuzuncu adım: B'nin fazlası kaynağa döndü; $s \to B$ akışı 3'e indi. (b) Son durum: aktif düğüm yok. $s \to A$ 9, $s \to B$ 3; hedefe giren akış 12.

Aktif düğüm kalmadığında maksimum akış hedefe giren akıştır: $7 + 5 = 12$. Kaynaktan çıkan akış da $9 + 3 = 12$'dir. Minimum kesim $S = \{s, A, B, C, D\}$, $T = \{t\}$'dir: $A \to t$ (7) + $D \to t$ (5) = 12.

Örnekteki seçim sırası tek seçenek değildir. Sunumdaki Java işlevleri, her turda indis sırasına göre ilk aktif düğümü seçen bir ana döngüyle de çalıştırılabilir. Bu döngü aynı ağda 38 işlem (yeniden etiketleme ve itme) yapar ve yine 12 bulur. Sıra işlem sayısını değiştirir, sonucu değiştirmez.

4.4.7.6 Sözde kod

Algoritmanın başlangıç kısmı şöyledir:

```
PUSH_RELABEL(G, kaynak, hedef):
  yükseklik[kaynak] = |V|
  yükseklik[v] = 0, v ≠ kaynak
  aşırı[v] = 0, her v için
  akış[e] = 0, her kenar e için
  // ön akış: kaynak kenarlarını doldur
  her bir (kaynak, v) ∈ E için:
    c = kapasite[kaynak, v]
    akış[kaynak, v] = c
    akış[v, kaynak] = -c
    aşırı[v] = c
    aşırı[kaynak] -= c
```

Akış ters simetrik bir tabloda tutulur: kaynak kenarı doldurulurken `akış[v, kaynak] = -c` yazılır. Kaynağın aşırı akışı negatif olur; kaynak bu bakımdan özel bir düğümdür. Ana döngü şöyledir:

```
döngü aşırı[u] > 0 olan u varken:
  // u ≠ kaynak, u ≠ hedef
  eğer kalan(u,v) > 0 ve
       yükseklik[u] == yükseklik[v]+1:
    // İtme
    delta = min(aşırı[u], kalan(u,v))
    akış[u,v] += delta
    akış[v,u] -= delta
    aşırı[u] -= delta
    aşırı[v] += delta
  değilse:
    // Yeniden etiketleme
    m = ∞
    her bir (u, v) için G[u] içinde:
      eğer akış[u,v] < kapasite[u,v]:
        m = min(m, yükseklik[v])
    yükseklik[u] = m + 1
döndür aşırı[hedef]
// kalan = kapasite - akış
```

Döngü aktif bir `u` düğümü bulunduğu sürece döner. `u`'nun bir `v` komşusuna itme koşulu sağlanıyorsa itme yapılır; sağlanmıyorsa `u` yeniden etiketlenir. Yeniden etiketleme kısmı artık ağdaki komşuların, yani `akış < kapasite` olan kenarların yüksekliklerinin en küçüğünü bulur ve bir fazlasını yazar. Döngü bittiğinde hedefin aşırı akışı döndürülür; bu, hedefe varmış toplam akıştır.

4.4.7.7 Java: it() ve etiketle()

Java uygulamasında her `Dugum` nesnesi kendi `yukseklik` ve `asiri` alanlarını ve çıkan kenarlarının listesini tutar. İtme işlemi şöyledir:

```
boolean it(Dugum d) {
  for (Kenar k : d.kenarlar) {
    if (d.yukseklik > k.hedef.yukseklik
        && k.akis != k.kapasite) {
      int akis = Math.min(
        k.kapasite - k.akis, d.asiri);
      d.asiri -= akis;
      k.hedef.asiri += akis;
      k.akis += akis;
      tersKenariGuncelle(k, akis);
      return true;   // itme yapıldı
    }
  }
  return false;
}
```

Metot düğümün kenarlarını sırayla dener. Koşuldaki `d.yukseklik > k.hedef.yukseklik` ifadesi, teorideki $h(u) = h(v) + 1$ koşulundan daha gevşek görünür. Ama değişmez nedeniyle ikisi aynıdır: artık ağdaki bir kenarda $h(u) \le h(v) + 1$ olduğundan $h(u) > h(v)$ ancak $h(u) = h(v) + 1$ iken mümkündür. `k.akis != k.kapasite` kenarın artık ağda olduğunu söyler.

Koşul sağlanınca itilecek miktar kalan kapasite ile aşırı akışın küçüğüdür. Düğümün fazlası azalır, hedef düğümünki artar, kenarın akışı büyür ve `tersKenariGuncelle` ters kenarın akışını aynı miktarda azaltır. Ters kenarlar kapasitesi 0, akışı negatif olan kenarlar olarak saklanır. Metot ilk başarılı itmeden sonra `true` döner. Hiçbir kenar uygun değilse `false` döner ve düğüm yeniden etiketlenir:

```
void etiketle(Dugum d) {
  int minY = Integer.MAX_VALUE;
  for (Kenar k : d.kenarlar) {
    // yalnızca artık ağdaki kenarlar
    if (k.akis != k.kapasite
        && k.hedef.yukseklik < minY) {
      minY = k.hedef.yukseklik;
      d.yukseklik = minY + 1;
    }
  }
}
```

Döngü artık ağdaki komşular arasında en alçak olanı bulur ve yüksekliği onun bir fazlasına ayarlar. Atama döngünün içinde yapılır, ama son yapılan atama en küçük yüksekliğe karşılık geldiği için sonuç doğrudur. Ana döngü bu iki metodu birleştirir: aktif düğüm varken `if (!it(d)) etiketle(d);`.

Sonuç hedefin aşırı akışıdır.

Bir itme $O(1)$, bir yeniden etiketleme düğümün derecesi kadar zaman alır. Kod kenar listesini her çağrıda baştan tarar. Bu basitlik uğruna yapılmış bir tercihtir; verimli uygulamalar Dinic'teki `basla` dizisine benzer bir "sıradaki kenar" göstergesi tutar.

4.4.7.8 Karmaşıklık

Algoritmanın süresi, her işlem türünün en fazla kaç kez yapılabileceği sayılarak bulunur. Tablo 4.4.10 sonuçları özetler.

İşlem	Toplam sayı	Gerekçe
Yeniden etiketleme	$O(V^2)$	her düğümün yüksekliği $\le 2\|V\| - 1$ ve yalnızca artar
Doyuran itme	$O(V \cdot E)$	bir kenar yeniden doyabilmek için iki uçtaki yükseklik ≥ 2 artmalı
Doyurmayan itme	$O(V^2 \cdot E)$	potansiyel fonksiyonu $\Sigma\, h(v)$ (aktif v) ile

Tablo 4.4.10: İtme–yeniden etiketleme işlemlerinin toplam sayısı.

Yeniden etiketleme. Aktif bir düğümün artık ağda kaynağa giden bir yolu vardır. Bu yol en fazla $|V| - 1$ kenarlıdır ve her kenarda yükseklik en fazla 1 düşer. Öyleyse $h(u) \le h(s) + |V| - 1 = 2|V| - 1$. Her yeniden etiketleme yüksekliği en az 1 artırdığı için bir düğüm en fazla $2|V| - 1$ kez etiketlenir; toplam $O(V^2)$. Örneğimizde $|V| = 6$ için sınır 11'dir; en yüksek etiket 8 olmuştur.

Doyuran itme. (u, v) kenarı doyurulduğunda $h(u) = h(v) + 1$'dir. Aynı yönde yeniden doyurulabilmesi için önce v'den u'ya itme yapılmalıdır; bunun için $h(v)$, $h(u) + 1$'e çıkmalıdır, yani en az 2 artmalıdır. Sonra u'nun da 2 artması gerekir. Yükseklikler $2|V|$'yi aşamadığından her kenar $O(V)$ kez doyurulur; toplam $O(V \cdot E)$.

Doyurmayan itme. Aktif düğümlerin yükseklikleri toplamı Φ bir potansiyel olarak kullanılır. Doyurmayan itme u'yu pasif yapar ve Φ'yi en az 1 azaltır (v aktifleşse bile $h(v) = h(u) - 1$'dir). Yeniden etiketlemeler Φ'yi toplamda $O(V^2)$, doyuran itmeler en fazla $O(V \cdot E) \cdot 2V$ artırır. Bu yüzden doyurmayan itme sayısı $O(V^2 \cdot E)$'dir.

Genel algoritma en kötü durumda $O(V^2 \cdot E)$ sürer. Bir yeniden etiketleme en fazla derece kadar kenarı tarar; toplam tarama $O(V \cdot E)$'dir ve baskın terimi değiştirmez. Aktif düğümün seçim kuralı sınırı iyileştirir. Aktif düğümler bir FIFO (ilk giren ilk çıkar) kuyruğunda tutulursa süre $O(V^3)$'e iner. Her seferinde en yüksek etiketli aktif düğüm seçilirse süre $O(V^2 \cdot \sqrt{E})$ olur.

Pratikte iki sezgi algoritmayı çok hızlandırır. **Boşluk** (*gap*) sezgisi, bir yükseklik değerinde hiç düğüm kalmadığını fark eder. Bu durumda o değerin üstündeki düğümler hedefe ulaşamaz; sezgi onları doğrudan $|V|$'nin üstüne taşır. Örnekteki uzun "top oynama" bölümleri böylece tek adımda kısalır. **Toplu yeniden etiketleme** (*global relabeling*) ise zaman zaman hedeften geriye bir BFS yaparak bütün yükseklikleri gerçek uzaklıklara eşitler. Bu sezgilerle itme–yeniden etiketleme, en hızlı genel amaçlı maksimum akış yöntemlerinden biridir.

4.4.8 Karşılaştırma ve uygulamalar

4.4.8.1 Algoritmaların karşılaştırması

Dört algoritma aynı ağ üzerinde çalıştırıldığında farkları somutlaşır. Tablo 4.4.11, maksimum akışı 19 olan ikinci örnek ağdaki sonuçları özetler.

Algoritma	Temel işlem	Süre	Örnek ağda
Ford-Fulkerson	herhangi bir artış yolu (DFS)	$O(E \cdot \lvert f^* \rvert)$	5 artış yolu (Örnek 2'deki sıra)
Edmonds-Karp	en kısa artış yolu (BFS)	$O(V \cdot E^2)$	4 artış yolu, 5 BFS
Dinic	düzey çizgesi + engelleyici akış	$O(V^2 \cdot E)$	2 faz, 3 BFS
İtme–yeniden etiketleme	yerel itme / yeniden etiketleme	$O(V^2 \cdot E)$, FIFO $O(V^3)$	yol aramaz

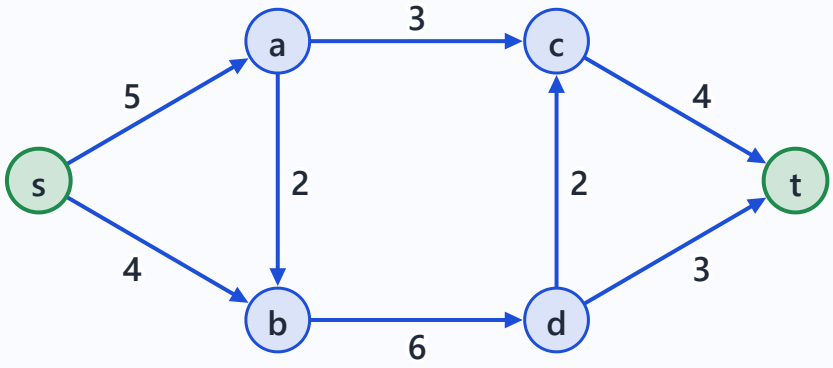
Tablo 4.4.11: Algoritmaların karşılaştırması. Son sütun $\lvert f^* \rvert = 19$ olan örnek ağdaki davranıştır.

Edmonds-Karp bu ağda şu yolları bulur: $S \to 1 \to 2 \to T$ (4), $S \to 1 \to 4 \to T$ (6), $S \to 3 \to 4 \to T$ (4) ve $S \to 3 \to 4 \to 2 \to T$ (5). İlk üç yol üç kenarlı, sonuncusu dört kenarlıdır. Beşinci BFS hedefe ulaşamaz. Dinic'in birinci fazı ilk üç yolu tek bir BFS'ten sonra işler; ikinci fazı dördüncü yolu bulur.

Seçim için kaba bir kural verilebilir. Kapasiteler küçük tam sayılarsa Ford-Fulkerson yeterlidir ve en kısa koddur. Genel amaçlı bir uygulama için Dinic iyi bir varsayılandır: kodu kısa, sınırı iyi, pratikte hızlıdır. Çok büyük ve yoğun ağlarda itme–yeniden etiketleme, sezgileriyle birlikte, genellikle en hızlısıdır. Bütün algoritmalar aynı $\lvert f^* \rvert$ değerini bulur. Akışın kenarlara dağılımı farklı olabilir. Minimum kesim her durumda son artık ağdan okunur.

4.4.8.2 Bir ağda bütün adımlar

Notta tanıtılan kavramların hepsi küçük bir ağda birlikte uygulanabilir.



Şekil 4.4.34: Problem ağı. Kapasiteler: $s \to a$ 5, $s \to b$ 4, $a \to b$ 2, $a \to c$ 3, $b \to d$ 6, $d \to c$ 2, $c \to t$ 4, $d \to t$ 3.

Örnek 4.4.6

Üst sınır. Kaynaktan çıkan kapasite $5 + 4 = 9$, hedefe giren $4 + 3 = 7$'dir. Öyleyse $|f^*| \le 7$.

Edmonds-Karp. BFS komşuları alfabetik dener. Birinci BFS'te kuyruk sırası a, b, c, d olur; c hedefi ilk bulur. Yol $s \to a \to c \to t$, darboğaz 3 ($a \to c$). İkinci BFS'te $a \to c$ doludur; $s \to b \to d \to t$ bulunur, darboğaz 3 ($d \to t$). Üçüncü BFS'te $d \to t$ doludur; $s \to b \to d \to c \to t$ bulunur. Darboğaz $\min(1, 3, 2, 1) = 1$'dir. İlk iki yol 3, sonuncusu 4 kenarlıdır. Toplam akış $3 + 3 + 1 = 7$.

Minimum kesim. Son artık ağda $s \to a$'da 2 birim vardır. $a \to b$ boştur; $b \to d$'de 2, $d \to c$'de 1 birim kalmıştır. Kaynaktan hedef dışındaki her düğüme ulaşılır: $S = \{s, a, b, c, d\}$. Kesim kenarları $c \to t$ (4) ve $d \to t$ (3); kapasite 7. Akış maksimumdur.

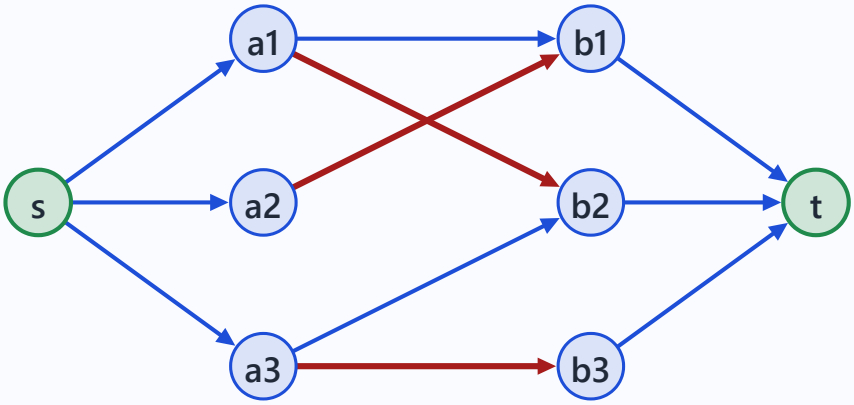
Artık kapasiteler. $c_f(s, a) = 5 - 3 = 2$. $c \to d$ özgün ağda bir kenar değildir; $d \to c$'de 1 birim akış olduğu için $c_f(c, d) = f(d, c) = 1$'dir (ters kenar).

İtme–yeniden etiketleme başlangıcı. Altı düğüm vardır: $h(s) = 6$, diğerleri 0. $s \to a$ ve $s \to b$ doldurulur: $e(a) = 5$, $e(b) = 4$, diğer aşırı akışlar 0.

4.4.8.3 Uygulamalar

Maksimum akışın gücü, ilk bakışta akışla ilgisi olmayan pek çok problemin bir akış ağına dönüştürülebilmesinden gelir. Dönüşüm yapıldıktan sonra notta anlatılan algoritmalardan herhangi biri kullanılır.

İki parçalı eşleştirme. Çalışanlar ve işler, öğrenciler ve projeler gibi iki grup düşünelim. Her çalışan yapabildiği işlerle bir kenarla bağlıdır. Her çalışan en çok bir iş, her iş en çok bir çalışan alabilir. En fazla sayıda eşleşme istenir. Bu problem bir akış ağına şöyle dönüştürülür: bir kaynak bütün çalışanlara, bütün işler bir hedefe bağlanır; çalışan–iş kenarları olduğu gibi kalır. Bütün kapasiteler 1'dir.



Şekil 4.4.35: İki parçalı eşleştirme ağı. Bütün kenarlar birim kapasitelidir. Vurgulu kenarlar en büyük eşleştirmedir: a_1–b_2, a_2–b_1, a_3–b_3.

Kaynaktan bir çalışana giden kenarın kapasitesi 1 olduğu için o çalışandan en fazla bir birim akış çıkar; çalışan en fazla bir işe atanır. İşten hedefe giden kenar da aynı sınırı işler için koyar. Tam sayılılık teoremi maksimum akışın her kenarda 0 ya da 1 olduğunu garanti eder. Akış taşıyan çalışan–iş kenarları bir eşleştirmedir ve akışın değeri eşleştirmenin büyüklüğüdür. Şekil 4.4.35'teki ağda a_1 çalışanı b_1 ve b_2'yi, a_2 yalnızca b_1'i, a_3 ise b_2 ve b_3'ü yapabilir. Maksimum akış 3'tür; herkes bir iş alır.

Kenar ayrık yollar. İki düğüm arasında ortak kenarı olmayan en fazla kaç yol bulunabilir? Bu soru ağların dayanıklılığıyla ilgilidir: kaç bağlantı koparsa iletişim kesilir? Her kenara kapasite 1 verilir; maksimum akış, kenar ayrık yolların sayısına eşittir. Menger teoremi aynı sayının, iki düğümü ayırmak için silinmesi gereken en az kenar sayısına eşit olduğunu söyler. Bu, maksimum akış minimum kesim teoreminin birim kapasiteli özel durumudur.

Düğüm kapasiteleri. Bazı problemlerde kısıt kenarlarda değil düğümlerdedir: bir kavşaktan saatte en fazla belli sayıda araç geçebilir. Her v düğümü iki düğüme bölünür: giriş v_g ve çıkış $v_ç$. Aralarına kapasitesi $c(v)$ olan bir $v_g \to v_ç$ kenarı konur. v'ye gelen kenarlar v_g'ye, v'den çıkanlar $v_ç$'den bağlanır. Düğümden geçen bütün akış bu tek kenarı kullanmak zorunda kalır.

Görüntü bölütleme. Bir fotoğrafta ön plan ile arka planı ayırmak için her piksel bir düğüm olur. Kaynak “ön plan”, hedef “arka plan”ı temsil eder. Bir pikselin ön plan olma olasılığı kaynaktan gelen kenarın, arka plan olma olasılığı hedefe giden kenarın kapasitesine dönüşür. Komşu pikseller arasındaki kenarlar, benzer piksellerin aynı tarafta kalmasını ödüllendirir. Minimum kesim, toplam “yanlış etiketleme maliyetini” en küçük yapan ayrımı verir.

Ağ güvenilirliği, ulaşım ve lojistik. Bir iletişim ağını ikiye bölmek için kesilmesi gereken en küçük toplam kapasite, bir minimum kesimdir. Boru hatlarında, trafik planlamada, havayolu çizelgelemede ve proje seçimi problemlerinde de maksimum akış ya da minimum kesim temel araçtır. Bu problemlerin çoğunda asıl iş, doğru ağı kurmaktır; algoritma hazırdır.

4.4.9 Sık yapılan hatalar

Ağ akışı problemlerinde yapılan hataların çoğu birkaç kavram karışıklığından doğar. Aşağıdaki liste bunları bir araya getirir.

- **Ters kenarları unutmak.** Yalnızca ileri kenarlarla aramak açgözlü bir algoritmadır ve maksimumu kaçırabilir; ikinci örnek ağda sonuç 19 yerine 18 çıkar.
- **Darboğazı kapasiteyle hesaplamak.** Darboğaz, kapasitelerin değil artık kapasitelerin en küçüğüdür: ileri kenarda $c - f$, ters kenarda f. İkinci artıştan sonra kenarların bir kısmı dolu olduğu için kapasiteye bakmak yanlış ve fazla bir darboğaz verir.
- **Minimum kesimi kenar sayısı sanmak.** Minimum kesim en az kenarı değil, en küçük kapasite toplamını keser.
- **T'den S'ye giden kenarları saymak.** Kesim kapasitesine yalnızca S'den T'ye giden kenarlar girer; geri yöndekiler eklenmez. Kesimden geçen *net akış* hesabında ise geri yöndeki akışlar çıkarılır. İki hesap birbirine karıştırılmamalıdır.

- **Akışın değerini kenar akışlarının toplamı sanmak.** Değer yalnızca kaynaktan çıkan net akıştır; aynı birim yol üzerindeki her kenarda yeniden sayılmaz.
- **Ford-Fulkerson'ı polinom sanmak.** $O(E \cdot |f^*|)$ kapasitelerin değerine bağlıdır. BFS ile çalışan Edmonds-Karp bu bağımlılığı kaldırır.
- **Edmonds-Karp'taki "en kısa"yı kapasiteyle ölçmek.** En kısa yol, kenar sayısı en az olan yoldur. Kapasitesi büyük yolu seçmek ayrı bir sezgidir ve Edmonds-Karp'ın sınırı onun için geçerli değildir.
- **Dinic'te düzey koşulunu gevşetmek.** DFS yalnızca $düzey(v) = düzey(u) + 1$ olan kenarları izlemelidir. Aynı düzeydeki kenarlara izin verilirse DFS döngüye girebilir ve faz sayısı sınırı geçersiz kalır.
- **İtme yönünü karıştırmak.** Akış yüksekten alçağa, yalnızca $h(u) = h(v) + 1$ iken itilir; başlangıçta $h(s) = |V|$ olmalıdır.
- **Minimum kesimi yanlış artık ağdan okumak.** Kesim, algoritma bittikten sonraki son artık ağda kaynaktan ulaşılabilen düğümlerden okunur. Ara bir adımın artık ağından okunan küme bir kesim verse de minimum olmayabilir.

4.4.10 Özet

- **Akış ağı:** kaynak, hedef ve kapasiteli kenarlardan oluşan yönlü çizge. Akış kapasite kısıtına ($0 \le f \le c$) ve ara düğümlerde korunuma (giren = çıkan) uyar. Değer $|f|$, kaynaktan çıkan net akıştır. Birden çok kaynak ve hedef, sonsuz kapasiteli kenarlarla bağlanan süper kaynak ve süper hedefle tek kaynaklı probleme dönüştürülür.
- **Artık ağ:** ileri kenarın artık kapasitesi $c - f$, ters kenarınki f'dir. Ters kenar, gönderilmiş akışın geri alınabilmesini sağlar. Artış yolu artık ağda bir s–t yoludur; darboğaz yoldaki en küçük artık kapasitedir. Yol boyunca darboğaz kadar artırma geçerli bir akış verir ve değeri darboğaz kadar büyütür.
- **Kesim:** $c(S, T)$, S'den T'ye giden kenarların kapasiteleri toplamıdır. Her kesimden geçen net akış $|f|$'ye eşittir; bu yüzden $|f| \le c(S, T)$. Maksimum akış minimum kesime eşittir. Minimum kesimin S tarafı, son artık ağda kaynaktan ulaşılabilen düğümlerdir.
- **Ford-Fulkerson:** herhangi bir artış yolunu kullanır. Tam sayı kapasitelerde sonlanır, tam sayı değerli bir akış bulur ve $O(E \cdot |f^*|)$ sürer. Kötü yol seçimi kapasite kadar artış gerektirebilir.
- **Edmonds-Karp:** BFS ile en kısa artış yolunu seçer. Uzaklıklar azalmaz, her kenar $O(V)$ kez kritik olur; $O(V \cdot E)$ artış ve $O(V \cdot E^2)$ süre.
- **Dinic:** her fazda BFS ile düzey çizgesi kurar, DFS ve kaldığı yer dizisiyle engelleyici akış bulur. En fazla $V - 1$ faz, faz başına $O(V \cdot E)$; toplam $O(V^2 \cdot E)$. Birim kapasitelerde çok daha hızlıdır.
- **İtme–yeniden etiketleme:** $h(s) = |V|$ ile başlar, kaynak kenarlarını doldurarak bir ön akış kurar. Akış yüksekten alçağa itilir; itilemiyorsa düğüm yeniden etiketlenir. Aktif düğüm kalmayınca ön akış maksimum akıştır. Genel sürüm $O(V^2 \cdot E)$, FIFO sürümü $O(V^3)$.
- **Uygulamalar:** iki parçalı eşleştirme, kenar ayrık yollar, düğüm kapasiteli ağlar, görüntü bölütleme ve ağ güvenilirliği maksimum akış ya da minimum kesim olarak modellenir.

4.4.11 Kavram listesi

akış ağı *flow network* · kaynak / hedef *source / sink* · kapasite *capacity* · akış korunumu *flow conservation* · maksimum akış *maximum flow* · artık ağ *residual network* · artık kapasite *residual capacity* · artış yolu *augmenting path* · darboğaz *bottleneck* · kesim *cut (s-t cut)* · maksimum akış minimum kesim teoremi *max-flow min-cut theorem* · Ford-Fulkerson yöntemi *Ford-Fulkerson method* · Edmonds-Karp algoritması *Edmonds-Karp algorithm* · Dinic algoritması *Dinic's algorithm* · düzey çizgesi *level graph* · engelleyici akış *blocking flow* · itme–yeniden etiketleme *push-relabel* · aşırı akış *excess flow* · iki parçalı eşleştirme *bipartite matching*

4.4.12 Kendini Dene

1. Şekil 4.4.2'deki akışta $B \to C$ (2/3) ve $C \to D$ (5/6) kenarları artık ağda hangi kenarları ve kapasiteleri üretir? Bu akış için artık ağda bir artış yolu var mıdır?

   Cevap

   $B \to C$: ileri $c_f(B, C) = 3 - 2 = 1$, ters $c_f(C, B) = 2$. $C \to D$: ileri $c_f(C, D) = 1$, ters $c_f(D, C) = 5$. Artış yolu yoktur: kaynaktan çıkan iki kenar doymuştur ve artık ağda S'den çıkan kenar kalmaz.

2. İkinci örnek ağda (Şekil 4.4.10) $S = \{S, 1, 3\}$ kesiminin kapasitesi kaçtır? Bu kesim neden minimum değildir?

   Cevap

   S'den T'ye giden kenarlar $1 \to 2$ (4), $1 \to 4$ (8) ve $3 \to 4$ (9)'dur: $4 + 8 + 9 = 21$. $S \to 1$, $S \to 3$ ve $1 \to 3$ kenarlarının iki ucu da S tarafındadır. Maksimum akış 19 olduğundan kapasitesi 21 olan kesim minimum olamaz; minimum kesim $\{S, 3\}$'tür (19).

3. Edmonds-Karp birinci örnek ağda (Şekil 4.4.1) hangi yolları bulur? Kaç artış ve kaç BFS yapar?

   Cevap

   $S \to A \to B \to T$ (2), $S \to C \to D \to T$ (3) ve $S \to A \to B \to C \to D \to T$ (2). İlk iki yol 3, üçüncüsü 5 kenarlıdır. 3 artış yapılır; dördüncü BFS hedefe ulaşamaz. Toplam 4 BFS, maksimum akış 7.

4. Şekil 4.4.17'deki ağda komşuları indis sırasıyla ($s, 1, 2, 3, 4, t$) deneyen DFS'li Ford-Fulkerson hangi

yolları bulur? Edmonds-Karp aynı ağda kaç artış yapar?

Cevap

Altı artış yapılır: $s→1→2→4→3→t$ (7), $s→1→2→4→t$ (3), $s→1→3→2→4→t$ (1), $s→1→3→t$ (5), $s→2→1→3→t$ (6), $s→2→3→t$ (1). Akış *7, 10, 11, 16, 22, 23* diye büyür. Beşinci yoldaki $2 → 1$ özgün bir kenardır; altıncı yoldaki $2 → 3$ ise $3 → 2$'nin ters kenarıdır. Edmonds-Karp aynı ağda 3 artışla 23'e ulaşır.

5. Dinic algoritması birinci örnek ağda kaç faz yapar? Her fazdaki düzeyler ve engelleyici akış nedir?

Cevap

Faz 1: düzeyler $S = 0$; $A, C = 1$; $B, D = 2$; $T = 3$. $B → C$ düzey geri gittiği için kullanılmaz. Engelleyici akış $S → A → B → T$ (2) ve $S → C → D → T$ (3); $|f| = 5$. Faz 2: düzeyler $S = 0$, $A = 1$, $B = 2$, $C = 3$, $D = 4$, $T = 5$; yol $S → A → B → C → D → T$ (2), $|f| = 7$. Üçüncü BFS hedefe ulaşamaz. 2 faz, 3 BFS.

6. Birinci örnek ağda $B → T$ kenarının kapasitesi 2'den 5'e çıkarılırsa maksimum akış değişir mi? Dinic kaç fazda biter?

Cevap

Değişmez; 7 kalır. *({S}, diğerleri)* kesiminin kapasitesi hâlâ $4 + 3 = 7$'dir ve bu kenar o kesimde yer almaz. Dinic tek fazda biter: düzey çizgesinde $S → A → B → T$ 4, $S → C → D → T$ 3 birim taşır. İkinci BFS hedefe ulaşamaz.

7. Birinci örnek ağda B düğümünden en fazla 3 birim akış geçebiliyorsa problem nasıl modellenir ve maksimum akış kaç olur?

Cevap

B, B_g ve $B_ç$ düğümlerine bölünür; aralarına kapasitesi 3 olan bir kenar konur. $A → B_g$ (4), $B_ç → C$ (3), $B_ç → T$ (2) olur. Maksimum akış 6'dır: örneğin B üzerinden 3 birim (2'si doğrudan hedefe, 1'i C'ye), $S → C$ üzerinden 3 birim. Minimum kesim $S → C$ (3) ile $B_g → B_ç$ (3) kenarlarından oluşur: 6.

8. İkinci örnek ağda S'den T'ye en fazla kaç kenar ayrık yol vardır? Bu sayıya maksimum akışla nasıl ulaşılır?

Cevap

Bütün kapasiteler 1 yapılıp maksimum akış hesaplanır. Sonuç 2'dir; örneğin $S → 1 → 2 → T$ ve $S → 3 → 4 → T$. Kaynaktan yalnızca iki kenar çıktığı için ({S} kesiminin kapasitesi 2) üçüncü bir kenar ayrık yol olamaz.

9. İtme–yeniden etiketleme birinci örnek ağda başlatılıyor. Başlangıçtaki yükseklikler ve aşırı akışlar nedir? A aktif düğüm olarak seçilirse ilk iki işlem ne olur?

Cevap

$h(S) = 6$, diğer bütün düğümlerde $h = 0$. $S \to A$ ve $S \to C$ doldurulur: $e(A) = 4$, $e(C) = 3$. A'nın artık komşuları B ($h = 0$) ve S (ters kenar, $h = 6$)'dir; hiçbiri bir basamak aşağıda değildir. Önce yeniden etiketleme: $h(A) = 1 + \min(0, 6) = 1$. Sonra itme: $\delta = \min(e(A), c_f(A, B)) = \min(4, 4) = 4$; $e(A) = 0$, $e(B) = 4$.

4.4.13 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 24: akış ağlarını, maksimum akış minimum kesim teoreminin ispatını, Edmonds-Karp analizini ve iki parçalı eşleştirmeyi ayrıntılı ispatlarla verir; notun Edmonds-Karp izleme ağı bu kitaptandır.
- J. Kleinberg ve É. Tardos, *Algorithm Design* (2006), Bölüm 7: maksimum akışı modelleme aracı olarak ele alır; görüntü bölütleme, proje seçimi ve havayolu çizelgeleme gibi uygulamaları çözülmüş örneklerle anlatır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 6.4: Ford-Fulkerson'ın en kısa yol sürümünü kenar nesneleriyle Java'da kodlar ve artık ağı nesne yönelimli bir tasarımla gösterir.
- A. V. Goldberg ve R. E. Tarjan, "A New Approach to the Maximum-Flow Problem" (1988): itme–yeniden etiketleme algoritmasını, değişmezlerini ve FIFO sürümünün $O(V^3)$ analizini ilk kaynağından verir.

4.4.14 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 24: Maximum Flow).
2. E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation", *Soviet Math. Doklady* 11, 1970.
3. J. Edmonds, R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM* 19(2), 1972.
4. L. R. Ford, D. R. Fulkerson, "Maximal Flow Through a Network", *Canadian Journal of Mathematics* 8, 1956.
5. A. V. Goldberg, R. E. Tarjan, "A New Approach to the Maximum-Flow Problem", *Journal of the ACM* 35(4), 1988.
6. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2006 (Bölüm 7: Network Flow).
7. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 6.4: Maximum Flow).

KISIM II · BÖLÜM 5

Dizgi Algoritmaları

Dizgi algoritmaları metin arama, veri sıkıştırma ve biyolojik dizi çözümlemesi gibi alanların temelini oluşturur. Bölümün ilk kısmında kaba kuvvet, Knuth-Morris-Pratt, Boyer-Moore ve Rabin-Karp örüntü eşleştirme algoritmaları ile RLE, LZW ve LZ78 sıkıştırma yöntemleri işlenir. İkinci kısım sözlüksel sıra ve LSD/MSD basamak sıralamasıyla dizgi sıralamayı, düzenli ifadeleri, sonlu durum makinelerini, Levenshtein düzenleme uzaklığını, sonek dizisini ve Burrows–Wheeler dönüşümünü kapsar. Üçüncü kısımda önek ağacı (trie) ekleme, arama, silme ve otomatik tamamlama işlemleriyle anlatılır; üçlü arama trie'si, sıkıştırılmış trie, sonek trie'si ve Aho–Corasick algoritması da tanıtılır.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum05/src`

BU BÖLÜMÜN NOTLARI

5.1 Dizgi Eşleştirme ve Sıkıştırma

5.2 İleri Dizgi Algoritmaları

5.3 Trie Veri Yapısı

KISIM II · BÖLÜM 5.1

Dizgi Eşleştirme ve Sıkıştırma

Bir metin düzenleyicide Ctrl+F ile kelime aramak, virüs tarayıcısının bilinen imzaları bulması ve DNA dizisinde gen aranması aynı problemin örnekleridir. Üçünde de kısa bir dizginin uzun bir metin içindeki yeri bulunur. Not bu problemin dört klasik çözümünü ele alır. Kaba kuvvet algoritması her konumu tek tek dener. Knuth-Morris-Pratt algoritması eşleşen kısımdan öğrendiğini bir tabloda saklar ve metinde hiç geri gitmez. Boyer-Moore algoritması örüntüyü sağdan sola karşılaştırır ve metnin büyük bölümünü hiç okumadan atlar. Rabin-Karp algoritması ise karakterler yerine hash değerlerini karşılaştırır. Notun ikinci bölümü dizgi sıkıştırmadır. Çalışma uzunluğu kodlaması (RLE) ardışık tekrarları sayıyla yazar. Lempel-Ziv-Welch (LZW) yöntemi tekrar eden dizgileri, kendi kendine büyüyen bir sözlüğün kısa kodlarıyla değiştirir. Her algoritma bir örnek üzerinde adım adım çalıştırılır, Java kodu satır satır izlenir ve maliyeti gerekçesiyle verilir.

ÖĞRENME ÇIKTILARI

Kavramlar
Metin, örüntü, kayma, önek ve sonek kavramlarını tanımlamak; dizgi eşleştirme problemini biçimsel olarak ifade etmek.

Eşleştirme
Kaba kuvvet, KMP, Boyer-Moore ve Rabin-Karp algoritmalarını bir örnek üzerinde adım adım çalıştırmak ve Java kodlarını izlemek.

Analiz
LPS tablosunun, kötü karakter tablosunun ve kayan hash'in zaman kazancını açıklamak; en iyi, ortalama ve en kötü durumları karşılaştırmak ve probleme göre algoritma seçmek.

Sıkıştırma
RLE ve LZW ile kayıpsız sıkıştırma yapmak, sıkıştırılmış veriyi geri açmak, LZW'nin özel durumunu çözmek ve sıkıştırma oranını hesaplamak.

5.1.1 Dizgi eşleştirme problemi

Bilgisayarların işlediği verinin büyük bölümü metindir: e-postalar, iletiler, sosyal medya paylaşımları, haber metinleri, program kaynak kodları, kayıt dosyaları. Bu metinler üzerinde en sık yapılan işlem aramadır. Bir kelimenin geçtiği yerler bulunur, bir ifade başka bir ifadeyle değiştirilir ya da iki metnin benzerliği ölçülür. Bu tür işlemleri yapan algoritmalara **dizgi algoritmaları** (*string algorithms*) denir. Bölümün iki ana konusu vardır: bir örüntüyü metinde bulmak (dizgi eşleştirme) ve metni daha az yerde saklamak (dizgi sıkıştırma). Bu kesim eşleştirme problemini tanımlar ve kullanılacak gösterimi kurar.

5.1.1.1 Metin arama her yerde

Dizgi eşleştirme günlük yazılımların içine gömülüdür. Metin düzenleyicilerdeki Ctrl+F araması, Unix'in `grep` komutu ve arama motorlarının dizinleri bunun en bilinen örnekleridir. Daha az görünen kullanımlar da vardır. Virüs tarayıcıları dosyalarda bilinen kötü amaçlı yazılımların bayt dizilerini (imzalarını) arar. İntihal tespit araçları iki belgede ortak geçen uzun parçaları bulur. Biyoinformatikte bir genin ya da kısa bir DNA parçasının milyonlarca harflik bir dizide nerede geçtiği sorulur.

Sıkıştırma da aynı ölçüde yaygındır. ZIP arşivleri, GIF ve PNG resimleri, PDF belgeleri ve faks makineleri veriyi daha az bitle saklamak için dizgi algoritmalarından yararlanır. Bu yöntemlerin çoğu da özünde bir aramadır: daha önce görülmüş bir parçanın tekrarını bulup onu kısa bir göstergeyle değiştirir.

Bu uygulamalarda metinler büyüktür. Bir kayıt dosyası gigabaytlarca, bir insan genomu yaklaşık üç milyar harf uzunluğundadır. Bu yüzden algoritmanın her karakteri kaç kez okuduğu doğrudan çalışma süresini belirler. Bölümdeki algoritmaların hepsi aynı soruyu yanıtlar. Farkları, bu soruyu kaç karşılaştırmayla yanıtladıklarıdır.

5.1.1.2 Temel kavramlar ve gösterim

Dizgiler bir karakter kümesinden oluşur. Bu kümeye **alfabe** (*alphabet*) denir ve Σ ile gösterilir. İngilizce küçük harfler için $\Sigma = \{a, b, \ldots, z\}$, DNA için $\Sigma = \{A, C, G, T\}$, ikili veri için $\Sigma = \{0, 1\}$'dir. Alfabenin büyüklüğü $\sigma = |\Sigma|$ ile yazılır. Bazı algoritmaların hızı bu sayıya bağlıdır.

Aramanın yapıldığı uzun dizgiye **metin** (*text*), aranan kısa dizgiye **örüntü** (*pattern*) denir. Metin $T[0..n-1]$, örüntü $P[0..m-1]$ biçiminde yazılır. Böylece n metnin, m örüntünün uzunluğudur ve genellikle $m \le n$'dir. İndisler Java'daki gibi 0'dan başlar. Kodlarda metin `t`, örüntü `p` adıyla geçer; `t.charAt(i)`, $T[i]$'nin karşılığıdır.

Tanım 5.1.1

Örüntünün ilk karakteri metnin s konumunun altına gelecek biçimde hizalanmasına **kayma** (*shift*) denir. $T[s..s+m-1] = P$ ise P, T içinde s kaymasında **geçer** ve s **geçerli bir kaymadır**. Örüntü metnin dışına taşamayacağı için olası kaymalar $0 \le s \le n - m$ aralığındadır; toplam $n - m + 1$ kayma vardır.

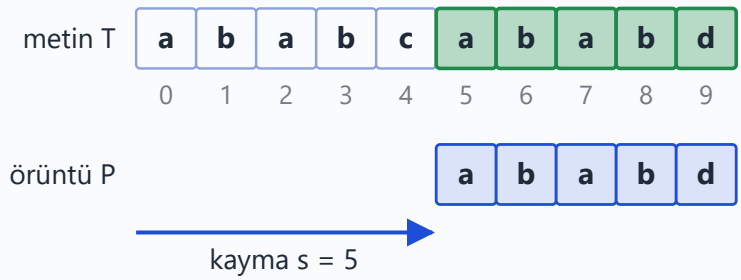
Dizgi eşleştirme (*string matching*) problemi, P'nin T içinde geçtiği bütün kaymaları (ya da yalnız ilkini) bulmaktır.

Algoritmaların çoğu, örüntünün baş ve son parçaları üzerine kuruludur. Bir dizginin başından alınan parçaya **önek** (*prefix*), sonundan alınan parçaya **sonek** (*suffix*) denir. P'nin k uzunluklu öneki $P[0..k-1]$, k uzunluklu soneki $P[m-k..m-1]$'dir. Dizginin kendisi de hem öneki hem sonekidir. Dizginin kendisine eşit olmayan öneke **uygun önek** (*proper prefix*), kendisine eşit olmayan sonek ise **uygun sonek** adını alır. Kaynaklarda "öz önek" adı da kullanılır.

Örnek 5.1.1

P = "ababd" ($m = 5$) olsun. Uygun önekleri a, ab, aba ve abab; uygun sonekleri d, bd, abd ve babd'dir. Boş dizgi de hem önek hem sonek sayılabilir, ama sorularda çoğunlukla işe yaramaz. Bu örüntüde hem uygun önek hem uygun sonek olan boş olmayan bir parça yoktur. Bütün sonekler d ile biter, hiçbir uygun önek ise d ile bitmez.

Aynı örüntü T = "ababcababd" ($n = 10$) metninde aranırsa olası kaymalar $0, 1, \ldots, 5$'tir. Yalnız $s = 5$'te $T[5..9]$ = "ababd" = P olur. Geçerli tek kayma 5'tir.



Şekil 5.1.1: Metin T = "ababcababd" ve örüntü P = "ababd". Örüntü $s = 5$ kaymasında metinle bütünüyle eşleşir; daha küçük kaymalarda en az bir karakter uyuşmaz.

Karakter karşılaştırmasının sonucu iki türlüdür. Karakterler aynıysa **eşleşme**, farklıysa **uyuşmazlık** (*mismatch*) vardır. Notun şekillerinde eşleşen karakterler yeşil, uyuşmayan karakterler kırmızı kutuyla gösterilir. Algoritmaların maliyeti çoğunlukla toplam karakter karşılaştırma sayısıyla ölçülür. Bu sayı, makineden ve dilden bağımsız, sayılabilir bir büyüklüktür.

5.1.1.3 Algoritmaların haritası

Bölümde altı algoritma ele alınır. İlk dördü eşleştirme, son ikisi sıkıştırma algoritmasıdır:

- **Kaba kuvvet** (*brute force, naive*): metindeki her konum, örüntüyle eşleşip eşleşmediği için ayrı ayrı denetlenir. En fazla karşılaştırmayı bu yöntem yapabilir.
- **Knuth-Morris-Pratt (KMP)**: aramadan önce örüntüden bir tablo (LPS tablosu) kurulur. Arama sırasında metinde hiç geri gidilmez; en kötü durumda bile süre doğrusaldır.
- **Boyer-Moore**: karşılaştırma sağdan sola yapılır. Kötü karakter ve iyi sonek kuralları uzun atlamalar sağlar; uzun örüntülerde çok etkilidir.
- **Rabin-Karp**: karakterler yerine hash (özet) değerleri karşılaştırılır. Pencere kaydıkça hash *O(1)* sürede güncellenir; algoritma beklenen durumda hızlıdır.
- **Çalışma uzunluğu kodlaması** (*run-length encoding, RLE*): ardışık tekrar eden değerler tek değer ve bir tekrar sayısıyla saklanır.
- **Lempel-Ziv-Welch (LZW)**: sözlük tabanlı sıkıştırmadır. Tekrar eden dizgiler, sıkıştırma sırasında büyüyen bir sözlüğün kısa kodlarıyla temsil edilir. GIF biçimi bu yöntemi kullanır.

Eşleştirme algoritmalarının hepsi aynı çıktıyı üretir. Aralarındaki fark, uyuşmazlık anında ne yaptıklarıdır. Kaba kuvvet uyuşmazlıktan hiçbir şey öğrenmez ve örüntüyü bir konum kaydırır. KMP, eşleşmiş parçanın yapısından yararlanarak gereksiz kaymaları atlar. Boyer-Moore, uyuşmayan metin karakterinin örüntüde nerede geçtiğine bakar. Rabin-Karp ise karşılaştırmanın kendisini ucuzlatır. Bu dört bakış açısı, aynı problemin neden birden çok çözümü olduğunu da açıklar.

5.1.2 Kaba kuvvet algoritması

Kaba kuvvet algoritması, metin içinde bir örüntü arayan en basit yöntemdir. İngilizcedeki “naive” (saf) adı da bu sadelikten gelir. Algoritma hiçbir ön hazırlık yapmaz ve hiçbir ek bellek kullanmaz. Bu yüzden hem öbür algoritmaların karşılaştırıldığı ölçüttür hem de kısa örüntülerde hâlâ kullanılan pratik bir yöntemdir.

5.1.2.1 İşleyiş

Algoritma, olası her kaymayı soldan sağa sırayla dener:

1. **Hizalama.** *s = 0, 1, …, n − m* kaymalarının her biri için örüntü metnin altına, ilk karakteri *T[s]*'nin altına gelecek biçimde yerleştirilir.
2. **Karşılaştırma.** *P[0]* ile *T[s]*, *P[1]* ile *T[s+1]* diye soldan sağa karşılaştırılır. İlk uyuşmazlıkta bu hizanın denetimi biter.
3. **Karar.** *m* karakterin hepsi eşleşmişse *s* rapor edilir. Aksi hâlde örüntü bir konum sağa kaydırılır ve bir sonraki kayma denenir.

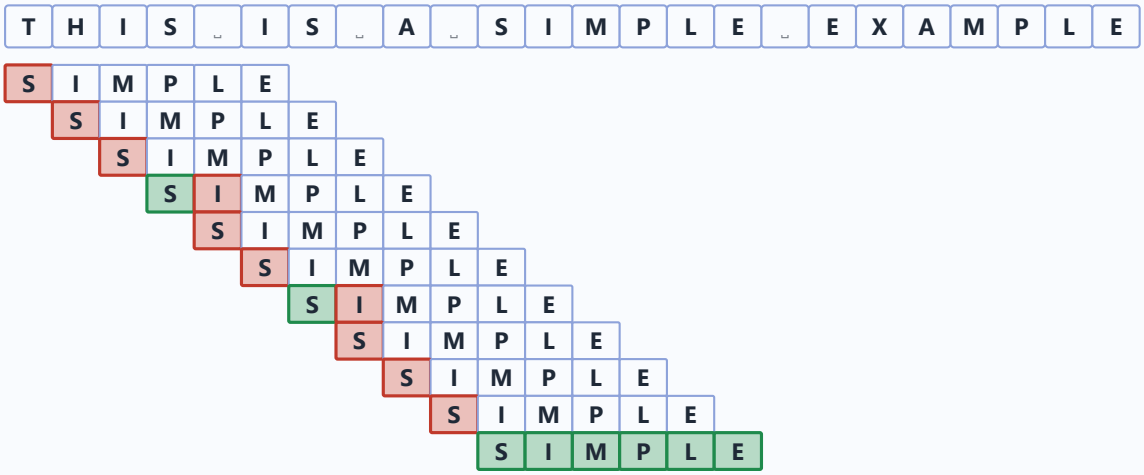
Karşılaştırma sayısı metne ve örüntüye bağlıdır. En kötü durumda her hizada m karakterin hepsi karşılaştırılır: $(n - m + 1) \cdot m$ karşılaştırma, yani $O(n \cdot m)$. En iyi durumda her hizada ilk karakter uyuşmaz: $n - m + 1$ karşılaştırma, yani $O(n)$.



Şekil 5.1.2: Kaba kuvvetin en kötü durumu. "AAAAAA" metninde "AAB" aranırken her hizada ilk iki karakter eşleşir, uyuşmazlık hep son karakterde çıkar: 4 hiza × 3 = 12 karşılaştırma.

5.1.2.2 Örnek: SIMPLE araması

Şekil 5.1.3 "THIS IS A SIMPLE EXAMPLE" metninde ($n = 24$) "SIMPLE" örüntüsünün ($m = 6$) aranışını gösterir. Her satır bir kaymadır. Boşluk karakteri ␣ işaretiyle gösterilmiştir.



Şekil 5.1.3: Kaba kuvvetle SIMPLE araması. Her satır bir kaymadır; yeşil kutular eşleşen, kırmızı kutular uyuşmayan karşılaştırmalardır. Örüntü 11. hizada ($s = 10$) bulunur.

Algoritma $s = 0$'dan başlar. $s = 0, 1, \ldots, 9$ kaymalarında ya örüntünün ilk karakteri S metindeki karakterle uyuşmaz ya da yalnız S eşleşir. $s = 3$'te $T[3]$ = S eşleşir, ardından $T[4]$ = ␣ ile I uyuşmaz. $s = 6$'da da aynı şey olur: S eşleşir, ␣ uyuşmaz. Diğer kaymaların hepsi ilk karşılaştırmada biter. $s = 10$'da altı karakterin altısı da eşleşir. Tablo 5.1.1 karşılaştırmaları kayma kayma sayar.

Kayma *s*	Hizalanan metin parçası	Sonuç	Karşılaştırma
0, 1, 2	THI…, HIS…, IS␣…	ilk karakter ≠ S	3 × 1 = 3
3	S␣IS␣A	S eşleşir, ␣ ≠ I	2
4, 5	␣IS…, IS␣…	ilk karakter ≠ S	2 × 1 = 2
6	S␣A␣SI	S eşleşir, ␣ ≠ I	2
7, 8, 9	␣A…, A␣…, ␣SI…	ilk karakter ≠ S	3 × 1 = 3
10	SIMPLE	6 karakter eşleşir: bulundu	6
Toplam (11 hiza)			18

Tablo 5.1.1: SIMPLE aramasında kaymalar ve karşılaştırma sayıları.

Toplam 18 karşılaştırma, metnin uzunluğuna (24) yakındır. Bunun nedeni, doğal dil metninde iki rastgele karakterin çoğunlukla farklı olmasıdır. Uyuşmazlık genellikle ilk karakterde çıkar. Kaba kuvvetin pratikte çoğu zaman kabul edilebilir hızda çalışması bu gözleme dayanır.

5.1.2.3 Java kodu ve izleme

Algoritmanın Java kodu iki iç içe döngüden oluşur. Dış döngü kaymaları, iç döngü bir hizadaki karakterler üzerinde gezinir. Metot ilk geçişin konumunu döndürür; örüntü yoksa −1 döner.

```
int kabaKuvvet(String t, String p) {
  int n = t.length(), m = p.length();
  for (int i = 0; i <= n - m; i++) {
    int j = 0;
    while (j < m &&
      t.charAt(i + j) == p.charAt(j))
      j++;
    if (j == m) return i;
  }
  return -1;
}
```

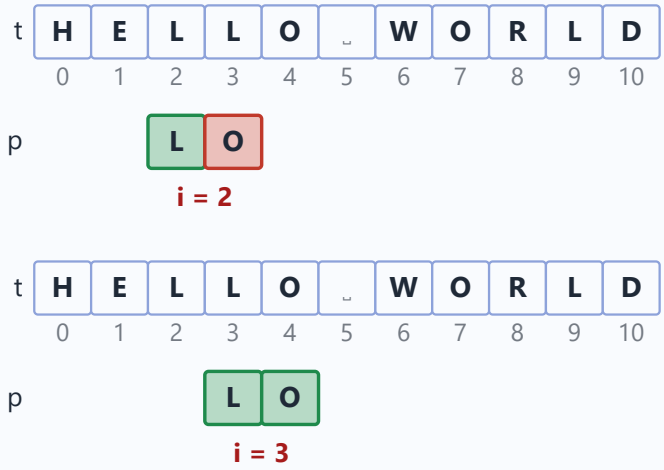
Kod satır satır şöyle okunur:

- **Dış döngü:** `i` kaymadır ve 0'dan `n - m`'ye kadar artar. Koşul `i <= n - m` biçimindedir, çünkü son geçerli kayma *n* − *m*'dir. `i + j` en çok *n* − *m* + *m* − *1* = *n* − *1* olur; dizinin dışına çıkılmaz.
- **İç döngü:** `j`, o hizada eşleşen karakter sayısıdır. `j < m` ve karakterler eşit olduğu sürece `j` artar. İki koşulun sırası önemlidir: `&&` işleci ikinci koşulu ancak birincisi doğruysa değerlendirir. Böylece `j == m` olduğunda `p.charAt(m)` çağrılmaz.
- **Karar:** iç döngü `j == m` ile bittiyse bütün karakterler eşleşmiştir ve `i` döndürülür. Döngü daha erken bittiyse bir uyuşmazlık vardır ve dış döngü bir sonraki kaymaya geçer.

Kodu `kabaKuvvet("HELLO WORLD", "LO")` çağrısıyla izleyelim. *n* = *11*, *m* = *2*'dir; kaymalar *i* = *0, 1, …, 9*'dur. Tablo 5.1.2 her karşılaştırmayı sırasıyla verir.

Adım	`i`	`j`	Karşılaştırma	Sonuç
1	0	0	`t[0]` = 'H' ile `p[0]` = 'L'	uyuşmaz; örüntü 1 kayar
2	1	0	`t[1]` = 'E' ile `p[0]` = 'L'	uyuşmaz; örüntü 1 kayar
3	2	0	`t[2]` = 'L' ile `p[0]` = 'L'	eşleşir; `j = 1`
4	2	1	`t[3]` = 'L' ile `p[1]` = 'O'	uyuşmaz; örüntü 1 kayar
5	3	0	`t[3]` = 'L' ile `p[0]` = 'L'	eşleşir; `j = 1`
6	3	1	`t[4]` = 'O' ile `p[1]` = 'O'	eşleşir; `j = 2 = m`
7	3	2	—	`j == m` : 3 döndürülür

Tablo 5.1.2: `kabaKuvvet("HELLO WORLD", "LO")` izlemesi.



Şekil 5.1.4: İzlemenin iki anı. Solda *i = 2*: L eşleşmiş, ikinci karakterde L ile O uyuşmamıştır. Sağda *i = 3*: iki karakter de eşleşmiş ve örüntü 3 konumunda bulunmuştur.

Toplam 6 karşılaştırma yapılır. 4. ve 5. adımlara dikkat: *T[3]* = 'L' iki kez okunur. İlkinde örüntünün ikinci karakteriyle, sonra birinci karakteriyle karşılaştırılır. Kaba kuvvetin temel kusuru budur: örüntü kaydığında metinde geri gidilir ve aynı karakterler yeniden okunur.

Bellek maliyeti *O(1)*'dir; yalnız `i`, `j`, `n` ve `m` tutulur. Bütün geçişler isteniyorsa `return i` yerine `i` bir listeye eklenir ve döngü sürer; karmaşıklık değişmez.

Sık yapılan hata

Dış döngünün koşulunu `i < n - m` yazmak. Bu koşul son kaymayı ($s = n - m$) atlar. Örneğin `kabaKuvvet("kalem", "em")` çağrısında doğru kaymalar 0, 1, 2, 3'tür. Hatalı koşul yalnız 0, 1, 2'yi dener. "em" yalnız *s = 3*'te geçtiği için bulunamaz ve metot −1 döndürür. Örüntü metnin sonunda değilse hata görünmez; bu yüzden testlerden kolayca kaçar.

5.1.2.4 En iyi, ortalama ve en kötü durum

Kaba kuvvetin karşılaştırma sayısı, her hizada kaç karakterin eşleştiğine bağlıdır. Tablo 5.1.3 üç durumu özetler.

Durum	Ne zaman?	Karşılaştırma	Zaman
En iyi	$P[0]$ metnin hiçbir karakteriyle eşleşmez (ör. P = "xyz", T = "aaaa...")	$n - m + 1$	$O(n)$
Ortalama	Rastgele metin, büyük alfabe: uyuşmazlık genellikle ilk bir iki karakterde çıkar	$\approx (n - m + 1) \cdot \sigma/(\sigma - 1)$	$O(n)$
En kötü	T = "aaaaaaaaaa", P = "aaab": her hizada m karşılaştırma	$(n - m + 1) \cdot m$	$O(n \cdot m)$

Tablo 5.1.3: Kaba kuvvetin en iyi, ortalama ve en kötü durumu.

Ortalama durumdaki $\sigma/(\sigma - 1)$ çarpanı basit bir olasılık hesabından gelir. Metnin karakterleri alfabeden bağımsız ve eşit olasılıkla seçilmiş olsun. Bir hizada ilk karşılaştırma her zaman yapılır. İkinci karşılaştırma ancak ilk karakter eşleşirse yapılır; bunun olasılığı $1/\sigma$'dır. Üçüncüsü için ilk iki karakterin eşleşmesi gerekir: olasılık $1/\sigma^2$. Bir hizadaki beklenen karşılaştırma sayısı bu yüzden en çok $1 + 1/\sigma + 1/\sigma^2 + \ldots = \sigma/(\sigma - 1)$'dir. İngilizce harflerde ($\sigma = 26$) bu sayı yaklaşık 1,04'tür; ikili alfabede ($\sigma = 2$) 2'dir. İki durumda da hiza başına iş sabittir ve toplam süre $O(n)$ olur.

Örnek 5.1.2

T = "aaaaaaaaaa" ($n = 10$) ve P = "aaab" ($m = 4$) için kaba kuvvetin karşılaştırma sayısı bulunacak.

1. **Hiza sayısı.** Kaymalar $0, 1, \ldots, 6$'dır: $n - m + 1 = 7$ hiza.
2. **Hiza başına iş.** Her hizada örüntünün ilk üç karakteri "aaa" metindeki a'larla eşleşir. Dördüncü karakter b, metindeki a ile uyuşmaz. Her hizada 4 karşılaştırma yapılır.
3. **Toplam.** $7 \times 4 = 28$ karşılaştırma. Uyuşmazlık hep son karakterde çıkar ve örüntü metinde geçmez.

Aynı yapı $n = 12$, $m = 4$ için $9 \times 4 = 36$ karşılaştırma verir. Genel formül $(n - m + 1) \cdot m$'dir. $m = n/2$ alınırsa bu sayı yaklaşık $n^2/4$ olur: süre metin uzunluğunun karesiyle büyür.

En kötü durumda neler olduğu dikkatle incelenmeye değer. $s = 0$'da ilk üç karakter eşleşmiştir. Bu bilgi, metnin $T[0..2]$ = "aaa" olduğunu söyler. $s = 1$'e geçildiğinde algoritma $T[1]$ ve $T[2]$'yi yeniden okur, oysa bu karakterler zaten biliniyordu. Kaba kuvvetin asıl kusuru buradadır: uyuşmazlıkta öğrenilen bilgi atılır, metinde geri gidilir ve aynı karakterler tekrar karşılaştırılır. KMP algoritması tam olarak bu bilgiyi saklayıp kullanır.

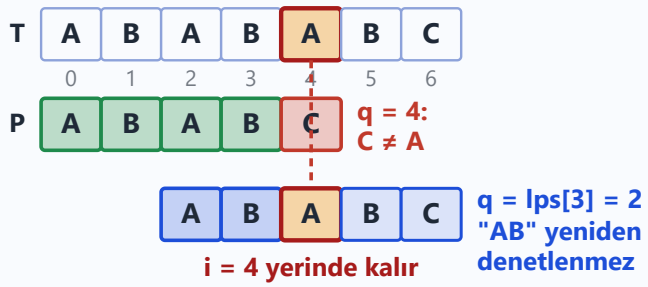
5.1.3 Knuth-Morris-Pratt algoritması

Knuth-Morris-Pratt (KMP) algoritması, kaba kuvvetin attığı bilgiyi saklayan ilk doğrusal zamanlı eşleştirme algoritmasıdır. James H. Morris ve Vaughan Pratt algoritmayı 1970'te birlikte buldu. Donald Knuth da aynı fikre otomatlar kuramından bağımsız olarak ulaştı. Üçü sonuçlarını 1977'de ortak bir makalede yayımladı. Algoritmanın adı bu üç kişinin soyadlarından gelir.

5.1.3.1 Eşleşen kısmı unutmamak

KMP'nin temel fikri şudur: uyuşmazlık anında örüntünün zaten eşleşmiş kısmı hakkında bilinenler kullanılır. Bir hizada *P[0..q−1]* metinle eşleşmiş ve *P[q]* uyuşmamış olsun. O zaman metnin son *q* karakteri tam olarak *P[0..q−1]*'dir. Bu karakterleri yeniden okumaya gerek yoktur; örüntünün kendisine bakarak ne oldukları bilinir.

Bu bilgi iki şey kazandırır. Birincisi, metin işaretçisi hiç geri gitmez. Metin soldan sağa bir kez okunur. İkincisi, kesinlikle eşleşemeyecek hizalar hiç denenmez. Örüntü, eşleşmiş parçanın yapısının izin verdiği en uzak konuma doğrudan kaydırılır.



Şekil 5.1.5: *T* = "ABABABC" metninde *P* = "ABABC" aranıyor. "ABAB" eşleşmiş (*q = 4*), C ile A uyuşmamıştır. Eşleşmiş "ABAB"ın sonundaki "AB" aynı zamanda öneki olduğu için örüntü *q = lps[3] = 2* konumundan devam eder. "AB" yeniden denetlenmez ve metin işaretçisi *i = 4* yerinde kalır.

Şekil 5.1.5 bu fikri küçük bir örnekle gösterir. Metin "ABABABC", örüntü "ABABC" olsun. *s = 0*'da ilk dört karakter "ABAB" eşleşir, *T[4]* = A ile *P[4]* = C uyuşmaz. Kaba kuvvet örüntüyü bir kaydırıp *T[1]*'den yeniden başlardı. Oysa metnin son dört karakterinin "ABAB" olduğu biliniyor. "ABAB"ın soneki olan "AB", örüntünün de önekidir. Öyleyse örüntü iki konum kaydırılırsa önündeki "AB" metindeki "AB" ile zaten eşleşmiş olur. Karşılaştırma *P[2]* ile *T[4]* arasında sürer. Gerçekten de bu hizada örüntünün kalan "ABC" kısmı da eşleşir ve örüntü *s = 2*'de bulunur.

Bu kaydırmanın ne kadar olacağı yalnız örüntüye bağlıdır, metne bağlı değildir. Bu yüzden her olası *q* için önceden hesaplanıp bir tabloya yazılabilir. Bu tabloya **LPS tablosu** ya da **önek fonksiyonu** (*prefix function*) denir ve kaynaklarda π ile de gösterilir.

5.1.3.2 LPS tablosu

LPS, “en uzun uygun önek ve aynı zamanda sonek” anlamındaki *longest proper prefix which is also suffix* ifadesinin kısaltmasıdır. Tablonun her hücresi, örüntünün bir önekinin içindeki en uzun “kendine benzeyen” parçanın uzunluğudur.

Tanım 5.1.2

0 ≤ q < m için *lps[q]*, *P[0..q]* dizgisinin hem uygun öneki hem de soneki olan en uzun parçasının uzunluğudur. Böyle boş olmayan bir parça yoksa *lps[q] = 0*'dır. Tek karakterin uygun öneki boş dizgi olduğundan her zaman *lps[0] = 0*'dır.

Tanımdaki “uygun” sözcüğü önemlidir. Dizginin kendisi de hem öneki hem sonekidir. Tanım onu dışarıda bırakmasaydı her hücre *q + 1* olurdu ve tablo hiçbir bilgi taşımazdı. Uygunluk koşulu ayrıca *lps[q] ≤ q* sonucunu verir. Bu eşitsizlik, algoritmanın her geri çekilmede ilerleme kaydettiğini gösteren kanıtta kullanılır.

Örnek 5.1.3

Metin "ababcababcababababd", örüntü "ababd" olsun. Örüntünün LPS tablosu Tablo 5.1.4'te hücre hücre gerekçesiyle verilmiştir.

q	*P[q]*	*lps[q]*	Gerekçe (*P[0..q]*)
0	a	0	"a": uygun önek yok
1	b	0	"ab": a ≠ b
2	a	1	"aba": "a" hem önek hem sonek
3	b	2	"abab": "ab" hem önek hem sonek
4	d	0	"ababd": d ile biten önek yok

Tablo 5.1.4: "ababd" örüntüsünün LPS tablosu.

Sonuç *lps = [0, 0, 1, 2, 0]*'dır. *q = 3* satırında "aba" da bir aday gibi görünebilir. Ama "aba", "abab"ın soneki değildir; "abab"ın üç harfli soneki "bab"dır. Bu yüzden değer 3 değil 2'dir.

Tablonun anlamı aramada ortaya çıkar. Örüntünün ilk *q* karakteri eşleşmiş ve *(q + 1)*'inci karakter uyuşmamış olsun. Eşleşmiş parça *P[0..q−1]*'dir ve onun “kendine benzeyen” en uzun parçası *lps[q − 1]* uzunluğundadır. Örüntü, bu parça metindeki aynı parçanın altına gelecek kadar kaydırılır. Kayma miktarı *q − lps[q − 1]*, karşılaştırmanın süreceği yeni örüntü konumu ise *lps[q − 1]*'dir.

Teorem 5.1.1 (KMP kaydırmasının güvenliği)

P[0..q−1] metinde *s* kaymasında eşleşmiş ve *P[q]* uyuşmamış olsun. *s* ile *s + q − lps[q − 1]* arasında kalan kaymaların hiçbirinde örüntü metinle eşleşemez.

İspat. **Kanıt.** $s < s' < s + q$ olan bir s' kaymasında örüntünün eşleşebildiğini varsayalım. Bu hizada örüntünün ilk $k = s + q - s'$ karakteri, metnin *T[s'..s+q−1]* parçasının altına gelir. Bu parça, eşleşmiş *P[0..q−1]*'in son k karakteridir. Eşleşme için *P[0..k−1]*'in bu parçaya eşit olması gerekir. Yani *P[0..k−1]*, *P[0..q−1]*'in hem uygun öneki hem sonekidir ($0 < k < q$). Tanım gereği böyle bir parçanın uzunluğu en çok *lps[q − 1]*'dir. Buradan $k \le lps[q-1]$, yani $s' \ge s + q - lps[q-1]$ çıkar. Daha küçük kaymalar bu koşulu bozduğu için eşleşemez. ■

Teorem, KMP'nin hiçbir geçişi kaçırmadığını söyler. Atlanan kaymalar, örüntüyle metin arasında kesin bir çelişki içerir. Algoritmanın doğruluğu bu gözleme dayanır.

5.1.3.3 LPS tablosunu kurmak

LPS tablosu tanımdan doğrudan hesaplanabilir: her q için bütün önek uzunlukları denenir ve sonekle karşılaştırılır. Bu yol $O(m^3)$ sürer. KMP'nin tabloyu kurma yöntemi daha akıllıcadır: tablo, örüntünün kendi içinde aranmasıyla, yani KMP'nin kendisiyle kurulur. Toplam süre $O(m)$'dir.

```
int[] lpsHesapla(String p) {
  int m = p.length(), k = 0;
  int[] lps = new int[m];
  for (int q = 1; q < m; q++) {
    while (k > 0 &&
      p.charAt(k) != p.charAt(q))
      k = lps[k - 1];
    if (p.charAt(k) == p.charAt(q))
      k++;
    lps[q] = k;
  }
  return lps;
}
```

Kodda `k`, o ana kadar eşleşen önek uzunluğudur. Döngünün her adımında `q` bir artar ve `p[q]` karakteri işlenir:

- **Geri çekilme:** `k > 0` iken `p[k]` ile `p[q]` farklıysa, uzunluğu `k` olan önek uzatılamaz. Bir sonraki aday, bu önekin kendi içindeki en uzun “kendine benzeyen” parçasıdır: `k = lps[k - 1]`. Bu adım, aramadaki uyuşmazlık adımının aynısıdır.
- **Uzatma:** `p[k] == p[q]` ise eşleşen önek bir karakter uzar ve `k` bir artar.
- **Yazma:** `k`, *P[0..q]* için en uzun uygun önek-sonek uzunluğudur ve `lps[q]`'ya yazılır.

Geri çekilmenin neden `lps[k - 1]` olduğu şöyle açıklanır. *P[0..q]*'nun önek-sonek parçalarını uzunluğa göre azalan sırayla denemek gerekir. Bu parçaların her biri, *P[0..q−1]*'in bir önek-sonek parçasının bir karakter uzatılmış hâlidir. *P[0..q−1]*'in önek-sonek parçaları da bir zincir oluşturur. En uzunu *k*, ondan sonraki *lps[k − 1]*, sonraki *lps[lps[k − 1] − 1]*... Zincirdeki her uzunluk, bir önceki parçanın içindeki en uzun önek-sonek parçasıdır. Algoritma zinciri büyükten küçüğe izler ve uzatılabilen ilk adayda durur.

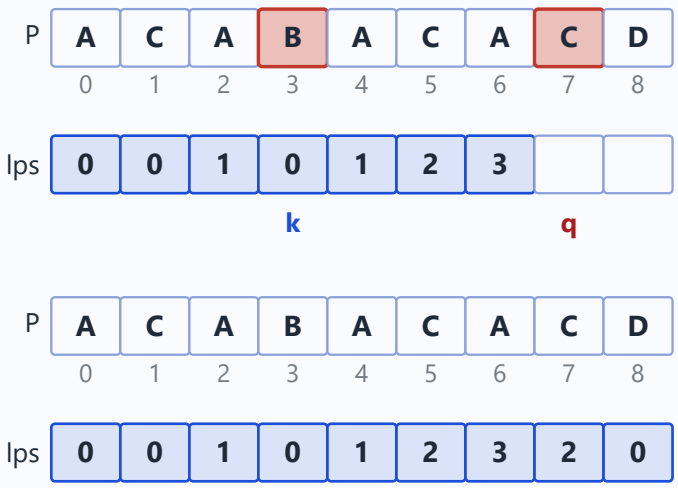
Örnek 5.1.4

`lpsHesapla("ACABACACD")` çağrısı izlenecek. Örüntünün indisleri 0–8'dir. Tablo 5.1.5 her karşılaştırmayı ve `k`'nin değişimini verir.

`q`	`k` (önce)	Karşılaştırma	İşlem	`lps[q]`
0	—	—	her zaman 0	0
1	0	`p[0]` = A, `p[1]` = C	farklı; `k = 0` kalır	0
2	0	`p[0]` = A, `p[2]` = A	aynı; `k = 1`	1
3	1	`p[1]` = C, `p[3]` = B	farklı; geri çekil: `k = lps[0] = 0`	
3	0	`p[0]` = A, `p[3]` = B	farklı; `k = 0` kalır	0
4	0	`p[0]` = A, `p[4]` = A	aynı; `k = 1`	1
5	1	`p[1]` = C, `p[5]` = C	aynı; `k = 2` ("AC")	2
6	2	`p[2]` = A, `p[6]` = A	aynı; `k = 3` ("ACA")	3
7	3	`p[3]` = B, `p[7]` = C	farklı; geri çekil: `k = lps[2] = 1`	
7	1	`p[1]` = C, `p[7]` = C	aynı; `k = 2` ("AC")	2
8	2	`p[2]` = A, `p[8]` = D	farklı; geri çekil: `k = lps[1] = 0`	
8	0	`p[0]` = A, `p[8]` = D	farklı; `k = 0` kalır	0

Tablo 5.1.5: `lpsHesapla("ACABACACD")` izlemesi. Her satır bir karşılaştırmadır.

Sonuç *lps = [0, 0, 1, 0, 1, 2, 3, 2, 0]*'dır. En öğretici satır *q = 7*'dir. "ACABACA"nın en uzun önek-sonek parçası "ACA"dır (*k = 3*). "ACA"nın ardından gelen *P[3]* = B, *P[7]* = C ile uyuşmaz; "ACAB" ile "ACAC" farklıdır. Bir sonraki aday, "ACA"nın içindeki en uzun önek-sonek parçası "A"dır: *lps[2] = 1*. "A"nın ardındaki *P[1]* = C, *P[7]* ile eşleşir ve "AC" bulunur. Toplam 3 geri çekilme yapılmıştır.



Şekil 5.1.6: Solda *q = 7*, *k = 3* anı: "ACA"dan sonraki B ile C uyuşmuyor; *k*, *lps[2] = 1*'e çekilecek. Sağda tamamlanmış tablo: *[0, 0, 1, 0, 1, 2, 3, 2, 0]*.

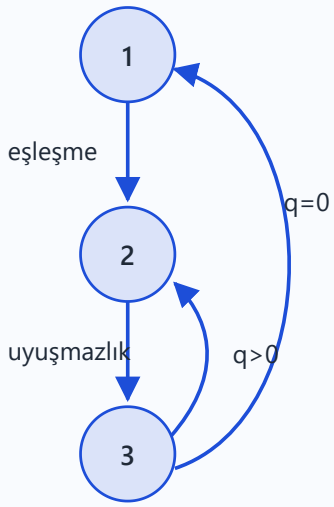
Kurma işleminin süresi *O(m)*'dir. Gerekçe, aramanın doğrusallık kanıtının aynısıdır: `k` her adımda en çok bir artar; toplam artış *m − 1*'i geçemez. Her geri çekilme `k` 'yi en az bir azaltır ve `k` negatif olamaz. Bu yüzden toplam geri çekilme sayısı da *m − 1*'i geçemez. Örnekte 8 artış fırsatına karşılık 3 geri çekilme vardır. Tablo için *O(m)* ek bellek gerekir.

> **Sık yapılan hata**
>
> LPS'yi dizginin kendisini de sayarak hesaplamak. "aaaa" için doğru tablo *[0, 1, 2, 3]*'tür, *[1, 2, 3, 4]* değildir. Örneğin "aaaa"nın en uzun uygun önek-sonek parçası "aaa"dır ve uzunluğu 3'tür. Tablo yanlış kurulursa geri çekilme *q*'yu hiç küçültmez ve arama sonsuz döngüye girer.

5.1.3.4 Aramanın üç evresi

KMP araması iki işaretçiyle çalışır. *i* metindeki, *q* örüntüdeki konumdur; *q* aynı zamanda o ana kadar eşleşen karakter sayısıdır. Algoritmanın davranışı üç evreye ayrılabilir:

1. **İlk karakteri arama.** *q = 0*'dır. Örüntünün ilk karakteriyle eşleşen bir metin karakteri bulunana kadar *i* ilerletilir.
2. **Birlikte ilerleme.** Karakterler eşleştikçe iki işaretçi de ilerler. *q = m* olursa örüntünün sonuna ulaşılmıştır ve *i − m + 1* konumu döndürülür.
3. **LPS ile geri çekilme.** Uyuşmazlıkta yalnız örüntü işaretçisi geri çekilir: *q = lps[q − 1]*. Metin işaretçisi *i* yerinde kalır. Bir eşleşme bulununca 2. evreye, *q* 0'a inerse 1. evreye dönülür.

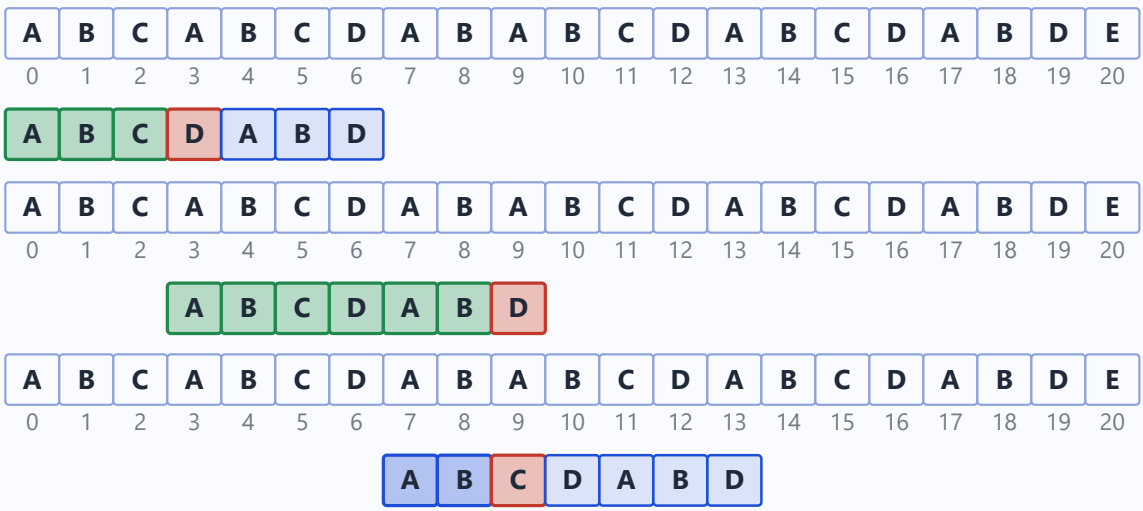


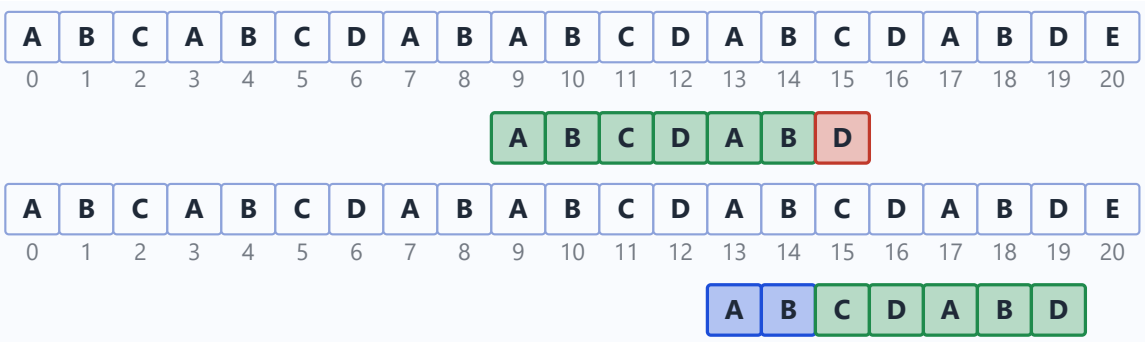
Şekil 5.1.7: KMP aramasının evreleri arasındaki geçişler. Eşleşme 1'den 2'ye götürür; uyuşmazlık 2'den 3'e götürür. 3. evreden, $q > 0$ iken eşleşme bulunursa 2'ye, $q = 0$'a inilirse 1'e dönülür.

Evrelerin hiçbirinde i azalmaz. Kaba kuvvetle en önemli fark budur. Kaba kuvvet her yeni hizada metin işaretçisini s'ye geri çeker. KMP'de ise metin bir akış gibi okunur: her karakter bir kez gelir, işlenir ve bir daha istenmez.

5.1.3.5 Örnek: ABCDABD araması

Örüntü "ABCDABD", metin "ABCABCDABABCDABCDABDE" olsun ($m = 7$, $n = 21$). Örüntünün LPS tablosu *[0, 0, 0, 0, 1, 2, 0]*'dır: "ABCDA"nın sonundaki "A" ve "ABCDAB"ın sonundaki "AB" aynı zamanda örüntünün önekidir. Şekil 5.1.8 aramanın beş hizasını gösterir.





Şekil 5.1.8: KMP ile "ABCDABD" araması, yukarıdan aşağıya *s = 0, 3, 7, 9, 13* hizaları. Yeşil eşleşen, kırmızı uyuşmayan, mavi ise LPS sayesinde eşleştiği bilinen ve yeniden karşılaştırılmayan karakterlerdir.

1. ***s = 0.*** "ABC" eşleşir; *T[3]* = A ile *P[3]* = D uyuşmaz. *q = 3*'tür ve *q = lps[2] = 0* olur. Örüntü 3 kayar; *T[3]* bu kez *P[0]* ile karşılaştırılır.
2. ***s = 3.*** "ABCDAB" eşleşir; *T[9]* = A ile D uyuşmaz. *q = 6*'dır ve *q = lps[5] = 2* olur. Eşleşmiş "ABCDAB"ın sonundaki "AB" önek olduğu için örüntü 4 kayar.
3. ***s = 7.*** "AB"nin eşleştiği bilinir, yeniden karşılaştırılmaz. *T[9]* = A ile *P[2]* = C uyuşmaz; *q = lps[1] = 0*.
4. ***s = 9.*** *T[9]* = A, *P[0]* ile eşleşir. "ABCDAB" eşleşir; *T[15]* = C ile D uyuşmaz. *q = 6 → lps[5] = 2*; örüntü 4 kayar.
5. ***s = 13.*** "AB" bilinir; ardından "CDABD" de eşleşir. Örüntü 13. konumda bulunur.

Arama boyunca metin işaretçisi hiç geri gitmez. Toplam karakter karşılaştırması 24'tür. Metnin 20 karakteri birer kez okunur. Üç uyuşmazlık noktasında (*i = 3, 9, 15*) aynı metin karakteri, örüntünün başka bir karakteriyle bir kez daha karşılaştırılır. *i = 9*'da bu iki kez olur. Kaba kuvvet aynı metinde 14 hiza dener ve 37 karşılaştırma yapar.

5.1.3.6 Java kodu ve izleme

Aramanın Java kodu, LPS kurma kodunun neredeyse aynısıdır. Tek fark, örüntünün kendisiyle değil metinle karşılaştırılmasıdır.

```
int kmpAra(String t, String p) {
  int n = t.length(), m = p.length();
  int[] lps = lpsHesapla(p);
  int q = 0;  // eşleşen uzunluk
  for (int i = 0; i < n; i++) {
    while (q > 0 &&
      p.charAt(q) != t.charAt(i))
      q = lps[q - 1];
    if (p.charAt(q) == t.charAt(i))
      q++;
    if (q == m)
      return i - m + 1;
  }
  return -1;
}
```

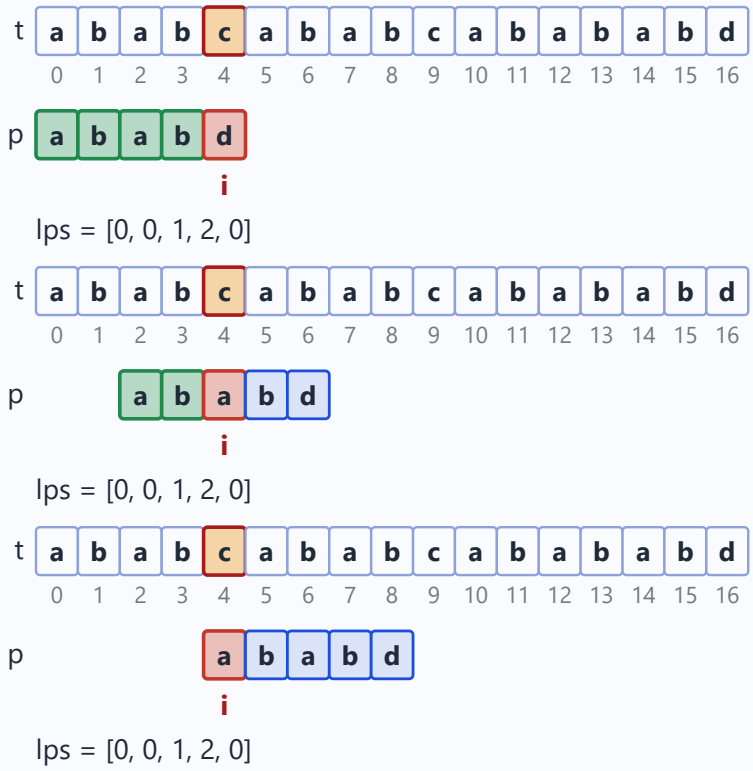
- **Hazırlık:** LPS tablosu kurulur (*O(m)*) ve `q = 0` yapılır. `q`, eşleşen önek uzunluğudur.
- **Dış döngü:** `i` metinde soldan sağa bir kez gezinir. Kodda `i`'yi azaltan hiçbir satır yoktur.
- `while` **döngüsü:** uyuşmazlık olduğu sürece `q` LPS zinciri boyunca geri çekilir. `q == 0` olunca döngü durur; daha geriye gidilecek yer yoktur.
- `if` **(eşleşme):** `p[q]` ile `t[i]` eşitse eşleşen önek bir uzar.
- `if (q == m)`**:** örüntünün tamamı eşleşmiştir. Son eşleşen karakter `i`'de olduğu için başlangıç `i - m + 1`'dir.

Kod `kmpAra("ababcababcababababd", "ababd")` çağrısıyla izlenecek. *n = 17*, *m = 5*, *lps = [0, 0, 1, 2, 0]*'dır. Tablo 5.1.6 her metin karakteri için `q`'nun değişimini verir.

`i`	`t[i]`	`q` (önce)	Geri çekilme	Eşleşme	`q` (sonra)
0–3	a b a b	0 → 3	—	her biri eşleşir	4
4	c	4	d ≠ c: `q = lps[3] = 2`; a ≠ c: `q = lps[1] = 0`	`p[0]` = a ≠ c	0
5–8	a b a b	0 → 3	—	her biri eşleşir	4
9	c	4	d ≠ c: `q = 2`; a ≠ c: `q = 0`	`p[0]` = a ≠ c	0
10–13	a b a b	0 → 3	—	her biri eşleşir	4
14	a	4	d ≠ a: `q = lps[3] = 2`	`p[2]` = a = a	3
15	b	3	—	`p[3]` = b = b	4
16	d	4	—	`p[4]` = d = d	5 = m

Tablo 5.1.6: `kmpAra("ababcababcababababd", "ababd")` izlemesi. "Geri çekilme" sütunu `while` döngüsünün yaptıklarıdır.

i = 16'da `q == m` olur ve *16 − 5 + 1 = 12* döndürülür. Metin yalnız bir kez taranmıştır. *i = 14* satırı KMP'nin kazancını en iyi gösteren adımdır. Uyuşmazlık anında metnin son dört karakteri "abab"dır. Bunun sonundaki "ab" örüntünün önekidir; *q = 2* ile örüntü iki konum ileriden sürer ve *T[14]* = a hemen eşleşir. Kaba kuvvet bu noktada *T[11]*'e geri dönerdi.



Şekil 5.1.9: *i = 4*'teki geri çekilme zinciri. Yukarıdan aşağıya: *q = 4*'te d ≠ c, *q* 2'ye iner; *q = 2*'de a ≠ c, *q* 0'a iner; *q = 0*'da da a ≠ c olur ve *i* ilerler. Bu üç adım boyunca metin işaretçisi yerinde kalır.

Bu izlemede toplam 22 karakter karşılaştırması yapılır: 17 metin karakterinin her biri en az bir kez, uyuşmazlık noktaları birkaç kez okunur. Kaba kuvvet aynı aramada 13 hiza dener ve 33 karşılaştırma yapar.

5.1.3.7 Neden O(n + m)?

KMP kodunda iç içe iki döngü vardır. İlk bakışta bu, *O(n · m)* süre gibi görünür. Oysa `while` döngüsünün toplam dönme sayısı sınırlıdır. Bu tür, işlemlerin tek tek değil toplamda sayıldığı çözümlemeye **amortize analiz** (*amortized analysis*) denir.

> **Teorem 5.1.2 (KMP'nin doğrusallığı)**
>
> KMP araması en çok *2n* karakter karşılaştırması yapar ve *O(n)* sürede çalışır. LPS tablosunun kurulmasıyla birlikte toplam süre *O(n + m)*, ek bellek *O(m)*'dir. Bu sınırlar en kötü durumda da geçerlidir.

İspat. **Kanıt.** *q*'nun değerini izleyelim. Dış döngünün her turunda *q* en çok bir kez, bir artar (`q++`). Dış döngü *n* kez döndüğü için toplam artış en çok *n*'dir. `while` döngüsünün her turu *q*'yu *lps[q − 1]*'e indirir. *lps[q − 1] ≤ q − 1* olduğu için her tur *q*'yu en az bir azaltır. *q* 0'dan başlar ve hiçbir zaman nega-

tif olmaz. Bu yüzden toplam azalma, toplam artışı aşamaz. Sonuç olarak `while` döngüsü bütün arama boyunca en çok *n* kez döner.

Her `while` turu bir karşılaştırma yapar. Bunlara her *i* için `if`'teki en çok bir karşılaştırma eklenir. Toplam en çok *n + n = 2n*'dir. Aynı argüman `lpsHesapla` için *O(m)* verir. ■

Kanıttaki düşünce bir banka hesabı benzetmesiyle de anlatılabilir. Her eşleşme hesaba bir lira yatırır, her geri çekilme hesaptan bir lira çeker. Hesap hiçbir zaman eksiye düşemez. Öyleyse çekilen toplam para, yatırılan toplam paradan fazla olamaz. Tek bir *i* için `while` birçok kez dönebilir; Tablo 5.1.6'daki *i = 4* satırında iki kez döner. Ama bu ancak önceki adımlarda aynı miktarda eşleşme biriktirilmişse mümkündür.

> **Sık yapılan hata**
>
> Uyuşmazlıkta `lps[q - 1]` yerine `lps[q]` kullanmak. `q` karakter eşleşmişse eşleşen parça *P[0..q−1]*'dir ve ona ait tablo değeri `lps[q - 1]`'dir. Örneğin "abcabd" örüntüsünün tablosu *[0, 0, 0, 1, 2, 0]*'dır. "abcab" eşleşmişken (*q = 5*) doğru değer *lps[4] = 2*'dir ve örüntü 3 konum kayar. `lps[5] = 0` kullanılırsa örüntü 5 konum kayar; bu kaymanın atladığı hizalarda bir geçiş varsa kaçırılır.

5.1.3.8 Bütün geçişleri bulmak; artılar ve eksiler

Sunumdaki `kmpAra` ilk geçişte durur. Bütün geçişler isteniyorsa `q == m` olduğunda konum bir listeye eklenir ve arama sürdürülür. Sürdürmeden önce *q*, *lps[m − 1]*'e çekilir. Bu adım, uyuşmazlık adımıyla aynı mantıktadır: örüntünün tamamı eşleşmiştir ve bir sonraki olası geçiş, eşleşen örüntünün en uzun önek-sonek parçasından başlar. Böylece birbiriyle çakışan geçişler de bulunur.

```
List<Integer> kmpHepsi(String t, String p) {
  int n = t.length(), m = p.length();
  int[] lps = lpsHesapla(p);
  List<Integer> konumlar = new ArrayList<>();
  int q = 0;
  for (int i = 0; i < n; i++) {
    while (q > 0 &&
      p.charAt(q) != t.charAt(i))
      q = lps[q - 1];
    if (p.charAt(q) == t.charAt(i))
      q++;
    if (q == m) {
      konumlar.add(i - m + 1);
      q = lps[q - 1];  // çakışan geçişler için
    }
  }
  return konumlar;
}
```

Kod Java ile denenmiştir. `kmpHepsi("abababa", "aba")` çağrısı *[0, 2, 4]* döndürür. Geçişler birbirine biner: 0'daki "aba"nın son a'sı 2'deki "aba"nın ilk a'sıdır. "aba"nın LPS tablosu *[0, 0, 1]* olduğu için her bulunuştan sonra *q = 1* olur ve tek a yeniden okunmaz. `kmpHepsi("aaaaa", "aa")` ise *[0, 1, 2, 3]* verir. *q = lps[q − 1]* satırı unutulursa `q` bir sonraki turda *m*'yi geçer ve `p.charAt(q)` dizinin dışına çıkar.

KMP'nin güçlü ve zayıf yanları şöyle özetlenebilir:

- **En kötü durumda bile doğrusal.** Süre girdinin yapısından bağımsız olarak *O(n + m)*'dir. Kaba kuvvetin en kötü girdisi olan *T* = a…a, *P* = a…ab KMP için sorun değildir.
- **Metinde geri gitmez.** Bu özellik, metnin bir ağdan ya da diskten parça parça geldiği **akış** (*stream*) verisinde önemlidir. Metnin önceki parçalarını bellekte tutmak gerekmez.
- **Pratikte Boyer-Moore'dan genellikle yavaştır.** KMP metnin her karakterini en az bir kez okur. Boyer-Moore ise birçok karakteri hiç okumadan atlar.
- **Küçük alfabede kazanç sınırlıdır.** DNA gibi dört harfli alfabelerde LPS değerleri büyük çıkabilir ve kaymalar kısa kalır. Sınır yine doğrusaldır, ama sabit çarpan büyür.

5.1.4 Boyer-Moore algoritması

Robert S. Boyer ve J Strother Moore 1977'de yayımladıkları algoritmayla şaşırtıcı bir sonuç gösterdi: bir örüntüyü bulmak için metnin her karakterini okumak gerekmez. KMP metnin her karakterine en az bir kez bakar. Boyer-Moore ise doğal dil metninde karakterlerin çoğunu hiç okumadan atlar. Metin düzenleyicilerin ve `grep` gibi araçların aramaları bu algoritma ailesine dayanır.

5.1.4.1 Sağdan sola karşılaştırma

Algoritmanın ilk farkı karşılaştırma yönüdür. Örüntü, kaba kuvvette olduğu gibi metin boyunca soldan sağa kaydırılır. Ama bir hizadaki karakterler sağdan sola, örüntünün son karakterinden başlanarak karşılaştırılır. Bu tercihin nedeni, uyuşmazlığın en çok bilgiyi sağ uçta vermesidir. Sağ uçtaki metin karakteri örüntüde hiç yoksa, o karakteri kapsayan hiçbir hiza eşleşemez. Örüntü tümüyle bu karakterin ötesine, yani *m* konum kaydırılabilir.

Algoritma dört adımda çalışır:

1. **Ön işleme.** Örüntüden kaydırma tabloları kurulur: her karakterin örüntüdeki son görülme konumu (kötü karakter tablosu) ve her sonek için kaydırma miktarı (iyi sonek tablosu).
2. **Arama.** Örüntü metnin altına hizalanır. Karşılaştırma örüntünün son karakterinden başlar ve sola doğru ilerler.
3. **Eşleşme denetimi.** Bütün karakterler eşleşirse konum rapor edilir.
4. **Kaydırma.** Uyuşmazlıkta iki kural ayrı ayrı bir kaydırma önerir: kötü karakter kuralı ve iyi sonek kuralı. Örüntü, ikisinden büyük olanı kadar kaydırılır.

İki kural da **güvenlidir**: önerdikleri kaydırma, eşleşme olabilecek hiçbir hizayı atlamaz. Bu yüzden ikisinin büyüğünü almak da güvenlidir.



Şekil 5.1.10: "ABDCABDCABDE" metninde "ABDE" araması. Kırmızı sayılar karşılaştırma sırasıdır. İlk iki hizada sağ uçtaki C örüntüde yoktur ve örüntü 4 konum atlar. Üçüncü hizada sağdan sola dört karakter eşleşir: 12 karakterlik metinde yalnız 6 karşılaştırma.

5.1.4.2 Kötü karakter kuralı

Bir hizada sağdan sola karşılaştırırken j konumunda uyuşmazlık çıktığını varsayalım: $P[j] \neq T[s + j]$. Uyuşmayan metin karakterine $c = T[s + j]$ denir ve **kötü karakter** (*bad character*) adını alır. Kural, örüntüyü c'nin örüntüdeki yerine göre kaydırır.

> **Tanım 5.1.3**
>
> **Son görülme tablosu** her c karakteri için $son[c]$ değerini tutar: c'nin örüntüdeki en sağdaki konumu, örüntüde yoksa -1. **Kötü karakter kuralı** (*bad character rule*) kaymayı $s \leftarrow s + max(1, j - son[c])$ biçiminde artırır.

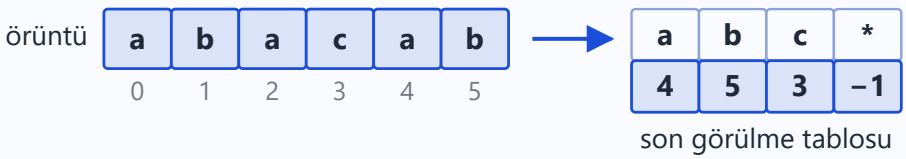
Kuralın üç durumu vardır:

- **c örüntüde j'nin solunda geçiyor.** Örüntü, en sağdaki c metindeki c'nin altına gelecek biçimde kaydırılır. Kaydırma miktarı $j - son[c]$'dir.
- **c örüntüde hiç yok.** $son[c] = -1$ olduğu için kaydırma $j + 1$ olur. Örüntü, c'nin tamamen ötesine geçer.
- **c'nin son geçişi j'nin sağında.** Bu durumda $j - son[c] \leq 0$ çıkar. Kural, örüntünün geri gitmesini önlemek için en az 1 kaydırır.

Kuralın güvenliği son görülme konumunun tanımından gelir. $son[c]$ en sağdaki geçiş olduğu için $son[c] + 1$ ile j arasındaki örüntü konumlarında c yoktur. $1, 2, \ldots, j - son[c] - 1$ kadarlık daha küçük kaydırmalar, bu konumlardan birini metindeki c'nin altına getirir. Orada bir uyuşmazlık kesindir; bu hizalar atlanabilir.

```
int[] sonGorulme(String p) {
  int[] son = new int[256];
  Arrays.fill(son, -1);
  for (int i = 0; i < p.length(); i++)
    son[p.charAt(i)] = i;
  return son;
}
```

`sonGorulme` metodu 256 hücreli bir dizi kurar ve hepsini −1 yapar. Dizi her ASCII karakteri için bir hücre içerir. Ardından örüntü soldan sağa taranır ve her karakterin hücresine o anki konum yazılır. Aynı karakter birden çok kez geçiyorsa son yazılan, yani en sağdaki konum kalır. Süre $O(m + \sigma)$, bellek $O(\sigma)$'dır; burada $\sigma = 256$'dır.



Şekil 5.1.11: "abacab" örüntüsünün son görülme tablosu: *son[a] = 4*, *son[b] = 5*, *son[c] = 3*, öbür bütün karakterler için −1.

Örnek 5.1.5

Metin "GCAATGCCTATGTGACC", örüntü "TATGTG" ($m = 6$) olsun. Yalnız kötü karakter kuralıyla ilk kaydırma bulunacak.

1. **Tablo.** Örüntü T A T G T G'dir. *son[T] = 4*, *son[A] = 1*, *son[G] = 5*; C örüntüde yoktur.
2. **Karşılaştırma.** $s = 0$'da sağdan sola: $P[5] = T[5]$ = G eşleşir, $P[4] = T[4]$ = T eşleşir. $P[3]$ = G ile $T[3]$ = A uyuşmaz; $j = 3$, kötü karakter A'dır.
3. **Kaydırma.** $son[A] = 1$; kaydırma $\max(1, 3 - 1) = 2$. Yeni hizada örüntünün A'sı (konum 1) metindeki A'nın (konum 3) altına gelir.

Arama sürdürülürse ikinci hizada ($s = 2$) sağ uçtaki C örüntüde olmadığı için örüntü 6 konum atlar. Üçüncü hizada ($s = 8$) altı karakter de eşleşir. Toplam 3 hiza ve 10 karşılaştırma yeterlidir.

Sık yapılan hata

Kaydırmayı `max(1, …)` olmadan, yalnız `j - son[c]` diye yazmak. c'nin son geçişi j'nin sağındaysa bu fark sıfır ya da negatiftir. Örneğin "abacab"da $son[a] = 4$'tür; uyuşmazlık $j = 3$'te çıkarsa fark $3 - 4 = -1$ olur. Örüntü geri gider ya da yerinde kalır; algoritma sonsuz döngüye girebi-

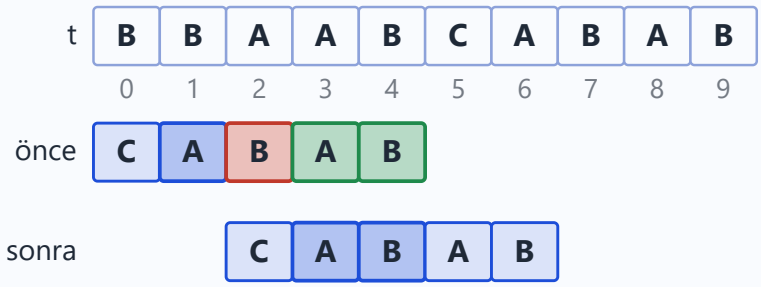
lir.

5.1.4.3 İyi sonek kuralı

Kötü karakter kuralı yalnız uyuşmayan karaktere bakar; o ana kadar eşleşen karakterlerden yararlanmaz. İkinci kural bu boşluğu doldurur. Uyuşmazlıktan önce eşleşen sonek $u = P[j+1..m-1]$, metinde de bulunmuştur. Bu sonek "iyi sonek" adını alır. **İyi sonek kuralı** (*good suffix rule*), örüntüyü u'nun örüntüdeki başka bir geçişine göre kaydırır:

1. **Durum 1.** u örüntüde başka bir yerde, önünde farklı bir karakterle geçiyorsa örüntü, o geçiş metindeki u'nun altına gelecek biçimde kaydırılır. Önündeki karakterin farklı olması gerekir; aynı karakter aynı uyuşmazlığı tekrarlardı.
2. **Durum 2.** Böyle bir geçiş yoksa, u'nun bir sonekine eşit olan en uzun P öneki hizalanır.
3. **Durum 3.** İkisi de yoksa örüntü m konum kaydırılır.

İyi sonek tablosu her j için bu kaydırmayı önceden hesaplar. Tablo, KMP'nin LPS fikrine benzer bir yöntemle, bu kez sonekler üzerinde $O(m)$ sürede kurulur. Kurma yönteminin ayrıntısı uzundur; ders kapsamında kuralın anlamı ve elle uygulanması yeterlidir.



Şekil 5.1.12: İyi sonek kuralı. "CABAB" örüntüsünün son iki karakteri "AB" eşleşmiş, $T[2]$ = A ile $P[2]$ = B uyuşmamıştır. "AB" örüntüde 1–2 konumlarında da geçer ve önünde farklı bir karakter (C) vardır. Örüntü bu geçiş metindeki "AB"nin altına gelecek biçimde 2 konum kaydırılır.

Şekil 5.1.12 iki kuralın farkını gösterir. Kötü karakter A'dır ve $son[A] = 3 > j = 2$'dir; kötü karakter kuralı yalnız 1 kaydırma önerir. İyi sonek kuralı ise 2 önerir. Algoritma büyüğünü seçer ve örüntüyü 2 kaydırır. Tam Boyer-Moore'da iki tablo birlikte kullanılır.

Küçük alfabelerde iyi sonek kuralı daha da önem kazanır. DNA'da ($\sigma = 4$) dört harfin her biri uzun bir örüntüde neredeyse kesin olarak sağ uca yakın bir yerde geçer. Kötü karakter kuralı bu yüzden çoğunlukla 1–2 konumluk kaydırmalar önerir. İyi sonek kuralı ise eşleşmiş parçanın yapısına baktığı için uzun kaydırmalar bulabilir.

5.1.4.4 Örnek: HELLO araması

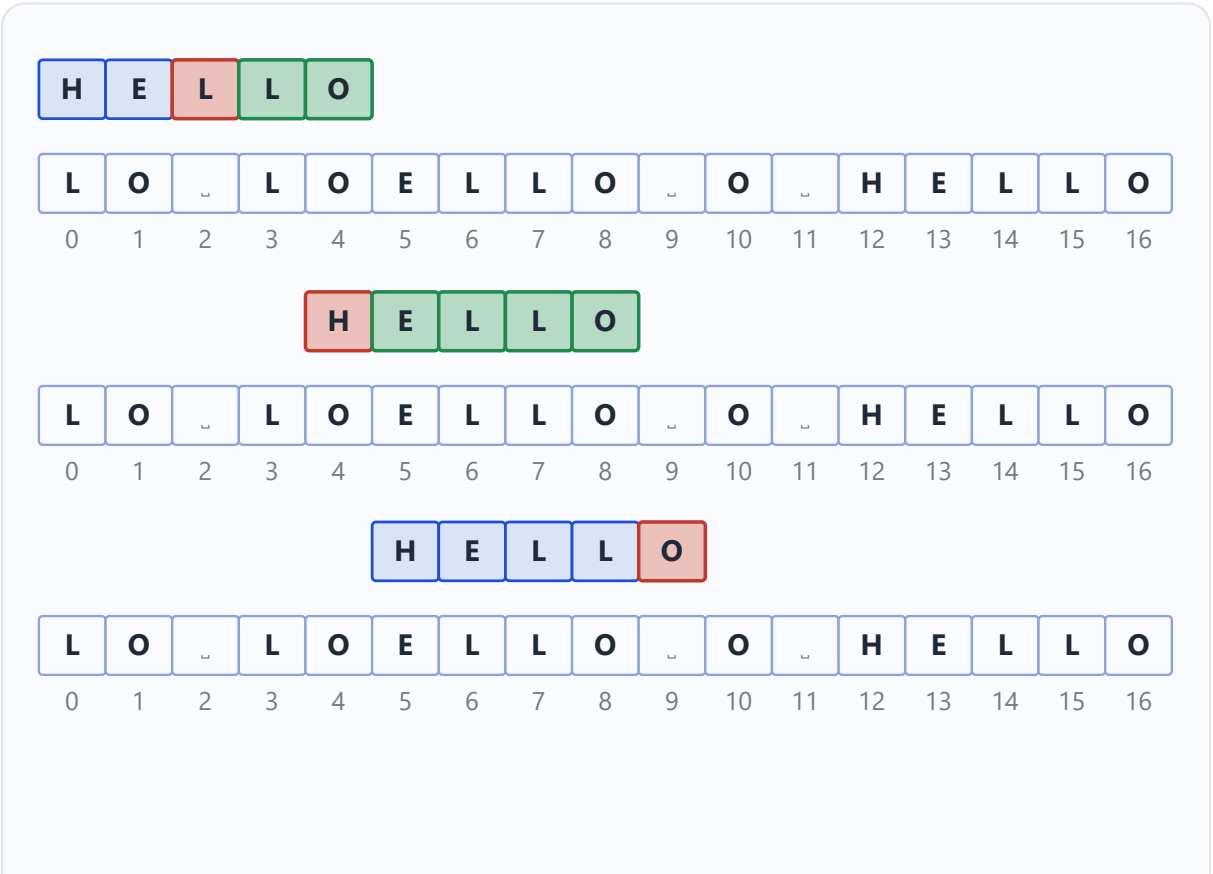
Metin "LO LOELLO O HELLO" ($n = 17$), örüntü "HELLO" ($m = 5$) olsun. Örnek yalnız kötü karakter kuralıyla çalıştırılacak. Son görülme tablosu Tablo 5.1.7'de verilmiştir; L iki kez geçtiği için en sağdaki konum (3) yazılmıştır.

c	H	E	L	O	diğer
son[c]	0	1	3	4	−1

Tablo 5.1.7: "HELLO" örüntüsünün son görülme tablosu.

Hiza	*s*	Sağdan sola karşılaştırma	Kötü karakter	Kaydırma	Karş.
1	0	O, L eşleşir; *T[2]* = ␣ ≠ *P[2]* = L	␣ (yok)	*2 − (−1) = 3*	3
2	3	*T[7]* = L ≠ *P[4]* = O	L (3)	*4 − 3 = 1*	1
3	4	O, L, L, E eşleşir; *T[4]* = O ≠ *P[0]* = H	O (4)	*0 − 4 < 1* → 1	5
4	5	*T[9]* = ␣ ≠ *P[4]* = O	␣ (yok)	*4 − (−1) = 5*	1
5	10	*T[14]* = L ≠ *P[4]* = O	L (3)	1	1
6	11	*T[15]* = L ≠ *P[4]* = O	L (3)	1	1
7	12	beş karakter de eşleşir	—	bulundu	5
Toplam					17

Tablo 5.1.8: Boyer-Moore (kötü karakter kuralı) ile HELLO araması. Kaydırma *max(1, j − son[c])*'dir.





Şekil 5.1.13: HELLO aramasının dört hizası, yukarıdan aşağıya hiza 1, 3, 4 ve 7. Yeşil eşleşen, kırmızı kötü karakterdir. Hiza 4'te örüntüde olmayan boşluk karakteri 5 konumluk atlama sağlar.

Örüntü 7 hizada ve 17 karşılaştırmada $s = 12$'de bulunur. Hiza 3 dikkat çekicidir: dört karakter eşleştikten sonra uyuşmazlık sol uçta çıkar. Kötü karakter O'nun son geçişi $j = 0$'ın sağındadır ve kural yalnız 1 kaydırır. Bu hizada iyi sonek kuralı daha fazlasını önerirdi: eşleşen "ELLO", örüntüde başka yerde geçmez ve hiçbir önekle de çakışmaz; kaydırma 5 olurdu. Kuralların birbirini nasıl tamamladığı bu örnekte açıkça görülür.

5.1.4.5 Horspool basitleştirmesi

R. Nigel Horspool 1980'de Boyer-Moore'un pratikte çok kullanılan bir sadeleştirmesini önerdi. İyi sonek tablosu tümüyle atılır. Kötü karakter kuralı da değiştirilir: kaydırma, uyuşmazlığın nerede çıktığına bakmadan, hizanın son karakterinin altındaki metin karakterine göre yapılır.

Tanım 5.1.4

Horspool kaydırma tablosu her c karakteri için $kayd[c] = m - 1 - r$ değerini tutar. Burada r, c'nin $P[0..m-2]$ içindeki en sağ konumudur. c bu parçada yoksa $kayd[c] = m$'dir. Her hizadan sonra kayma $s \leftarrow s + kayd[T[s + m - 1]]$ olur.

Tabloya örüntünün son karakteri katılmaz. Katılsaydı son karakterin kaydırması $m - 1 - (m - 1) = 0$ olurdu ve örüntü yerinde sayardı. "HELLO" için tablo şöyledir: H → 4, E → 3, L → 1, O → 5, diğer → 5. L iki kez geçer (konum 2 ve 3); en sağdaki 3 kullanılır ve kaydırma 1 olur. O yalnız son konumda geçtiği için tabloya hiç girmez ve “diğer” gibi 5 alır.

Hiza	s	Karşılaştırma	Son karakter	Kaydırma
1	0	O, L eşleşir; $T[2] = ␣ \neq P[2]$	$T[4] =$ O	5
2	5	$T[9] = ␣ \neq P[4]$	$T[9] = ␣$	5
3	10	$T[14] =$ L $\neq P[4]$	$T[14] =$ L	1
4	11	$T[15] =$ L $\neq P[4]$	$T[15] =$ L	1
5	12	tümü eşleşir	—	bulundu

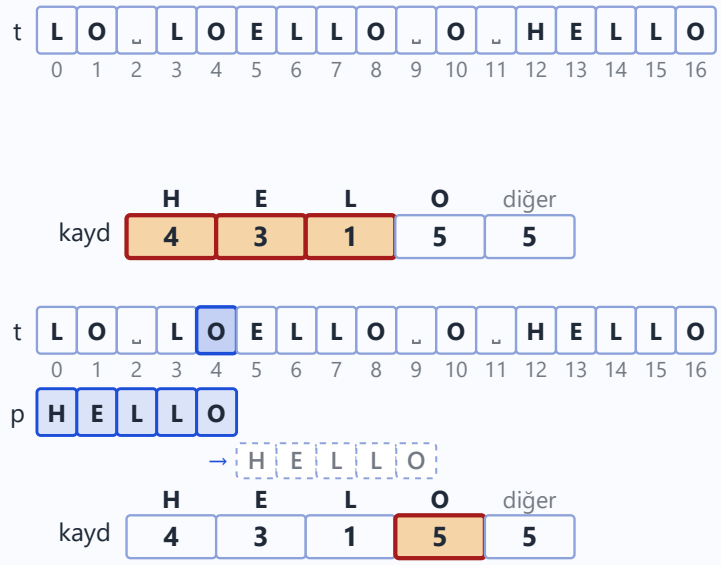
Tablo 5.1.9: Horspool ile aynı HELLO araması.

Aynı örnekte Horspool 5 hiza ve 11 karşılaştırmayla yetinir; temel kötü karakter kuralı 7 hiza ve 17 karşılaştırma yapmıştı. Farkın kaynağı ilk hizadır. Temel kural uyuşmayan boşluk karakterine bakıp 3 kaydırır. Horspool ise son karakter O'ya bakar. O, örüntünün ilk dört karakterinde geçmediği için örüntü 5 kayar. Basitleştirme her zaman daha iyi sonuç vermez, ama tek tablosu ve kısa koduyla pratikte çok hızlıdır.

```
int horspool(String t, String p) {
  int n = t.length(), m = p.length();
  int[] kayd = new int[256];
  Arrays.fill(kayd, m);
  for (int i = 0; i < m - 1; i++)
    kayd[p.charAt(i)] = m - 1 - i;
  int s = 0;
  while (s <= n - m) {
    int j = m - 1;
    while (j >= 0 &&
      p.charAt(j) == t.charAt(s + j))
      j--;
    if (j < 0) return s;
    s += kayd[t.charAt(s + m - 1)];
  }
  return -1;
}
```

- **Tablo:** `kayd` dizisinin 256 hücresi önce `m` ile doldurulur. Ardından `P[0..m−2]` soldan sağa taranır ve her karakter için `m - 1 - i` yazılır. Soldan sağa tarandığı için en sağdaki geçiş en son yazılır ve kalır.
- **Karşılaştırma:** `j` sağ uçtan başlar ve karakterler eşleştikçe azalır. `j < 0` olursa bütün karakterler eşleşmiştir ve `s` döndürülür.
- **Kaydırma:** uyuşmazlık hangi `j`'de çıkarsa çıksın, `s` hizanın son karakteri `t[s + m - 1]`'in tablo değeri kadar artar. Bu değer en az 1 olduğu için örüntü hiç geri gitmez.

`horspool("LO LOELLO O HELLO", "HELLO")` çağrısı Tablo 5.1.9'daki adımları birebir izler ve 12 döndürür. Kod Java ile çalıştırılarak 5 hiza ve 11 karşılaştırma sayısı doğrulanmıştır.



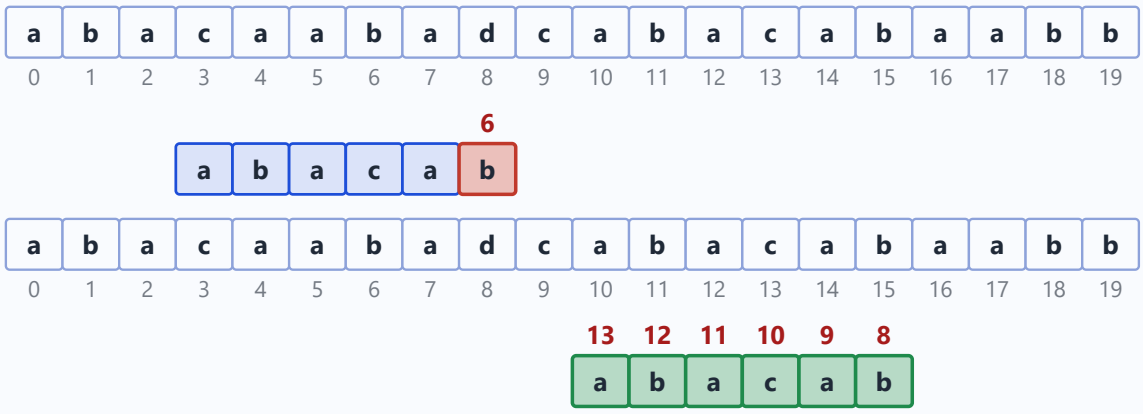
Şekil 5.1.14: Solda Horspool kaydırma tablosu. Sağda ilk hizadan sonraki kaydırma: uyuşmazlık $j = 2$'de çıkmıştır, ama kaydırma hizanın son karakteri $T[4]$ = O'ya göre yapılır ve s 5 olur.

5.1.4.6 Örnek: abacab ile bütün geçişler

Bu örnek, kötü karakter kuralının bütün durumlarını bir arada gösterir. Metin "abacaabadcabacabaabb" ($n = 20$), örüntü "abacab" ($m = 6$) olsun. Son görülme tablosu $a = 4$, $b = 5$, $c = 3$, diğerleri −1'dir. Bütün geçişler aranacak; bir eşleşmeden sonra örüntü 1 kaydırılıp aramaya devam edilir.

Hiza	s	Uyuşmazlık	Kaydırma	Karş.
1	0	$T[5]$ = a ≠ $P[5]$ = b	$5 - son[a] = 5 - 4 = 1$	1
2	1	b, a eşleşir; $T[4]$ = a ≠ $P[3]$ = c	$son[a] = 4 \geq j = 3 \rightarrow 1$	3
3	2	$T[7]$ = a ≠ $P[5]$ = b	1	1
4	3	$T[8]$ = d ≠ $P[5]$ = b	d yok: $j + 1 = 6$	1
5	9	$T[14]$ = a ≠ $P[5]$ = b	1	1
6	10	altı karakter eşleşir: geçiş, $s = 10$	1 (aramaya devam)	6
7	11	$T[16]$ = a ≠ $P[5]$ = b	1	1
8	12	$T[17]$ = a ≠ $P[5]$ = b	1	1
9	13	b, a eşleşir; $T[16]$ = a ≠ $P[3]$ = c	$son[a] = 4 \geq 3 \rightarrow 1$	3
10	14	b eşleşir; $T[18]$ = b ≠ $P[4]$ = a	$son[b] = 5 \geq 4 \rightarrow 1$	2
Toplam ($s = 15 > n - m$ olunca arama biter)				20

Tablo 5.1.10: "abacab" için bütün geçişlerin Boyer-Moore (kötü karakter) ile aranması.



Şekil 5.1.15: Üstte hiza 4: sağ uçtaki d örüntüde yoktur ve örüntü 6 konum atlar. Altta hiza 6: altı karşılaştırmanın hepsi eşleşir. Hücre üstündeki sayılar karşılaştırma sırasıdır.

Arama 10 hizada ve 20 karşılaştırmada biter; örüntü yalnız *s = 10*'da geçer. Kaydırmaların çoğu 1'dir, çünkü metindeki a ve b karakterleri örüntünün sağ ucuna yakın yerlerde geçer. Tek büyük atlamayı d sağlar. Bu tablo, Boyer-Moore'un hızının alfabeye ve metne bağlı olduğunu gösterir: örüntüde olmayan karakterler ne kadar sıksa atlamalar o kadar uzundur.

5.1.4.7 Java kodu ve izleme

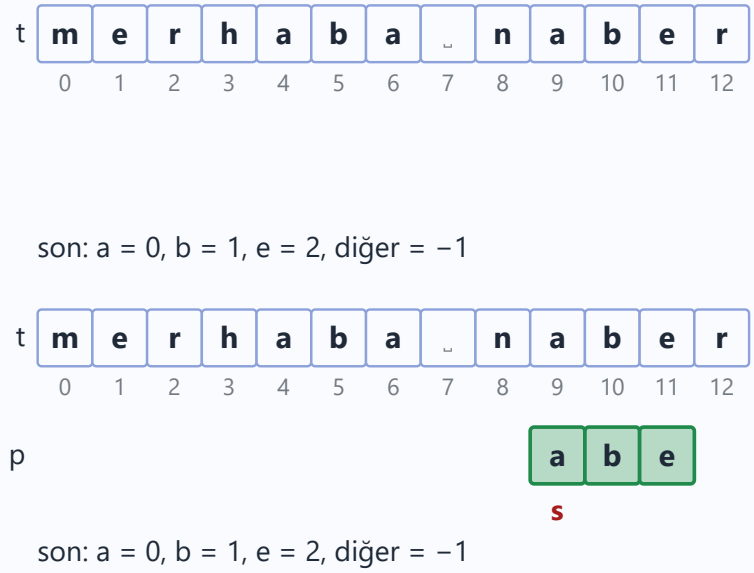
Yalnız kötü karakter kuralını kullanan Boyer-Moore kodu, Horspool koduna çok benzer. Tek fark kaydırma satırıdır: kaydırma, son karaktere göre değil uyuşmayan karaktere göre hesaplanır.

```
int boyerMoore(String t, String p) {
  int n = t.length(), m = p.length();
  int[] son = sonGorulme(p);
  int s = 0;
  while (s <= n - m) {
    int j = m - 1;
    while (j >= 0 &&
      p.charAt(j) == t.charAt(s + j))
      j--;
    if (j < 0) return s;
    s += Math.max(1,
      j - son[t.charAt(s + j)]);
  }
  return -1;
}
```

Kod `boyerMoore("merhaba naber", "abe")` çağrısıyla izlenecek. *n = 13*, *m = 3*'tür. Son görülme tablosu *a = 0*, *b = 1*, *e = 2*, diğerleri −1'dir.

`s`	Hiza	Karşılaştırma	Kaydırma
0	mer	`p[2]` = e ≠ `t[2]` = r	`son['r'] = -1` : *max(1, 2 − (−1)) = 3*
3	hab	`p[2]` = e ≠ `t[5]` = b	`son['b'] = 1` : *max(1, 2 − 1) = 1*
4	aba	`p[2]` = e ≠ `t[6]` = a	`son['a'] = 0` : *max(1, 2 − 0) = 2*
6	a␣n	`p[2]` = e ≠ `t[8]` = n	`son['n'] = -1` : 3
9	abe	e, b, a eşleşir; `j = -1`	bulundu: 9 döndürülür

Tablo 5.1.11: `boyerMoore("merhaba naber", "abe")` izlemesi.



Şekil 5.1.16: Solda başlangıç: son görülme tablosu kurulmuş, *s = 0*. Sağda son durum: "abe" 9. konumda ("naber" kelimesinin içinde) bulunmuştur.

Arama 5 hizada ve 7 karşılaştırmada biter. Kaba kuvvet aynı aramada 10 hiza dener ve 15 karşılaştırma yapar. 13 karakterlik metnin yalnız 7 karakterine bakılmıştır. 0, 1, 3, 4, 7 ve 12. konumlardaki karakterler hiç okunmamıştır. Boyer-Moore'u kaba kuvvetten ve KMP'den ayıran özellik budur.

5.1.4.8 Analiz

Boyer-Moore'un performansı geniş bir aralıkta değişir:

- **En iyi durum:** metindeki karakterler örüntüde hiç geçmez. Her hizada tek karşılaştırma yapılır ve örüntü *m* konum atlar. Toplam yaklaşık *n/m* karşılaştırma yeterlidir: *O(n/m)*. Süre, örüntü uzadıkça *kısalır*. Kaba kuvvette ve KMP'de bunun tersi olur.
- **Ortalama durum:** doğal dil metninde ve geniş alfabede çoğu uyuşmazlık sağ uçta çıkar ve kaydırmalar uzundur. Süre alt-doğrusaldır; yani metnin uzunluğundan daha az karakter okunur.

- **En kötü durum (yalnız kötü karakter):** T = aaaa…a, P = baaa…a. Her hizada sağdan $m - 1$ a eşleşir, sol uçtaki b uyuşmaz. Kötü karakter a'nın son geçişi sağ uçtadır ve kaydırma 1 olur. Süre $O(n \cdot m)$'dir. Örneğin $n = 20$, $m = 5$ için 16 hizanın her birinde 5 karşılaştırma, toplam 80 karşılaştırma yapılır.
- **En kötü durum (tam algoritma):** iyi sonek kuralı ve Zvi Galil'in 1979'da önerdiği iyileştirme birlikte kullanılırsa en kötü durum $O(n + m)$'ye iner.

Örnek 5.1.6

Metin "abccbaabccbaabcbcabbabcabc" ($n = 26$), örüntü "abcbcabbabcabc" ($m = 14$) olsun. Son görülme tablosu bulunup arama yalnız kötü karakter kuralıyla yapılacak.

1. **Tablo.** Örüntü a b c b c a b b a b c a b c'dir. En sağdaki a konum 11'de, b konum 12'de, c konum 13'tedir: $son[a] = 11$, $son[b] = 12$, $son[c] = 13$.
2. **İlk hiza.** $s = 0$'da $P[13]$ = c ile $T[13]$ = b uyuşmaz. Kaydırma $max(1, 13 - 12) = 1$.
3. **Sonraki hizalar.** Arama 10 hiza sürer. Kaydırmalar 1, 1, 1, 1, 2, 1, 2, 1, 2'dir; hiçbiri 2'yi geçmez.
4. **Sonuç.** Örüntü $s = 12$'de bulunur. Toplam 31 karşılaştırma yapılır.

Alfabe üç harflidir ve her harf örüntünün sağ ucunda geçer. Kötü karakter kuralı bu yüzden zayıf kalır. Bu örnek, küçük alfabede iyi sonek kuralının neden gerekli olduğunu gösterir.

Boyer-Moore'un ön işleme maliyeti $O(m + \sigma)$, ek belleği $O(m + \sigma)$'dır. Kötü karakter tablosu alfabenin her harfi için bir hücre tuttuğundan büyük alfabelerde (ör. Unicode'un bütün karakterleri) dizi yerine hash tablosu kullanılır.

5.1.5 Rabin-Karp algoritması

Michael O. Rabin ve Richard M. Karp'ın 1987'de yayımladığı algoritma soruna başka bir yerden yaklaşır. Kaydırmaları akıllıca seçmek yerine her hizadaki karşılaştırmanın kendisini ucuzlatır. Örüntü ve metin pencereleri birer sayıya dönüştürülür; iki dizginin eşitliği yerine iki sayının eşitliği sınanır.

5.1.5.1 Karakter yerine sayı karşılaştırmak

Rabin-Karp, örüntünün ve metindeki her m uzunluklu pencerenin **hash** değerini hesaplar. Hash, bir dizgiyi küçük bir tam sayıya eşleyen bir fonksiyondur ve dizginin bir tür özetidir. İki sayının karşılaştırılması tek işlemdir; iki dizginin karşılaştırılması ise m işleme kadar çıkabilir.

Bu fikrin iki sorunu vardır. Birincisi, her pencerenin hash'ini baştan hesaplamak $O(m)$ sürer; bu durumda hiçbir kazanç olmaz. Çözüm **kayan hash**'tir (*rolling hash*): pencere bir sağa kaydığında yeni hash eskisinden $O(1)$ sürede hesaplanır. İkincisi, farklı dizgiler aynı hash değerini verebilir. Hash eşit-

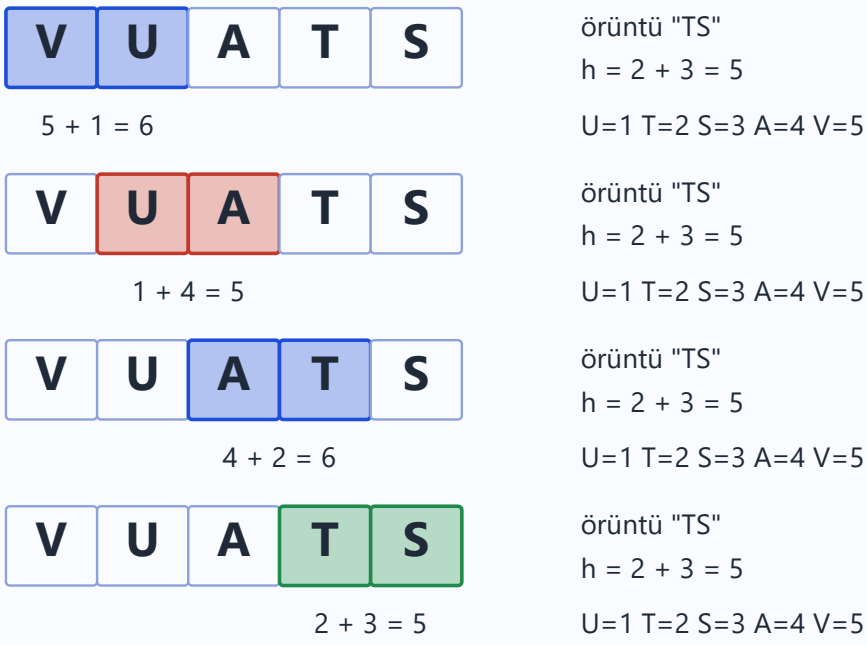
liği bu yüzden eşleşmeyi garanti etmez. Hash'leri eşit çıkan pencere yalnız bir adaydır ve karakter karakter doğrulanmalıdır.

Algoritmanın adımları şöyledir:

1. **Hash hesabı.** Örüntünün ve metnin ilk penceresinin hash değerleri hesaplanır.
2. **Eşleşme denetimi.** Hash değeri örüntününkine eşit olan pencere aday kabul edilir.
3. **Doğrulama.** Aday pencere karakter karakter örüntüyle karşılaştırılır.
4. **Kaydırma ve yeniden hesap.** Eşleşme yoksa pencere bir sağa kaydırılır ve hash güncellenir.
5. **Tekrar.** Adımlar metnin sonuna kadar sürer.

5.1.5.2 Toplam hash ve sahte eşleşme

Fikri en basit hash ile görelim: karakter değerlerinin toplamı. Harflere U = 1, T = 2, S = 3, A = 4, V = 5 değerleri verilsin. "VUATS" metninde "TS" aranıyor; örüntünün hash'i *2 + 3 = 5*'tir.



Şekil 5.1.17: Toplam hash ile "VUATS" içinde "TS" araması, yukarıdan aşağıya dört pencere. "UA" penceresinin hash'i örüntününkine eşittir ama karakterleri farklıdır: sahte eşleşme. "TS" penceresinde hash de karakterler de tutar.

- **"VU":** *5 + 1 = 6 ≠ 5*. Hash farklıdır; karakterlere hiç bakılmaz.
- **"UA":** *1 + 4 = 5*. Hash eşittir, ama "UA" ≠ "TS". Bu bir **sahte eşleşmedir** (*spurious hit*); doğrulama boşa harcanır.
- **"AT":** *4 + 2 = 6 ≠ 5*. Yeni hash eskisinden bulunur: çıkan U'nun değeri düşülür, giren T'ninki eklenir: *5 − 1 + 2 = 6*. İşlem pencere uzunluğundan bağımsızdır, *O(1)*'dir.

- **"TS":** *2 + 3 = 5*. Hash eşittir ve karakter karşılaştırması da tutar: eşleşme, *s = 3*.

Toplam hash kayan güncellemenin ne olduğunu iyi gösterir, ama zayıf bir hash'tir. Harflerin sırasına duyarsızdır: "ST" de 5 verir. Uzun pencerelerde değerler dar bir aralıkta toplanır ve birçok pencere aynı toplamı verir. Gerçek uygulamalarda bu yüzden konum ağırlıklı polinom hash ve büyük bir asal modül kullanılır.

5.1.5.3 Polinom hash ve kayan güncelleme

Polinom hash, dizgiyi *B* tabanında yazılmış bir sayı gibi okur. Karakter kodları basamaklardır; soldaki karakter en büyük basamaktır. Sayı çok büyüyebileceği için bir *Q* asalına göre kalanı alınır.

Tanım 5.1.5

B taban, *Q* asal bir modül olsun. *s* kaymasındaki pencerenin **polinom hash** değeri

$$h_s = (T[s]\cdot B^{m-1} + T[s+1]\cdot B^{m-2} + \ldots + T[s+m-1]) \bmod Q$$

biçimindedir. Pencere bir sağa kayınca yeni değer **kayan güncelleme** formülüyle bulunur:

$$h_{s+1} = (B \cdot (h_s - T[s] \cdot B^{m-1}) + T[s+m]) \bmod Q$$

Formül üç adımı tek satırda yapar. Önce çıkan karakterin katkısı $T[s] \cdot B^{m-1}$ düşülür. Ardından kalan basamaklar *B* ile çarpılarak bir basamak sola kaydırılır. Son olarak giren karakter en küçük basamağa eklenir. Ondalık bir benzetme bunu açıkça gösterir: "314"ten "141"e geçmek için 3 yüzlüğü atılır (14), bir basamak sola kaydırılır (140) ve 1 eklenir (141).

$B^{m-1} \bmod Q$ değeri her güncellemede gerekir ve pencere uzunluğu değişmediği için bir kez hesaplanıp saklanır. Modüler aritmetiğin kuralları her ara adımda kalan almaya izin verir. *(a + b) mod Q* ve *(a · b) mod Q* değerleri, *a* ile *b*'nin kalanlarıyla da hesaplanabilir. Bu sayede sayılar hiçbir zaman büyümez.

Örnek 5.1.7

Taban *B = 10*, *Q = 13*, karakterler rakam olsun. *T* = "31415" ve *m = 3* için pencerelerin hash'leri kayan formülle bulunacak.

1. **Sabit.** $B^{m-1} \bmod Q = 100 \bmod 13 = 9$.
2. **İlk pencere.** *h("314") = 314 mod 13 = 2* (*13 · 24 = 312*).
3. **İkinci pencere.** *h("141") = (10 · (2 − 3 · 9) + 1) mod 13 = (−249) mod 13 = 11*. Doğrudan hesap: *141 = 13 · 10 + 11* ✓.
4. **Üçüncü pencere.** *h("415") = (10 · (11 − 1 · 9) + 5) mod 13 = 25 mod 13 = 12*. Doğrudan hesap: *415 = 13 · 31 + 12* ✓.

İkinci adımda ara sonuç negatif çıkmıştır. Matematikte *(−249) mod 13 = 11*'dir, çünkü *−249 = 13 · (−20) + 11*. Java'nın `%` işleci ise negatif bir sayı için negatif kalan verir: `-249 % 13` ifadesi −2'dir. Sonuca 13 eklenince doğru değer olan 11 elde edilir.

5.1.5.4 Java kodu ve izleme

Sunumdaki kod ASCII kodlarını basamak olarak kullanır: taban *B = 256*, modül *Q = 101*'dir. `esit(t, p, s)` yardımcı metodu, aday pencereyi karakter karakter örüntüyle karşılaştırır.

```
int rabinKarp(String t, String p) {
  int n = t.length(), m = p.length();
  int B = 256, Q = 101, h = 1;
  for (int i = 0; i < m - 1; i++)
    h = (h * B) % Q;  // B^(m-1) mod Q
  int ph = 0, th = 0;
  for (int i = 0; i < m; i++) {
    ph = (B * ph + p.charAt(i)) % Q;
    th = (B * th + t.charAt(i)) % Q;
  }
  for (int s = 0; s <= n - m; s++) {
    if (ph == th && esit(t, p, s))
      return s;
    if (s < n - m) {
      th = (B * (th - t.charAt(s) * h)
            + t.charAt(s + m)) % Q;
      if (th < 0) th += Q;
    }
  }
  return -1;
}
```

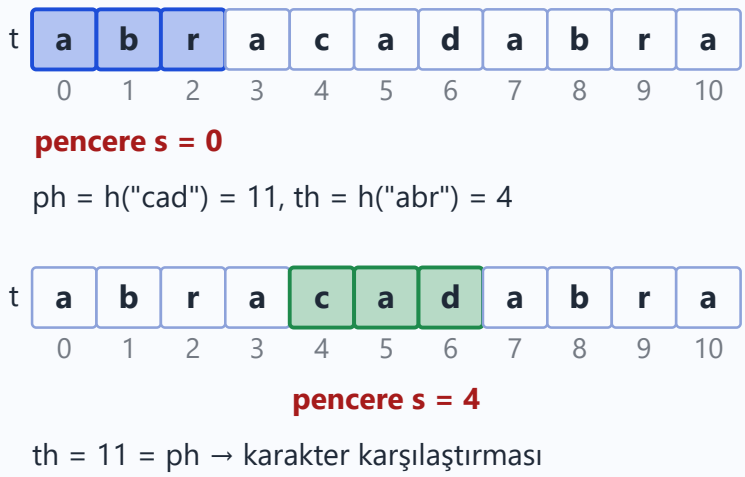
- **İlk döngü:** `h = B^(m-1) mod Q` hesaplanır. Bu, çıkan karakterin ağırlığıdır.
- **İkinci döngü:** `ph` (örüntü hash'i) ve `th` (ilk pencerenin hash'i) Horner yöntemiyle hesaplanır: her adımda önceki değer `B` ile çarpılır ve yeni karakter eklenir. Bu döngü *O(m)* sürer.
- **Ana döngü:** her kayma için önce hash'ler karşılaştırılır. `&&` sayesinde `esit` ancak hash'ler eşitse çağrılır. Eşleşme yoksa ve metnin sonuna gelinmemişse `th` kayan formülle güncellenir.
- `if (th < 0) th += Q;` satırı Java'nın negatif kalan sorununu giderir. `th - t.charAt(s) * h` çoğunlukla negatiftir ve `%` işleci negatif bir kalan döndürür.

Kod `rabinKarp("abracadabra", "cad")` çağrısıyla izlenecek. *n = 11*, *m = 3*'tür. *h = 256² mod 101 = 88*'dir. Örüntü hash'i *ph = (99 · 256² + 97 · 256 + 100) mod 101 = 11*'dir (c = 99, a = 97, d = 100). İlk pencere "abr"nin hash'i 4'tür.

`s`	Pencere	`th`	Karar	Çıkan / giren	Ara	`% 101`	Yeni `th`
0	abr	4	4 ≠ 11, geç	a (97) / a (97)	−2.184.095	−71	30

`s`	Pencere	`th`	Karar	Çıkan / giren	Ara	`% 101`	Yeni `th`
1	bra	30	30 ≠ 11, geç	b (98) / c (99)	−2.199.965	−84	17
2	rac	17	17 ≠ 11, geç	r (114) / a (97)	−2.563.743	−60	41
3	aca	41	41 ≠ 11, geç	a (97) / d (100)	−2.174.620	−90	11
4	cad	11	11 = 11: `esit` doğrular	"cad" = "cad": 4 döndürülür			

Tablo 5.1.12: `rabinKarp("abracadabra", "cad")` izlemesi (*B = 256*, *Q = 101*, *h = 88*, *ph = 11*). "Ara" sütunu `B * (th - çıkan * h) + giren` değeridir.



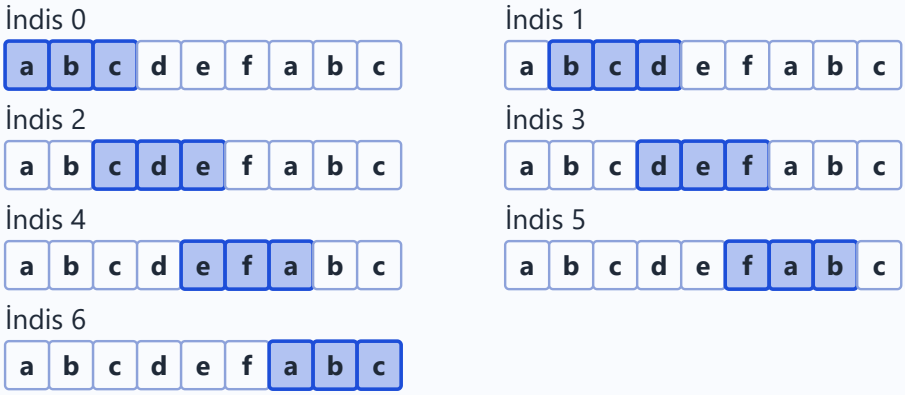
Şekil 5.1.18: Solda başlangıç: *ph = 11*, ilk pencere "abr" için *th = 4*. Sağda *s = 4*: hash'ler eşittir, karakter karşılaştırması da tutar ve "cad" bulunur.

İzlemede dört pencerede hash'ler farklı çıkar ve karakterlere hiç bakılmaz. Karakter karşılaştırması yalnız beşinci pencerede, üç karakter için yapılır. Tablodaki "ara" değerlerinin hepsi negatiftir. Java'nın `%` işleci bu yüzden her adımda negatif kalan verir ve düzeltme satırı her adımda çalışır. Bu satır unutulsaydı ikinci pencerenin hash'i 30 yerine −71 olurdu. Sonraki hash'ler de yanlış çıkardı ve "cad" penceresinin hash'i 11'e hiç eşit olmayabilirdi.

Sık yapılan hata

Büyük bir *Q* seçip `int` türünde kalmak. Sunumdaki kodda *Q = 101*'dir ve ara değerler *±6,6 · 10⁶* sınırını aşmaz; `int` yeterlidir. Sahte eşleşmeleri azaltmak için *Q = 10⁹ + 7* gibi bir asal seçilebilir. Bu durumda `B * th` çarpımı bile *2,56 · 10¹¹*'e ulaşır. `int` sınırı ise yaklaşık *2,1 · 10⁹*'dur. Taşma sessizdir; hash'ler yanlış çıkar ve gerçek eşleşmeler kaçırılabilir. Bu durumda hesaplar `long` türünde yapılmalıdır.

5.1.5.5 Analiz ve çok sayıda örüntü



Şekil 5.1.19: "abcdefabc" metninde 3 uzunluklu pencerenin 0'dan 6'ya kayması. $n - m + 1 = 7$ pencerenin her birinin hash'i bir öncekinden $O(1)$ sürede bulunur.

Rabin-Karp'ın maliyeti iki parçadan oluşur: hash hesapları ve doğrulamalar. Hash hesapları her durumda $O(m) + (n - m) \cdot O(1) = O(n + m)$ tutar. Doğrulamaların maliyeti ise kaç pencerenin aday olduğuna bağlıdır.

- **En iyi ve ortalama durum:** Q büyük bir asal seçilirse rastgele bir pencerenin hash'inin örüntününkine yanlışlıkla eşit çıkma olasılığı yaklaşık $1/Q$'dur. Beklenen sahte eşleşme sayısı $(n - m + 1)/Q$'dur. $n = 10^6$ ve $Q \approx 10^9$ için bu sayı yaklaşık 0,001'dir: bütün metin boyunca büyük olasılıkla hiç sahte eşleşme çıkmaz. Gerçek geçişler k tane ise beklenen süre $O(n + m + k \cdot m)$'dir; az sayıda geçiş için $O(n + m)$.
- **En kötü durum:** kötü bir Q seçilir ya da girdi kasıtlı olarak çakışmalar üretecek biçimde hazırlanırsa her pencere aday olur. Örneğin T = a…a ve P = a…a için bütün pencereler gerçek eşleşmedir ve her biri m karakterle doğrulanır. Süre $O(n \cdot m)$'dir.

Ek bellek $O(1)$'dir; yalnız birkaç tam sayı tutulur. Rabin-Karp'ın asıl gücü tek örüntüde değil, **çok sayıda örüntüde** ortaya çıkar. Aynı uzunlukta k örüntünün hash'leri bir kümeye (`HashSet`) konur. Metin bir kez taranır ve her pencerenin hash'i kümede aranır. Küme araması ortalama $O(1)$ olduğu için bütün örüntüler beklenen $O(n + k \cdot m)$ sürede bulunur. KMP ya da Boyer-Moore ile aynı iş, metni k kez taramayı gerektirirdi.

```
List<Integer> rkCoklu(String t, List<String> desenler) {
  int n = t.length(), m = desenler.get(0).length();
  int B = 256, Q = 101, h = 1;
  for (int i = 0; i < m - 1; i++)
    h = (h * B) % Q;
  Set<Integer> hashler = new HashSet<>();
  Set<String> kume = new HashSet<>(desenler);
  for (String p : desenler) {
    int ph = 0;
    for (int i = 0; i < m; i++)
      ph = (B * ph + p.charAt(i)) % Q;
    hashler.add(ph);
  }
  List<Integer> konumlar = new ArrayList<>();
  int th = 0;
  for (int i = 0; i < m; i++)
    th = (B * th + t.charAt(i)) % Q;
  for (int s = 0; s <= n - m; s++) {
    if (hashler.contains(th)
        && kume.contains(t.substring(s, s + m)))
      konumlar.add(s);
    if (s < n - m) {
      th = (B * (th - t.charAt(s) * h)
            + t.charAt(s + m)) % Q;
      if (th < 0) th += Q;
    }
  }
  return konumlar;
}
```

Kod Java ile denenmiştir: `rkCoklu("abracadabra", List.of("cad", "bra", "xyz"))` çağrısı *[1, 4, 8]* döndürür. "bra" 1 ve 8'de, "cad" 4'te geçer; "xyz" metinde yoktur. Doğrulama, pencerenin kendisinin örüntü kümesinde bulunup bulunmadığına bakar. Böylece hash'i bir örüntününkiyle çakışan ama hiçbir örüntüye eşit olmayan pencereler elenir.

Bu teknik intihal tespitinin ve belge benzerliği ölçümünün temelidir. Bir belgedeki bütün *k* kelimelik parçaların hash'leri çıkarılır ve başka bir belgenin parçalarıyla karşılaştırılır. Aynı düşünce iki boyuta da genişletilebilir: bir görüntünün içinde küçük bir görüntü bloğu, satır ve sütun hash'leriyle aranabilir.

5.1.6 Eşleştirme algoritmalarının karşılaştırması

Dört algoritma aynı soruyu yanıtlar, ama farklı varsayımlar altında en iyidir. Tablo 5.1.13 temel özelliklerini yan yana koyar.

Algoritma	Temel fikir	Ön işleme	Ortalama	En kötü	Ek bellek
Kaba kuvvet	Her kaymayı dener	yok	$O(n)$*	$O(n \cdot m)$	$O(1)$
KMP	LPS tablosu, metinde geri gitmez	$O(m)$	$O(n + m)$	$O(n + m)$	$O(m)$
Boyer-Moore	Sağdan sola; kötü karakter ve iyi sonek	$O(m + \sigma)$	alt-doğrusal	$O(n \cdot m)$**	$O(m + \sigma)$
Rabin-Karp	Kayan hash karşılaştırması	$O(m)$	$O(n + m)$	$O(n \cdot m)$	$O(1)$

Tablo 5.1.13: Dizgi eşleştirme algoritmalarının karşılaştırması. σ alfabe büyüklüğüdür.

* Rastgele metin ve geniş alfabe için; tekrarlı metinde $O(n \cdot m)$'ye yaklaşır. ** Yalnız kötü karakter kuralıyla; iyi sonek kuralı ve Galil iyileştirmesiyle $O(n + m)$.

Asimptotik sınırlar resmin yalnız bir parçasıdır. Tablo 5.1.14, notta kullanılan örneklerde dört yöntemin karakter karşılaştırma sayılarını verir. Sayılar algoritmalar Python ve Java ile çalıştırılarak elde edilmiştir. KMP'de her karakter çifti bir kez sayılmıştır; `while` döngüsünden eşitlikle çıkılıp `if`'te aynı çiftin yeniden denetlenmesi ayrıca sayılmamıştır. Boyer-Moore sütunu yalnız kötü karakter kuralını kullanır.

Metin / örüntü	n, m	Kaba kuvvet	KMP	Boyer-Moore	Horspool
"THIS IS A SIMPLE EXAMPLE" / "SIMPLE"	24, 6	18	18	8	8
"ABCABCDABABCDABCDABDE" / "ABCDABD"	21, 7	37	24	12	12
"ababcababcababababd" / "ababd"	17, 5	33	22	8	8
"LO LOELLO O HELLO" / "HELLO"	17, 5	17	17	17	11
"merhaba naber" / "abe"	13, 3	15	14	7	7
"aaaaaaaaaa" / "aaab" (geçiş yok)	10, 4	28	17	7	7

Tablo 5.1.14: İlk geçiş bulunana kadar yapılan karakter karşılaştırmaları.

Tablodan üç gözlem çıkar. Birincisi, KMP kaba kuvvetten hiçbir zaman kötü değildir. Kazancı, eşleşmiş parçaların uzun olduğu tekrarlı metinlerde büyür. Doğal dil metninde (SIMPLE satırı) iki yöntem neredeyse aynı işi yapar. İkincisi, Boyer-Moore çoğu satırda açık ara öndedir. Son satırda kaba kuvvetin en kötü girdisi, Boyer-Moore için en kolay girdilerden biridir. Örüntünün sağ ucundaki b her hizada ilk karşılaştırmada uyuşmaz ve örüntü her seferinde ilerler. Üçüncüsü, hiçbir yöntem her girdide en iyi değildir. HELLO satırında temel kötü karakter kuralı kaba kuvvetten iyi değildir, Horspool ise daha iyidir.

Seçim için kaba bir yol haritası şöyledir:

- **Garanti gerekiyorsa KMP.** Akış verisinde (ağdan gelen, geri okunamayan veride) ya da en kötü durum sınırının önemli olduğu gerçek zamanlı sistemlerde tercih edilir.
- **Uzun örüntü ve doğal dil metni için Boyer-Moore ya da Horspool.** Metin düzenleyicilerde ve komut satırı araçlarında pratikte en hızlı seçenektir.
- **Çok sayıda örüntü için Rabin-Karp.** Aynı uzunlukta birçok örüntü tek geçişte aranır: intihal kontrolü, virüs imzaları.

- **Kısa örüntü ve kısa metin için kaba kuvvet.** Ön işlemesi yoktur, kodu kısadır ve ek bellek istemez. Birkaç karakterlik örüntülerde öbür yöntemlerin hazırlık maliyeti kazancı aşabilir.

5.1.7 Dizgi sıkıştırma

Bölümün ikinci konusu, metni ya da genel olarak veriyi daha az bitle saklamak ve iletmektir. Sıkıştırma disk alanından ve ağ bant genişliğinden tasarruf sağlar. Bir dosyanın boyutu yarıya inerse indirme süresi de kabaca yarıya iner. Sıkıştırma algoritmaları verideki **fazlalığı**, yani tekrar eden ya da tahmin edilebilir kısımları bulur ve bunları daha kısa biçimde yazar.

5.1.7.1 Kayıpsız ve kayıplı sıkıştırma

Tanım 5.1.6

Kayıpsız sıkıştırmada (*lossless compression*) sıkıştırılmış veri açıldığında orijinal veri bit bit aynen elde edilir. **Kayıplı sıkıştırmada** (*lossy compression*) algıda önemsiz sayılan ayrıntılar atılır ve açılan veri orijinale yalnız yakındır.

Sıkıştırma oranı (*compression ratio*), sıkıştırılmış boyutun orijinal boyuta oranıdır. Oran ne kadar küçükse sıkıştırma o kadar iyidir; 1'den büyük oran, verinin büyüdüğü anlamına gelir.

Kayıplı sıkıştırma JPEG resimlerinde ve MP3 seslerinde kullanılır. Bir fotoğrafın gözün fark etmeyeceği renk geçişlerini atmak büyük kazanç sağlar. Metin için ise kayıplı sıkıştırma uygun değildir. Bir programdaki tek bir karakterin değişmesi programı bozar; bir sözleşmedeki tek bir rakamın değişmesi anlamı değiştirir. Bu yüzden metin, program, PNG ve GIF resimleri kayıpsız sıkıştırılır. Bölümdeki iki yöntem, RLE ve LZW, kayıpsızdır.

Oranın kullanımına bir örnek: 15 karakterlik bir metin 8 karaktere iniyorsa oran $8/15 \approx 0{,}53$'tür; veri yaklaşık yarıya inmiştir. Kaynaklarda oran bazen ters yönde (orijinal / sıkıştırılmış) ya da yüzde kazanç olarak da verilir. Bu notta sunumdaki tanım kullanılır.

5.1.7.2 Her girdiyi küçülten yöntem yoktur

Sıkıştırmanın aşılamaz bir sınırı vardır. Hiçbir kayıpsız yöntem her girdiyi küçültemez. Bu sonuç, basit bir sayma argümanına, **güvercin yuvası ilkesine** dayanır: $k + 1$ güvercin k yuvaya yerleşirse en az bir yuvada iki güvercin olur.

Teorem 5.1.3 (Kayıpsız sıkıştırmanın sınırı)

Hiçbir kayıpsız sıkıştırma yöntemi, uzunluğu N bit olan bütün girdileri daha kısa bitlik dizilere dönüştüremez.

İspat. **Kanıt.** *N* bitlik farklı girdilerin sayısı 2^N'dir. *N*'den kısa bit dizilerinin sayısı (boş dizi dahil) $2^0 + 2^1 + \ldots + 2^{N-1} = 2^N - 1$'dir. Her girdi daha kısa bir diziye dönüşseydi 2^N girdi $2^N - 1$ çıktıya dağılırdı. Güvercin yuvası ilkesine göre en az iki farklı girdi aynı çıktıya düşerdi. Açıcı bu çıktıdan hangi girdinin geldiğini bilemezdi; yöntem kayıpsız olamazdı. ■

Örneğin 8 bitlik 256 farklı bayt vardır, ama 8 bitten kısa yalnız 255 farklı bit dizisi vardır. Teoremin pratik sonucu şudur: sıkıştırma algoritmaları belirli türden fazlalığı hedefler. O fazlalığı içeren girdileri küçültür, içermeyenleri biraz büyütür. RLE ve LZW'yi değerlendirirken sorulacak soru, hangi fazlalığı yakaladıkları ve hangi girdide başarısız olduklarıdır. Zaten sıkıştırılmış bir dosyayı yeniden sıkıştırmanın neredeyse hiç kazanç sağlamaması da aynı nedenden kaynaklanır.

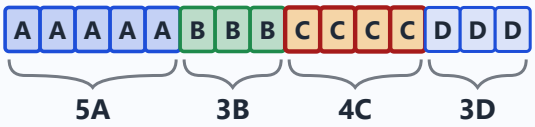
5.1.8 Çalışma uzunluğu kodlaması (RLE)

Çalışma uzunluğu kodlaması (*run-length encoding, RLE*) en basit sıkıştırma yöntemlerinden biridir. Yakaladığı fazlalık, aynı sembolün art arda tekrarıdır.

5.1.8.1 Koşular

> **Tanım 5.1.7**
>
> Bir dizgide aynı karakterin art arda gelen en uzun tekrarına **koşu** (*run*) denir. RLE her koşuyu, koşunun uzunluğu ve karakterden oluşan bir çiftle değiştirir.

Yöntemin adımları şöyledir: metin soldan sağa taranır, ardışık tekrarlar sayılır, her koşu için sayı ve karakter yan yana yazılır ve sonuç sıkıştırılmış metin olarak saklanır. Metin bir kez tarandığı için süre *O(n)*'dir.



Şekil 5.1.20: "AAAAABBBCCCCDDD" dört koşuya ayrılır: 5 A, 3 B, 4 C ve 3 D. Sıkıştırılmış metin "5A3B4C3D"dir.

15 karakterlik "AAAAABBBCCCCDDD" metni "5A3B4C3D" olur; 8 karakter, oran *8/15 ≈ 0,53*. Uzun koşular varsa kazanç büyüktür: 1000 tane a'dan oluşan bir metin "1000a" olur, yani 5 karaktere iner.

Tekrar az ise RLE işe yaramaz, hatta zararlıdır. "ABCD"de her koşu tek karakterliktir ve sonuç "1A1B1C1D" olur: 4 karakter 8'e çıkar, oran 2'dir. Güvercin yuvası ilkesinin öngördüğü gibi RLE, koşu içermeyen girdileri büyütür. "abababab" gibi tekrarlı ama ardışık tekrarı olmayan bir girdi de "1a1b1a1b1a1b1a1b" olur; 8 karakter 16'ya çıkar. RLE bu tür tekrarı göremez.

RLE'nin doğal kullanım alanları, uzun düz bölgeler içeren verilerdir. Faks makineleri siyah-beyaz sayfaların satırlarını RLE ile kodlar; bir satırın çoğu beyazdır. BMP, PCX ve TIFF gibi eski resim biçimleri ve basit simgeler de RLE kullanır.

5.1.8.2 Java kodu ve izleme

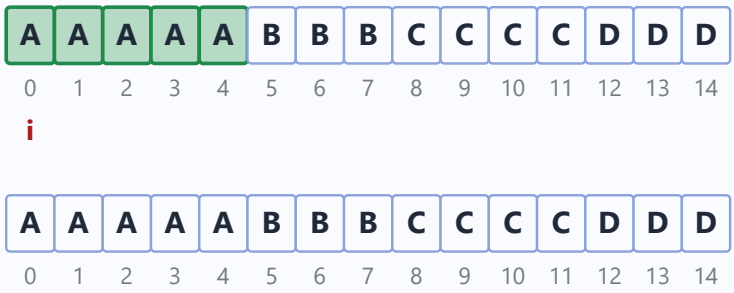
```
String rleSikistir(String s) {
  StringBuilder sonuc = new StringBuilder();
  int n = s.length();
  for (int i = 0; i < n; i++) {
    char c = s.charAt(i);
    int sayac = 1;
    while (i + 1 < n &&
      s.charAt(i + 1) == c) {
      sayac++;
      i++;
    }
    sonuc.append(sayac).append(c);
  }
  return sonuc.toString();
}
```

- **Dış döngü:** `i` her yeni koşunun ilk karakterinde durur. `c` bu karakterdir ve `sayac` 1'den başlar.
- **İç döngü:** bir sonraki karakter `c` ile aynı olduğu sürece `sayac` ve `i` birlikte artar. `i + 1 < n` koşulu önce denetlenir; metnin sonunda `charAt` dizinin dışına çıkmaz.
- **Yazma:** iç döngü bittiğinde `i` koşunun son karakterindedir. Sayı ve karakter sonuca eklenir. Dış döngünün `i++`'ı `i`'yi bir sonraki koşunun başına taşır.

İç döngü `i`'yi de artırdığı için kod iç içe döngüye rağmen doğrusaldır: her karakter bir kez ya dış döngüde ya da iç döngüde ziyaret edilir. Toplam süre $O(n)$'dir. Sonuç `StringBuilder` ile kurulur; `String` birleştirmesi her eklemede yeni bir nesne yaratır ve $O(n^2)$ süreye yol açabilirdi.

Koşu	Başta `i`	`c`	İç döngü turu	Sonda `i`, `sayac`	`sonuc`
1	0	A	4	4, 5	"5A"
2	5	B	2	7, 3	"5A3B"
3	8	C	3	11, 4	"5A3B4C"
4	12	D	2	14, 3	"5A3B4C3D"

Tablo 5.1.15: `rleSikistir("AAAAABBBCCCCDDD")` izlemesi ($n = 15$).



Şekil 5.1.21: Solda ilk koşunun sonu: *i = 4*, *sayac = 5*; "5A" yazılmıştır. Sağda son durum: 15 karakter yerine 8.

5.1.8.3 Açma ve çok basamaklı sayılar

RLE'nin açılması sıkıştırmanın tersidir: her (sayı, karakter) çifti için karakter sayı kadar yazılır. Metin biçimindeki RLE'de tek bir incelik vardır: sayılar birden çok basamaklı olabilir. "12A3B" çözülürken "1" tek başına okunursa çıktı “bir tane 2” gibi anlamsız bir şeye döner. Doğru okuma, rakamları bir harf gelene kadar biriktirip tek bir sayı oluşturmaktır.

```
String rleAc(String s) {
  StringBuilder sonuc = new StringBuilder();
  int sayi = 0;
  for (char c : s.toCharArray()) {
    if (Character.isDigit(c)) {
      sayi = sayi * 10 + (c - '0');
    } else {
      for (int k = 0; k < sayi; k++)
        sonuc.append(c);
      sayi = 0;
    }
  }
  return sonuc.toString();
}
```

Kodda `sayi`, o ana kadar okunan rakamlardan oluşan sayıdır. Her rakam geldiğinde onluk sistemdeki gibi eski değer 10 ile çarpılıp rakam eklenir: "12" için önce 1, sonra $1 \cdot 10 + 2 = 12$. Harf gelince harf `sayi` kez yazılır ve `sayi` sıfırlanır. Kod Java ile denenmiştir: `rleAc("12A3B1C")` on iki A, üç B ve bir C üretir; `rleAc("5A3B4C3D")` özgün metni geri verir. Süre, çıktının uzunluğuyla doğrusaldır.

Bu yöntemin bir sınırı vardır: metnin kendisinde rakam varsa sayılarla karakterler karışır. "3" karakterinin 12 kez tekrarı "123" olarak yazılırdı ve "123 tane ne?" sorusu yanıtsız kalırdı. Bayt düzeyinde RLE bu sorunu sayıları ayrı baytlarda tutarak çözer.

5.1.8.4 Bayt düzeyinde RLE ve bayrak baytı

Gerçek dosyalarda RLE baytlar üzerinde çalışır. En basit biçimde her koşu iki baytla yazılır: önce sayı, sonra karakterin kodu. Sayı tek bayta sığmalıdır; 255'ten uzun koşular birden çok koşuya bölünür. Bu biçim, rakam sorununu ortadan kaldırır, ama tekrarsız bölgelerde her karakteri iki bayta çıkarır.

Bayraklı RLE bu kaybı azaltır. Yalnız uzun koşular özel bir **bayrak baytıyla** başlayan üç baytlık bir grupla yazılır: bayrak, sayı, karakter. Kısa koşulardaki karakterler olduğu gibi bırakılır. Açıcı bir bayrak baytı görünce sonraki iki baytı sayı ve karakter olarak okur; başka bir bayt görünce onu doğrudan çıktıya yazar.

Örnek 5.1.8

Metin "aaaaaaaaaabbbbbbeceeceeceeceddddddddddddddddecb" (46 bayt) olsun. Bayrak baytı 255'tir; a = 97, b = 98, c = 99, d = 100, e = 101.

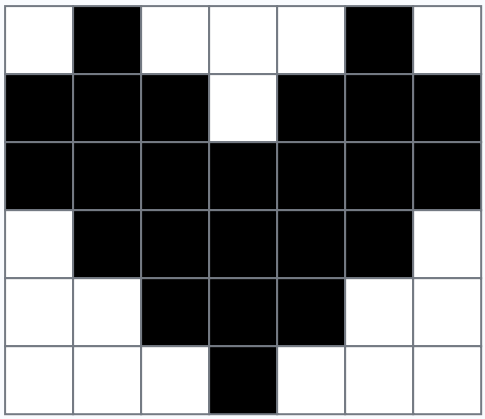
1. **Koşular.** 10 a ve 6 b'nin ardından tek karakterlik on iki koşu gelir (e c e c … e c). Sonra 15 d ve tek karakterlik e, c, b vardır: toplam 18 koşu.
2. **Bayraksız RLE.** Her koşu 2 bayt: $18 \cdot 2 = 36$ bayt. Dizi şöyle başlar: 10 97 06 98 01 101 01 99 …; tekrarsız bölgedeki her karakter 2 bayt tutar.
3. **Bayraklı RLE.** Üç uzun koşu üçer baytla yazılır: 255 10 97, 255 06 98 ve 255 15 100. Bu 9 bayttır. Kalan 15 tek karakter olduğu gibi kalır: 15 bayt. Toplam $9 + 15 = 24$ bayt.
4. **Oranlar.** Bayraksız $36/46 \approx 0{,}78$, bayraklı $24/46 \approx 0{,}52$.

Bayraklı biçimin tam dizisi: 255 10 97 255 06 98 101 99 101 99 101 99 101 99 101 99 101 99 255 15 100 101 99 98.

Bayrak baytının değeri metinde de geçebilir. 255 değerli bir karakter doğrudan yazılsaydı açıcı onu bayrak sanardı. Bu yüzden bu karakter de bir kaçış (*escape*) biçimiyle, örneğin 255 1 255 üçlüsüyle yazılmalıdır. Koşunun kaç karakterden itibaren “uzun” sayılacağı da bir tasarım kararıdır. Üç baytlık grup ancak dört ve daha uzun koşularda kazanç sağlar. Üç karakterlik koşuda iki yazım aynı uzunluktadır; iki karakterlik koşuyu olduğu gibi yazmak daha kısadır.

5.1.8.5 Görüntülerde RLE

RLE'nin en etkili olduğu yer, büyük düz alanlar içeren görüntülerdir. Siyah-beyaz bir görüntüde her satır soldan sağa taranır ve renk değişene kadar aynı renkteki pikseller sayılır.

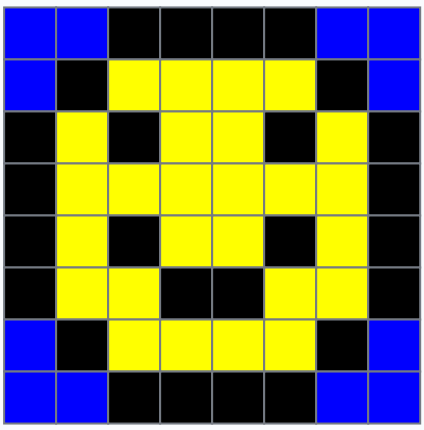


Şekil 5.1.22: 7 × 6 piksellik siyah-beyaz kalp görüntüsü ve satır satır RLE kodları. W beyaz, B siyah koşudur; harften sonraki sayı koşu uzunluğudur.

Şekil 5.1.22'deki 42 piksellik görüntü 18 koşuyla yazılır. Satırlar ayrı ayrı kodlanır; ilk satır "W1B1W3B1W1" olur ve beş koşudan oluşur. Her satırın koşu uzunlukları toplamı 7'dir. Bu bir denetim olanağı sağlar: açıcı her satırda 7 piksel üretmezse veri bozuktur.

Yalnız iki renk olduğu için renk adı bile yazılmayabilir. Koşuların dönüşümlü olarak beyaz ve siyah olduğu kabul edilir; satır siyahla başlıyorsa ilk beyaz koşu 0 uzunluklu yazılır. Faks standardı bu fikri kullanır ve koşu uzunluklarını ayrıca değişken uzunluklu kodlarla sıkıştırır. Büyük düz alanlı görüntülerde (çizimler, metin taramaları) RLE çok etkilidir. Fotoğraflarda ise komşu pikseller nadiren tam olarak aynı renktedir ve RLE etkisiz kalır.

Renkli görüntülerde koşular piksel düzeyinde tanımlanır. Her pikselin rengi üç baytlık bir (R, G, B) üçlüsüdür ve bir koşu, aynı renkteki ardışık piksellerden oluşur.



Şekil 5.1.23: 8 × 8 piksellik gülen yüz. Renkler mavi (0, 0, 255), siyah (0, 0, 0) ve sarıdır (255, 255, 0). Pikseller satır satır, soldan sağa okunur.

Örnek 5.1.9

Şekil 5.1.23'teki görüntünün ham ve RLE boyutları karşılaştırılacak. Her koşu bir sayı baytı ve üç renk baytıyla yazılır.

1. **Ham boyut.** 64 piksel, her biri 3 bayt: $64 \cdot 3 = 192$ bayt.
2. **Koşular.** Pikseller satır satır okunur. Satır sonunda aynı renk sürüyorsa koşu bir sonraki satıra taşar. İlk satırın son iki mavisi ile ikinci satırın ilk mavisi tek bir "3 mavi" koşusudur. Görüntü 33 koşuya ayrılır. İlk koşular 2 mavi, 4 siyah, 3 mavi, 1 siyah, 4 sarı, 1 siyahtır.
3. **RLE boyutu.** $33 \cdot (1 + 3) = 132$ bayt. Oran $132/192 \approx 0{,}69$.

Koşular satır sınırında kesilseydi koşu sayısı 38'e çıkardı ve RLE boyutu $38 \cdot 4 = 152$ bayt olurdu. Küçük ve ayrıntılı bir görüntüde kazanç sınırlıdır; aynı yüz 800 × 800 piksel çizilseydi koşular yüz kat uzar ve oran çok küçülürdü.

5.1.9 Lempel-Ziv-Welch (LZW)

RLE yalnız ardışık tekrarı görür. Doğal dil metninde ise tekrarlar çoğunlukla ardışık değildir: "ve", "bir", "için" gibi kelimeler ya da "-ler", "-dır" gibi ekler metnin her yerine dağılmıştır. LZW bu tür tekrarı yakalar.

5.1.9.1 Sözlük fikri

Abraham Lempel ve Jacob Ziv 1977 ve 1978'de sözlük tabanlı iki sıkıştırma yöntemi yayımladı (LZ77 ve LZ78). Terry Welch 1984'te LZ78'i sadeleştirdi ve donanımda hızla çalışacak bir biçime getirdi. Ortaya çıkan yöntem LZW adını aldı ve GIF, TIFF, PDF biçimlerinde ve Unix'in `compress` programında kullanıldı.

Yöntemin özü şudur: sıkıştırıcı daha önce gördüğü dizgileri bir sözlükte numaralandırır. Aynı dizgiyle yeniden karşılaşınca dizginin kendisi yerine sözlükteki numarasını, yani kodunu yazar. Sözlük **dinamiktir**: sıkıştırma ilerledikçe büyür. Sözlük çıktıyla birlikte gönderilmez. Açıcı aynı sözlüğü, sıkıştırıcıyla aynı kuralları uygulayarak kendisi yeniden kurar.

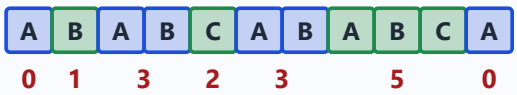
Tanım 5.1.8

LZW sıkıştırması metindeki bütün tek karakterleri içeren bir başlangıç sözlüğüyle başlar. w boş dizgiyle başlar. Metnin her c karakteri için:

- $w + c$ sözlükteyse $w \leftarrow w + c$ yapılır; eşleşme uzatılır.
- Değilse w'nin kodu çıktıya yazılır, $w + c$ sözlüğe bir sonraki boş kodla eklenir ve $w \leftarrow c$ yapılır.

Metin bitince boş değilse w'nin kodu da çıktıya yazılır.

Kural bir açgözlülük içerir: sıkıştırıcı her adımda sözlükte bulunan en uzun parçayı yazar. Ardından bu parçanın bir karakter uzatılmış hâlini sözlüğe ekler. Böylece sözlükteki dizgiler zamanla uzar. Sık tekrar eden parçalar giderek daha uzun sözlük girdileriyle ve tek bir kodla yazılır.



Şekil 5.1.24: "ABABCABABCA" metninin LZW parçaları ve kodları: A 0, B 1, AB 3, C 2, AB 3, ABC 5, A 0. 11 sembol yerine 7 kod yazılır.

5.1.9.2 Sıkıştırma: Java kodu ve izleme

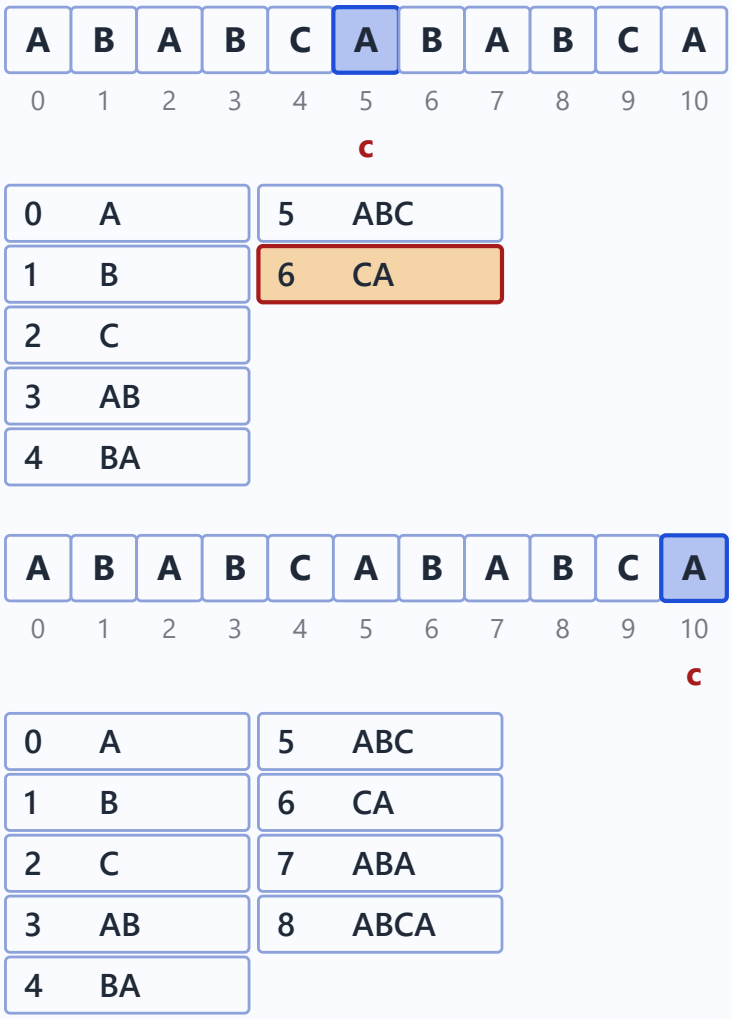
```
List<Integer> lzw(String s) {
  Map<String, Integer> sozluk =
      ilkSozluk(s);  // A=0, B=1, C=2
  String w = "";
  List<Integer> kod = new ArrayList<>();
  for (char c : s.toCharArray()) {
    String wc = w + c;
    if (sozluk.containsKey(wc)) {
      w = wc;
    } else {
      kod.add(sozluk.get(w));
      sozluk.put(wc, sozluk.size());
      w = "" + c;
    }
  }
  if (!w.isEmpty())
    kod.add(sozluk.get(w));
  return kod;
}
```

- **Sözlük:** `Map<String, Integer>`, her dizgiyi koduna eşler. `ilkSozluk(s)` metindeki farklı karakterleri 0'dan başlayarak numaralandırır; bu örnekte A = 0, B = 1, C = 2.
- **Uzatma:** `wc = w + c` sözlükteyse yalnız `w` uzar; çıktı yazılmaz.
- **Yazma ve ekleme:** `wc` sözlükte yoksa `w`'nin kodu yazılır. `wc`, `sozluk.size()` koduyla eklenir; bu, her zaman bir sonraki boş koddur. `w` yalnız `c`'den oluşan dizgiye döner.
- **Son `w`:** döngü bittiğinde `w`'de henüz yazılmamış bir parça kalır ve kodu eklenir.

Kod `lzw("ABABCABABCA")` çağrısıyla izlenecek. Tablo 5.1.16 her karakterden sonraki durumu verir.

Adım	`c`	`w + c`	Sözlükte?	Çıktı	Eklenen	Yeni `w`
1	A	A	evet (0)	—	—	A
2	B	AB	hayır	0	3 = AB	B
3	A	BA	hayır	1	4 = BA	A
4	B	AB	evet (3)	—	—	AB
5	C	ABC	hayır	3	5 = ABC	C
6	A	CA	hayır	2	6 = CA	A
7	B	AB	evet (3)	—	—	AB
8	A	ABA	hayır	3	7 = ABA	A
9	B	AB	evet (3)	—	—	AB
10	C	ABC	evet (5)	—	—	ABC
11	A	ABCA	hayır	5	8 = ABCA	A
son	—	—	—	0	—	—

Tablo 5.1.16: `lzw("ABABCABABCA")` izlemesi. Başlangıç sözlüğü A = 0, B = 1, C = 2.



Şekil 5.1.25: Solda 6. adım: "CA" sözlükte yoktur; C'nin kodu 2 yazılır ve 6 = CA eklenir. Sağda son durum: sözlük 0–8 kodlarına büyümüş, çıktı 0 1 3 2 3 5 0 olmuştur.

Çıktı *0 1 3 2 3 5 0*'dır: 11 sembol yerine 7 kod. 10. adım sözlüğün nasıl işe yaradığını gösterir. 5. adımda eklenen "ABC", metnin ikinci yarısında yeniden görülür ve tek bir kodla (5) yazılır. Sözlük bu kadar kısa bir metinde ancak birkaç kez kullanılabilir. Uzun metinlerde sözlükteki dizgiler uzar ve kazanç artar.

Sürenin hesabı şöyledir: her karakter için bir sözlük araması ve en çok bir ekleme yapılır. Sözlük hash tablosuyla tutulursa her işlem ortalama *O(1)* beklenir ve toplam süre *O(n)*'dir. Bu hesapta bir incelik vardır: `w + c` dizgisi her adımda yeniden oluşturulur ve hash'i hesaplanır; maliyeti `w`'nin uzunluğuyla orantılıdır. Pratik uygulamalar bu yüzden sözlüğü bir trie olarak tutar ya da her girdiyi (önek kodu, son karakter) çiftiyle saklar. Böylece her adım gerçekten sabit süre alır.

5.1.9.3 Açma

Açıcı yalnız başlangıç sözlüğünü bilir. Sözlüğün geri kalanını, kodları okurken sıkıştırıcıyla aynı sırada kendisi kurar. Kural şudur: her yeni k kodu için $e = sözlük[k]$ çıktıya yazılır, sözlüğe $w + e[0]$ eklenir ve $w \leftarrow e$ yapılır. $e[0]$, e'nin ilk karakteridir.

Eklenen dizginin neden $w + e[0]$ olduğu sıkıştırıcının davranışından anlaşılır. Sıkıştırıcı w'yi yazdığında sözlüğe $w + c$'yi eklemişti; c, bir sonraki parçanın ilk karakteriydi. Açıcı w'yi çözdüğü anda bir sonraki parçayı henüz bilmez. Bir sonraki kodu çözünce parçanın ilk karakterini öğrenir ve eksik girdiyi tamamlar. Açıcı bu yüzden sözlükte hep bir adım geride kalır.

Kod	Çıktı (e)	Eklenen ($w + e[0]$)
0	A	—
1	B	3 = AB
3	AB	4 = BA
2	C	5 = ABC
3	AB	6 = CA
5	ABC	7 = ABA
0	A	8 = ABCA

Tablo 5.1.17: *0 1 3 2 3 5 0* kodlarının açılması (A = 0, B = 1, C = 2).

Çıktılar art arda yazılınca A·B·AB·C·AB·ABC·A = ABABCABABCA elde edilir; özgün metin geri gelmiştir. Açıcının kurduğu 3–8 kodları, sıkıştırıcınınkilerle birebir aynıdır.

Bir durumda bu kural tıkanır. Açıcı, sözlüğünde henüz bulunmayan bir kod okuyabilir. Bu ancak tek bir biçimde olur: kod, açıcının bir sonra ekleyeceği koddur.

Teorem 5.1.4 (LZW'nin özel durumu)

Açıcı, sözlüğünde henüz tanımlı olmayan bir k kodu okursa k bir sonraki boş koddur ve $e = w + w[0]$'dır. Burada w, bir önceki kodun çözümüdür.

İspat. **Kanıt.** Açıcı bir adım geride olduğu için sıkıştırıcının tanımladığı ama açıcının henüz tanımlamadığı tek kod, sıkıştırıcının en son eklediği koddur. Sıkıştırıcı bu kodu w'yi yazarken $w + c$ olarak eklemiştir; c bir sonraki parçanın ilk karakteridir. Bir sonraki parça da tam bu yeni koddur: $e = w + c$. e, w ile başladığı için ilk karakteri $w[0]$'dır. c de bir sonraki parçanın, yani e'nin ilk karakteridir. Buradan $c = e[0] = w[0]$ ve $e = w + w[0]$ çıkar. ■

Bu durum, sıkıştırıcının bir girdiyi ekledikten hemen sonra kullandığı zaman ortaya çıkar. Bunun için metinde $c \cdot S \cdot c \cdot S \cdot c$ biçiminde bir parça bulunmalı ve cS sözlükte olmalıdır. Burada c bir karakter, S bir dizgidir. Bu yüzden kaynaklarda “cScSc durumu” diye anılır.

Örnek 5.1.10

A = 0, B = 1 sözlüğüyle "ABABABA" sıkıştırılıp geri açılacak.

1. **Sıkıştırma.** AB sözlükte yoktur: 0 yazılır, 2 = AB eklenir. BA yoktur: 1 yazılır, 3 = BA eklenir. AB vardır, ABA yoktur: 2 yazılır, 4 = ABA eklenir ve w = A olur. Son üç karakterde AB ve ABA sözlükte bulunur; w ABA'ya uzar. Metin bitince kalan w'nin kodu 4 yazılır. Çıktı *0 1 2 4*'tür.
2. **Açma.** 0 → A. 1 → B; 2 = AB eklenir. 2 → AB; 3 = BA eklenir.
3. **Özel durum.** Sıradaki kod 4'tür, ama açıcının sözlüğü 3'e kadardır. Teoreme göre $e = w + w[0]$ = AB + A = ABA. Çıktıya ABA yazılır ve 4 = ABA eklenir.
4. **Sonuç.** A·B·AB·ABA = ABABABA ✓.

5.1.9.4 Örnek: DNA dizisi

LZW alfabeye bağlı değildir. DNA dizilerinde başlangıç sözlüğü dört nükleotittir: A = 0, T = 1, G = 2, C = 3. Metin "ATGATCATGAG" (11 sembol) olsun.

Kod	Sözlüğe eklenen	Çıktı
4	AT	0 (A)
5	TG	1 (T)
6	GA	2 (G)
7	ATC	4 (AT)
8	CA	3 (C)
9	ATG	4 (AT)
10	GAG	6 (GA)
—	(son w = G)	2 (G)

Tablo 5.1.18: "ATGATCATGAG" metninin LZW ile sıkıştırılması. Her satırda sözlüğe eklenen dizgi ve o anda yazılan kod verilmiştir.

Çıktı *0 1 2 4 3 4 6 2*'dir: 11 sembol yerine 8 kod. "AT" ikinci ve üçüncü kez görüldüğünde tek kodla (4) yazılır; "GA" da ikinci kez görüldüğünde tek kodla (6) yazılır. Döngüden sonra kalan w = G'nin kodu (2) en sona eklenir. Gerçek genomlarda tekrar eden diziler çok uzun olduğu için sözlük girdileri de uzar.

5.1.9.5 ASCII sözlük ve özel durum

Pratikte başlangıç sözlüğü 256 ASCII karakterinin hepsidir; metinde hangi karakterlerin geçtiğine bakılmaz. Örneğin a = 97, b = 98, c = 99'dur. Yeni dizgiler 256'dan itibaren numaralanır. Böylece başlangıç sözlüğünü gönderme gereği de ortadan kalkar.

Çıktı	Sözlüğe eklenen
97 (a)	256 = aa
97 (a)	257 = ab
98 (b)	258 = ba
256 (aa)	259 = aac
99 (c)	260 = ca
257 (ab)	261 = aba
261 (aba)	262 = abac
99 (c)	263 = cb
98 (b)	(son *w* = b)

Tablo 5.1.19: "aabaacababacb" (13 karakter) metninin ASCII sözlüklü LZW ile sıkıştırılması.

Metin 9 koda iner: 97 97 98 256 99 257 261 99 98. Tablonun altıncı ve yedinci satırları özel durumu içerir. "ab" (257) yazılırken sözlüğe 261 = "aba" eklenir. Bir sonraki parça tam olarak "aba"dır ve 261 hemen kullanılır. Açıcı 261'i okuduğunda sözlüğünde henüz 260'a kadar kod vardır.

Açma kodu, özel durumu tek bir koşullu ifadeyle ele alır:

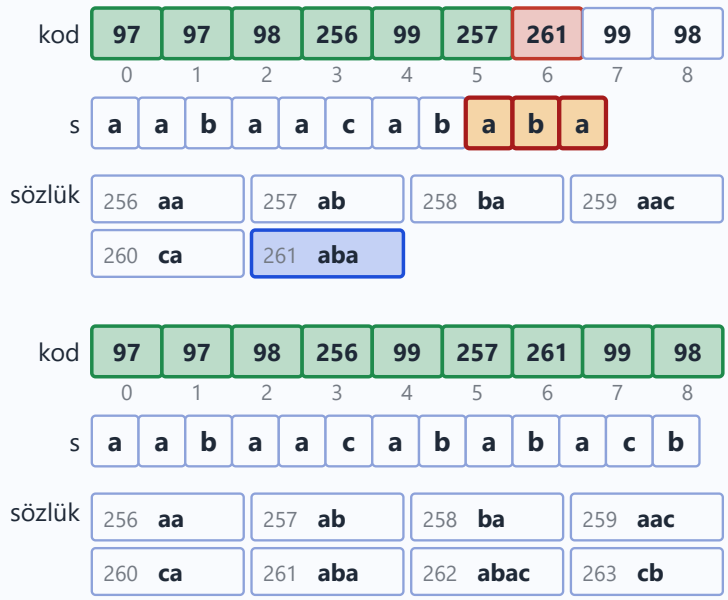
```
String lzwAc(List<Integer> kod) {
  List<String> sozluk =
      ilkSozluk();      // 0..255: ASCII
  String w = sozluk.get(kod.get(0));
  StringBuilder s = new StringBuilder(w);
  for (int i = 1; i < kod.size(); i++) {
    int k = kod.get(i);
    String e = k < sozluk.size()
        ? sozluk.get(k)
        : w + w.charAt(0); // özel durum
    s.append(e);
    sozluk.add(w + e.charAt(0));
    w = e;
  }
  return s.toString();
}
```

- **Başlangıç:** sözlük 0–255 ASCII karakterleriyle dolar. İlk kod her zaman tek karakterdir; doğrudan çözülür ve çıktıya yazılır.

- **Çözme:** `k < sozluk.size()` ise kod tanımlıdır ve `e = sozluk.get(k)`. Değilse özel durumdur ve `e = w + w.charAt(0)`.
- **Ekleme:** her adımda `w + e.charAt(0)` sözlüğün sonuna eklenir; `List` yapısında yeni girdinin kodu kendiliğinden bir sonraki indistir.

`k`	Sözlükte?	`e`	Eklenen (`w + e[0]`)	Çıktı (birikimli)
97	ASCII	a	—	a
97	ASCII	a	256 = a + a = aa	aa
98	ASCII	b	257 = a + b = ab	aab
256	evet	aa	258 = b + a = ba	aabaa
99	ASCII	c	259 = aa + c = aac	aabaac
257	evet	ab	260 = c + a = ca	aabaacab
261	hayır (en büyük 260)	ab + a = aba	261 = ab + a = aba	aabaacababa
99	ASCII	c	262 = aba + c = abac	aabaacababac
98	ASCII	b	263 = c + b = cb	aabaacababacb

Tablo 5.1.20: `lzwAc([97, 97, 98, 256, 99, 257, 261, 99, 98])` izlemesi.



Şekil 5.1.26: Solda özel durum: 261 okunduğunda açıcının sözlüğü 260'ta biter; *e = w + w[0]* = "aba". Sağda son durum: 9 kod 13 karakterlik metne dönmüş, 256–263 sözlüğü sıkıştırıcınınkiyle aynı kurulmuştur.

Kod Java ile çalıştırılmış ve "aabaacababacb" çıktısı doğrulanmıştır. Açmanın süresi çıktının uzunluğuyla doğrusaldır.

5.1.9.6 Sözlük büyüklüğü ve kod genişliği

Kodların kaç bitle yazılacağı sözlüğün büyüklüğüne bağlıdır. 256 ASCII karakteri 8 bitle yazılabilir. Sözlüğe yeni girdiler eklendiğinde kodlar 8 bite sığmaz. Bu yüzden LZW uygulamaları 9 bitlik kodlarla başlar ve sözlük 512, 1024… sınırlarını geçtikçe kod genişliğini birer bit artırır.

Bu hesap, kod sayısındaki azalmanın neden doğrudan bit kazancı olmadığını gösterir. ASCII örneğinde metin 13 bayt, yani 104 bittir. 9 kod 9'ar bitle $9 \cdot 9 = 81$ bit tutar; oran $81/104 \approx 0{,}78$'dir. "ABABCABABCA" örneği ASCII ile kodlansaydı 11 bayt (88 bit) yerine 7 kod (63 bit) yazılırdı. Kısa metinlerde kazanç küçüktür. Uzun ve tekrarlı metinlerde tek bir kod onlarca karakterin yerini tutar ve kazanç büyür.

Sözlük büyüdükçe iki sorun çıkar: kod genişliği (bit sayısı) artar ve sözlüğün kapladığı bellek büyür. Pratikte sözlük sınırlandırılır. GIF'te en büyük kod genişliği 12 bittir; sözlük en çok $2^{12} = 4096$ girdi tutar. Sözlük dolunca ya donar (yeni girdi eklenmez) ya da sıfırlanıp yeniden kurulur. Sıfırlama, metnin içeriği değiştiğinde (örneğin bir dosyanın metin bölümünden sayısal bölümüne geçildiğinde) eski girdilerin yer kaplamasını önler.

> **Sık yapılan hata**
>
> İki hata LZW'de çok sık görülür. Birincisi, sıkıştırmada döngüden sonra kalan *w*'nin kodunu yazmayı unutmaktır. "ABABCABABCA" örneğinde bu, son A'nın (0) kaybolmasına yol açar ve açılan metin "ABABCABABC" olur. İkincisi, açmada henüz tanımlanmamış kodu ele almamaktır. `sozluk.get(k)`, `k == sozluk.size()` olduğunda `IndexOutOfBoundsException` fırlatır. Hata ancak cScSc biçimli metinlerde ortaya çıktığı için basit testlerden kaçabilir.

5.1.9.7 LZ78: LZW'nin atası

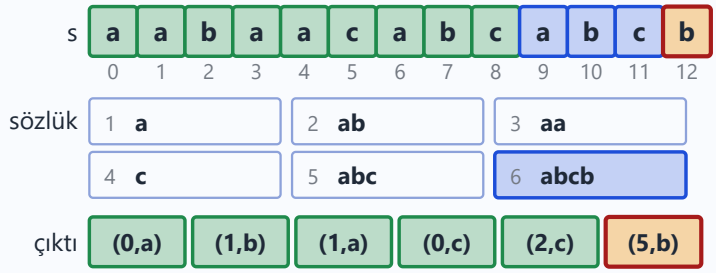
LZW'nin dayandığı LZ78 yönteminde sözlük boş başlar. Çıktı kodlardan değil (indis, karakter) çiftlerinden oluşur: sözlükteki en uzun eşleşmenin indisi (eşleşme yoksa 0) ve onu izleyen karakter. Her çift aynı zamanda sözlüğe yeni bir girdi ekler.

```
List<String> lz78(String s) {
  Map<String, Integer> sozluk =
      new HashMap<>();
  List<String> cikti = new ArrayList<>();
  String w = "";
  for (char c : s.toCharArray()) {
    if (sozluk.containsKey(w + c)) {
      w = w + c;               // uzat
    } else {
      int k = w.isEmpty() ? 0
                          : sozluk.get(w);
      cikti.add("(" + k + ", " + c + ")");
      sozluk.put(w + c, sozluk.size() + 1);
      w = "";
    }
  }
  if (!w.isEmpty())        // son parça
    cikti.add("(" + sozluk.get(w) + ")");
  return cikti;
}
```

Kod LZW koduna çok benzer. Farklar şunlardır: `w + c` sözlükte yoksa hem `w`'nin indisi hem de `c` karakteri yazılır; ardından `w` boş dizgiye döner. İndisler 1'den başlar, çünkü 0 boş önek için ayrılmıştır. Metin bittiğinde `w`'de yazılmamış bir parça kalabilir; bu parça sözlükte vardır ve döngüden sonra yalnız indisiyle yazılır. Örneğin "aabaacababacb" metninde son b, sözlükte 6 numaralı girdidir ve çıktının sonuna "(6)" eklenir. Bu satır unutulursa metnin sonu kaybolur. Tablodaki örnekte metin bir çiftle tam biter, `w` boş kalır ve son satır çalışmaz.

Konum	`c`	`w + c`	Sözlükte?	Çıktı	Eklenen
0	a	a	hayır	(0, a)	1 = a
1	a	a	evet	—	—
2	b	ab	hayır	(1, b)	2 = ab
3	a	a	evet	—	—
4	a	aa	hayır	(1, a)	3 = aa
5	c	c	hayır	(0, c)	4 = c
6–7	a, b	a, ab	evet, evet	—	—
8	c	abc	hayır	(2, c)	5 = abc
9–11	a, b, c	a, ab, abc	evet ×3	—	—
12	b	abcb	hayır	(5, b)	6 = abcb

Tablo 5.1.21: `lz78("aabaacabcabcb")` izlemesi. "—" çıktı yazılmayan adımlardır.



Şekil 5.1.27: LZ78'in son durumu: metin a | ab | aa | c | abc | abcb parçalarına ayrılmış, altı çift yazılmış ve sözlük 1–6 girdilerine büyümüştür.

Metin 6 çifte dönüşür: (0, a) (1, b) (1, a) (0, c) (2, c) (5, b). Parçalar a, ab, aa, c, abc ve abcb'dir; her parça, sözlükteki bir parçanın bir karakter uzatılmış hâlidir.

Welch'in değişikliği tek bir fikirden ibarettir: sözlüğü bütün tek karakterlerle başlatmak. LZ78'de bir karakter ilk kez görüldüğünde (0, c) çifti gerekir, çünkü sözlükte yoktur. LZW'de her tek karakter baştan sözlüktedir. Bu yüzden çıktıda karakter yazmaya gerek kalmaz; yalnız kodlar yazılır ve her kodun ardından gelen karakter, bir sonraki kodun ilk karakterinden çıkarılır.

Lempel ve Ziv'in 1977 tarihli öbür yöntemi LZ77 ise sözlük tutmaz. Onun yerine son birkaç kilobaytlık bir kayan pencerede geriye bakar ve tekrarları (geri uzaklık, uzunluk) çiftleriyle yazar. LZ77 ailesi, ZIP ve gzip'in kullandığı DEFLATE biçiminin ve PNG resimlerinin temelidir.

5.1.9.8 RLE ile LZW'nin karşılaştırması

	RLE	LZW
Yakaladığı fazlalık	aynı sembolün ardışık tekrarı	tekrar eden dizgiler (ardışık olmasa da)
Ek yapı	yok	sözlük (hash tablosu ya da trie)
Zaman	*O(n)*	*O(n)* (sözlük erişimi ortalama *O(1)*)
En kötü durum	tekrar yoksa çıktı iki katına çıkar	kısa ya da rastgele veride kazanç az, kod genişliği büyür
Kullanım	faks, BMP/PCX, basit grafikler	GIF, TIFF, PDF, Unix `compress`

Tablo 5.1.22: RLE ile LZW'nin karşılaştırması.

İki yöntem de kayıpsızdır ve ikisi de doğrusal sürede çalışır. Seçimi verinin yapısı belirler. Uzun düz alanlı görüntülerde RLE basitliğiyle öne çıkar. Doğal dil metninde ya da programlarda ardışık tekrar azdır, ama tekrar eden kelimeler ve kalıplar boldur; burada LZW çok daha etkilidir. Güvercin yuvası ilkesi ikisinin de bazı girdileri büyüteceğini söyler: RLE için "abababab", LZW için kısa ve rastgele veriler bu türdendir.

5.1.10 Sık yapılan hatalar

Bölüm boyunca anılan hatalar burada bir arada toplanmıştır. Hepsi küçük bir kod ayrıntısından doğar, ama yanlış sonuç, çökme ya da sonsuz döngü üretir.

- **Döngü sınırı.** Kaba kuvvette `i < n - m` yazmak son kaymayı ($s = n - m$) atlar. Doğrusu `i <= n - m`'dir. Örüntü metnin en sonunda geçiyorsa bulunamaz.
- **LPS'yi yanlış tanımlamak.** Örüntünün kendisi sayılmaz: *lps("aaaa") = [0, 1, 2, 3]*'tür, *[1, 2, 3, 4]* değildir. KMP'de uyuşmazlıkta `lps[q - 1]` kullanılır, `lps[q]` değil.
- **Negatif kaydırma.** Boyer-Moore'da $j - son[c]$ sıfır ya da negatif olabilir: c'nin son geçişi j'nin sağındaysa böyle olur. `max(1, …)` unutulursa örüntü geri gider ya da algoritma sonsuz döngüye girer.
- **Horspool tablosuna son karakteri katmak.** Son karakter tabloya girerse onun kaydırması 0 olur ve örüntü yerinde sayar. Tablo yalnız *P[0..m−2]*'den kurulur.
- **Hash eşitliğine güvenmek.** Rabin-Karp'ta hash eşitliği eşleşme demek değildir; aday pencere karakter karakter doğrulanmalıdır. Java'da `%` negatif sonuç verebilir: `if (th < 0) th += Q;` satırı gerekir. Büyük Q ile `int` taşar; `long` kullanılmalıdır.
- **RLE'de çok basamaklı sayılar.** "12A" çözülürken sayıyı tek rakam okumak hatalıdır (bir tane '2' mi?). Rakamlar bir harf gelene kadar biriktirilmeli ya da sayılar ayrı baytlarda yazılmalıdır.
- **LZW'de son *w* ve özel durum.** Döngüden sonra kalan w'nin kodunu yazmayı unutmak metnin sonunu kaybettirir; aynı satır LZ78'de de gereklidir. Açarken henüz tanımlanmamış kodu ($w + w[0]$) ele almamak cScSc biçimli metinlerde çökmeye yol açar.
- **Karşılaştırma sayısını yanlış saymak.** Bir hizadaki uyuşmazlık da bir karşılaştırmadır. Kaba kuvvette her hiza en az bir karşılaştırma yapar; toplam en az $n - m + 1$'dir.

5.1.11 Özet

- Dizgi eşleştirme, m uzunluklu bir P örüntüsünün n uzunluklu bir T metni içinde geçtiği bütün kaymaları bulma problemidir. Olası kaymalar $0 \le s \le n - m$'dir.
- **Kaba kuvvet** her kaymayı soldan sağa dener. Ön işlemesi ve ek belleği yoktur. Ortalama durumda $O(n)$, en kötü durumda $(n - m + 1) \cdot m$ karşılaştırmayla $O(n \cdot m)$'dir. Kusuru, uyuşmazlıkta öğrenilen bilgiyi atmasıdır.
- **KMP**, $lps[q]$ = $P[0..q]$'nun en uzun uygun önek-sonek uzunluğu tablosunu $O(m)$'de kurar. Uyuşmazlıkta $q = lps[q - 1]$ yapılır ve metin işaretçisi geri gitmez. En kötü durumda bile $O(n + m)$'dir; amortize argüman en çok $2n$ karşılaştırma verir.
- **Boyer-Moore** sağdan sola karşılaştırır ve $s \mathrel{+}= max(1, j - son[c])$ (kötü karakter) ile iyi sonek kuralının büyüğü kadar kaydırır. En iyi durumda $O(n/m)$'dir, pratikte en hızlısıdır. Horspool, kaydırmayı hizanın son karakterine göre yapan tek tablolu sadeleştirmedir.

- **Rabin-Karp**, polinom kayan hash'i *O(1)*'de günceller. Hash eşitse pencere doğrulanır. Beklenen süre *O(n + m)*, en kötü süre *O(n · m)*'dir. Aynı uzunlukta çok sayıda örüntüyü tek geçişte arayabilir.
- Hiçbir kayıpsız sıkıştırma yöntemi her girdiyi küçültemez (güvercin yuvası ilkesi). Sıkıştırma oranı, sıkıştırılmış boyutun orijinal boyuta oranıdır.
- **RLE** her koşuyu (sayı, sembol) çiftiyle yazar ve *O(n)* sürer. Tekrar yoksa veriyi iki katına çıkarır; bayrak baytı bu kaybı azaltır. Düz alanlı görüntülerde etkilidir.
- **LZW**'de *w + c* sözlükteyse *w* uzatılır; değilse *w*'nin kodu yazılır ve *w + c* sözlüğe eklenir. Açıcı sözlüğü kendisi kurar ve bir adım geride kalır. Tanımsız kod okununca *e = w + w[0]*'dır. LZ78'in farkı, sözlüğün boş başlaması ve (indis, karakter) çiftleri yazmasıdır.

5.1.12 Kavram listesi

dizgi *string* · alfabe *alphabet* · alt dizgi *substring* · örüntü *pattern* · dizgi eşleştirme *string matching* · önek *prefix* · sonek *suffix* · kaba kuvvet *brute force (naive)* · Knuth-Morris-Pratt (KMP) *Knuth-Morris-Pratt* · LPS tablosu *LPS table* · amortize analiz *amortized analysis* · Boyer-Moore algoritması *Boyer-Moore algorithm* · kötü karakter kuralı *bad character rule* · iyi sonek kuralı *good suffix rule* · Rabin-Karp algoritması *Rabin-Karp algorithm* · kayan hash *rolling hash* · sıkıştırma *compression* · çalışma uzunluğu kodlaması (RLE) *run-length encoding* · Lempel-Ziv-Welch (LZW) *Lempel-Ziv-Welch*

Sözlükte ayrıca yer almayan, notta tanımlanan terimler: kayma (*shift*), uygun önek (*proper prefix*), son görülme tablosu, Horspool kaydırma tablosu, sahte eşleşme (*spurious hit*), polinom hash, kayıpsız ve kayıplı sıkıştırma, sıkıştırma oranı, koşu (*run*), bayrak baytı, LZ78.

5.1.13 Kendini Dene

1. "abaabab" örüntüsünün LPS tablosu nedir?

   Cevap

   [0, 0, 1, 1, 2, 3, 2]. *q = 3*'te "abaa"nın en uzun önek-sonek parçası "a"dır. *q = 5*'te "abaaba" için "aba" (3) bulunur. *q = 6*'da "abaa" ile "abab" uyuşmaz; *k = lps[2] = 1*'e çekilir ve "ab" (2) bulunur.

2. `kabaKuvvet("aaaab", "ab")` hangi değeri döndürür ve kaç karşılaştırma yapar? Yalnız kötü ka-

rakter kuralıyla Boyer-Moore kaç karşılaştırma yapar?

Cevap

3 döner. Kaba kuvvet *s = 0, 1, 2*'de a eşleşir ve b uyuşmaz (3 × 2 = 6), *s = 3*'te iki karakter eşleşir (2): toplam 8. Boyer-Moore her hizada sağ uçtaki b'yi metindeki a ile karşılaştırır; *son[a] = 0* olduğundan 1 kayar. *s = 0, 1, 2*'de 1'er, *s = 3*'te 2 karşılaştırma: toplam 5.

3. "aabaab" örüntüsü için KMP aramasında "aabaa" eşleşmişken (*q = 5*) sıradaki metin karakteri b değil. *q* hangi değeri alır, örüntü kaç konum kaymış olur?

Cevap

LPS tablosu *[0, 1, 0, 1, 2, 3]*'tür. *q = lps[4] = 2*: "aabaa"nın sonundaki "aa" aynı zamanda önektir. Kayma *5 − 2 = 3*'tür. Ardından aynı metin karakteri *P[2]* = b ile karşılaştırılır; o da uyuşmazsa *q = lps[1] = 1* olur.

4. "ANANAS" örüntüsünün son görülme tablosu nedir? "BANANAS ANANAS" metninde yalnız kötü karakter kuralıyla ilk kaydırma kaç olur?

Cevap

son[A] = 4, *son[N] = 3*, *son[S] = 5*, diğerleri −1. *s = 0*'da *P[5]* = S ile *T[5]* = A uyuşmaz; kaydırma *max(1, 5 − 4) = 1*. *s = 1*'de altı karakter eşleşir: toplam 7 karşılaştırma.

5. "ANANAS" için Horspool kaydırma tablosu nedir? Hizanın son karakterinin altına S gelirse örüntü kaç kayar?

Cevap

Tabloya yalnız *P[0..4]* = ANANA katılır: A → *5 − 4 = 1*, N → *5 − 3 = 2*, diğer → 6. S yalnız son konumda geçtiği için tabloda yoktur ve 6 kaydırır.

6. *B = 10*, *Q = 7*, karakterler rakam. "27182818" metninde "18" aranıyor. Hangi pencereler aday olur, hangileri sahte eşleşmedir?

Cevap

h("18") = 18 mod 7 = 4. Pencere hash'leri: 27 → 6, 71 → 1, 18 → 4, 82 → 5, 28 → 0, 81 → 4, 18 → 4. Adaylar *s = 2, 5, 6*'dır. *s = 5*'teki "81" sahte eşleşmedir; *s = 2* ve *s = 6* gerçek geçişlerdir.

7. "WWWWWWWWWWWWBWWW" (16 karakter) metni karakter RLE'siyle nasıl yazılır? Bayt dü-

zeyinde bayraksız ve bayraklı (dört ve daha uzun koşular bayrakla) biçimler kaç bayt tutar?

Cevap

"12W1B3W": 7 karakter, oran *7/16 ≈ 0,44*. Bayraksız: 3 koşu × 2 = 6 bayt. Bayraklı: 12 W için 3 bayt, B ve üç W olduğu gibi 4 bayt: 7 bayt. Bu girdide tekrarsız bölge olmadığı için bayraksız biçim daha kısadır.

8. A = 0 sözlüğüyle "AAAAAAA" (7 tane A) LZW ile sıkıştırılırsa hangi kodlar çıkar? Açıcı bu kodları çözerken kaç kez özel durumla karşılaşır?

Cevap

Çıktı *0 1 2 0*; sözlüğe 1 = AA, 2 = AAA, 3 = AAAA eklenir. Açarken 0 → A. 1 tanımsızdır: *e* = A + A = AA. 2 de tanımsızdır: *e* = AA + A = AAA. 0 → A. Özel durum iki kez çıkar; çıktı A·AA·AAA·A = 7 A.

9. LZ78, on tane a'dan oluşan "aaaaaaaaaa" dizgisini hangi çiftlere dönüştürür?

Cevap

(0, a) (1, a) (2, a) (3, a). Parçalar a, aa, aaa, aaaa'dır; uzunlukları *1 + 2 + 3 + 4 = 10*. Parça uzunlukları her adımda bir arttığı için *n* uzunluklu bir koşu yaklaşık $\sqrt{(2n)}$ çiftle yazılır.

10. Bir milyon karakterlik bir metinde, metinde hiç geçmeyen karakterlerden oluşan 10 harflik bir örüntü aranıyor. Boyer-Moore yaklaşık kaç karşılaştırma yapar? Kaba kuvvet ve KMP en az kaç karşılaştırma yapar?

Cevap

Boyer-Moore'da her hizada sağ uçtaki karakter uyuşmaz ve örüntü 10 konum atlar: yaklaşık $n/m = 10^5$ karşılaştırma. Kaba kuvvet ve KMP her metin karakterine en az bir kez bakar: yaklaşık 10^6 karşılaştırma. Bu en iyi durumda Boyer-Moore on kat daha az iş yapar.

5.1.14 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 32: Rabin-Karp'ı, sonlu otomatla eşleştirmeyi ve KMP'nin önek fonksiyonunu kanıtlarıyla verir; sonek dizilerine de değinir.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 5.3 ve 5.5: KMP'nin otomat sürümünü, Boyer-Moore ve Rabin-Karp'ın Java kodlarını, RLE, Huffman ve LZW sıkıştırmasını anlatır.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (1997): iyi sonek tablosunun *O(m)* sürede kurulmasını, Z algoritmasını ve biyoinformatik uygulamalarını ayrıntılı ele alır.

- K. Sayood, *Introduction to Data Compression* (5. baskı, 2017): sözlük tabanlı yöntemleri (LZ77, LZ78, LZW) ve kayıpsız sıkıştırmanın bilgi kuramsal sınırlarını anlatır.

5.1.15 Kaynaklar

1. R. S. Boyer, J S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM* 20(10), 1977.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 32: String Matching).
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014 (Bölüm 13).
4. R. N. Horspool, "Practical Fast Searching in Strings", *Software: Practice and Experience* 10(6), 1980.
5. R. M. Karp, M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms", *IBM Journal of Research and Development* 31(2), 1987.
6. D. E. Knuth, J. H. Morris, V. R. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing* 6(2), 1977.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Kesim 5.3: Substring Search, 5.5: Data Compression).
8. T. A. Welch, "A Technique for High-Performance Data Compression", *IEEE Computer* 17(6), 1984.
9. J. Ziv, A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding", *IEEE Transactions on Information Theory* 24(5), 1978.

KISIM II · BÖLÜM 5.2

İleri Dizgi Algoritmaları

Metin, bilgisayarın en çok işlediği veri türlerinden biridir. Bir sözlüğün kelimeleri sıraya konur, bir form alanındaki e-posta adresi denetlenir. Bir yazım denetleyicisi yanlış yazılmış kelimeye en yakın doğru kelimeyi önerir. Bir sıkıştırma programı ise dosyayı küçültmeden önce harflerin yerini değiştirir. Bu işlerin hepsi dizgiler üzerinde çalışan algoritmalara dayanır. Not dört problem ailesini sırayla ele alır. İlk aile sıralamadır: dizgilerin sözlüksel sırası, karşılaştırmalı sıralama ve karşılaştırma yapmayan LSD ve MSD basamak sıralaması. İkinci aile ayrıştırmadır: düzenli ifadeler ve onların kuramsal karşılığı olan sonlu durum makineleri. Üçüncü aile benzerliktir: Levenshtein uzaklığı, dinamik programlama tablosu ve Hamming, en uzun ortak alt dizi gibi öteki ölçüler. Dördüncü aile dönüşümdür: sonek dizisi ve Burrows–Wheeler dönüşümü. Her algoritma için Java kodu, adım adım izleme ve maliyet hesabı verilir.

ÖĞRENME ÇIKTILARI

Sıralama

Dizgileri sözlüksel sırayla karşılaştırmak; LSD ve MSD basamak sıralamasını adım adım uygulamak ve $O(W \cdot (N + R))$ maliyetini açıklamak.

Ayrıştırma

Düzenli ifadeler yazıp okumak; bir sonlu durum makinesini (DFA) tasarlamak, çizmek ve bir girdi üzerinde çalıştırmak.

Benzerlik

Levenshtein uzaklığını dinamik programlama tablosuyla hesaplamak, düzenleme dizisini tablodan geri okumak; Hamming, LCS ve en uzun ortak alt dizgi ölçüleriyle karşılaştırmak.

Dönüşüm

Sonek dizisini kurmak ve aramada kullanmak; Burrows–Wheeler dönüşümünü hesaplamak ve tersine çevirmek.

5.2.1 Dört problem ailesi

Bir **dizgi** (*string*), belirli bir alfabeden alınmış karakterlerin sonlu dizisidir. **Alfabe** (*alphabet*) kullanılabilecek karakterlerin kümesidir ve Σ ile gösterilir. İkili dizgilerde $\Sigma = \{0, 1\}$'dir, DNA dizilerinde $\Sigma = \{A, C, G, T\}$'dir. Java'da ise bir `String` içindeki her karakter 16 bitlik bir `char` değeridir. Alfabenin büyüklüğü R ile gösterilir: ASCII için $R = 128$ ya da genişletilmiş biçimiyle $R = 256$'dır.

Dizgiler üzerine kurulan problemler çok çeşitlidir; ama çoğu dört aileden birine girer. Bu ailelerin her biri farklı bir soruya cevap verir ve farklı bir algoritma tekniği kullanır.

Aile	Soru	Algoritmalar	Temel teknik
Sıralama (*string sorting*)	Dizgiler hangi sırada dizilir?	sözlüksel karşılaştırma, LSD ve MSD basamak sıralaması	karşılaştırma, sayma
Ayrıştırma (*parsing*)	Dizgi belirli bir biçime uyuyor mu? Hangi parçalardan oluşuyor?	düzenli ifadeler, sonlu durum makineleri	otomat
Benzerlik (*similarity*)	İki dizgi birbirine ne kadar yakın?	Levenshtein, Hamming, LCS, en uzun ortak alt dizgi	dinamik programlama
Dönüşüm (*transformation*)	Dizgi, aramayı ya da sıkıştırmayı kolaylaştıran bir biçime nasıl getirilir?	sonek dizisi, Burrows–Wheeler dönüşümü	sıralama, ikili arama

Tablo 5.2.1: Dizgi algoritmalarının dört ailesi.

Sıralama ailesinde dizgiler sözlükteki gibi dizilir. Dizgileri karakter karakter karşılaştıran yöntemler genel amaçlıdır. Basamak sıralaması ise hiç karşılaştırma yapmaz; her karakteri bir basamak gibi kullanıp dizgileri gruplara dağıtır. Ayrıştırma ailesinde bir arama örüntüsü **düzenli ifadeyle** yazılır ve bu ifadeyi tanıyan bir **sonlu durum makinesi** girdiyi karakter karakter okur. Benzerlik ailesinde iki dizgi arasındaki fark bir sayıyla ölçülür. En bilineni, bir dizgiyi ötekine çeviren en az tek karakterlik düzenleme sayısı olan Levenshtein uzaklığıdır. Dönüşüm ailesinde dizgi yeniden düzenlenir: **sonek dizisi** bütün sonekleri sıralı tutarak aramayı hızlandırır, **Burrows–Wheeler dönüşümü** ise aynı harfleri bir araya toplayarak sıkıştırmaya hazırlık yapar.

Dört ailenin ortak bir noktası vardır: hepsi dizginin bir *yapısını* kullanır. Sözlüksel sıra, karakterlerin soldan sağa önemini kullanır. Sonlu durum makinesi, girdinin şimdiye kadar okunan kısmını tek bir durumla özetler. Levenshtein uzaklığı, iki dizginin önekleri arasındaki uzaklıklardan kurulur. Sonek dizisi ve BWT ise dizginin bütün sonekleri ya da döndürmeleri sıralandığında ortaya çıkan düzene dayanır. Notta önek, sonek ve alt dizgi terimleri sık geçer. **Önek** (*prefix*) dizginin başından, **sonek** (*suffix*) sonundan alınan parçadır. **Alt dizgi** (*substring*) ardışık karakterlerden oluşan herhangi bir parçadır. Örneğin “banana” için “ban” bir önek, “ana” bir sonek, “nan” bir alt dizgidir.

5.2.2 Sözlüksel sıra

Bir sözlükte kelimeler belirli bir düzende dizilir: “armut”, “elma”dan önce gelir, “elma” da “elmas”tan önce. Bu düzenin kuralı basittir ve bilgisayarda da aynen kullanılır. Tek fark, bilgisayarın harflerin sırasını bir alfabe ezberinden değil, karakter kodlarından öğrenmesidir.

5.2.2.1 Karşılaştırma kuralı

Tanım 5.2.1

a ve *b* iki dizgi olsun. *a*'nın *b*'den sözlüksel olarak küçük olması ($a < b$) için iki durumdan biri geçerli olmalıdır. (1) İki dizginin ilk farklı karakteri bir k konumundadır ve $a[k] < b[k]$'dir. (2) İki dizgi *a*'nın sonuna kadar eşittir ve *a*, *b*'den kısadır, yani *a*, *b*'nin bir önekidir. Bu sıraya **sözlüksel sıra** (*lexicographic order*) denir.

Kural karşılaştırmayı ilk karakterlerle başlatır. İlk karakterler eşitse ikinci karakterlere, onlar da eşitse üçüncülere bakılır. İlk farklı karakter sonucu belirler ve sonraki karakterlere hiç bakılmaz. Bir dizgi tükenirse kısa olan önce gelir. Tablo 5.2.2 dört karşılaştırmada kararı neyin verdiğini gösterir.

Karşılaştırma	Karar veren	Sonuç
apple · banana	$a < b$ (1. karakter)	apple önce
apple · apricot	$p < r$ (3. karakter)	apple önce
car · cart	“car” tükendi	car önce
Zebra · apple	'Z' (90) < 'a' (97)	Zebra önce

Tablo 5.2.2: Dört sözlüksel karşılaştırma ve kararı veren karakter.

Son satır ilk bakışta şaşırtıcıdır: bir insan “apple”ı “Zebra”dan önce yazar. Bilgisayar ise büyük Z'nin kodunu (90) küçük a'nın kodundan (97) küçük bulur ve harfin büyük ya da küçük olmasını ayrıca dikkate almaz.

Sözlüksel sıra bir **tam sıradır**: herhangi iki farklı dizgiden biri ötekinden küçüktür ve sıra geçişlidir. Bu yüzden dizgiler tam sayılar gibi sıralanabilir; sıralama algoritmalarının hepsi dizgilerle de çalışır. Yalnız bir karşılaştırmanın maliyeti artık sabit değildir. İki tam sayı tek işlemle karşılaştırılır. İki dizgi ise en kötü durumda kısa olanın uzunluğu kadar karakter karşılaştırması gerektirir.

5.2.2.2 Kod izleme: apple ve apricot

Kuralı doğrudan uygulayan Java metodu aşağıdadır. Metot negatif bir sayı döndürürse *a* önce, pozitif bir sayı döndürürse *b* önce gelir; sıfır, iki dizginin eşit olduğunu gösterir. Java'nın `String.compareTo` metodu da aynı biçimde çalışır.

```
int karsilastir(String a, String b) {
  int n = Math.min(a.length(), b.length());
  for (int k = 0; k < n; k++) {
    char x = a.charAt(k), y = b.charAt(k);
    if (x != y)
      return x - y;   // ilk farklı karakter
  }
  return a.length() - b.length();
}
```

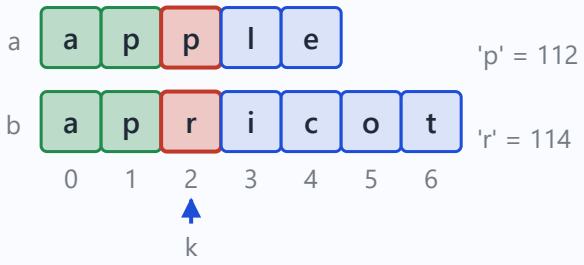
`n`, iki dizgiden kısa olanın uzunluğudur; döngü en fazla bu kadar karakter karşılaştırabilir. Döngünün her turunda iki dizginin `k`. karakterleri `x` ve `y` olarak okunur. Karakterler farklıysa `x - y` döndürülür. Java'da iki `char` çıkarıldığında karakter kodlarının farkı elde edilir; işaret, hangi karakterin önce geldiğini gösterir. Döngü bir fark bulmadan biterse kısa dizgi uzun olanın önekidir. Bu durumda `a.length() - b.length()` döner: kısa olan önce gelir, uzunluklar eşitse sonuç sıfırdır.



Şekil 5.2.1: `karsilastir("apple", "apricot")` çağrısının başlangıcı. `n = min(5, 7) = 5`; en çok beş karakter karşılaştırılabilir.

`k`	`x`	`y`	**Karşılaştırma**	**Sonuç**
0	'a' (97)	'a' (97)	eşit	sonraki karaktere geçilir
1	'p' (112)	'p' (112)	eşit	sonraki karaktere geçilir
2	'p' (112)	'r' (114)	farklı	`return 112 - 114 = -2`

Tablo 5.2.3: `karsilastir("apple", "apricot")` izlemesi.



Şekil 5.2.2: İlk farklı karakter 2 numaralı konumdadır: 'p' (112) ile 'r' (114). Sonuç *−2 < 0*'dır; “apple”, “apri-

cot"tan önce gelir.

Sonuç negatif olduğu için "apple", "apricot"tan önce gelir. Karşılaştırma üç karakterde bitti; "apple"ın son iki harfine ve "apricot"un son dört harfine hiç bakılmadı. Aynı metot başka girdilerle şu değerleri döndürür: `karsilastir("car", "cart")` için ilk üç karakter eşittir ve kısa dizgi tükenir; sonuç *3 − 4 = −1*'dir. `karsilastir("Zebra", "apple")` ilk karakterde *90 − 97 = −7* döndürür. `karsilastir("kale", "kaba")` ise üçüncü karakterde *108 − 98 = 10* döndürür; pozitif sonuç "kaba"nın önce geldiğini gösterir.

Metodun maliyeti, ilk farklı karaktere kadar okunan karakter sayısıyla orantılıdır. En kötü durumda iki dizgi uzun bir ortak önek paylaşır ve maliyet *O(min(|a|, |b|))* olur. Ek bellek kullanılmaz.

Sık yapılan hata

Dönüş değerinin yalnız −1, 0 ya da 1 olabileceğini sanmak. `compareTo` ve `karsilastir` karakter kodlarının farkını ya da uzunluk farkını döndürür: −2, −7, 10 gibi değerler olağandır. Bu yüzden sonucu `== -1` ile değil, `< 0` ile sınamak gerekir.

5.2.2.3 Karakter kodları ve Türkçe

Bilgisayar için "alfabe", karakter kodlarının oluşturduğu tablodur: ASCII ya da onu içine alan Unicode. Bu tabloda bütün büyük harfler (A = 65 … Z = 90) bütün küçük harflerden (a = 97 … z = 122) önce gelir. Rakamlar (0 = 48 … 9 = 57) ise harflerden de öncedir. Türkçeye özgü ç, ğ, ı, ö, ş, ü harflerinin kodları ASCII tablosunun dışındadır: örneğin ç = 231, ğ = 287, ı = 305, ş = 351'dir. Bu yüzden kod sırasıyla dizildiklerinde hepsi z'den sonra gelir.

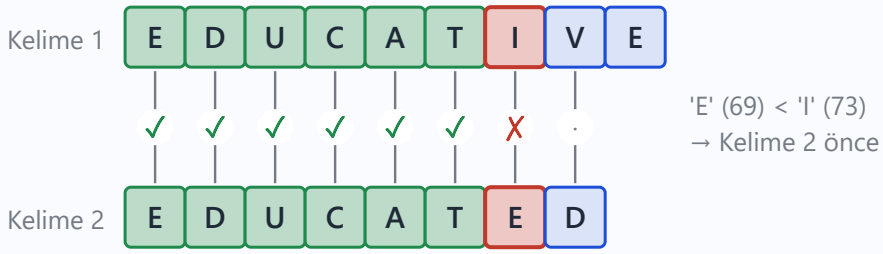
Sonuç olarak `compareTo` ile yapılan sıralama Türkçe sözlük sırasından farklıdır. "şeker" ile "zeytin" karşılaştırıldığında 'ş' (351) 'z'den (122) büyük olduğu için "zeytin" önce gelir. Türkçe sözlükte ise ş, z'den öncedir. Benzer biçimde "Zebra" "apple"dan önce, "Çorum" ise "Zonguldak"tan sonra gelir. Türkçe sözlük sırası gerekiyorsa Java'nın `java.text.Collator` sınıfı Türkçe yerel ayarla (`Collator.getInstance(new Locale("tr", "TR"))`) kullanılır. Bu sınıf büyük/küçük harf farkını ve Türkçe harflerin alfabedeki yerini dikkate alır. Yalnız büyük/küçük harf farkını gidermek için `compareToIgnoreCase` da yeterlidir, ama bu metot Türkçe harflerin yerini düzeltmez.

Sık yapılan hata

Kod sırasını alfabe sırası sanmak. `{"b2", "B10", "ab", "Ab", "a"}` dizisi `Arrays.sort` ile sıralanınca sonuç `Ab, B10, a, ab, b2` olur: büyük harfle başlayanlar öne geçer. "a", "ab"nin öneki olduğu için ondan önce gelir. İnsan alfabesine uygun sıra için `Collator` ya da harfleri önceden aynı biçime getiren bir normalleştirme gerekir.

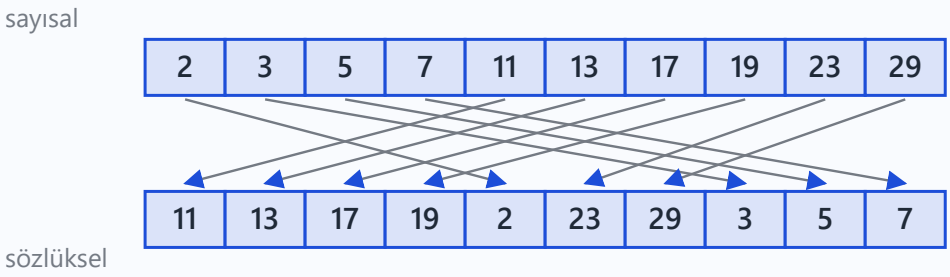
5.2.2.4 Kelimeler ve sayılar

Uzun ortak önekli iki kelimede karşılaştırma birkaç karakter sürebilir. Şekil 5.2.3 “EDUCATIVE” ile “EDUCATED” kelimelerini karşılaştırır. İlk altı harf (EDUCAT) eşittir. Yedinci harfte 'I' (73) ile 'E' (69) karşılaşır; 'E' daha küçük olduğu için “EDUCATED” önce gelir. Kelimelerin uzunluğu sonuca etki etmez: “EDUCATIVE” dokuz, “EDUCATED” sekiz harflidir, ama karar yedinci harfte verilmiştir.



Şekil 5.2.3: EDUCATIVE ile EDUCATED: ilk altı harf eşittir, yedinci harfte 'E' (69) < 'I' (73) olduğu için EDUCATED önce gelir.

Sayılar dizgi olarak saklandığında sözlüksel sıra, sayısal sıradan ayrılır. Karşılaştırma yine soldan sağa, karakter karakter yapılır. “11” ile “2” karşılaştırılırken ilk karakterler '1' ve '2'dir; '1' küçük olduğu için “11” önce gelir. Sayının kaç basamaklı olduğu hiç dikkate alınmaz. Şekil 5.2.4, 2 ile 29 arasındaki asal sayıları iki sırada gösterir.



Şekil 5.2.4: 2–29 arasındaki asal sayılar. Üst satır sayısal sıradır; alt satır aynı sayıların dizgi olarak sözlüksel sırasıdır: 11, 13, 17, 19, 2, 23, 29, 3, 5, 7.

Aynı nedenle “123”, “45”, “6”, “789” dizgileri sözlüksel olarak tam bu sıradadır: ilk karakterler 1 < 4 < 6 < 7'dir. Sayısal sıra isteniyorsa iki yol vardır. Birincisi, dizgiler tam sayıya çevrilip sayı olarak sıralanır. İkincisi, bütün sayılar soldan sıfırla aynı uzunluğa getirilir: “02” < “11” olur, çünkü artık ilk karakterler '0' ve '1'dir. Dosya adlarında “resim01, resim02, ..., resim10” biçimi bu yüzden kullanılır.

Örnek 5.2.1

"10", "9", "100", "2" dizgileri sözlüksel sıraya konsun. İlk karakterler '1', '9', '1', '2'dir. '1' ile başlayan "10" ve "100" en öne gelir. Bu ikisinden "10", "100"ün önekidir ve önce gelir. Ardından '2' ile başlayan "2", en son '9' ile başlayan "9" gelir. Sonuç: *"10" < "100" < "2" < "9"*. Sayısal sıra ise 2, 9, 10, 100'dür.

5.2.2.5 Karşılaştırmalı dizgi sıralama

Sözlüksel karşılaştırma elde olduğuna göre bilinen herhangi bir sıralama algoritması dizgilere uygulanabilir. Sunumdaki örnek en basit yöntemi kullanır: her konuma, geri kalan dizgilerin sözlüksel olarak en küçüğü takaslarla getirilir.

```
void sozlukselSirala(String[] a) {
  int n = a.length;
  for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
      if (a[i].compareTo(a[j]) > 0) {
        String gecici = a[i];
        a[i] = a[j];
        a[j] = gecici;
      }
}
```

Dış döngü `i` konumunu soldan sağa ilerletir. İç döngü `i`'den sonraki her `j` için `a[i]` ile `a[j]`'yi karşılaştırır. `a[i].compareTo(a[j]) > 0` ise `a[j]` daha küçüktür ve iki öğe takas edilir. İç döngü bittiğinde `i` konumunda kalanların en küçüğü bulunur. Bu yöntem seçmeli sıralamaya benzer; farkı, en küçük öğeyi bir kez değil, her bulduğunda takas etmesidir. Tablo 5.2.4 kodun iki girdi üzerindeki çıktısını verir.

Girdi	Çıktı
eabcdf, abcde, bcdea, cdeab, deabcg	abcde, bcdea, cdeab, deabcg, eabcdf
banana, orange, grape, apple, kiwi	apple, banana, grape, kiwi, orange

Tablo 5.2.4: `sozlukselSirala` ile iki girdi.

Maliyet iki çarpandan oluşur. İç içe iki döngü $N(N-1)/2$ karşılaştırma yapar; bu $O(N^2)$'dir. Her `compareTo` çağrısı ise en fazla W karakter okur; W en uzun dizginin uzunluğudur. Toplam zaman $O(N^2 \cdot W)$ olur. Takaslar yalnız başvuruları (referansları) değiştirir; dizgilerin karakterleri kopyalanmaz. Bu yüzden takas sabit zamanlıdır ve ek bellek $O(1)$'dir.

Daha iyi bir karşılaştırmalı sıralama, örneğin birleştirme sıralaması, $O(N \log N)$ karşılaştırma yapar. Dizgilerde bu, $O(W \cdot N \log N)$ zaman demektir. Java'nın `Arrays.sort(String[])` metodu da birleştirme sıralamasının bir türevini kullanır ve kararlıdır. Karşılaştırmaya dayanan her sıralama en kötü du-

rumda en az $\Omega(N \log N)$ karşılaştırma yapmak zorundadır. Bu alt sınırın nedeni, $N!$ farklı olası sıranın her birini ayırt etmek için gereken evet/hayır sorusu sayısıdır: $\log_2(N!) \approx N \log_2 N$. Basamak sıralaması ise karakterleri karşılaştırmaz, sayar. Bu yüzden alt sınır ona uygulanmaz.

Bir sayısal örnek farkı somutlaştırır. $N = 10^6$ dizgi, her biri $W = 10$ karakter olsun. Birleştirme sıralaması yaklaşık $N \log_2 N \approx 2 \cdot 10^7$ karşılaştırma yapar; her karşılaştırma en çok 10 karakter okursa üst sınır $2 \cdot 10^8$ karakter işlemidir. LSD basamak sıralaması ise $W \cdot (N + R) = 10 \cdot (10^6 + 256) \approx 10^7$ adımda biter. Rakamlar yalnız büyüklük sırası verir; sabit çarpanlar ve önbellek davranışı gerçek süreyi değiştirir.

5.2.3 Basamak sıralaması

Karşılaştırmalı sıralama iki dizgiyi her seferinde baştan karşılaştırır. Ortak önekleri uzun dizgilerde aynı karakterler defalarca okunur. **Basamak sıralaması** (*radix sort*) başka bir yol izler: dizgiyi karakter karakter ele alır ve her karakteri bir sayının basamağı gibi kullanır. Her geçişte dizgiler tek bir karakter konumuna göre kovalara dağıtılır. Dağıtma işi karşılaştırma değil sayma ile yapılır.

İki yön vardır. **LSD** (*least significant digit*, en az anlamlı basamak) sağdan, yani son karakterden başlar ve sola doğru ilerler. **MSD** (*most significant digit*, en anlamlı basamak) soldan, ilk karakterden başlar ve her kovayı ayrı ayrı sıralar. İki yöntem de her geçişte aynı alt işlemi kullanır: tek bir karakter konumuna göre kararlı saymalı sıralama.

5.2.3.1 Saymalı sıralama ve kararlılık

Saymalı sıralama (*counting sort*), anahtarları küçük tam sayılar olan öğeleri sıralar. Burada anahtar, dizginin `d`. karakteridir ve 0–255 arasında bir koddur. Yöntem üç adımdan oluşur: her anahtarın kaç kez geçtiği sayılır, sayılar önek toplamıyla konumlara çevrilir, öğeler bu konumlara yerleştirilir.

> **Tanım 5.2.2**
>
> Bir sıralama yöntemi, anahtarları eşit olan öğelerin giriş dizisindeki göreli sırasını çıktıda da koruyorsa **kararlıdır** (*stable*). Örneğin anahtarları eşit olan x ve y girişte bu sırada geliyorsa, çıktıda da x, y'den önce bulunur.

Sunumdaki LSD kodu üç metottan oluşur. `radixSirala` geçişleri yönetir, `sayarakSirala` tek bir karakter konumuna göre kararlı saymalı sıralamayı yapar, `kar` ise dizginin `d`. karakterini güvenli biçimde okur:

```
void radixSirala(String[] a) {
  int W = enUzun(a);               // en uzun dizgi
  for (int d = W - 1; d >= 0; d--)
    sayarakSirala(a, d);           // sağdan sola
}
void sayarakSirala(String[] a, int d) {
  int[] say = new int[256];        // ASCII
  for (String s : a) say[kar(s, d)]++;
  for (int r = 1; r < 256; r++) say[r] += say[r - 1];
  String[] gec = new String[a.length];
  for (int i = a.length - 1; i >= 0; i--)   // kararlı
    gec[--say[kar(a[i], d)]] = a[i];
  System.arraycopy(gec, 0, a, 0, a.length);
}
int kar(String s, int d) {         // kısa dizgi → 0
  return d < s.length() ? s.charAt(d) : 0;
}
```

`sayarakSirala` metodunun satırları sırayla şu işleri yapar.

1. **Sayma.** `say` dizisi 256 hücrelidir; her hücre bir karakter koduna karşılık gelir. `for (String s : a) say[kar(s, d)]++;` satırı, `d`. karakteri *c* olan dizgilerin sayısını `say[c]` 'ye yazar.
2. **Önek toplamı.** `say[r] += say[r - 1]` döngüsünden sonra `say[c]`, anahtarı *c*'ye eşit ya da ondan küçük olan dizgilerin toplam sayısıdır. Bu sayı, *c* kovasının çıktı dizisinde *bittiği* konumu gösterir: *c* kovası `say[c - 1]` 'den başlar ve `say[c] - 1` 'de biter.
3. **Yerleştirme.** Dizi *sondan başa* taranır. Her dizgi kendi kovasının son boş hücresine yazılır: `gec[--say[kar(a[i], d)]] = a[i]`. Önce sayaç bir azaltılır, sonra o konuma yazılır.
4. **Geri kopyalama.** `System.arraycopy`, sıralı `gec` dizisini `a` 'ya kopyalar.

Yerleştirmenin sondan başa yapılması kararlılığı sağlar. Bir kovaya ait dizgilerden girişte en sonda olanı, kovanın en sonuna yazılır. Ondan önceki, bir önceki hücreye yazılır. Böylece kova içindeki sıra girişteki sırayla aynı kalır. `kar` metodu ise `d`, dizginin uzunluğunu aşıyorsa 0 döndürür. 0 bütün harflerin kodundan küçük olduğu için kısa dizgi, o konumda karakteri olan dizgilerin önüne geçer. Böylece "car", "cart"tan önce gelir; bu, sözlüksel sıradaki "önek önce gelir" kuralının karşılığıdır.

Örnek 5.2.2

On dizgi verilsin: `cab, dab, bad, dad, fad, ace, bee, bed, add, ebb`. Hepsi üç karakterlidir, $W = 3$. LSD'nin ilk geçişi `sayarakSirala(a, 2)`, yani son karaktere göre sıralamadır.

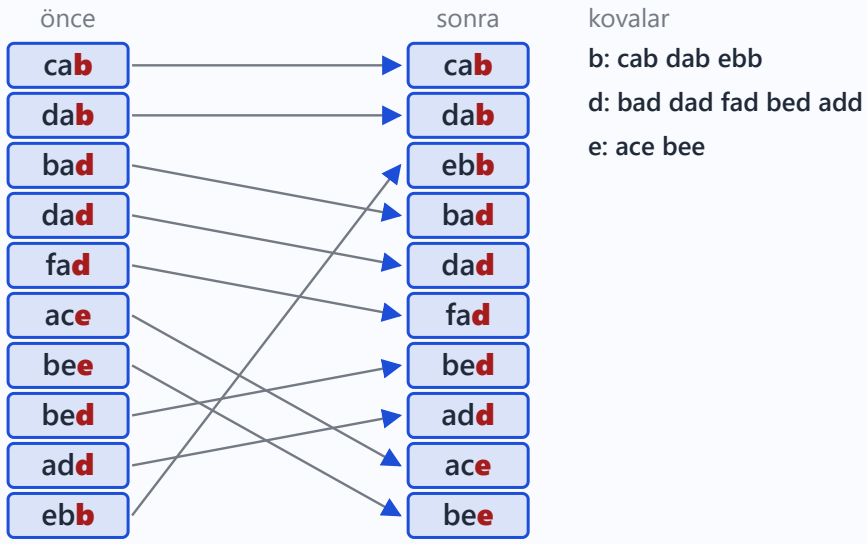
1. **Sayma.** Son karakterler sırasıyla b, b, d, d, d, e, e, d, d, b'dir. `say['b'] = 3`, `say['d'] = 5`, `say['e'] = 2`; öteki hücreler 0'dır.
2. **Önek toplamı.** `say['b'] = 3`, `say['c'] = 3`, `say['d'] = 8`, `say['e'] = 10`. b kovası 0–2, d kovası 3–7, e kovası 8–9 konumlarını kaplar.

3. **Yerleştirme (sondan başa).** `i = 9` : ebb, b kovası → `say['b']` 2'ye iner, `gec[2] = ebb`. `i = 8` : add → `gec[7]`. `i = 7` : bed → `gec[6]`. `i = 6` : bee → `gec[9]`. `i = 5` : ace → `gec[8]`. `i = 4` : fad → `gec[5]`. `i = 3` : dad → `gec[4]`. `i = 2` : bad → `gec[3]`. `i = 1` : dab → `gec[1]`. `i = 0` : cab → `gec[0]`.
4. **Sonuç.** `cab, dab, ebb, bad, dad, fad, bed, add, ace, bee`. Her kovada dizgiler giriş-teki sırayı korumuştur: b kovasında cab, dab, ebb; d kovasında bad, dad, fad, bed, add.

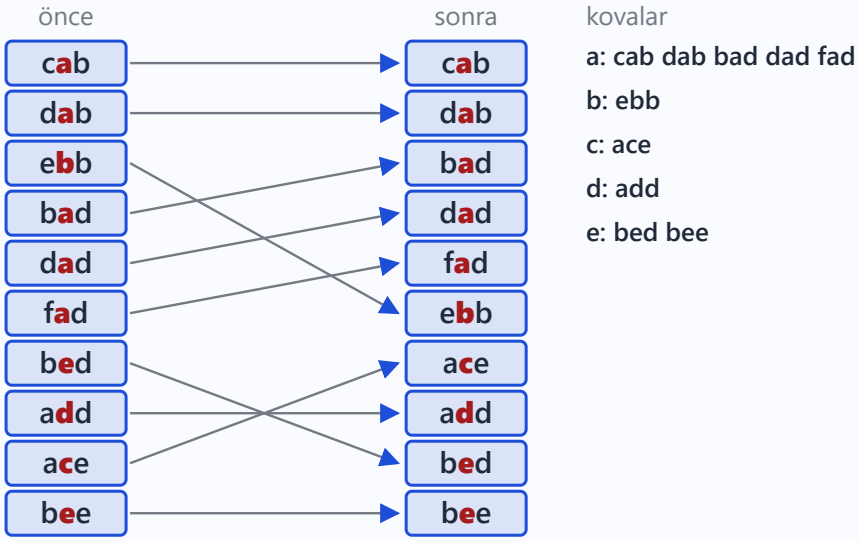
Tek bir geçişin maliyeti açıktır. Sayma ve yerleştirme döngüleri N adım, önek toplamı döngüsü $R = 256$ adım sürer. Bir geçiş $O(N + R)$ zaman ve `gec` ile `say` dizileri için $O(N + R)$ ek bellek kullanır.

5.2.3.2 LSD: sağdan sola

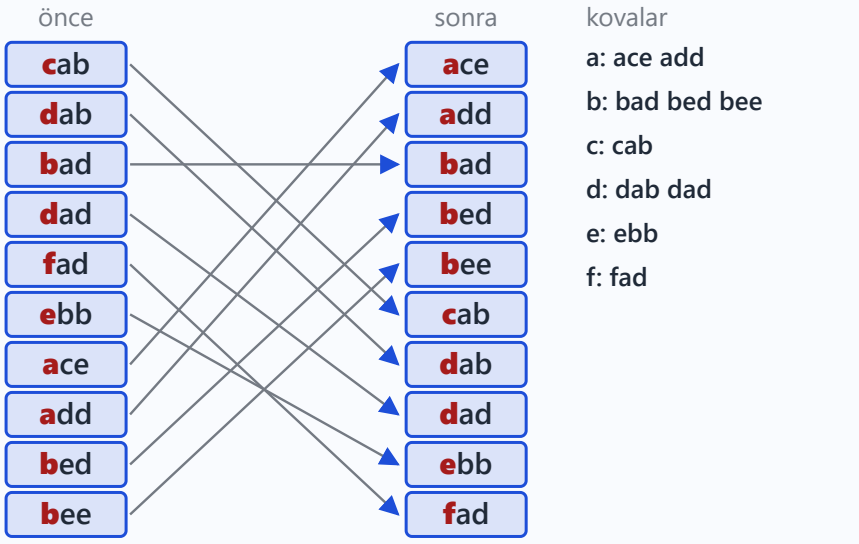
`radixSirala`, `d`'yi $W - 1$'den 0'a indirerek `sayarakSirala`'yı W kez çağırır. Önce son karaktere, sonra ortanca karaktere, en son ilk karaktere göre sıralanır. Her geçişte eşit anahtarlı dizgiler bir önceki geçişin kurduğu sırayı korur. Şekiller önceki örneğin üç geçişini gösterir.



Şekil 5.2.5: 1. geçiş (3. karakter). Dizgiler son karakterlerine göre b, d ve e kovalarına dağılır; aynı kovadakiler (ör. cab, dab) giriş sırasını korur.



Şekil 5.2.6: 2. geçiş (2. karakter). 'a' kovasında cab, dab, bad, dad, fad önceki sırayla kalır: son harfe göre b'liler d'lilerden öncedir.



Şekil 5.2.7: 3. geçiş (1. karakter). Dizi artık sözlüksel olarak sıralıdır: ace, add, bad, bed, bee, cab, dab, dad, ebb, fad.

Geçiş	Anahtar	Dizi
giriş	—	cab, dab, bad, dad, fad, ace, bee, bed, add, ebb
1 (`d = 2`)	3. karakter	cab, dab, ebb, bad, dad, fad, bed, add, ace, bee
2 (`d = 1`)	2. karakter	cab, dab, bad, dad, fad, ebb, ace, add, bed, bee
3 (`d = 0`)	1. karakter	ace, add, bad, bed, bee, cab, dab, dad, ebb, fad

Tablo 5.2.5: LSD basamak sıralamasında her geçişten sonra dizi.

İkinci geçişin sonucuna dikkatle bakmak yararlıdır. Dizi artık son iki karaktere göre sıralıdır: “ab” ile bitenler (cab, dab), “ad” ile bitenler (bad, dad, fad), sonra “bb”, “ce”, “dd”, “ed”, “ee”. Üçüncü geçiş ilk karaktere göre gruplar. Aynı ilk harfli dizgiler, örneğin bad, bed ve bee, ikinci geçişin bıraktığı sırayla kalır. O sıra da zaten son iki karaktere göre doğru sıradır.

Dizgilerin uzunlukları farklıysa `kar` eksik konumlar için 0 döndürür ve kısa dizgi sanki sonuna en küçük karakter eklenmiş gibi davranır. Örneğin `car, cart, ca, cat` için $W = 4$'tür. Dördüncü karaktere göre yapılan ilk geçişte “car”, “ca” ve “cat”ın anahtarı 0, “cart”ınki 't'dir; dizi `car, ca, cat, cart` olur. Üçüncü karaktere göre geçişte “ca”nın anahtarı 0'dır ve öne geçer: `ca, car, cart, cat`. Kalan iki geçiş bu sırayı değiştirmez. Sonuç sözlüksel sıradır.

5.2.3.3 LSD neden doğrudur?

LSD'nin en az önemli karakterden başlaması ilk bakışta ters görünür. Doğruluğunun anahtarı bir değişmezdir: her geçişten sonra dizi, o ana kadar işlenen son karakterlere göre sıralıdır.

Teorem 5.2.1 (LSD değişmezi)

Hepsi W karakterli dizgilere LSD uygulansın ve her geçiş kararlı olsun. k geçişten sonra dizi, dizgilerin son k karakterden oluşan soneklerine göre sözlüksel olarak sıralıdır. $k = W$ olduğunda dizi tamamen sıralıdır.

İspat. Tümevarım k üzerinedir. $k = 1$ için ilk geçiş son karaktere göre sıralar; iddia doğrudur. $k - 1$ geçişten sonra iddianın doğru olduğu varsayılsın. k. geçiş, sondan k. karaktere göre sıralar. İki dizgi x ve y alınsın. Bu karakterleri farklıysa geçiş onları bu karaktere göre doğru sıraya koyar; son k karakterlik soneklerin sırasına da zaten bu karakter karar verir. Bu karakterleri eşitse son k karakterlik soneklerin sırasına geri kalan $k - 1$ karakter karar verir. Varsayıma göre x ile y girişte bu $k - 1$ karaktere göre doğru sıradadır. Geçiş kararlı olduğu için eşit anahtarlı x ve y'nin göreli sırası değişmez. İki durumda da x ve y son k karaktere göre doğru sıradadır. □

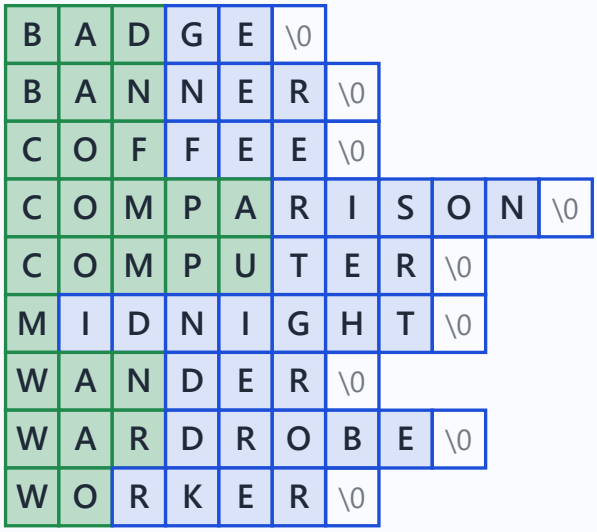
Kanıt kararlılığın neden vazgeçilmez olduğunu da gösterir. Eşit anahtarlı iki dizginin sırasına son geçiş karar vermez; o kararı önceki geçişler vermiştir. Kararsız bir alt sıralama bu kararı bozar. Örneğin yerleştirme döngüsü sondan başa değil de baştan sona yazılırsa her kova tersine dolar. Sunumdaki on dizgiyle bu bozuk sürüm `add, ace, bed, bee, bad, cab, dab, dad, ebb, fad` üretir. İlk harfler doğru sıradadır, ama aynı ilk harfli dizgiler karışmıştır.

Sık yapılan hata

LSD'de alt sıralama olarak hızlı sıralama ya da heap sort gibi kararsız bir yöntem kullanmak. Her geçiş kendi anahtarına göre doğru sıralasa da önceki geçişlerin kurduğu sıra bozulur; “bed” ile “bee” son geçişte yer değiştirebilir. Saymalı sıralamada yerleştirme sondan başa yapılmalıdır.

5.2.3.4 MSD: soldan sağa

MSD basamak sıralaması ilk karakterden başlar. Bütün dizgiler önce ilk karakterlerine göre kovalara ayrılır. Ardından her kova, bir sonraki karaktere göre *ayrı ayrı* ve özyinelemeli olarak sıralanır. Tek öğeli bir kovada ya da dizginin sonuna gelindiğinde özyineleme durur. Dizgi sonu, sunumdaki gösterimde `\0` karakteriyle belirtilir ve bütün harflerden küçük kabul edilir.



Şekil 5.2.8: MSD ile sıralanmış dokuz kelime. Yeşil hücreler, her kelimenin yerini belirlemek için okunması gereken karakterlerdir (ayırt edici önek).

Şekil 5.2.8'deki dokuz kelime ilk karaktere göre B, C, M ve W kovalarına ayrılır. M kovasında yalnız MIDNIGHT vardır; tek öğeli kova hemen biter ve MIDNIGHT için tek karakter yeter. B kovasında BADGE ve BANNER vardır. İkinci karakterleri aynı (A) olduğu için üçüncü karaktere bakılır: D < N. Bu kelimeler için üç karakter okunur. C kovasında COFFEE, COMPARISON ve COMPUTER vardır. COFFEE üçüncü karakterde (F < M) ayrılır. COMPARISON ile COMPUTER ise ancak beşinci karakterde (A < U) ayrılır; ikisi için beş karakter gerekir. W kovasında WORKER ikinci karakterde (O), WANDER ve WARDROBE üçüncü karakterde (N < R) ayrılır.

Toplam okunan karakter sayısı yeşil hücrelerin sayısıdır: *3 + 3 + 3 + 5 + 5 + 1 + 3 + 3 + 2 = 28*. Dokuz kelimenin toplam uzunluğu ise 63 karakterdir. MSD karakterlerin yarısından azını okuyarak sıralamayı bitirmiştir. LSD ise her kelimeyi en uzun kelimenin uzunluğu kadar, yani 10 kez ele alırdı. Bu, MSD'nin asıl üstünlüğüdür: yalnız **ayırt edici önekleri** inceler.

Sunum MSD'nin kodunu vermez; özyinelemeli yazımı alıştırma olarak bırakır. Aşağıdaki sürüm sunumdaki fikri doğrudan uygular. `karM`, dizgi bittiyse 0, bitmediyse karakter kodunun bir fazlasını döndürür. Böylece 0 kovası “dizgi burada bitti” anlamına gelir ve bu kova sıralamada en öne düşer.

```
void msdSirala(String[] a) {
  String[] yardimci = new String[a.length];
  msdSirala(a, yardimci, 0, a.length - 1, 0);
}
int karM(String s, int d) {          // dizgi bittiyse 0, değilse kod + 1
  return d < s.length() ? s.charAt(d) + 1 : 0;
}
void msdSirala(String[] a, String[] yardimci, int alt, int ust, int d) {
  if (alt >= ust) return;            // 0 ya da 1 öğeli kova
  int[] say = new int[258];
  for (int i = alt; i <= ust; i++) say[karM(a[i], d) + 1]++;
  for (int r = 0; r < 257; r++) say[r + 1] += say[r];
  for (int i = alt; i <= ust; i++) yardimci[say[karM(a[i], d)]++] = a[i];
  for (int i = alt; i <= ust; i++) a[i] = yardimci[i - alt];
  for (int r = 0; r < 256; r++)      // her kova bir sonraki karakterle
    msdSirala(a, yardimci, alt + say[r], alt + say[r + 1] - 1, d + 1);
}
```

Kod, LSD'deki sayma adımını bir alt dizi (`alt` – `ust`) üzerinde yapar. `say` dizisi 258 hücrelidir: dizgi sonu için bir kova, 256 karakter için 256 kova ve önek toplamı için bir fazladan hücre. Önek toplamından sonra `say[r]` , `r` kovasının alt dizideki başlangıç konumudur. Dağıtma adımı öğeleri yardımcı diziye yazar ve geri kopyalar. Son döngü, her karakter kovasını `d + 1` ile özyinelemeli sıralar. 0 kovası (bitmiş dizgiler) yeniden sıralanmaz; içindekiler birbirine eşittir. Kod sunumdaki dokuz kelimeyi `BADGE, BANNER, COFFEE, COMPARISON, COMPUTER, MIDNIGHT, WANDER, WARDROBE, WORKER` sırasına koyar. `car, cart, ca, cat, a, ""` girdisini de `"", a, ca, car, cart, cat` olarak sıralar.

MSD'nin zayıf yanları da vardır. Her özyinelemeli çağrı 258 hücrelik bir `say` dizisi oluşturur ve tarar. Birkaç öğeli küçük kovalarda bu tarama, öğelerin kendisinden çok daha pahalıdır. Bu yüzden uygulamalarda küçük alt dizilere (ör. 15 öğenin altı) eklemeli sıralama uygulanır. İkinci zayıflık, uzun ortak önekli dizgilerdir. Bütün dizgiler aynı uzun önekle başlıyorsa MSD de bütün karakterleri okumak zorunda kalır. Örneğin URL listelerinde “https://www.” öneki her dizgide ortaktır.

5.2.3.5 Karmaşıklık ve yöntem seçimi

LSD W geçiş yapar; W en uzun dizginin uzunluğudur. Her geçiş N dizgiyi ve R kovayı işler. Toplam zaman $O(W \cdot (N + R))$'dir. Alfabe sabitse ($R = 256$) ve N, R'den çok büyükse maliyet $O(W \cdot N)$'ye iner: en uzun dizginin uzunluğu çarpı öğe sayısı. Bu, girdinin toplam karakter sayısıyla aynı büyüklüktedir. Girdiyi okumak bile bu kadar sürdüğünden daha iyisi beklenemez.

Yöntem	Zaman	Ek bellek	Kararlı
Yer değiştirmeli sıralama + `compareTo`	$O(N^2 \cdot W)$	$O(1)$	hayır
Birleştirme sıralaması + `compareTo`	$O(W \cdot N \log N)$	$O(N)$	evet
LSD basamak sıralaması	$O(W \cdot (N + R))$	$O(N + R)$	evet

Yöntem	Zaman	Ek bellek	Kararlı
MSD basamak sıralaması	$O(N \cdot W)$ en kötü	$O(N + W \cdot R)$	evet

Tablo 5.2.6: Dizgi sıralama yöntemlerinin karşılaştırması (N dizgi, en uzun dizgi W karakter, alfabe R).

MSD'nin ek belleğindeki $W \cdot R$ terimi özyinelemeden gelir. Özyineleme derinliği en çok W'dir ve her düzeyde bir `say` dizisi canlı kalır. MSD'nin zamanı en kötü durumda $O(N \cdot W)$'dir; ama rastgele dizgilerde yalnız ayırt edici önekler okunduğu için çok daha azdır. R'nin büyük olduğu küçük kovalarda ise sayma dizisinin maliyeti baskın çıkar.

Seçim anahtarların yapısına bağlıdır. Anahtarlar kısa ve sabit uzunluklu olduğunda LSD idealdir: araç plakaları, posta kodları, IP adresleri, üç harfli havalimanı kodları. Örneğin $N = 10^6$ adet üç harfli kod LSD ile $3 \cdot (10^6 + 256) \approx 3 \cdot 10^6$ adımda sıralanır; birleştirme sıralaması ise yaklaşık $2 \cdot 10^7$ karşılaştırma yapar. Uzunlukları çok farklı dizgilerde LSD her dizgiyi en uzun dizginin uzunluğu kadar ele alır ve gereksiz iş yapar. Bu durumda MSD ya da karşılaştırmalı sıralama tercih edilir.

Sık yapılan hata

Basamak sıralamasını her durumda karşılaştırmalı sıralamadan hızlı sanmak. $O(W \cdot N)$, W küçükken iyidir. W, $\log N$'den büyükse, örneğin bin karakterlik birkaç yüz dizgide, karşılaştırmalı sıralama daha az iş yapabilir. Ayrıca R büyükse (ör. bütün Unicode için $R = 65.536$) her geçişteki $O(R)$ terimi küçük dizilerde maliyete egemen olur.

5.2.4 Düzenli ifadeler

Bir metinde bütün tarihleri bulmak, bir form alanına yazılan telefon numarasının doğru biçimde olup olmadığını denetlemek ya da bir günlük dosyasından hata satırlarını ayıklamak gerekebilir. Bu işlerin her biri için ayrı bir döngü yazmak yorucudur ve hataya açıktır. Düzenli ifadeler bu tür örüntüleri kısa bir dizgiyle tanımlamayı sağlar.

5.2.4.1 Düzenli ifade nedir?

Tanım 5.2.3

Düzenli ifade (*regular expression*, kısaca *regex*), bir arama örüntüsünü tanımlayan dizgidir. Bir düzenli ifade, bir dizgi kümesini betimler: ifadeye "uyan" bütün dizgiler bu kümenin öğeleridir.

Düzenli ifadeler metinde örüntüye uyan parçaları bulmak, bir dizginin bütünüyle örüntüye uyup uymadığını doğrulamak, parçaları ayıklamak ya da başka metinle değiştirmek için kullanılır. Günlük (*log*) analizi, form doğrulama, veri temizleme ve derleyicilerdeki sözcük çözümleyici (*lexer*) bunların başında gelir. Sözcük çözümleyici, kaynak kodu anahtar sözcük, değişken adı, sayı gibi parçalara ayıran ilk derleyici aşamasıdır.

Düzenli ifadelerin bir de kuramsal temeli vardır. **Kleene teoremi**, her düzenli ifadeye eşdeğer bir sonlu durum makinesi bulunduğunu ve bunun tersinin de geçerli olduğunu söyler. Düzenli ifadelerle tanımlanabilen dizgi kümelerine **düzenli diller** denir. Bu bağ, düzenli ifadelerin nasıl hızlı çalıştırıldığını açıklar: ifade önce bir otomata çevrilir, sonra metin bu otomatla taranır.

Örüntü	Metin	Eşleşmeler
`colou?r`	color colour	color, colour
`gr(a|e)y`	gray grey groy	gray, grey
`\d+`	20.99 TL, 5 adet	20, 99, 5
`\bkod\b`	kod kodlama	yalnız “kod”

Tablo 5.2.7: Dört düzenli ifade ve bir metindeki eşleşmeleri.

İlk satırda `u?`, u harfinin olabileceğini ya da olmayabileceğini söyler; Amerikan ve İngiliz yazımı birlikte yakalanır. İkinci satırda `(a|e)` iki seçenekten birini ister; “groy” uymaz. Üçüncü satırda `\d+` bir ya da daha çok rakamdan oluşan her parçayı bulur. Nokta bir rakam olmadığı için “20.99” iki ayrı parçaya bölünür. Dördüncü satırda `\b` kelime sınırıdır; “kodlama” içindeki “kod” bir kelimenin tamamı olmadığı için alınmaz.

Java'da düzenli ifadeler `java.util.regex` paketindeki `Pattern` ve `Matcher` sınıflarıyla kullanılır. `String` sınıfının `matches`, `replaceAll` ve `split` metotları da düzenli ifade alır. Java dizgisinde ters bölü bir kaçış karakteri olduğu için düzenli ifadedeki her ters bölü iki kez yazılır: `\d+` ifadesi kodda `"\\d+"` olarak görünür.

5.2.4.2 Temel işleçler

Düzenli ifadedeki sıradan karakterler kendileriyle eşleşir: `abc` yalnız “abc” parçasına uyar. Bazı karakterlerin ise özel anlamı vardır. Tablo 5.2.8 en sık kullanılan işleçleri ve her biri için bir örneği verir. Tablonun son iki sütunu, verilen dizginin *içinde* örüntüye uyan bir parça bulunup bulunmadığını gösterir; bu, Java'daki `Matcher.find` davranışıdır.

Op.	Anlamı	Örnek	İçinde bulunur	Bulunmaz
`.`	herhangi bir tek karakter (yeni satır hariç)	`a.c`	abc, a7c	ac
`*`	önceki öğenin 0 ya da daha fazla tekrarı	`ab*c`	ac, abbc	adc
`+`	önceki öğenin 1 ya da daha fazla tekrarı	`ab+c`	abc, abbc	ac
`?`	önceki öğenin 0 ya da 1 kez geçmesi	`ab?c`	ac, abc	abbc

Op.	Anlamı	Örnek	İçinde bulunur	Bulunmaz
`[ ]`	kümedeki karakterlerden biri	`[ck]ar`	car, kar	bar
`^`	dizginin başı	`^ab`	abc	cab
`$`	dizginin sonu	`ab$`	cab	abc
`( )`	gruplama; alt ifade tanımlar	`^(ab)+$`	ab, abab	aba

Tablo 5.2.8: Temel işleçler. "İçinde bulunur" sütunundaki dizgilerde `find` bir eşleşme bulur, "Bulunmaz" sütunundakilerde bulamaz.

`*`, `+` ve `?` yalnız hemen önlerindeki öğeye uygulanır. Bu öğe tek bir karakter, bir küme ya da parantezle kurulmuş bir grup olabilir. `ab*c` ifadesinde yıldız yalnız b'ye aittir: a, ardından sıfır ya da daha çok b, ardından c. Bu yüzden "ac" (sıfır b) ve "abbc" uyar, "adc" uymaz. Bütün bir parçanın tekrarlanması isteniyorsa parça gruplanır: `(ab)+`, "ab", "abab", "ababab" … ile eşleşir.

`^` ve `$` karakter değil, *konum* belirtir. `^ab` ancak dizginin en başındaki "ab" ile eşleşir; "cab" içindeki "ab" başta olmadığı için alınmaz. `^(ab)+$` ise baştan sona yalnız "ab" tekrarlarından oluşan dizgileri kabul eder. "aba"nın sonunda fazladan bir a vardır ve uymaz.

Sık yapılan hata

`ab?c` ifadesinin "abbc" içinde bir eşleşme bulacağını sanmak. `?` yalnız bir b'ye izin verir. "abbc" içinde a ile c arasında iki b vardır; a'dan hemen sonra c gelen bir parça da yoktur. Bu yüzden `find` hiçbir eşleşme bulamaz.

5.2.4.3 Özel karakterler ve kaçış

Temel işleçlerin yanında sık kullanılan birkaç kısaltma vardır. Bu kısaltmalar ters bölüyle başlar ve bir karakter sınıfını ya da bir konumu tanımlar.

İfade	Anlamı
`a`	tek bir 'a' karakteri
`abc`	tam olarak "abc"
`{ }`	belirli sayıda tekrar: `\d{4}` dört rakam
`|`	seçenekler: `kedi|köpek`
`\b`	kelime sınırı (kelimenin başı ya da sonu)
`\d`	bir rakam: `[0-9]`
`\w`	bir kelime karakteri: `[a-zA-Z0-9_]`
`\s`	boşluk karakteri: boşluk, sekme, yeni satır

Tablo 5.2.9: Özel karakterler ve kısaltmalar.

Kısaltmaların büyük harfli biçimleri tümleyendir. `\D` rakam olmayan, `\W` kelime karakteri olmayan, `\S` boşluk olmayan bir karakterle eşleşir. `\w` tek bir karakterdir; bir kelimenin tamamı için `\w+` yazılır.

Özel anlamı olan bir karakterin kendisi aranıyorsa önüne ters bölü konur. Buna **kaçış** denir. `\.` gerçek bir nokta, `\+` artı işareti, `\(` açma parantezi ile eşleşir. Kaçış unutulursa ifade beklenenden fazlasını yakalar: `a.b`, nokta herhangi bir karakter anlamına geldiği için "a+b" ile de eşleşir.

Java'da bir ayrıntı Türkçe metinlerde önemlidir. `\w` varsayılan olarak yalnız ASCII harflerini kapsar. "şeker" kelimesine `\w+` uygulandığında ilk eşleşme "eker" olur; 'ş' bir kelime karakteri sayılmaz. Türkçe harfleri de kapsayan bir harf sınıfı için Unicode özelliği `\p{L}` kullanılır: `\p{L}+`, "şeker"in tamamıyla eşleşir. Bir başka yol, örüntüyü `Pattern.UNICODE_CHARACTER_CLASS` bayrağıyla derlemektir; bu bayrak `\w`'yi de Unicode'a genişletir.

5.2.4.4 Karakter sınıfları

Köşeli parantez içine yazılan karakterler bir **karakter sınıfı** oluşturur. Sınıf, içindeki karakterlerden herhangi *biriyle* eşleşir. Tire (-) bir aralık bildirir: `[a-z]`, a ile z arasındaki kodlara sahip her küçük harfle eşleşir.

Sınıf	Eşleştiği karakter
`[abc]`	'a', 'b' ya da 'c'
`[a-z]`	herhangi bir küçük harf (ASCII)
`[A-Z]`	herhangi bir büyük harf (ASCII)
`[0-9]`	herhangi bir rakam
`[a-zA-Z_]`	harf ya da alt çizgi
`[^abc]`	'a', 'b', 'c' dışındaki herhangi bir karakter
`[^0-9]`	rakam olmayan karakter
`[^0-9]+`	"ab12cd" içinde "ab" ve "cd"

Tablo 5.2.10: Karakter sınıfları ve tümleyenleri.

`^` işaretinin iki ayrı anlamı vardır. Köşeli parantezin içinde ve başında "değil" anlamına gelir; dışarıda ise dizginin başını belirtir. Küme içinde `.`, `*` ve `+` özel değildir: `[.+]` bir nokta ya da bir artı ile eşleşir. Tire, aralık bildirdiği için kendisi aranıyorsa kümenin başına ya da sonuna yazılır: `[-+]`.

Sık yapılan hata

`[a-z]` sınıfının Türkçe küçük harfleri de kapsadığını sanmak. Aralık karakter kodlarına dayanır; ç, ğ, ı, ö, ş, ü bu aralığın dışındadır. "çiçek" kelimesine `[a-z]+` uygulandığında yalnız "i" ve "ek" parçaları bulunur. Türkçe harfler için ya kümeye açıkça eklenir (`[a-zçğıöşü]`) ya da `\p{L}` kullanılır.

5.2.4.5 Niceleyiciler ve gruplama

Bir öğenin kaç kez tekrarlanacağını belirten işleçlere **niceleyici** (*quantifier*) denir. `*`, `+` ve `?` en kısa niceleyicilerdir. Süslü parantez daha ince ayar sağlar.

İfade	Anlamı
`a*`	sıfır ya da daha fazla 'a'
`a+`	bir ya da daha fazla 'a'
`a?`	sıfır ya da bir 'a'
`a{3}`	tam olarak üç 'a'
`a{2,4}`	iki ile dört arası 'a'
`a{2,}`	en az iki 'a'
`(abc)`	tam olarak “abc”; grup olarak yakalanır
`(a|b|c)`	'a', 'b' ya da 'c'; grup olarak yakalanır
`(ab)+`	“ab”, “abab”, …

Tablo 5.2.11: Niceleyiciler ve gruplama.

Parantezin iki görevi vardır. Birincisi, niceleyicinin bütün bir parçaya uygulanmasını sağlar. İkincisi, eşleşen parçayı **yakalar**: eşleşme bulunduktan sonra parantezin içine düşen metin ayrıca okunabilir. Gruplar soldan sağa, açma parantezlerinin sırasıyla numaralanır. Java'da 1. grup `m.group(1)` ile, eşleşmenin tamamı `m.group()` ya da `m.group(0)` ile okunur.

Niceleyiciler varsayılan olarak **açgözlüdür** (*greedy*): olabildiğince uzun bir parça alırlar. `a{2,4}`, “aaaaa” içinde mümkün olan en uzun parçayı, yani “aaaa”yı alır. Niceleyiciden sonra bir `?` yazılırsa niceleyici **tembel** (*lazy*) olur ve olabildiğince kısa bir parça seçer. Fark en iyi HTML etiketleriyle görülür.

Örnek 5.2.3

Metin `<b>x</b>` olsun. Açgözlü `<.+>` ilk `<` 'den başlar; `.+` satırın sonuna kadar her şeyi alır. Sonra bir `>` bulmak için geri çekilir ve *son* `>` 'de durur. İlk eşleşme metnin tamamıdır: `<b>x</b>`. Tembel `<.+?>` ise `.+?` ile önce tek karakter (b) alır, ardından `>` gelip gelmediğine bakar. Gelmiştir; ilk eşleşme `<b>` olur. `find` aramayı sürdürürse ikinci eşleşme `</b>` 'dir. İki örüntüde de eşleşme en soldaki uygun konumdan (0) başlar. Tembellik eşleşmenin başlangıcını değil, yalnız uzunluğunu etkiler.

5.2.4.6 İleri ve geri bakış

Bazen bir parçanın yalnız belirli bir bağlamda geçtiği durumlar aranır, ama bağlamın kendisi eşleşmeye katılmamalıdır. Örneğin “TL”den önce gelen sayılar bulunmak istenir, ama “TL” sonuca girmemelidir. Bakış ifadeleri bu işi görür.

Örüntü	Adı	Eşleşme koşulu	“ab ac ba”
`a(?=b)`	ileri bakış (*lookahead*)	'a', ardından 'b' geliyorsa	0
`a(?!b)`	olumsuz ileri bakış	'a', ardından 'b' gelmiyorsa	3, 7
`(?<=b)a`	geri bakış (*lookbehind*)	'a', önünde 'b' varsa	7
`(?<!b)a`	olumsuz geri bakış	'a', önünde 'b' yoksa	0, 3

Tablo 5.2.12: Bakış ifadeleri. Son sütun, “ab ac ba” metninde eşleşen a'ların konumlarını (0'dan) verir.

“ab ac ba” metninde üç a vardır: 0, 3 ve 7 numaralı konumlarda. 0'daki a'dan sonra b gelir; ileri bakış bu a'yı seçer. 3'teki a'dan sonra c, 7'deki a'dan sonra metnin sonu gelir; olumsuz ileri bakış bu ikisini seçer. 7'deki a'nın önünde b vardır; geri bakış yalnız onu seçer. Öteki iki a'nın önünde ise boşluk ya da metin başı vardır.

Bakış ifadeleri bir *konumu* sınar ve karakter tüketmez. `a(?=b)` eşleşmesinin metni yalnız “a”dır; b eşleşmeye katılmaz ve sonraki aramada yeniden kullanılabilir. Bu yüzden “abacab” metninde `a(?=b)` iki kez, 0 ve 4 konumlarında, her seferinde “a” olarak eşleşir.

Bakış ifadelerinin en yaygın kullanımı, birden çok koşulu aynı anda sınamaktır. `^(?=.*\d)(?=.*[A-Z]).{8,}$` ifadesi bir parola kuralını denetler. İlk ileri bakış, dizginin bir yerinde bir rakam olduğunu; ikincisi, bir büyük harf olduğunu sınar. İkisi de karakter tüketmediği için konum hâlâ dizginin başındadır. Son parça `.{8,}$`, dizginin en az 8 karakter olduğunu ister. “Parola12” üç koşulu da sağlar. “parola12”de büyük harf, “PAROLAXY”de rakam yoktur; “Pa1” ise kısadır.

5.2.4.7 Doğrulama ifadeleri

Düzenli ifadeler en çok kullanıcıdan gelen verinin biçimini denetlemek için yazılır. Tablo 5.2.13 üç örnek verir. Hepsi `^` ile başlar ve `$` ile biter; dizginin bir parçasının değil, *tamamının* biçime uyması istenir.

Amaç	Düzenli ifade	Uyan	Uymayan
E-posta	`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`	ali.veli@giresun.edu.tr	ali@tr; a b@x.com
Telefon	`^(\+\d{1,3}[- ]?)?\d{10}$`	+90 5321234567; 5321234567	0532 123 45 67
URL	`^https?:\/\/[^\s/$.?#].[^\s]*$`	https://giresun.edu.tr	ftp://x.com; http://x

Tablo 5.2.13: Doğrulama ifadeleri ve sınama dizgileri.

E-posta ifadesi üç parçadan oluşur. `[a-zA-Z0-9._%+-]+` kullanıcı adıdır: harf, rakam ve birkaç işaretten oluşan, en az bir karakterlik bir parça. Ardından @ gelir. `[a-zA-Z0-9.-]+` alan adıdır. `\.[a-zA-Z]{2,}` ise gerçek bir nokta ve en az iki harfli uzantıdır. “ali@tr” uymaz, çünkü @'den sonra nokta ve uzantı yoktur. “a b@x.com” uymaz, çünkü boşluk kullanıcı adı kümesinde yer almaz.

Telefon ifadesinde `(\+\d{1,3}[- ]?)?` isteğe bağlı ülke kodudur. Grup bir artı, bir ile üç arası rakam ve isteğe bağlı bir tire ya da boşluktan oluşur; sondaki `?` grubun tamamını isteğe bağlı yapar. Ardından tam 10 rakam gelmelidir. "0532 123 45 67" uymaz: hem 11 rakamdır hem de araya boşluklar girmiştir.

URL ifadesinde `s?` "http" ve "https"i birlikte kabul eder. Ardından `:\/\/` gelir. Java'da eğik çizgi özel değildir ve kaçış gerekmez; başka dillerden gelen bu yazım yine de zararsızdır. `[^\s/$.?#]`, ilk karakterin boşluk ya da bu işaretlerden biri olamayacağını söyler. Son parça boşluk içermeyen herhangi bir karakter dizisidir. "ftp://x.com" uymaz, çünkü şema http ya da https değildir. "http:// x" uymaz, çünkü // işaretinden sonraki ilk karakter boşluktur.

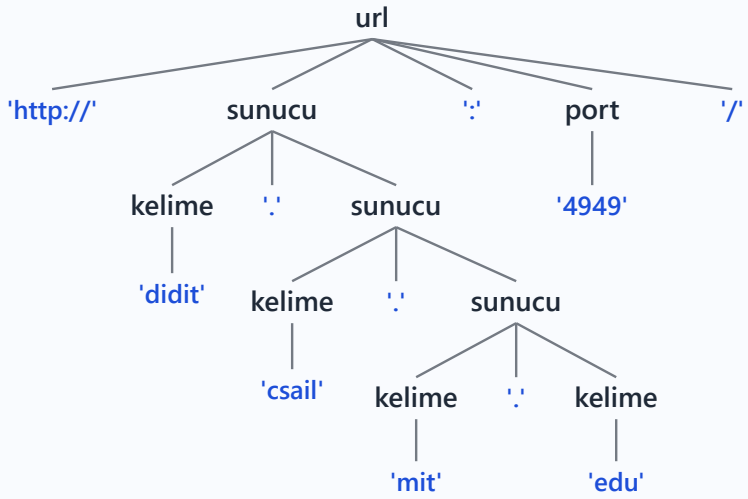
Bu ifadeler pratik yaklaşımlardır; ilgili standartların tamamını kapsamaz. E-posta adreslerinin tam söz dizimi RFC 5322'de, URI söz dizimi RFC 3986'da tanımlanır ve çok daha geniştir. Gerçek uygulamalarda düzenli ifade ilk elemeyi yapar; asıl doğrulama, örneğin adrese bir onay iletisi göndermek, başka yolla yapılır.

5.2.4.8 Dilbilgisi ve ayrıştırma ağacı

Bir dizginin yapısını tanımlamanın bir başka yolu **dilbilgisi** (*grammar*) yazmaktır. Dilbilgisi, büyük bir yapının hangi küçük parçalardan oluştuğunu kurallarla anlatır. Örnek olarak `http://didit.csail.mit.edu:4949/` biçimindeki adresler ele alınsın:

```
url    ::= 'http://' sunucu
           ':' port '/'
sunucu ::= kelime '.' sunucu
         | kelime
kelime ::= [a-z]+
port   ::= [0-9]+
```

İlk kural bir `url`'nin "http://", bir sunucu, iki nokta üst üste, bir port ve bir eğik çizgiden oluştuğunu söyler. İkinci kural özyinelemelidir: bir sunucu ya "kelime . sunucu" biçimindedir ya da tek bir kelimedir. Son iki kural kelimeyi ve portu karakter sınıflarıyla tanımlar. Dilbilgisinin bir dizgiye uygulanması bir **ayrıştırma ağacı** üretir.



Şekil 5.2.9: `http://didit.csail.mit.edu:4949/` adresinin ayrıştırma ağacı. Kök `url` 'dir; `sunucu` kuralı üç kez özyinelemeli olarak açılır ve didit, csail, mit, edu kelimelerini üretir.

Dilbilgisi özyinelemeli görünse de bu özyineleme kuralın en sonundadır; kuyruk özyinelemesidir. Böyle bir özyineleme bir döngüye çevrilebilir. Bu yüzden aynı dil bir düzenli ifadeyle de tanımlanır: `http://([a-z]+\.)*[a-z]+:[0-9]+/`. Bu ifade gerçekten de örnek adresle eşleşir.

Her yapı için bu mümkün değildir. Dengeli parantezler, yani "(()())" gibi her açılanın kapandığı dizgiler, düzenli ifadeyle tanımlanamaz. Bir sonlu durum makinesinin belleği yalnız şu anki durumdur ve sınırsız derinlikteki iç içe açılmış parantezleri sayamaz. İç içe yapılar için **bağlamdan bağımsız dilbilgisi** ve onu işleyen bir **ayrıştırıcı** (*parser*) gerekir. Programlama dillerinin derleyicileri bu ikili yapıyı kullanır: sözcük çözümleyici düzenli ifadelerle, ayrıştırıcı bağlamdan bağımsız dilbilgisiyle çalışır.

Sık yapılan hata

HTML ya da JSON gibi iç içe yapıları yalnız düzenli ifadeyle ayrıştırmaya çalışmak. `<.*>` açgözlü olduğu için "x y" satırının tamamını tek eşleşme olarak yutar. Tembel sürüm tek tek etiketleri bulur, ama iç içe etiketlerin hangisinin hangisini kapattığını bilemez. Bu iş için bir ayrıştırıcı kullanılır.

5.2.4.9 Java'da Pattern ve Matcher

Java'da bir düzenli ifade önce `Pattern.compile` ile derlenir. Derleme, ifadeyi içeride bir otomat benzeri yapıya çevirir; aynı ifade birçok metne uygulanacaksa derlenmiş nesne saklanır ve yeniden kullanılır. Derlenmiş örüntüden `matcher` metoduyla belirli bir metin için bir `Matcher` elde edilir.

```
String girdi = "Ad: Ali Veli, Yaş: 30, E-posta: ali.veli@ornek.com";
Pattern p = Pattern.compile("Yaş: (\\d+),");  // derle
Matcher m = p.matcher(girdi);
if (m.find()) {                           // ilk eşleşmeyi bul
  int yas = Integer.parseInt(m.group(1)); // 1. grup: "30"
  System.out.println("Yaş: " + yas);
}
```

Kod bir satırlık kayıttan yaşı ayıklar. `"Yaş: (\\d+),"` örüntüsü "Yaş: " metnini, ardından bir ya da daha çok rakamı, ardından bir virgülü arar. Rakamlar parantez içinde olduğu için 1. grup olarak yakalanır. `m.find()` metinde ilk eşleşmeyi arar ve bulursa `true` döndürür. `m.group(1)` "30" dizgisini verir; `Integer.parseInt` bunu sayıya çevirir. Kod `Yaş: 30` yazdırır.

Örüntü	`group(1)`	Açıklama
`Ad: (.*?),`	Ali Veli	tembel `.*?` : ilk virgülde durur
`Yaş: (\d+),`	30	bir ya da daha çok rakam
`E-posta: (.*?)$`	ali.veli@ornek.com	satır sonuna kadar

Tablo 5.2.14: Aynı kayıt üzerinde üç örüntü ve yakalanan 1. grup.

İlk satırdaki tembellik önemlidir. Açgözlü `Ad: (.*),` yazılsaydı `.*` son virgüle kadar uzanır ve grup "Ali Veli, Yaş: 30" olurdu. Üçüncü satırda `.*?` tembel olsa da `$` onu satır sonuna kadar gitmeye zorlar.

`Matcher`'ın iki temel metodu farklı sorulara cevap verir. `find()` metnin bir *parçasını* arar; her çağrıda bir sonraki eşleşmeye geçer. `matches()` ise metnin *tamamını* eşleştirmeye çalışır; ifadenin başına `^`, sonuna `$` konmuş gibi davranır. `String.matches` da bu ikinci türdendir. Örneğin `"abc123".matches("\\d+")` `false` döndürür, çünkü dizgi rakamla başlamaz. Aynı örüntüyle `find()` ise "123" parçasını bulur ve `true` döndürür. Benzer biçimde `"2024".matches("\\d{4}")` `true`, `"20245".matches("\\d{4}")` `false`'tur: beşinci rakam fazladır.

Aynı iş düzenli ifade olmadan da yapılabilir. Başlangıç işareti (ör. "Yaş: ") `indexOf` ile bulunur, bitiş işaretine (",") kadar olan kısım `substring` ile kesilir. Basit bir ayıklamada bu yol daha hızlı ve okunaklı olabilir. Örüntü karmaşıklaştıkça, örneğin isteğe bağlı parçalar ve seçenekler arttıkça, düzenli ifade daha kısa ve daha güvenilir kalır.

Sık yapılan hata

İç içe niceleyicileri dikkatsiz kullanmak. Java'nın düzenli ifade motoru eşleşme bulamadığında geri dönüp (geri izleme) başka seçenekleri dener. `(a+)+$` gibi bir ifadede "aaaa…ab" metnini bölmenin üstel sayıda yolu vardır. Geri izlemeli motorlar en kötü durumda bu yolların hepsini

deneyebilir ve program kilitlenmiş gibi görünür. Bu duruma **felaket geri izleme** (*catastrophic backtracking*) denir. İç içe `+` ve `*` kullanmamak ya da kaçış karakterini unutmamak çoğu durumda yeterlidir.

5.2.5 Sonlu durum makineleri

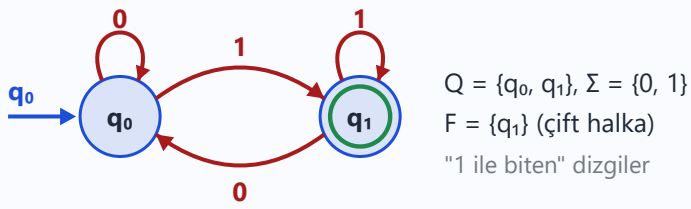
Bir turnike düşünülsün. Kilitliyken jeton atılırsa açılır; açıkken biri geçerse yeniden kilitlenir. Turnikenin geçmişte ne kadar jeton yuttuğu önemli değildir; davranışını yalnız şu anki durumu, kilitli ya da açık olması belirler. Sonlu durum makinesi bu fikrin matematiksel biçimidir ve dizgileri karakter karakter işlemenin en sade modelidir.

5.2.5.1 Biçimsel tanım

Tanım 5.2.4

Bir **sonlu durum makinesi** (*finite state machine*, FSM) beş öğeyle tanımlanır: $M = (Q, \Sigma, \delta, q_0, F)$. Q sonlu bir **durumlar** kümesidir. Σ girdi **alfabesidir**. δ **geçiş fonksiyonudur**: $\delta(q, c)$, makine q durumundayken c sembolünü okuyunca gideceği durumu verir. $q_0 \in Q$ **başlangıç durumudur**. $F \subseteq Q$ **kabul durumlarının** kümesidir. Makine dizgiyi soldan sağa birer sembol okuyarak işler; dizgi bittiğinde F'deki bir durumdaysa dizgiyi **kabul eder**, değilse **reddeder**.

Makinenin tek belleği şu anki durumdur. Durum sayısı sonlu olduğu için makine, okuduğu girdinin yalnız sınırlı bir özetini tutabilir. Bu kısıt ilk bakışta zayıflık gibi görünür. Yine de sözcük çözümleme, ağ protokolleri ve örüntü arama gibi pek çok gerçek problem bu kadar bellekle çözülür. Üstelik makine her karakter için tek bir işlem yaptığı için çok hızlıdır.



Şekil 5.2.10: İki durumlu bir DFA: $Q = \{q_0, q_1\}, \Sigma = \{0, 1\}, F = \{q_1\}$ (çift halka). Makine 1 ile biten ikili dizgileri kabul eder.

Durum diyagramında her durum bir daire, her geçiş etiketli bir oktur. Başlangıç durumuna dışarıdan bir ok girer; kabul durumları çift halkayla çizilir. Şekil 5.2.10'daki makinede $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 1) = q_1$, $\delta(q_1, 0) = q_0$'dır. Durumların anlamı açıktır: q_1 "son okunan sembol 1'di", q_0 "son okunan sembol 0'dı ya da henüz bir şey okunmadı" demektir.

Örnek 5.2.4

Makine "0101" girdisiyle çalıştırılsın. Başlangıç q_0'dır. 0 okunur: q_0'da kalınır. 1 okunur: q_1'e geçilir. 0 okunur: q_0'a dönülür. 1 okunur: q_1'e geçilir. Girdi bitmiştir ve $q_1 \in F$'dir; dizgi kabul edilir. "0110" ise son 0 ile q_0'da biter ve reddedilir.

5.2.5.2 DFA, NFA ve ε-NFA

Sonlu durum makinelerinin üç yaygın türü vardır. Farkları, geçiş fonksiyonunun bir (durum, sembol) çiftine kaç hedef verebildiğidir.

- **Deterministik sonlu otomat** (*deterministic finite automaton*, DFA): her (durum, sembol) çifti için *tam olarak bir* geçiş vardır. Makine her an tek bir durumdadır. Çalıştırması basittir ve n uzunluklu bir girdi $O(n)$ zamanda işlenir.
- **Deterministik olmayan sonlu otomat** (*nondeterministic finite automaton*, NFA): bir (durum, sembol) çifti için sıfır, bir ya da birden fazla geçiş tanımlanabilir. Makine aynı anda bir durum *kümesinde* bulunur. Girdi bittiğinde kümede bir kabul durumu varsa dizgi kabul edilir.
- **ε-NFA**: sembol okumadan yapılan boş geçişlere (epsilon, ε) de izin veren NFA. Bir düzenli ifadeden doğrudan, parça parça kurulabilir. Bu kurulumun adı Thompson yapısıdır.



Şekil 5.2.11: Düzenli ifadeden DFA'ya: Thompson yapısıyla ε-NFA, ε-kapanışla NFA, alt küme kurulumuyla DFA elde edilir.

Teorem 5.2.2 (Eşdeğerlik)

DFA'lar, NFA'lar, ε-NFA'lar ve düzenli ifadeler aynı dil sınıfını, yani düzenli dilleri tanır. Her NFA, **alt küme kurulumu** (*subset construction*) ile aynı dili tanıyan bir DFA'ya çevrilebilir; n durumlu bir NFA'dan elde edilen DFA'nın durum sayısı en kötü durumda 2^n'dir.

İspat. Alt küme kurulumunun fikri kanıtı da verir. NFA her an bir durum kümesindedir. DFA'nın her durumu, NFA'nın bir durum *kümesine* karşılık gelsin. DFA'nın başlangıç durumu $\{q_0\}$'dır. DFA bir S kümesindeyken c okursa, S'deki her durumun c ile gidebildiği durumların birleşimine geçer. Bu geçiş her (küme, sembol) çifti için tektir; elde edilen makine bir DFA'dır. DFA'nın durumu, girdinin her önekinde NFA'nın bulunabileceği durumların tam kümesidir. Bu yüzden DFA, NFA'nın kabul ettiği dizgileri ve yalnız onları kabul eder. n elemanlı bir kümenin 2^n alt kümesi olduğundan durum sayısı bu sınırı aşamaz. □

Üstel büyüme gerçekten de ortaya çıkabilir. “Sondan üçüncü sembolü a olan” *{a, b}* dizgilerini tanıyan NFA dört durumludur. Başlangıç durumu her sembolde kendinde kalır ya da bir a okuyunca “a görüldü” durumuna geçer. Ardından iki sembol daha okunur ve kabul durumuna varılır. Bu dili tanıyan DFA ise son üç sembolün hepsini hatırlamak zorundadır ve $2^3 = 8$ durum ister. “Sondan k. sembol” için NFA $k + 1$, DFA 2^k durumludur. Uygulamada bu patlama nadiren görülür. `grep`, `lex` / `flex` ve derleyicilerin sözcük çözümleyicileri Şekil 5.2.11'deki zinciri kullanır: düzenli ifadeyi NFA'ya, gerekirse DFA'ya çevirir ve metni bu otomatla tarar.

5.2.5.3 DFA simülasyonu

Bir DFA'yı çalıştırmak üç adımdan oluşur. Başlangıç durumuna geçilir. Her karakter için durum geçiş tablosundan güncellenir. Dizgi bitince son durumun kabul durumu olup olmadığı denetlenir. Geçiş fonksiyonu bir tabloyla saklanır: satırlar durumlar, sütunlar sembollerdir. Durumlar 0'dan numaralanır ve 0 başlangıç durumudur.

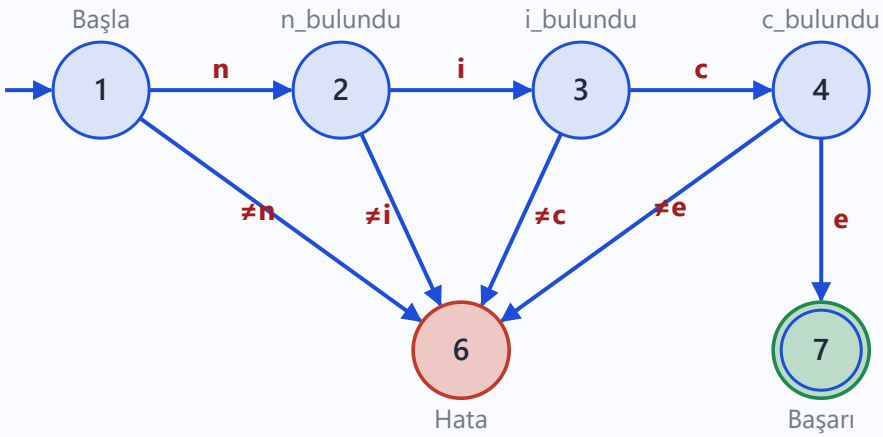
```
boolean kabulEder(int[][] gecis, boolean[] kabul,
                  String s) {
  int durum = 0;                    // başlangıç durumu
  for (char c : s.toCharArray())
    durum = gecis[durum][c - '0'];  // tek geçiş
  return kabul[durum];              // son durum kabul mü?
}
```

`gecis` tablosu `int[|Q|][|Σ|]` boyutundadır. Bu örnekte alfabe *{0, 1}*'dir; `c - '0'` ifadesi '0' karakterini 0'a, '1' karakterini 1'e çevirir ve sütun indisi olarak kullanılır. `kabul` dizisi her durumun kabul durumu olup olmadığını tutar. Döngü her karakter için tek bir tablo okuması yapar. n uzunluklu dizgi $O(n)$ zamanda, geri dönmeden işlenir; ek bellek yalnız `durum` değişkenidir. Tablonun kendisi $O(|Q| \cdot |\Sigma|)$ yer kaplar.

Geri dönmemek önemli bir özelliktir. Düzenli ifade motorlarının bir kısmı, `(a+)+$` örneğinde olduğu gibi geri izleme yapar ve kötü durumda üstel süre harcayabilir. DFA ise her karakteri tam bir kez okur; en kötü durumda da doğrusal zamanlıdır.

5.2.5.4 Örnek: “nice” tanıyıcısı

Belirli tek bir kelimeyi tanıyan makine, sonlu durum makinelerinin en yalın örneğidir. Şekil 5.2.12'deki makine yalnız “nice” dizgisini kabul eder.



Şekil 5.2.12: “nice” kelimesini tanıyan makine. Başla (1), n_bulundu (2), i_bulundu (3), c_bulundu (4) durumlarından beklenen karakterle ilerlenir; e ile Başarı (7) durumuna varılır. Beklenmeyen her karakter Hata (6) durumuna götürür.

Her durum, o ana kadar okunan öneki temsil eder: 1 boş öneki (ε), 2 “n”yi, 3 “ni”yi, 4 “nic”i, 7 ise “nice”ı. Beklenen karakter gelirse bir sonraki duruma geçilir. Beklenen karakter gelmezse, örneğin 2 durumunda i'den başka bir karakter okunursa, Hata durumuna gidilir. Hata durumundan çıkış yoktur; her karakter makineyi orada tutar. Böyle bir duruma **tuzak durum** (*trap state*) denir. Diyagramda tuzak durumun kendine dönen okları çizilmez, ama makinenin DFA olması için var oldukları kabul edilir. Aynı biçimde Başarı durumundan sonra bir karakter daha gelirse makine Hata'ya geçer; “nicer” reddedilir.

“nice” girdisi 1 → 2 → 3 → 4 → 7 yolunu izler ve kabul edilir. “nine” ise 1 → 2 → 3 yolundan sonra üçüncü karakterde 'n' okur. Beklenen 'c'dir; makine 6'ya gider ve dördüncü karakter onu orada tutar. Sonuç rettir.

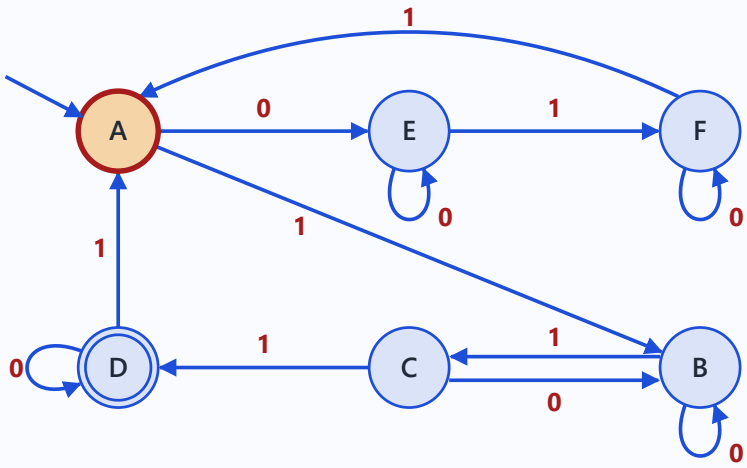
5.2.5.5 Örnek: ikili alfabe üzerinde DFA

Daha büyük bir DFA altı durumludur: A, B, C, D, E ve F. Başlangıç A, tek kabul durumu D'dir. Geçişler Tablo 5.2.15'te ve Şekil 5.2.13'te verilir. Tablonun her hücresinde tek bir durum vardır; bu yüzden makine bir DFA'dır.

δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F

δ	0	1
F	F	A

Tablo 5.2.15: Altı durumlu DFA'nın geçiş tablosu. → başlangıç, * kabul durumunu gösterir.

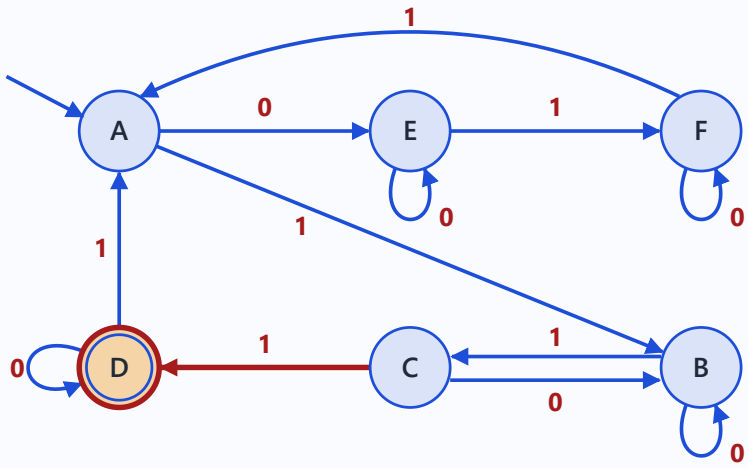


Şekil 5.2.13: İkili alfabe üzerinde altı durumlu DFA. Başlangıç A (vurgulu), kabul D (çift halka).

Örnek 5.2.5

Girdi “10011” olsun.

1. Başlangıç: A.
2. 1. sembol 1: $\delta(A, 1) = B$.
3. 2. sembol 0: $\delta(B, 0) = B$.
4. 3. sembol 0: $\delta(B, 0) = B$.
5. 4. sembol 1: $\delta(B, 1) = C$.
6. 5. sembol 1: $\delta(C, 1) = D$. Girdi bitti; D kabul durumudur ve dizgi kabul edilir.



Şekil 5.2.14: "10011" okunduktan sonra makine D'dedir; son geçiş *δ(C, 1) = D* kırmızıyla gösterilmiştir.

Kabul kararı yalnız *son* duruma bağlıdır. "1011010" girdisi A → B → B → C → D yolundan sonra D'ye varır; ama girdi bitmemiştir. Sonraki 0 makineyi D'de tutar, 1 A'ya götürür, son 0 ise E'ye götürür. E kabul durumu olmadığından dizgi reddedilir. Yolda kabul durumundan geçmiş olmak bir şey değiştirmez. Benzer biçimde "111" A → B → C → D yoluyla kabul edilir, "11" ise C'de biter ve reddedilir.

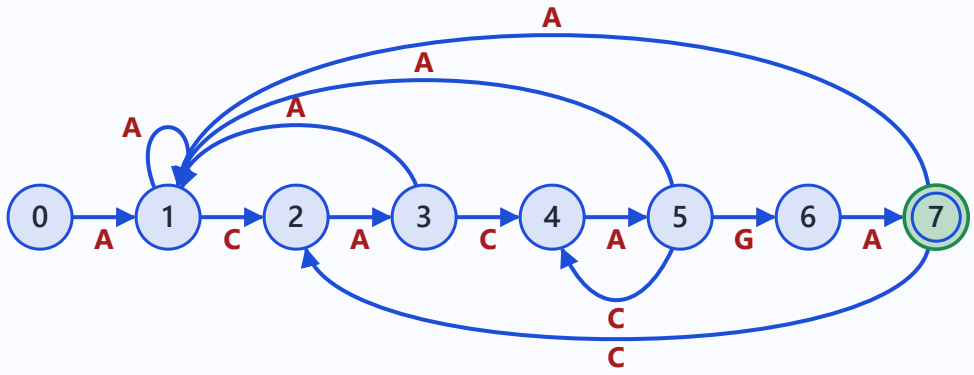
Tablonun `kabulEder` metoduna verilmesi kolaydır: A–F durumları 0–5 olarak numaralanır, tablo `{{4, 1}, {1, 2}, {1, 3}, {3, 0}, {4, 5}, {5, 0}}` olur ve `kabul` dizisinde yalnız 3. hücre `true`'dur. Metot "10011" için `true`, "1011010" için `false` döndürür.

5.2.5.6 Örüntü eşleme otomatı

Sonlu durum makineleri bir metin içinde bir örüntüyü aramak için de kullanılır. Örüntü "ACACAGA" olsun. Metin soldan sağa okunurken makinenin durumu, şimdiye kadar okunan metnin sonunun örüntünün hangi önekiyle uyuştuğunu gösterir.

Tanım 5.2.5

m uzunluklu bir *P* örüntüsü için **örüntü eşleme otomatının** durumları *0, 1, ..., m*'dir. Metnin ilk *i* karakteri okunduktan sonra durum *q*, okunan metnin bir soneki olan *P*'nin en uzun önekinin uzunluğudur. *q = m* durumuna varılması, örüntünün metinde o noktada bittiği anlamına gelir.



Şekil 5.2.15: "ACACAGA" örüntüsü için eşleme otomatı. 0'dan 7'ye ileri geçişler örüntünün karakterleridir. Geri geçişler, en uzun uyan öneke dönüşü gösterir. Çizilmeyen bütün geçişler 0'a gider.

İleri geçişler kolaydır: q durumunda örüntünün q. karakteri (0'dan sayarak) okunursa $q + 1$'e geçilir. Asıl ilginç olan geri geçişlerdir. Örneğin 5 durumunda (son okunan beş karakter "ACACA") C okunsun. Beklenen G'dir, uyuşma bozulmuştur. Ama okunan metnin sonu artık "ACACAC"dır ve bunun soneki "ACAC", örüntünün 4 uzunluklu önekidir. Makine 0'a değil 4'e döner; uyuşmanın bir kısmı korunur. Otomattaki bütün sıfırdan farklı geri geçişler şunlardır: 1, 3, 5 ve 7 durumlarından A ile 1'e; 5'ten C ile 4'e; 7'den C ile 2'ye.

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
metin	G	A	C	A	C	A	C	A	G	A
durum	0	1	2	3	4	5	4	5	6	7

Tablo 5.2.16: "GACACACAGA" metni okunurken otomatın durumu. Durum 7'ye 9 numaralı karakterde ulaşılır.

Bu fikir, Knuth–Morris–Pratt (KMP) algoritmasındaki önek fonksiyonunun otomat biçimidir. KMP uyuşmazlıkta örüntüyü ne kadar kaydıracağını önek fonksiyonundan okur. Otomat ise bu bilgiyi her (durum, karakter) çifti için önceden hesaplayıp tabloya yazar. Tablonun kurulumu $O(m \cdot |\Sigma|)$, arama $O(n)$ sürer. Sunum otomatın nasıl kurulduğunu göstermez; aşağıdaki metot bu kurulumu yapar:

```
int[][] otomatKur(String p) {        // örüntü otomatı, O(m · 256)
  int m = p.length();
  int[][] dfa = new int[m + 1][256];
  dfa[0][p.charAt(0)] = 1;
  for (int x = 0, j = 1; j <= m; j++) {
    for (int c = 0; c < 256; c++)
      dfa[j][c] = dfa[x][c];         // uyuşmazlık: x durumundaki gibi
    if (j < m) {
      dfa[j][p.charAt(j)] = j + 1;   // uyuşma: ilerle
      x = dfa[x][p.charAt(j)];       // yeniden başlama durumu
    }
  }
  return dfa;
}
```

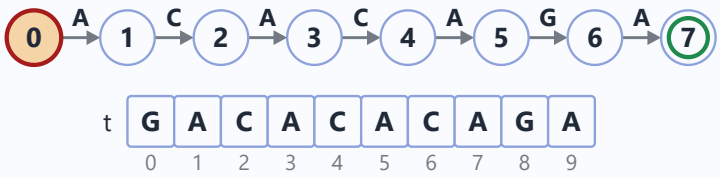
Metodun anahtarı `x` değişkenidir: `x`, örüntünün `p[1..j-1]` parçası otomata verildiğinde varılacak durumdur. `j` durumunda bir uyuşmazlık olduğunda makine, "son okunanlar örüntünün `j` karakterlik öneki, ardından yeni karakter" bilgisine sahiptir. En uzun uyan önek, bu metnin ilk karakteri atıldığında elde edilen durumdan, yani `x`'ten yeni karakterle gidilen durumdur. Bu yüzden `j` satırı önce `x` satırından kopyalanır. Sonra uyuşma geçişi `j + 1` olarak üzerine yazılır ve `x` bir adım ilerletilir. Metot "ACACAGA" için tam olarak Şekil 5.2.15'teki geçişleri üretir.

5.2.5.7 Kod izleme: dfaAra

Otomat kurulduktan sonra arama birkaç satırdır:

```
int dfaAra(int[][] dfa, String t, int m) {
  int q = 0;                     // durum
  for (int i = 0; i < t.length(); i++) {
    q = dfa[q][t.charAt(i)];
    if (q == m)
      return i - m + 1;          // eşleşme
  }
  return -1;
}
```

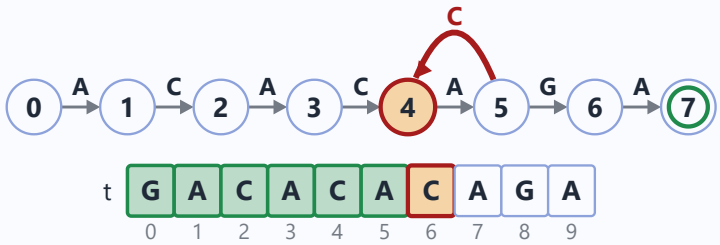
`dfa` tablosunun satırları durumlar, sütunları karakter kodlarıdır; `dfa[q][t.charAt(i)]` doğrudan bir sonraki durumu verir. `q == m` olduğunda örüntünün son karakteri metnin `i` konumundadır. Örüntü `i - m + 1` konumunda başlar ve bu değer döndürülür. Döngü eşleşme bulamadan biterse `-1` döner. Kod `dfaAra(dfa, "GACACACAGA", 7)` çağrısıyla izlensin.



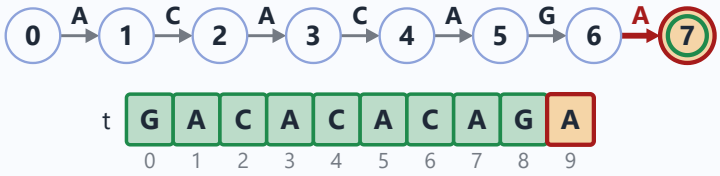
Şekil 5.2.16: Başlangıç: otomat "ACACAGA" için önceden kurulmuştur, `q = 0`.

`i`	`t[i]`	`q`	**Açıklama**
0	G	0 → 0	G örüntünün hiçbir önekini sürdürmez
1	A	0 → 1	örüntünün sıradaki karakteri
2	C	1 → 2	örüntünün sıradaki karakteri
3	A	2 → 3	örüntünün sıradaki karakteri
4	C	3 → 4	örüntünün sıradaki karakteri
5	A	4 → 5	örüntünün sıradaki karakteri
6	C	5 → 4	beklenen G değil; "ACACAC"ın sonu "ACAC" önekidir
7	A	4 → 5	örüntünün sıradaki karakteri
8	G	5 → 6	örüntünün sıradaki karakteri
9	A	6 → 7	`q == m`: `return 9 - 7 + 1 = 3`

Tablo 5.2.17: `dfaAra(dfa, "GACACACAGA", 7)` izlemesi.



Şekil 5.2.17: `t[6] = 'C'` : beklenen G değildir, ama `q` sıfırlanmaz; 5 → 4 geçişi yapılır. Metinde geri dönülmez.



Şekil 5.2.18: `t[9] = 'A'` ile 6 → 7 = `m` : örüntü bulundu ve *9 − 7 + 1 = 3* döndürülür.

Örüntü metinde 3 numaralı konumdan başlar: `t[3..9]` = “ACACAGA”. Dikkat çekici adım `i = 6`'dır. Saf bir arama yöntemi burada örüntüyü bir kaydırıp metinde geri dönerek yeniden karşılaştırırdı. Otomat ise 4 durumuna geçer ve metnin hiçbir karakterini ikinci kez okumaz. Toplam on karakter, on tablo okumasıyla işlenmiştir.

Sık yapılan hata

Uyuşmazlıkta durumu her zaman 0'a döndürmek. “abab” örüntüsünü arayan otomat “abaa” metnini okusun. İlk üç karakter 3 durumuna getirir. Dördüncü karakter a, beklenen b değildir. Ama “abaa”nın “abab”a önek olan en uzun soneki “a”dır; doğru durum 1'dir. 0'a dönen bir makine, “abaabab” gibi bir metinde 3'te başlayan eşleşmeyi kaçırır.

5.2.5.8 FSM ile sayı ayrıştırma

Sonlu durum makineleri metinden parça ayıklamak için de kullanılır. Aşağıdaki metot bir metindeki ondalık sayıları bulup yazdırır. Makinenin üç durumu vardır: BASLA (bir sayının içinde değiliz), RAKAM (tam sayı kısmı okunuyor) ve ONDALIK (nokta görüldü, kesir kısmı okunuyor). Okunan sayı karakterleri bir `StringBuilder` tamponunda biriktirilir.

```
void sayilariBul(String s) {
  Durum d = Durum.BASLA;
  StringBuilder tampon = new StringBuilder();
  for (char c : s.toCharArray()) {
    boolean rakam = Character.isDigit(c);
    if (rakam) {
      if (d == Durum.BASLA) d = Durum.RAKAM;
      tampon.append(c);
    } else if (c == '.' && d != Durum.ONDALIK) {
      d = Durum.ONDALIK;
      tampon.append(c);
    } else if (d != Durum.BASLA) {  // sayı bitti
      System.out.println(tampon);
      tampon.setLength(0);
      d = Durum.BASLA;
    }
  }
  if (d != Durum.BASLA) System.out.println(tampon);
}
```

Döngü her karakter için üç koşuldan birini uygular. Karakter bir rakamsa ve makine BASLA'daysa RAKAM'a geçilir; her durumda rakam tampona eklenir. Karakter bir nokta ise ve makine henüz ONDALIK'ta değilse ONDALIK'a geçilir ve nokta da tampona eklenir. Bu iki koşul tutmazsa ve makine bir sayının içindeyse (BASLA dışında), sayı bitmiştir: tampon yazdırılır, boşaltılır ve BASLA'ya dönülür. Döngüden sonra makine hâlâ bir sayının içindeyse, yani girdi bir sayıyla bitiyorsa, son sayı da yazdırılır.

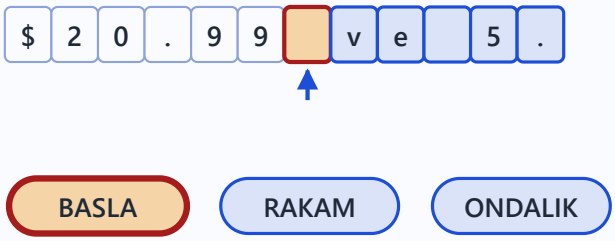


Şekil 5.2.19: Girdi “$20.99 ve 5.”; durum BASLA, tampon boş.

Karakter	Uygulanan dal	Durum	Tampon	Çıktı
$	hiçbiri (BASLA)	BASLA	“”	
2	rakam	RAKAM	“2”	
0	rakam	RAKAM	“20”	
.	nokta	ONDALIK	“20.”	
9	rakam	ONDALIK	“20.9”	
9	rakam	ONDALIK	“20.99”	

Karakter	Uygulanan dal	Durum	Tampon	Çıktı
␣	sayı bitti	BASLA	“”	20.99
v, e, ␣	hiçbiri (BASLA)	BASLA	“”	
5	rakam	RAKAM	“5”	
.	nokta	ONDALIK	“5.”	
(son)	döngü sonrası `if`	—	“5.”	5.

Tablo 5.2.18: `sayilariBul("$20.99 ve 5.")` izlemesi (␣ boşluk karakterini gösterir).



Şekil 5.2.20: Boşluk karakteri sayı karakteri değildir: sayı bitti, 20.99 yazdırıldı ve BASLA'ya dönüldü.



Şekil 5.2.21: Girdi bitti; durum ONDALIK olduğu için tampon yazdırılır. Sondaki nokta da sayıya katılmıştır.

Metot iki satır yazdırır: “20.99” ve “5.”. İkincisi bir kusurdur: cümle sonundaki nokta sayının parçası sanılmıştır. Kusurun nedeni, makinenin ONDALIK durumuna noktadan sonra rakam gelip gelmeyeceğini bilmeden geçmesidir. Aynı tasarım başka girdilerde de beklenmedik sonuçlar verir. “v1.2.3 ve 7” girdisinde ikinci nokta geldiğinde makine zaten ONDALIK'tadır. Nokta koşulu tutmaz, üçüncü dal çalışır: “1.2” yazdırılır, nokta atılır ve BASLA'ya dönülür. Ardından “3” ve “7” ayrı sayılar olarak yazdırılır. Sürüm numarası iki sayıya bölünmüştür.

Kusur iki yoldan giderilebilir. Birincisi, yazdırmadan önce tampon nokta ile bitiyorsa son karakteri silmektir. İkincisi, noktayı ancak ardından rakam gelirse sayıya katmaktır; bunun için makineye “nokta görüldü ama henüz kesir rakamı yok” anlamında yeni bir durum eklenir. Aşağıdaki sürüm birinci yolu izler ve noktayı yalnız bir rakamdan sonra kabul eder:

```
void sayilariBulDuzgun(String s) {
  Durum d = Durum.BASLA;
  StringBuilder tampon = new StringBuilder();
  for (char c : (s + " ").toCharArray()) {        // sona ayırıcı eklenir
    if (Character.isDigit(c)) {
      if (d == Durum.BASLA) d = Durum.RAKAM;
      tampon.append(c);
    } else if (c == '.' && d == Durum.RAKAM) {     // nokta yalnız rakamdan sonra
      d = Durum.ONDALIK;
      tampon.append(c);
    } else if (d != Durum.BASLA) {
      if (tampon.charAt(tampon.length() - 1) == '.')
        tampon.setLength(tampon.length() - 1);     // sondaki nokta sayıya ait değil
      System.out.println(tampon);
      tampon.setLength(0);
      d = Durum.BASLA;
    }
  }
}
```

Girdinin sonuna bir boşluk eklenmesi, döngüden sonraki ayrı `if`'i gereksiz kılar: son sayı da boşlukla biter ve aynı dalda yazdırılır. Düzeltilmiş metot “$20.99 ve 5.” için “20.99” ve “5” yazdırır. “v1.2.3 ve 7” için davranış değişmez: “1.2”, “3” ve “7”. Noktanın yalnız rakamdan sonra kabul edilmesinin bir yan etkisi vardır: “.5” gibi baştaki sıfırı yazılmamış bir sayı “5” olarak okunur. Hangi davranışın doğru olduğu uygulamanın kurallarına bağlıdır. FSM tasarımının asıl yararı da budur: her kural bir duruma ya da geçişe karşılık gelir ve açıkça görülür.

5.2.6 Düzenleme uzaklığı

Bir arama kutusuna “algortima” yazılır ve arama motoru “algoritma”yı önerir. Bunun için iki kelimenin birbirine ne kadar yakın olduğunu ölçen bir sayıya gerek vardır. Eşitlik denetimi yalnız “aynı” ya da “farklı” der; “bir harf eksik” ile “tamamen başka bir kelime” arasında ayrım yapmaz. Düzenleme uzaklığı bu ayrımı yapar.

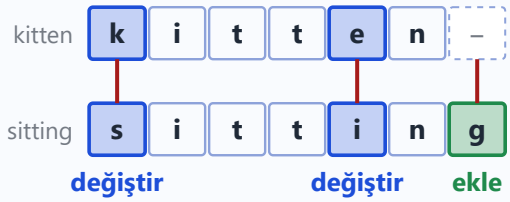
5.2.6.1 Levenshtein uzaklığı

Tanım 5.2.6

İki dizgi arasındaki **Levenshtein uzaklığı** (*Levenshtein distance*), birinci dizgiyi ikinciye dönüştürmek için gereken en az tek karakterlik düzenleme sayısıdır. İzin verilen düzenlemeler üç tanedir ve her biri 1 sayılır: **ekleme** (*insertion*), **silme** (*deletion*) ve **değiştirme** (*substitution*).

Ölçü, adını 1965'te bu kavramı tanımlayan Sovyet matematikçi Vladimir Levenshtein'dan alır; makalenin İngilizce çevirisi 1966'da yayımlanmıştır. Genel adı **düzenleme uzaklığıdır** (*edit distance*). Bu ad, düzenlemelere farklı maliyetler verilen türleri de kapsar. Kullanım alanları geniştir: yazım denetimi ve otomatik düzeltme, klavyede tahmin, DNA dizilerinin karşılaştırılması ve küçük yazım farklarını hoş gören bulanık (*fuzzy*) arama.

Üç düzenlemenin her birine bir örnek verilebilir. “sittin” → “sitting”: sona g *eklenir*. “shine” → “shin”: son e *silinir*. “kitten” → “sitten”: k, s ile *değiştirilir*. Bu üç işlem art arda kullanılarak “kitten”, “sitting”e üç adımda dönüştürülür: kitten → sitten (k → s) → sittin (e → i) → sitting (+g).

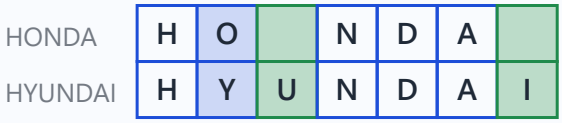


Şekil 5.2.22: “kitten” ile “sitting”in hizalaması: iki değiştirme (k → s, e → i) ve bir ekleme (g). Uzaklık 3'tür.

Üçten kısa bir dönüşüm yoktur. Bunu görmek için her olasılığı denemek gerekmez; dinamik programlama tablosu bunu kanıtlar. Yine de kısa bir gerekçe verilebilir. “sitting”, “kitten”den bir harf uzundur; en az bir ekleme gerekir. İlk harfler (k ve s) farklıdır ve öteki harflerle hizalanarak giderilemez; bir değiştirme daha gerekir. Ortadaki e/i farkı da üçüncü bir işlem ister.

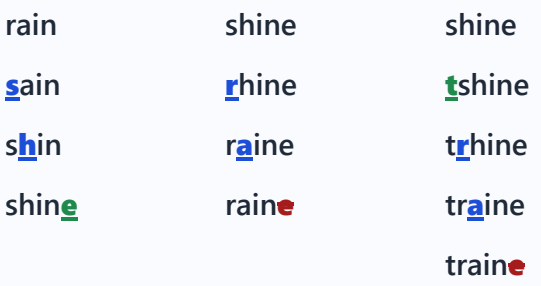
Levenshtein uzaklığı matematiksel anlamda bir **metriktir**. Üç özelliği vardır. Bir dizginin kendisine uzaklığı sıfırdır: $d(a, a) = 0$, farklı dizgiler için uzaklık pozitiftir. Uzaklık simetriktir: $d(a, b) = d(b, a)$, çünkü her ekleme tersine çevrilince silme, her değiştirme tersine çevrilince yine değiştirme olur. Üçgen eşitsizliği geçerlidir: $d(a, c) \le d(a, b) + d(b, c)$; a'dan b'ye, oradan c'ye giden düzenlemeler art arda uygulanarak a'dan c'ye varılır.

5.2.6.2 Düzenleme örnekleri



Şekil 5.2.23: HONDA → HYUNDAI: O → Y değiştirme, U ekleme, I ekleme; $d = 3$.

Bir dönüşümü görmenin en kolay yolu iki dizgiyi alt alta hizalamaktır. Eşit harfler aynı sütuna yazılır. Bir dizgide olup ötekinde olmayan harfin karşısına boşluk konur. Şekil 5.2.23'te HONDA'nın H, N, D, A harfleri HYUNDAI'deki karşılıklarıyla hizalanmıştır. O, Y ile değiştirilir; U ve I'nın karşısında boşluk vardır, bu iki harf eklenir. Toplam üç düzenleme yapılır.



Şekil 5.2.24: (a) rain → sain → shin → shine; (b) shine → rhine → raine → rain (e silinir); (c) shine → tshine → trhine → traine → train (e silinir). Altı çizili mavi harf değiştirilmiş, altı çizili yeşil harf eklenmiş, üstü çizili turuncu harf silinecek harftir.

(a) ve (b) birbirinin tersidir. “rain”den “shine”a iki değiştirme ve bir ekleme ile gidilir; “shine”dan “rain”e dönerken aynı iki değiştirme tersine yapılır ve ekleme silmeye dönüşür. İki yön de 3 düzenleme sürer; bu, simetrinin somut bir örneğidir. (c)'de “shine” ile “train” aynı uzunluktadır, ama uzaklık 4'tür: t eklenir, s → r ve h → a değiştirilir, sondaki e silinir.

5.2.6.3 Dinamik programlama bağıntısı

Bütün düzenleme dizilerini denemek olanaksızdır; sayıları dizgi uzunluğuyla üstel artar. Çözüm, problemi önekler üzerinde küçük alt problemlere bölmektir. a dizgisi m, b dizgisi n karakterli olsun.

Tanım 5.2.7

D[i][j], *a*'nın ilk *i* karakterini *b*'nin ilk *j* karakterine dönüştürmenin en az maliyetidir ($0 \le i \le m$, $0 \le j \le n$). Aranan uzaklık *D[m][n]*'dir.

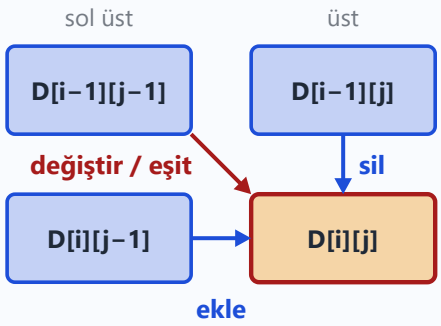
Teorem 5.2.3 (Levenshtein bağıntısı)

> D[i][0] = i ve D[0][j] = j
> $a_i = b_j$ ise D[i][j] = D[i−1][j−1]
> aksi hâlde D[i][j] = 1 + min(D[i−1][j], D[i][j−1], D[i−1][j−1])

Burada a_i, *a*'nın *i*. karakteridir (1'den sayarak). Minimumun üç terimi sırasıyla silmeye, eklemeye ve değiştirmeye karşılık gelir.

İspat. Kenar değerleri açıktır: *i* karakterlik bir önekten boş dizgiye ancak *i* silmeyle, boş dizgiden *j* karakterlik bir öneke ancak *j* eklemeyle gidilir. Genel durumda en iyi dönüşümün son karakterlerle ne yaptığına bakılır. Üç olasılık vardır. (1) a_i silinir; geri kalan iş, *a*'nın ilk $i-1$ karakterini *b*'nin ilk *j* karakterine çevirmektir: maliyet $1 + D[i-1][j]$. (2) Sona b_j eklenir; geri kalan iş ilk *i* karakteri ilk $j-1$ ka-

raktere çevirmektir: *1 + D[i][j−1]*. (3) a_i, b_j'nin karşısına düşer. Karakterler eşitse bu adım bedavadır ve maliyet *D[i−1][j−1]*'dir. Farklılarsa bir değiştirme gerekir: *1 + D[i−1][j−1]*. En iyi dönüşüm bu üç seçenekten birini kullanır; bu yüzden değer, seçeneklerin en küçüğüdür. Karakterler eşitken bedava çaprazın her zaman en iyisi olduğu da gösterilebilir: *D*'nin komşu hücreleri arasındaki fark en çok 1 olduğundan, öteki iki seçenek *D[i−1][j−1]*'den küçük bir değer veremez. □



Şekil 5.2.25: *D[i][j]* üç komşudan hesaplanır: sol üst (değiştirme ya da eşitlik), üst (silme) ve sol (ekleme).

Bu, **dinamik programlama** (*dynamic programming*, DP) yönteminin tipik bir örneğidir. Her alt problem bir kez çözülür ve *(m + 1) × (n + 1)* boyutlu bir tabloda saklanır; cevap tablonun sağ alt köşesindedir. Bir hücre yalnız üst, sol ve sol üst komşularına bağlı olduğu için tablo satır satır, her satırda soldan sağa doldurulabilir. Hücre başına sabit iş yapılır. Zaman *O(m · n)*, bellek *O(m · n)*'dir. Yalnız uzaklık gerekiyorsa bellek iki satıra, yani *O(min(m, n))*'ye indirilebilir.

5.2.6.4 DP tablosu: CARS → PAIRS

a = CARS satırlara, *b* = PAIRS sütunlara yazılsın. Tablonun 0. satırı ve 0. sütunu boş öneke (*ε*) karşılık gelir.

Örnek 5.2.6

1. **Başlangıç.** İlk satır 0, 1, 2, 3, 4, 5'tir: boş dizgiden P, PA, … önekine ekleme sayısı. İlk sütun 0, 1, 2, 3, 4'tür: C, CA, … önekinden boş dizgiye silme sayısı.
2. **1. satır (C).** C, PAIRS'in hiçbir harfine eşit değildir. *D[1][1] = 1 + min(0, 1, 1) = 1*: C, P ile değiştirilir. Satırın kalanı her adımda bir artar: 1, 2, 3, 4, 5.
3. **2. satır (A).** *D[2][1] = 1 + min(1, 2, 1) = 2*. A = A olduğu için *D[2][2] = D[1][1] = 1*; maliyet eklenmez. A ≠ I: *D[2][3] = 1 + min(üst 3, sol 1, sol üst 2) = 2*. Satır 2, 1, 2, 3, 4 olur.
4. **3. satır (R).** R = R olduğu için *D[3][4] = D[2][3] = 2*. Satır 3, 2, 2, 2, 3 olur.
5. **4. satır (S).** S = S olduğu için *D[4][5] = D[3][4] = 2*. Satır 4, 3, 3, 3, 2 olur.

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1	**1**	2	3	4	5
A	2	2	**1**	**2**	3	4
R	3	3	2	2	**2**	3
S	4	4	3	3	3	**2**

Tablo 5.2.19: CARS → PAIRS için doldurulmuş DP tablosu. Sonuç sağ alt köşe-dedir: *D[4][5] = 2*.

Tablo yalnız uzaklığı değil, düzenleme dizisini de verir. Sağ alt köşeden başlanır ve her hücrede değerin hangi komşudan geldiğine bakılarak sol üst köşeye geri yürünür. Bu işleme **geri okuma** (*traceback*) denir. *D[4][5] = 2*, S = S olduğu için çaprazdan gelir. *D[3][4] = 2*, R = R olduğu için yine çaprazdan gelir. *D[2][3] = 2*, soldaki *D[2][2] = 1*'den bir ekleme ile gelir: I eklenir. *D[2][2] = 1*, A = A ile çaprazdan; *D[1][1] = 1*, C → P değiştirmesiyle çaprazdan gelir. Kalın hücreler bu yolu gösterir. Sonuç: C → P değiştirme, A eşit, I ekleme, R ve S eşit. Dönüşüm CARS → PARS → PAIRS'tir.

> **Sık yapılan hata**
>
> Tablo indisiyle dizgi indisini karıştırmak. Tablo *(m + 1) × (n + 1)* boyutludur, çünkü 0. satır ve sütun boş önek içindir. *D[i][j]* hesaplanırken karşılaştırılan karakterler `a.charAt(i - 1)` ve `b.charAt(j - 1)` 'dir. İlk satırı ve sütunu başlatmayı unutmak da sonucu bozar: Java diziyi sıfırla başlattığı için kenarlar 0 kalır ve uzaklık küçük çıkar.

5.2.6.5 Java kodu ve kod izleme

Bağıntı doğrudan koda çevrilir:

```
int levenshtein(String a, String b) {
  int m = a.length(), n = b.length();
  int[][] D = new int[m + 1][n + 1];
  for (int i = 0; i <= m; i++) D[i][0] = i;  // silme
  for (int j = 0; j <= n; j++) D[0][j] = j;  // ekleme
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        D[i][j] = D[i - 1][j - 1];            // eşit
      else
        D[i][j] = 1 + Math.min(D[i - 1][j - 1], // değiştir
                  Math.min(D[i][j - 1],         // ekle
                           D[i - 1][j]));       // sil
  return D[m][n];
}
```

İlk iki döngü tablonun kenarlarını doldurur. İç içe iki döngü kalan hücreleri satır satır doldurur. Karakterler eşitse sol üst değer kopyalanır; değilse üç komşunun en küçüğüne 1 eklenir. Kod sunumdaki örneklerde şu değerleri verir: kitten → sitting = 3, ercan → sercan = 1 (baştaki s eklenir), apple → banana = 5. Kod “kar” → “ark” çağrısıyla izlensin. *a* = “kar” satırlara, *b* = “ark” sütunlara yazılır.

	ε	a	r	k
ε	0	1	2	3
k	1			
a	2			
r	3			

Şekil 5.2.26: `levenshtein("kar", "ark")` : kenarlar dolduruldu. 0. sütun *i* silmeyi, 0. satır *j* eklemeyi gösterir.

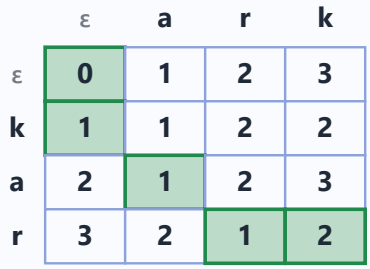
Hücre	Karakterler	Hesap	Değer
D[1][1]	k ≠ a	1 + min(değiştir 0, ekle 1, sil 1)	1
D[1][2]	k ≠ r	1 + min(değiştir 1, ekle 1, sil 2)	2
D[1][3]	k = k	sol üst	2
D[2][1]	a = a	sol üst	1
D[2][2]	a ≠ r	1 + min(değiştir 1, ekle 1, sil 2)	2
D[2][3]	a ≠ k	1 + min(değiştir 2, ekle 2, sil 2)	3
D[3][1]	r ≠ a	1 + min(değiştir 2, ekle 3, sil 1)	2
D[3][2]	r = r	sol üst	1

Hücre	Karakterler	Hesap	Değer
D[3][3]	r ≠ k	1 + min(değiştir 2, ekle 1, sil 3)	2

Tablo 5.2.20: `levenshtein("kar", "ark")` izlemesi. "değiştir", "ekle", "sil" sırasıyla sol üst, sol ve üst komşunun değeridir.



Şekil 5.2.27: `a[0] = b[2] = 'k'` : maliyetsiz; *D[1][3]*, sol üstteki 2'yi alır.



Şekil 5.2.28: Tamamlanmış tablo, *D[3][3] = 2*. Yeşil yol iki düzenlemeyi gösterir: baştaki k silinir (ar), sona k eklenir (ark).

Sonuç 2'dir. Geri okuma *D[3][3]*'ten başlar. Değer, soldaki *D[3][2] = 1*'den bir ekleme ile gelmiştir: sona k eklenir. *D[3][2]*, r = r ile *D[2][1]*'den, o da a = a ile *D[1][0] = 1*'den gelir. *D[1][0]* ise k'nin silinmesidir. Yalnız değiştirmeyle üç düzenleme gerekirdi (k → a, a → r, r → k); "kar"ın son iki harfi "ark"ın ilk iki harfiyle aynı olduğundan silme–ekleme yolu daha kısadır.

Bu örnek, Levenshtein uzaklığının harf *takasını* tek işlem saymadığını da hatırlatır. "saat" → "asat" dönüşümünde ilk iki harf yer değiştirmiştir; uzaklık 2'dir. İki komşu harfin takasını tek işlem sayan ölçünün adı **Damerau–Levenshtein uzaklığıdır**. Yazım hatalarının önemli bir kısmı komşu harf takası olduğu için yazım denetleyicileri bu türü sık kullanır.

5.2.6.6 İki satırlık bellek

Bir satırı doldurmak için yalnız bir önceki satır gerekir. Yalnız uzaklık isteniyorsa bütün tabloyu saklamak gereksizdir. İki dizi yeter: `onceki` ve `simdi`. Her satır bitince iki dizi yer değiştirir.

```
int levenshteinIkiSatir(String a, String b) {
  if (a.length() < b.length()) { String t = a; a = b; b = t; }  // b kısa olan
  int n = b.length();
  int[] onceki = new int[n + 1], simdi = new int[n + 1];
  for (int j = 0; j <= n; j++) onceki[j] = j;
  for (int i = 1; i <= a.length(); i++) {
    simdi[0] = i;
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        simdi[j] = onceki[j - 1];
      else
        simdi[j] = 1 + Math.min(onceki[j - 1], Math.min(simdi[j - 1], onceki[j]));
    int[] t = onceki; onceki = simdi; simdi = t;   // satırlar yer değiştirir
  }
  return onceki[n];
}
```

Metot önce kısa dizgiyi `b` yapar; böylece dizilerin uzunluğu *min(m, n) + 1* olur. `simdi[0] = i` kenar değeridir. İç döngü aynı bağıntıyı uygular: sol üst `onceki[j − 1]`, üst `onceki[j]`, sol `simdi[j − 1]`'dir. Satır bitince başvurular takas edilir ve yeni satır `onceki` olur. Döngüden sonra cevap `onceki[n]`'dedir. Metot, sunumdaki bütün örneklerde tam tabloyla aynı sonucu verir.

Kazanç büyüktür. İki dizgi 10.000'er karakter olsun. Tam tablo $10.001^2 \approx 10^8$ tam sayı, yani yaklaşık 400 MB tutar. İki satırlık sürüm $2 \cdot 10.001$ tam sayı, yaklaşık 80 KB kullanır. Zaman yine $O(m \cdot n)$'dir. Bedeli, geri okumanın artık yapılamamasıdır; düzenleme dizisi de isteniyorsa tam tablo ya da daha ileri bir teknik (Hirschberg algoritması) gerekir.

5.2.6.7 Öteki benzerlik ölçüleri

Levenshtein tek benzerlik ölçüsü değildir. Problemin yapısına göre daha basit ya da farklı bir ölçü daha uygun olabilir. Tablo 5.2.21 dört ölçüyü aynı dizgi çifti, AGGTAB ve GXTXAYB üzerinde karşılaştırır.

Ölçü	Ne ölçer?	Zaman	AGGTAB · GXTXAYB
Hamming	eşit uzunluktaki iki dizgide farklı konum sayısı	$O(n)$	tanımsız (6 ≠ 7)
Levenshtein	ekleme, silme, değiştirme sayısı	$O(m \cdot n)$	4
LCS	en uzun ortak alt dizinin uzunluğu	$O(m \cdot n)$	4 (GTAB)
En uzun ortak alt dizgi	en uzun ardışık ortak parçanın uzunluğu	$O(m \cdot n)$	1 (ör. “G”)

Tablo 5.2.21: Dört benzerlik ölçüsünün karşılaştırması.

Hamming uzaklığı yalnız eşit uzunluktaki dizgiler için tanımlıdır ve karşılıklı konumlardaki farklı karakterleri sayar. Tek bir geçişle, $O(n)$ zamanda hesaplanır. Örneğin “karolin” ile “kathrin”in Hamming uzaklığı 3'tür (r/t, o/h, l/r). Hamming ölçüsü, kayıp ya da fazla karakterin olmadığı durumlar için uygundur: sabit uzunluklu ikili kodlar, iletişimde bit hataları.

```
int hamming(String a, String b) {
  if (a.length() != b.length())
    throw new IllegalArgumentException("uzunluklar farklı");
  int d = 0;
  for (int i = 0; i < a.length(); i++)
    if (a.charAt(i) != b.charAt(i)) d++;
  return d;
}
```

Eşit uzunluktaki iki dizgi için Levenshtein uzaklığı Hamming uzaklığını aşamaz. Hamming'in saydığı her farklı konum bir değiştirmeyle düzeltilir; bu, Levenshtein'a izin verilen bir dönüşümdür. Levenshtein ise ekleme ve silmeyi de kullanarak bazen daha kısa bir yol bulur. “karolin” · “kathrin” çiftinde iki ölçü de 3'tür.

Tanım 5.2.8

Bir dizginin bazı karakterleri silinerek elde edilen diziye o dizginin **alt dizisi** denir; kalan karakterlerin göreli sırası korunur ama ardışık olmaları gerekmez. İki dizginin **en uzun ortak alt dizisi** (*longest common subsequence*, LCS), ikisinin de alt dizisi olan en uzun dizidir. **En uzun ortak alt dizgi** (*longest common substring*) ise ikisinde de *ardışık* olarak geçen en uzun parçadır.

AGGTAB ile GXTXAYB'nin LCS'si GTAB'dir. G, T, A, B harfleri iki dizgide de bu sırayla geçer: AGGTAB'de 2., 4., 5., 6. konumlarda; GXTXAYB'de 1., 3., 5., 7. konumlarda. Arada başka harfler bulunması önemli değildir. En uzun ortak alt dizgi ise yalnız 1 karakterliktir: iki dizgide de ardışık geçen iki harflik bir parça yoktur. LCS de Levenshtein'a benzer bir DP tablosuyla bulunur:

```
int lcs(String a, String b) {
  int m = a.length(), n = b.length();
  int[][] L = new int[m + 1][n + 1];            // 0. satır ve sütun 0
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        L[i][j] = L[i - 1][j - 1] + 1;          // çapraz + 1
      else
        L[i][j] = Math.max(L[i - 1][j], L[i][j - 1]);
  return L[m][n];
}
```

Bağıntı şudur: karakterler eşitse *L[i][j] = L[i−1][j−1] + 1*, yani çapraz değere 1 eklenir; eşit değilse *L[i][j] = max(L[i−1][j], L[i][j−1])*. Levenshtein en küçüğü ararken LCS en büyüğü arar. Metot AGGTAB · GXTXAYB için 4, CARS · PAIRS için 3 (ARS), kitten · sitting için 4 (ittn) döndürür.

LCS ile düzenleme uzaklığı arasında basit bir bağ vardır. Yalnız ekleme ve silmeye izin verilirse, değiştirme yasaklanırsa, uzaklık $m + n - 2 \cdot LCS$ olur. LCS'ye girmeyen her karakter ya silinir ya eklenir. AGGTAB · GXTXAYB için bu $6 + 7 - 8 = 5$'tir. Değiştirmeye de izin veren Levenshtein bir silme ile bir eklemeyi tek değiştirmede birleştirerek uzaklığı 4'e indirir.

En uzun ortak alt dizgi için bağıntı daha da basittir. Karakterler eşitse $L[i][j] = L[i-1][j-1] + 1$; eşit değilse $L[i][j] = 0$, çünkü ardışıklık bozulmuştur. Cevap sağ alt köşede değil, tablonun en büyük değerindedir:

```
String enUzunOrtakAltDizgi(String a, String b) {
  int m = a.length(), n = b.length(), enIyi = 0, bitis = 0;
  int[][] L = new int[m + 1][n + 1];
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1)) {
        L[i][j] = L[i - 1][j - 1] + 1;          // ardışık parça uzar
        if (L[i][j] > enIyi) { enIyi = L[i][j]; bitis = i; }
      }                                          // eşleşmezse 0 kalır
  return a.substring(bitis - enIyi, bitis);
}
```

Metot “ercan” · “sercan” için “ercan”ı, kitten · sitting için “itt”yi döndürür. Eşit uzunlukta birden çok parça varsa ilk bulunanı verir; AGGTAB · GXTXAYB çiftinde “A” döner. Kopya metin tespiti ve iki dosyanın ortak bloklarını bulma bu ölçünün kullanıldığı yerlerdir.

Uzaklıklar, karşılaştırılabilir bir **benzerlik oranına** da çevrilebilir. Yaygın bir tanım $1 - d / max(m, n)$'dir: özdeş dizgilerde 1, hiç ortak yanı olmayanlarda 0'a yakın bir değer verir. kitten · sitting için bu $1 - 3/7 \approx 0{,}57$'dir. Bir yazım denetleyicisi, örneğin benzerliği 0,7'nin üstündeki sözlük kelimelerini öneri olarak gösterebilir.

> **Sık yapılan hata**
>
> Alt dizi ile alt dizgiyi karıştırmak. LCS ardışıklık istemez; en uzun ortak alt dizgi ister. AGGTAB · GXTXAYB çiftinde biri 4, öteki 1'dir. Hamming uzaklığını farklı uzunluktaki dizgilere uygulamak da bir başka hatadır: tanım gereği hesaplanamaz ve yukarıdaki `hamming` metodu bu durumda hata fırlatmalıdır.

5.2.7 Sonek dizisi

Bir metinde aynı örüntü birçok kez aranacaksa her aramada metni baştan taramak pahalıdır. Kitapların sonundaki dizin bu sorunu çözer: kelimeler sıralı bir listede durur ve her birinin geçtiği sayfalar yanında yazılıdır. Sonek dizisi, bir metnin bütün alt dizgileri için böyle bir dizin kurar.

5.2.7.1 Tanım

Tanım 5.2.9

n karakterli bir *t* dizgisinin *i*. soneki, *i* konumundan başlayıp dizginin sonuna kadar giden alt dizgidir: *t[i..n−1]*. *t*'nin **sonek dizisi** (*suffix array*, SA), bütün soneklerin sözlüksel sıraya dizilmiş hâlinin başlangıç indisleridir: *SA[k]*, sıralı listede *k*. sıradaki sonekin başladığı konumdur.

i	sonek	sıra	SA	sıralı sonek
0	banana	0	5	a
1	anana	1	3	ana
2	nana	2	1	anana
3	ana	3	0	banana
4	na	4	4	na
5	a	5	2	nana

Tablo 5.2.22: “banana” dizgisinin sonekleri (solda, başlangıç konumuna göre) ve sıralı hâli (sağda). Sağdaki SA sütunu sonek dizisidir.

“banana”nın altı soneki vardır. Sıralandıklarında “a” en öne gelir, çünkü “ana” ve “anana”nın önekidir. Ardından “ana”, “anana”, “banana”, “na”, “nana” gelir. Sonek dizisi *SA = [5, 3, 1, 0, 4, 2]*'dir. Dizide sonekler değil, yalnız başlangıç indisleri saklanır. Sonekin kendisi gerektiğinde `t.substring(SA[k])` ile elde edilir.

Sonek dizisinin gücü basit bir gözleme dayanır: bir dizginin her alt dizgisi, bir sonekin önekidir. “nan” alt dizgisi, 2 konumundan başlayan “nana” sonekinin önekidir. Bu yüzden bir örüntünün metinde nerede geçtiğini bulmak, o örüntüyle başlayan sonekleri bulmak demektir. Sonekler sıralı olduğundan bu sonekler listede yan yana durur.

Aynı işi yapan daha eski bir yapı **sonek trie'sidir**: bütün sonekleri bir trie'ye ekleyen ağaç. Sonek trie'si çok hızlı arama yapar, ama düğüm sayısı farklı alt dizgi sayısıyla orantılıdır ve en kötü durumda $O(n^2)$'dir. “banana” gibi altı harflik bir dizginin bile 15 farklı alt dizgisi, dolayısıyla kökle birlikte 16 düğümü vardır. Sonek dizisi ise yalnız *n* tam sayı tutar: $O(n)$ bellek. Dizgi arama, sıralama, genetik dizilim analizi, BWT ve veri sıkıştırma sonek dizisinin başlıca kullanım alanlarıdır.

5.2.7.2 Oluşturma yöntemleri

Sonek dizisini kurmanın en kolay yolu, bütün sonekleri bir karşılaştırmalı sıralamayla sıralamaktır:

```
int[] sonekDizisi(String t) {        // kaba kuvvet: O(n² log n)
  int n = t.length();
  Integer[] ix = new Integer[n];
  for (int i = 0; i < n; i++) ix[i] = i;
  Arrays.sort(ix, (x, y) -> t.substring(x).compareTo(t.substring(y)));
  int[] sa = new int[n];
  for (int i = 0; i < n; i++) sa[i] = ix[i];
  return sa;
}
```

Bu kaba kuvvet yöntemi *O(n log n)* karşılaştırma yapar. Her karşılaştırma iki soneki karakter karakter okur ve en kötü durumda *O(n)* sürer; ayrıca `substring` her çağrıda yeni bir dizgi kopyalar. Toplam zaman *O(n^2 log n)*'dir. Kısa metinlerde yeterlidir, ama birkaç milyon karakterlik bir genom dizisinde kullanılamaz. Daha hızlı yöntemler, sonekleri karakter karakter karşılaştırmaktan kaçınır.

Yöntem	Fikir	Zaman
Kaba kuvvet	bütün sonekler oluşturulur, `Arrays.sort` ile sıralanır; her karşılaştırma *O(n)*	*O(n^2 log n)*
Manber–Myers (1990)	önek ikiye katlama: *k* karakterlik sıralardan *2k* karakterlik sıra, sıra çiftleri basamak sıralamasıyla sıralanarak bulunur	*O(n log n)*
Larsson–Sadakane (1999)	aynı ikiye katlama; yalnız henüz ayrışmamış gruplar yeniden sıralanır	*O(n log n)*
DC3 (2003), SA-IS (2009)	soneklerin bir kısmı özyinelemeli sıralanır, kalanı onlardan türetilip birleştirilir	*O(n)*

Tablo 5.2.23: Sonek dizisi oluşturma yöntemleri.

Önek ikiye katlama yöntemleri *O(n log n)* zamanlıdır, doğrusal değildir. Doğrusal zamanlı kurulum ancak DC3 ve SA-IS gibi sonraki algoritmalarla mümkün olmuştur. SA-IS bugün pratikte de en hızlı yöntemlerden biridir ve birçok biyoinformatik aracında kullanılır.

5.2.7.3 Önek ikiye katlama

Manber ve Myers'ın fikri şudur. Önce sonekler yalnız ilk karakterlerine göre sıralanır ve her sonekin bir **sırası** (*rank*) belirlenir: ilk karakteri aynı olan soneklerin sırası da aynıdır. Sonra ilk iki karaktere, ilk dört karaktere, ilk sekiz karaktere göre sıralanır. Her turda bakılan önek uzunluğu ikiye katlanır. Bir turu hızlandıran gözlem şudur: *i*. sonekin ilk *2k* karakteri, iki parçadan oluşur. İlk *k* karakter *i*'de, sonraki *k* karakter *i + k*'de başlayan sonekin önekidir. İkisinin de *k* karakterlik sırası önceki turdan bilinir. Bu yüzden *2k* karakterlik sıralama, harf karşılaştırmadan, *(sıra[i], sıra[i + k])* sayı çiftleri sıralanarak yapılır.

Örnek 5.2.7

"banana" üzerinde üç tur:

1. **1 karakter.** Sonekler ilk karakterlerine göre sıralanır: a → 0, b → 1, n → 2. Sıralar: *sıra[0..5] = 1, 0, 2, 0, 2, 0*. 1, 3, 5'te başlayan sonekler (hepsi a ile başlar) henüz ayrışmamıştır.
2. **2 karakter.** Anahtar *(sıra[i], sıra[i + 1])*'dir; *i + 1 ≥ n* ise ikinci öğe −1'dir. 5: (0, −1); 1: (0, 2); 3: (0, 2); 0: (1, 0); 2: (2, 0); 4: (2, 0). Sıralanınca yeni sıralar 5 → 0, 1 ve 3 → 1, 0 → 2, 2 ve 4 → 3 olur. “an” ile başlayan iki sonek ve “na” ile başlayan iki sonek hâlâ eşittir.
3. **4 karakter.** Anahtar *(sıra[i], sıra[i + 2])*'dir. 5: (0, −1); 3: (1, 0); 1: (1, 1); 0: (2, 3); 4: (3, −1); 2: (3, 3). Bütün anahtarlar farklıdır; yeni sıralar 0'dan 5'e kadar dağılır. Sıralama tamamlanmıştır: *SA = [5, 3, 1, 0, 4, 2]*.

i	ilk 4 karakter	anahtar	yeni sıra
5	a	(0, −1)	0
3	ana	(1, 0)	1
1	anan	(1, 1)	2
0	bana	(2, 3)	3
4	na	(3, −1)	4
2	nana	(3, 3)	5

Tablo 5.2.24: Önek ikiye katlamanın son turu (ilk 4 karakter). Anahtar *(sıra[i], sıra[i + 2])*; harf karşılaştırılmaz, yalnız sayı çiftleri sıralanır.

Üçüncü turdaki bir satır yöntemi özetler. 3'te başlayan “ana” ile 1'de başlayan “anana”nın ilk iki karakteri aynıdır; ikisinin de 2 karakterlik sırası 1'dir. Fark sonraki iki karakterdedir: 3 + 2 = 5'te başlayan “a”nın sırası 0, 1 + 2 = 3'te başlayan “an…”ın sırası 1'dir. (1, 0) < (1, 1) olduğu için “ana”, “anana”dan önce gelir. Hiçbir harf yeniden okunmamıştır.

Tur sayısı en çok $\lceil \log_2 n \rceil$'dir, çünkü $2^k \geq n$ olduğunda bakılan önek her soneki baştan sona kapsar ve bütün sıralar farklı olmak zorundadır. Her turda n sayı çifti iki geçişli basamak sıralamasıyla $O(n)$'de sıralanabilir. Toplam süre $O(n \log n)$'dir. Aşağıdaki sürüm fikri anlaşılır kılmak için sayı çiftlerini `Arrays.sort` ile sıralar; bu yüzden her tur $O(n \log n)$, toplam $O(n \log^2 n)$ sürer:

```
int[] onekKatla(String t) {          // önek ikiye katlama (sıralama Arrays.sort ile)
  int n = t.length();
  Integer[] sa = new Integer[n];
  int[] sira = new int[n], yeni = new int[n];
  for (int i = 0; i < n; i++) { sa[i] = i; sira[i] = t.charAt(i); }
  for (int k = 1; ; k *= 2) {
    final int kk = k;
    final int[] r = sira;
    Comparator<Integer> cmp = (x, y) -> {
      if (r[x] != r[y]) return Integer.compare(r[x], r[y]);
      int rx = x + kk < n ? r[x + kk] : -1, ry = y + kk < n ? r[y + kk] : -1;
      return Integer.compare(rx, ry);
    };
    Arrays.sort(sa, cmp);            // anahtar: (sira[i], sira[i + k])
    yeni[sa[0]] = 0;
    for (int i = 1; i < n; i++)
      yeni[sa[i]] = yeni[sa[i - 1]] + (cmp.compare(sa[i - 1], sa[i]) < 0 ? 1 : 0);
    System.arraycopy(yeni, 0, sira, 0, n);
    if (sira[sa[n - 1]] == n - 1) break;   // bütün sıralar farklı
  }
  int[] s = new int[n];
  for (int i = 0; i < n; i++) s[i] = sa[i];
  return s;
}
```

`sira` dizisi başlangıçta karakter kodlarını tutar. Her turda `cmp` karşılaştırıcısı önce `sira[x]`, eşitse `sira[x + k]` değerine bakar. Sıralamadan sonra yeni sıralar verilir: bir sonek kendinden öncekinden farklıysa sırası bir artar, değilse aynı kalır. En son sonekin sırası $n - 1$ olduğunda bütün sıralar farklıdır ve döngü biter. Metot “banana” için *[5, 3, 1, 0, 4, 2]*, “mississippi” için *[10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]* döndürür; ikisi de kaba kuvvetle aynıdır.

5.2.7.4 Sonek dizisiyle arama

Sonekler sıralı olduğundan bir P örüntüsüyle başlayan sonekler SA'da bitişik bir aralık oluşturur. Aralığın iki ucu ikili aramayla bulunur. İlk uç, P'den küçük olmayan ilk sonektir. İkinci uç, bu noktadan sonra P ile başlamayan ilk sonektir.

```
int[] ara(String t, int[] sa, String p) {
  int lo = 0, hi = sa.length;
  while (lo < hi) {   // ilk >= p
    int mid = (lo + hi) / 2;
    if (t.substring(sa[mid])
          .compareTo(p) < 0) lo = mid + 1;
    else hi = mid;
  }
  int bas = lo;
  hi = sa.length;
  while (lo < hi) {   // p ile başlamayan
    int mid = (lo + hi) / 2;
    if (t.startsWith(p, sa[mid]))
      lo = mid + 1;
    else hi = mid;
  }
  return new int[] { bas, lo };
}
```

İlk döngü klasik bir “alt sınır” ikili aramasıdır. Ortadaki sonek *P*'den küçükse aranan uç sağdadır: `lo = mid + 1`. Değilse uç `mid`'de ya da solundadır: `hi = mid`. Döngü bittiğinde `lo`, *P*'den küçük olmayan ilk sonekin sırasıdır ve `bas`'a yazılır. İkinci döngü `lo`'yu olduğu yerde bırakıp `hi`'yi dizinin sonuna çeker. Bu kez soru “bu sonek *P* ile başlıyor mu?”dur. Başlıyorsa aralığın sonu sağdadır. Döngü bittiğinde `lo`, aralığın hemen sonrasını gösterir. Metot yarı açık `[bas, lo)` aralığını döndürür.



Şekil 5.2.29: `ara("banana", sa, "an")` başlangıcı: `lo = 0`, `hi = 6` (dizi sonu). “an” ile başlayan sonekler mavi önekle gösterilmiştir.

Döngü	`mid`	Sonek	Soru ve cevap	Yeni aralık
1	3	banana	“banana” < “an”? hayır → `hi = 3`	[0, 3)
1	1	ana	“ana” < “an”? hayır → `hi = 1`	[0, 1)
1	0	a	“a” < “an”? evet → `lo = 1`	[1, 1)
—	—	—	`bas = 1`, `hi = 6`	[1, 6)
2	3	banana	“an” ile başlıyor mu? hayır → `hi = 3`	[1, 3)

Döngü	`mid`	Sonek	Soru ve cevap	Yeni aralık
2	2	anana	“an” ile başlıyor mu? evet → `lo = 3`	[3, 3)

Tablo 5.2.25: `ara("banana", sa, "an")` izlemesi, *SA = [5, 3, 1, 0, 4, 2]*.



Şekil 5.2.30: İlk döngü bitti: “an”dan küçük olmayan ilk sonek sıra 1'dedir, `bas = 1`. İkinci arama için `hi` yeniden dizi sonuna çekilir.



Şekil 5.2.31: Sonuç `[1, 3)`: sıra 1 ve 2 seçilir; SA değerleri 3 ve 1'dir.

Sonuç `[1, 3)`'tür: sıra 1 ve 2. Bu sıralardaki SA değerleri 3 ve 1'dir. “an”, “banana” içinde 1 ve 3 konumlarında geçer. Toplam 5 karşılaştırma yapılmıştır. Her ikili arama en çok $\lfloor \log_2 6 \rfloor + 1 = 3$ karşılaştırma yapar. Her karşılaştırma ise en çok $m = |P|$ karakter okur. Toplam zaman $O(m \log n)$'dir. Bir milyar karakterlik bir metinde bile her arama yaklaşık 60 sonek karşılaştırmasıyla biter.

Sonek dizisinin yanına bir **LCP dizisi** (*longest common prefix*) eklenirse daha fazla soru cevaplanır. LCP dizisi, sıralı listede komşu iki sonekin en uzun ortak önekinin uzunluğunu tutar. “banana” için “a” ile “ana”nın ortak öneki 1, “ana” ile “anana”nınki 3 karakterdir. Metinde iki kez geçen en uzun alt dizgi, LCP dizisinin en büyük değeridir. “banana”da bu “ana”dır ve LCP dizisinin bir taramasıyla, $O(n)$'de bulunur.

Sık yapılan hata

Sonek dizisinin değerlerini sıra numarasıyla karıştırmak. `ara`'nın döndürdüğü `[1, 3)` aralığı sıralı listedeki *sıralardır*; örüntünün metindeki konumları bu sıralardaki `sa[k]` değerleridir (3 ve 1). Bir başka hata, aralığın konumlarını sıralı sanmaktır: SA sıralı sonekleri tutar, ko-

numlar ise karışık sırada gelir.

5.2.8 Burrows–Wheeler dönüşümü

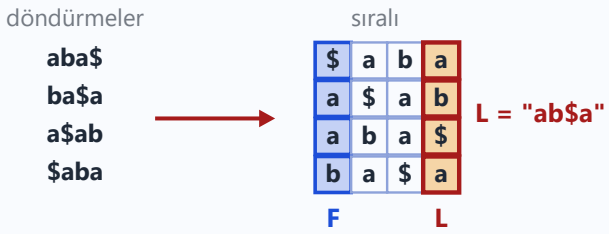
Sıkıştırma algoritmaları tekrarlardan beslenir: “AAAAA”, “A” harfinin beş kez yazıldığı bilgisiyle kısaltılabilir. Ama doğal metinde aynı harfler genellikle yan yana durmaz. Burrows–Wheeler dönüşümü harflerin yerini, aynı harfler bir araya toplanacak biçimde değiştirir. Üstelik bunu geri alınabilir biçimde yapar.

5.2.8.1 Tanım ve adımlar

Tanım 5.2.10

Bir *s* dizgisinin **Burrows–Wheeler dönüşümü** (*Burrows–Wheeler transform*, BWT) üç adımda hesaplanır. Önce *s*'nin bütün döndürmeleri (*cyclic rotations*) oluşturulur. Sonra bu döndürmeler sözlüksel olarak sıralanır. En son sıralı döndürmelerin son karakterleri yukarıdan aşağıya okunur. Elde edilen dizgiye **L sütunu** denir. Ters dönüşüm için ayrıca orijinal dizginin sıralı listedeki satır numarası saklanır ya da *s*'nin sonuna, alfabedeki bütün karakterlerden küçük ve yalnız bir kez geçen bir *$* işareti eklenir.

Bir döndürme, dizginin ilk karakterinin sona taşınmasıyla elde edilir; bu işlem *n* kez yapılınca dizgi başa döner. *n* karakterlik bir dizginin *n* döndürmesi vardır. BWT karakterleri değiştirmez, yalnız yerlerini değiştirir: L sütunu, *s*'nin harflerinin bir başka sıralamasıdır. Dönüşümü Michael Burrows ve David Wheeler 1994'te bir teknik raporla yayımladı.



Şekil 5.2.32: “aba$” dizgisinin döndürmeleri (solda) sıralanır (sağda). İlk sütun F, son sütun L'dir; BWT çıktısı *L = “ab$a”*'dır.

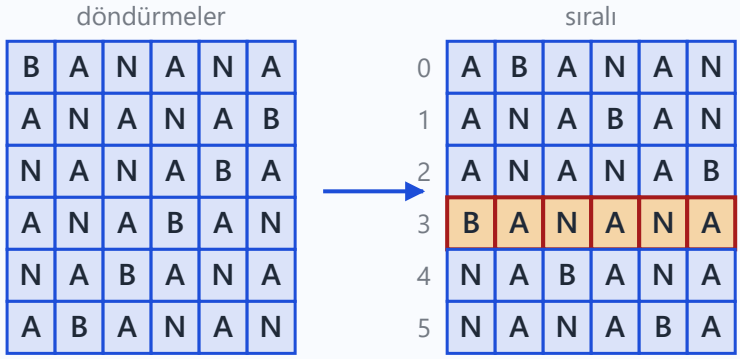
Şekil 5.2.32'deki küçük örnekte “aba$”ın dört döndürmesi aba$, baa, aab ve $aba'dır. *$* en küçük karakter olduğu için sıralı listede $aba en öne geçer; ardından aab, aba ve ba$a gelir. Son karakterler a, b, $, a'dır: *L = “ab$a”*. Sıralı listenin ilk sütunu ise **F sütunudur**: $, a, a, b.

5.2.8.2 Örnekler: BANANA ve abraca

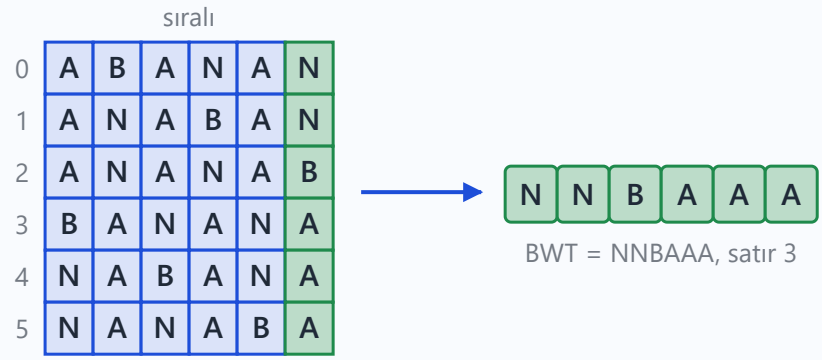
BWT'nin harfleri nasıl gruplandırdığı “BANANA” örneğinde açıkça görülür. Bu örnekte *$* kullanılmaz; orijinal satırın numarası saklanır.



Şekil 5.2.33: 1. adım: BANANA'nın altı döndürmesi. Her satır, bir öncekinin ilk karakterinin sona taşınmasıyla elde edilir.



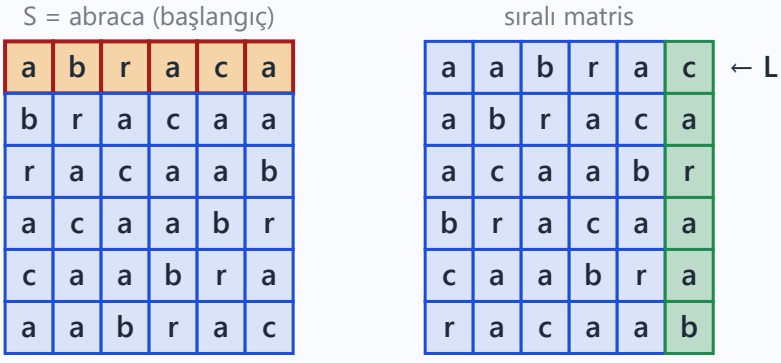
Şekil 5.2.34: 2. adım: döndürmeler alfabetik olarak sıralanır. Orijinal dizgi BANANA, 0'dan sayarak 3 numaralı satıra düşer.



Şekil 5.2.35: 3. adım: son karakterler (L sütunu) okunur. Çıktı (NNBAAA, 3)'tür; aynı harfler gruplanmıştır.

Döndürmeler BANANA, ANANAB, NANABA, ANABAN, NABANA ve ABANAN'dır. Sıralı liste ABANAN, ANABAN, ANANAB, BANANA, NABANA, NANABA'dır. Son karakterler N, N, B, A, A, A'dır. Çıktı (NNBAAA, 3) olur. Orijinal dizgide hiçbir harf komşusuyla aynı değildi; dönüşümden sonra iki N ve üç A yan yana gelmiştir.

Gruplanmanın nedeni sıralamanın yapısıdır. Sıralı listede, aynı karakterlerle başlayan döndürmeler yan yana durur. Örneğin “ANA…” ile başlayan iki döndürme komşudur. L sütunu, her döndürmenin ilk karakterinden hemen *önce* gelen karakteri tutar. Doğal metinde aynı bağlamdan önce genellikle aynı karakter gelir: İngilizcede “he” bağlamından önce çoğu zaman t gelir (the), Türkçede “ğı” bağlamından önce çoğu zaman a gelir (ağı, dağı). Benzer bağlamlar sıralamada bir araya gelince onlardan önce gelen aynı karakterler de L'de yan yana düşer.



Şekil 5.2.36: “abraca”: başlangıç matrisi (solda, döndürmeler) ve sıralı matris (sağda). Son sütun *L = “caraab”*dır.

“abraca” için sıralı döndürmeler aabrac, abraca, acaabr, bracaa, caabra, racaab'dır. Son sütun *L = “caraab”*dır. Orijinal “abraca” sıralı matriste 0'dan sayarak 1 numaralı satırdadır; ters dönüşüm için bu indis saklanır. İlk sütun F = aaabcr'dir ve L'nin harflerinin sıralanmış hâlidir. Bu her zaman doğru-

dur: her döndürme aynı harfleri içerdiğinden F ve L aynı harf çoklu kümesini taşır, F ise sıralıdır. L bilindiğinde F'yi bulmak için L'yi sıralamak yeter.

Doğrudan yöntem *n* döndürmeyi açıkça üretip sıralar:

```
String bwt(String s) {              // doğrudan yöntem; s sonunda $ var
  int n = s.length();
  String[] dondurme = new String[n];
  for (int i = 0; i < n; i++)
    dondurme[i] = s.substring(i) + s.substring(0, i);
  Arrays.sort(dondurme);            // O(n² log n) zaman, O(n²) bellek
  StringBuilder L = new StringBuilder();
  for (String r : dondurme) L.append(r.charAt(n - 1));
  return L.toString();
}
```

Bu yöntem $O(n^2)$ bellek kullanır, çünkü *n* döndürmenin her biri *n* karakterdir. Sıralama $O(n \log n)$ karşılaştırma yapar ve her karşılaştırma $O(n)$ sürebilir: toplam $O(n^2 \log n)$ zaman. Bir megabaytlık bir blok için 10^{12} karakterlik bir matris gerekir; bu açıkça uygulanamaz. Pratikte BWT sonek dizisinden hesaplanır. Metnin sonunda *$* varsa döndürmelerin sırası soneklerin sırasıyla aynıdır. İki döndürme en geç *$*'a gelince ayrışır; *$*'dan sonrası karşılaştırmaya katılmaz. *k*. sıralı döndürme *SA[k]*'da başlar; son karakteri ondan hemen önceki karakterdir. Bu yüzden *L[k] = T[SA[k] − 1]*'dir; *SA[k] = 0* ise *L[k] = $*'dır. Sonek dizisi doğrusal zamanda kurulursa BWT de *O(n)*'de elde edilir.

5.2.8.3 Tersine çevirme: LF eşlemesi

BWT'nin asıl şaşırtıcı yanı, yalnız L sütunundan (ve *$*'dan ya da satır indisinden) orijinal metnin geri elde edilebilmesidir. Örnek olarak *T = "banana$"* alınsın. *$* en küçük ve tekil karakterdir.

i	F	…	L
0	$	banan	a
1	a	$bana	n
2	a	na$ba	n
3	a	nana$	b
4	b	anana	$
5	n	a$ban	a
6	n	ana$b	a

Tablo 5.2.26: "banana$" için sıralı döndürme matrisi. F ilk, L son sütundur; ortadaki karakterler ters dönüşümde bilinmez.

BWT çıktısı *L = "annb$aa"*'dır. F, L'nin sıralanmış hâlidir: *F = "$aaabnn"*. Her satır bir döndürme olduğundan, satırın son karakteri *L[i]*, metinde ilk karakter *F[i]*'den hemen önce gelir. Ters dönüşüm bu komşuluğu zincirleme kullanır. Eksik olan tek bilgi, L'deki bir karakterin F'de hangi satıra karşılık geldiğidir. Aynı harf birden çok kez geçtiğinde bu açık değildir. Aşağıdaki teorem bu belirsizliği giderir.

Teorem 5.2.4 (LF eşlemesi)

Bir *c* karakteri L'de *k*. kez geçiyorsa (0'dan sayarak), metindeki aynı karakter F'de de *c*'nin *k*. geçişidir. Buna göre *L[i]*'nin F'deki satırı *LF(i) = C[c] + sıra[i]*'dir. Burada *c = L[i]*, *C[c]* metinde *c*'den küçük karakterlerin sayısı (yani *c*'nin F'de başladığı satır), *sıra[i]* ise *L[i]*'nin L'de kaçıncı *c* olduğudur.

İspat. L'de *c* ile biten iki satır $x \cdot c$ ve $y \cdot c$ biçimindedir ve sıralı listede $x \cdot c$ önce gelsin; yani $x \cdot c < y \cdot c$. Bu iki satırı birer kez sağa döndürmek $c \cdot x$ ve $c \cdot y$ satırlarını verir; bunlar da matrisin satırlarıdır ve F'de *c* ile başlar. İki dizgi aynı karakterle başladığında sıralarına geri kalan kısım karar verir. $x \cdot c$ ile $y \cdot c$ eşit uzunluktadır ve farklı oldukları ilk konum *x* ile *y*'nin içindedir. Bu yüzden $x < y$'dir ve $c \cdot x < c \cdot y$ olur. Demek ki L'de *c* ile biten satırların sırası, onların döndürülmüş hâllerinin F'deki sırasıyla aynıdır. L'deki *k*. *c*, F'deki *k*. *c*'dir. F sıralı olduğu için *c*'nin F'deki ilk satırı *C[c]*'dir ve *k*. *c*, *C[c] + k* satırındadır. □

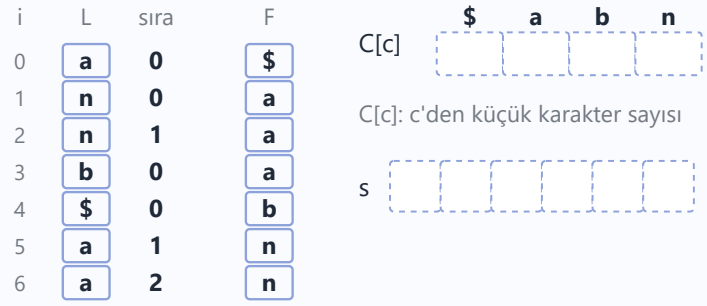
Ters dönüşüm artık basittir. *$* ile başlayan satır 0'dır; bu satır metnin kendisinin bir döndürmesidir ve son karakteri *L[0]*, metnin *$*'dan önceki son karakteridir. *L[0]* okunur ve *LF(0)* satırına gidilir. O satırın L'deki karakteri metinde bir önceki karakterdir. Bu yürüyüş sürdürüldükçe metin sondan başa doğru okunur. "banana$" için satırlar $0 \to 1 \to 5 \to 2 \to 6 \to 3 \to 4$ sırasıyla gezilir. Okunan L karakterleri a, n, a, n, a, b'dir. Ters çevrilince "banana" elde edilir.

Sıra sayıları ve *C* tablosu önceden bir geçişte hesaplanırsa her adım sabit zamanlıdır ve ters dönüşüm *O(n)* sürer. Metinde *$* yoksa, BANANA örneğinde olduğu gibi orijinal satır indisi (3) saklanır ve yürüyüş 0 yerine o satırdan başlatılır.

5.2.8.4 Kod izleme: bwtTers

```
String bwtTers(String L) {   // $ ile
  int n = L.length();
  int[] sira = new int[n];
  int[] say = new int[256];
  int[] C = new int[256];
  for (int i = 0; i < n; i++)
    sira[i] = say[L.charAt(i)]++;
  for (int c = 1; c < 256; c++)
    C[c] = C[c - 1] + say[c - 1];
  StringBuilder s = new StringBuilder();
  int i = 0;                    // $ satırı
  for (int k = 0; k < n - 1; k++) {
    char c = L.charAt(i);
    s.insert(0, c);
    i = C[c] + sira[i];         // LF(i)
  }
  return s.toString();
}
```

Metot dört bölümden oluşur. İlk döngü her *i* için `sira[i]`'yi hesaplar: `say[c]`, o ana kadar L'de kaç *c* görüldüğünü tutar ve sonra bir artar. İkinci döngü `C` tablosunu bir önek toplamıyla kurar: `C[c]`, *c*'den küçük bütün karakterlerin sayısıdır. Üçüncü bölüm yürüyüştür. `i = 0`, yani *$* satırından başlanır. Her adımda `L.charAt(i)` okunur ve oluşan metnin *başına* eklenir (`s.insert(0, c)`); metin sondan başa doğru okunduğu için ters çevirmeye gerek kalmaz. Sonra `i`, *LF(i)*'ye güncellenir. Döngü *n − 1* tur sürer; *$* metne eklenmez.



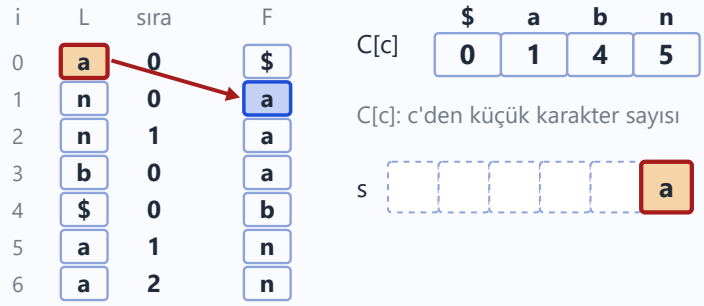
Şekil 5.2.37: `bwtTers("annb$aa")`: L, sıra sayıları ve F. `sira[6] = 2`, çünkü L[6] üçüncü a'dır.

Hazırlık adımlarının sonuçları şunlardır. *sıra = [0, 0, 1, 0, 0, 1, 2]*: L'deki a'lar (0, 5, 6 konumları) 0, 1, 2 sırasını, n'ler (1, 2) 0, 1 sırasını alır; b ve *$* tek oldukları için 0'dır. C tablosu: *C[$] = 0*, *C[a] = 1*, *C[b] = 4*, *C[n] = 5*. Bu değerler F sütununda her harfin başladığı satırlardır.

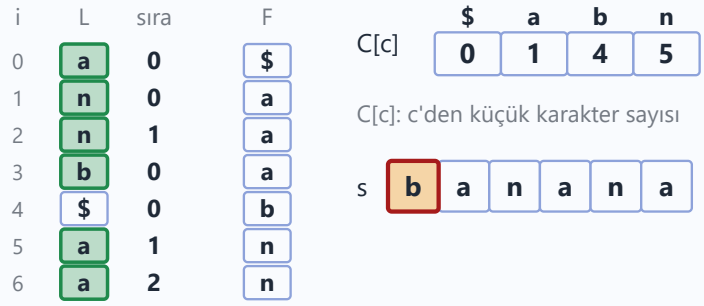
k	i	L[i]	s	*LF(i) = C[c] + sıra[i]*
0	0	a	a	1 + 0 = 1

k	i	L[i]	s	*LF(i) = C[c] + sıra[i]*
1	1	n	na	5 + 0 = 5
2	5	a	ana	1 + 1 = 2
3	2	n	nana	5 + 1 = 6
4	6	a	anana	1 + 2 = 3
5	3	b	banana	4 + 0 = 4

Tablo 5.2.27: `bwtTers("annb$aa")` yürüyüşü.



Şekil 5.2.38: İlk adım: `L[0] = 'a'` metne eklenir; *LF(0) = C[a] + sıra[0] = 1*. Ok, L'deki ilk a'nın F'deki ilk a'ya karşılık geldiğini gösterir.



Şekil 5.2.39: Satırlar 0 → 1 → 5 → 2 → 6 → 3 → 4 izlendi. Sıradaki satır *$* ile biteceği için döngü *n − 1 = 6* turda durur. Sonuç: banana.

Altıncı turdan sonra `i = 4`'tür. 4 numaralı satırın L karakteri *$*'dır; bu satır “banana$”ın kendisidir ve yürüyüş başa dönmüştür. Döngü tam bu noktada sona erer. Zaman *O(n + R)*, ek bellek `sira`, `say` ve `C` dizileri için *O(n + R)*'dir. Bir uygulama ayrıntısı vardır: `s.insert(0, c)` her çağrıda bütün dizgiyi kaydırdığı için aslında *O(n)* sürer. Büyük girdilerde karakterler bir diziye sondan başa yazılır ya da sona eklenip en sonda bir kez ters çevrilir.

Sık yapılan hata

BWT çıktısını yalnız L sütunu olarak saklamak. Ne *$* işareti ne de orijinal satır indisi saklanırsa dönüşüm geri alınamaz: "BANANA"nın bütün döndürmeleri aynı L sütununu (NNBAAA) verir. Bir başka hata, *$*'ı alfabenin en küçük karakteri olarak seçmemektir. Java'da '$' (36) harflerden ve rakamlardan küçüktür; ama metinde boşluk (32) varsa bu koşul bozulur ve LF eşlemesi yanlış satırlara gider.

5.2.8.5 Sıkıştırmadaki yeri

BWT kendi başına bir sıkıştırma değildir; çıktı girdiyle aynı uzunluktadır. Sıkıştırmaya hazırlık yapar. Yaygın `bzip2` programı BWT'yi dört aşamalı bir zincirin başında kullanır:

1. **BWT.** Metin bloklara bölünür ve her bloğun dönüşümü alınır. Aynı harfler bir araya toplanır.
2. **Öne taşı** (*move-to-front*, MTF). Her karakter, bir listedeki konumuyla kodlanır ve sonra listenin başına taşınır. Yakın zamanda görülen harfler küçük sayılarla kodlanır; art arda gelen aynı harfler 0'lara dönüşür.
3. **Çalışma uzunluğu kodlaması** (*run-length encoding*, RLE). Art arda gelen aynı değerler (değer, tekrar sayısı) çiftiyle yazılır. MTF'nin ürettiği uzun 0 dizileri burada kısalır.
4. **Huffman kodlaması.** Sık görülen değerlere kısa, seyrek görülenlere uzun bit dizileri verilir.

Zincirin her halkası bir öncekinin hazırladığı yapıdan yararlanır. BWT olmadan MTF'nin çıktısında bu kadar çok 0 bulunmaz; MTF olmadan RLE uzun tekrarlar bulamaz. BWT ayrıca biyoinformatikte de önemlidir. FM-index adı verilen yapı, BWT ile LF eşlemesini birleştirerek milyarlarca harflik bir genomda kısa okumaları sıkıştırılmış veri üzerinde arar.

5.2.9 Sık yapılan hatalar

Notun içine dağılmış uyarılar ve sunumdaki hata listesi burada bir arada toplanır.

Sık yapılan hata

Kod sırasını alfabe sanmak. `compareTo` karakter kodlarını karşılaştırır: "Zebra" < "apple" ve "şeker" > "zeytin" çıkar. Büyük/küçük harf farkı ve Türkçe harfler için `Collator` ya da önceden bir normalleştirme gerekir.

Sık yapılan hata

Sayıları dizgi olarak sıralamak. "10" < "9" olur, çünkü '1' < '9'dur. Sayısal sıra için dizgiler tam sayıya çevrilir ya da soldan sıfırla aynı uzunluğa getirilir.

Sık yapılan hata

LSD'de kararsız alt sıralama. Her geçiş kararlı olmalıdır; aksi hâlde önceki geçişlerin kurduğu sıra bozulur. Saymalı sıralamada yerleştirme sondan başa yapılır. Baştan sona yerleştiren sürüm sunumdaki on dizgiyi `add, ace, bed, bee, bad, …` sırasına koyar.

Sık yapılan hata

Kaçışsız nokta ve açgözlü `.*` **.** `a.b` , "a+b" ile de eşleşir; `<.*>` bütün satırı yutar. Doğrusu `\.` ve tembel `.*?` 'dır. `(a+)+$` gibi iç içe niceleyiciler, geri izlemeli motorlarda üstel sayıda denemeye yol açabilir.

Sık yapılan hata

`find` **ile** `matches` **'ı karıştırmak.** `find` bir parça arar, `matches` dizginin tamamını ister. Doğrulama için ya `matches` kullanılır ya da ifade `^` ve `$` ile çevrelenir; yoksa "abc123" gibi bir girdi "sayı" diye kabul edilebilir.

Sık yapılan hata

FSM'de ara durumu sonuç sanmak. DFA'nın kararı yalnız girdi bittiğindeki duruma bağlıdır. Yolda bir kabul durumundan geçmek dizgiyi kabul ettirmez. Örüntü otomatında ise uyuşmazlıkta 0'a dönmek eşleşmeleri kaçırır; doğru durum en uzun uyan önektir.

Sık yapılan hata

DP tablosunda indis kayması. Tablo *(m + 1) × (n + 1)*'dir; *D[i][j]* için `charAt(i - 1)` ve `charAt(j - 1)` karşılaştırılır. İlk satırı ve sütunu başlatmamak sonucu bozar.

Sık yapılan hata

Alt dizi ile alt dizgiyi karıştırmak. LCS ardışıklık istemez; en uzun ortak alt dizgi ister. Hamming uzaklığı ise yalnız eşit uzunluktaki dizgiler için tanımlıdır.

Sık yapılan hata

BWT'de satır indisini ya da *$*'ı saklamamak. Bir dizginin bütün döndürmeleri aynı L sütununu verir; ek bilgi olmadan dönüşüm geri alınamaz. Sonek dizisinde ise sıralar ile metindeki

konumlar (SA değerleri) birbirine karıştırılmamalıdır.

5.2.10 Özet

- **Sözlüksel sıra:** ilk farklı karakter karar verir; bir dizgi ötekinin önekiyse kısa olan önce gelir. Bilgisayar karakter kodlarını kullanır: büyük harfler küçüklerden önce, Türkçe harfler z'den sonra gelir. Bir karşılaştırma $O(W)$ sürer.
- **Karşılaştırmalı sıralama:** basit yöntemle $O(N^2 \cdot W)$, birleştirme sıralamasıyla $O(W \cdot N \log N)$. Karşılaştırmaya dayanan her yöntem $\Omega(N \log N)$ karşılaştırma yapar.
- **LSD basamak sıralaması:** sağdan sola, her geçişte o karaktere göre kararlı saymalı sıralama; $O(W \cdot (N + R))$ zaman, $O(N + R)$ bellek. Doğruluğu, "k geçişten sonra son k karaktere göre sıralı" değişmezine ve kararlılığa dayanır.
- **MSD basamak sıralaması:** soldan sağa, her kova ayrı ayrı özyinelemeli sıralanır; yalnız ayırt edici önekler okunur. Küçük kovalarda eklemeli sıralamaya geçilir.
- **Düzenli ifadeler:** `. * + ? [ ] ^ $ ( ) { } |` ve `\d \w \s \b`; niceleyiciler varsayılan olarak açgözlüdür, `?` eklenince tembel olur. Bakış ifadeleri bir konumu sınar, karakter tüketmez. Java'da `Pattern` / `Matcher`; `find` parça, `matches` bütün dizgi arar.
- **Sonlu durum makinesi:** $(Q, \Sigma, \delta, q_0, F)$. DFA'da her (durum, sembol) için tek geçiş vardır ve tarama $O(n)$'dir. DFA, NFA ve düzenli ifadeler aynı dilleri tanır; alt küme kurulumu en kötü 2^n durum üretir.
- **Örüntü eşleme otomatı:** durum, metnin sonunda örüntüyle uyuşan en uzun önekin uzunluğudur. Kurulum $O(m \cdot |\Sigma|)$, arama $O(n)$; KMP'nin otomat biçimidir.
- **Levenshtein:** eşitse sol üst, değilse $1 + \min(\text{sil}, \text{ekle}, \text{değiştir})$; $O(m \cdot n)$ zaman. Düzenleme dizisi tablodan geri okunur; yalnız uzaklık için iki satır yeter.
- **Öteki ölçüler:** Hamming eşit uzunlukta, $O(n)$. LCS ve en uzun ortak alt dizgi $O(m \cdot n)$. Yalnız ekleme ve silmeyle uzaklık $m + n - 2 \cdot LCS$'dir.
- **Sonek dizisi:** sıralı soneklerin başlangıç indisleri; $O(n)$ bellek. Kaba kuvvetle $O(n^2 \log n)$, önek ikiye katlamayla $O(n \log n)$, SA-IS ile $O(n)$ kurulur. Arama iki ikili aramayla $O(m \log n)$.
- **BWT:** sıralı döndürmelerin son sütunu ve satır indisi (ya da $\$$). Aynı harfleri gruplar; LF eşlemesiyle $O(n)$'de geri çevrilir. `bzip2`'nin ilk adımıdır: BWT → öne taşı → RLE → Huffman.

5.2.11 Kavram listesi

dizgi *string* · alfabe *alphabet* · önek *prefix* · sonek *suffix* · alt dizgi *substring* · sözlüksel sıra *lexicographic order* · kararlı sıralama *stable sort* · saymalı sıralama *counting sort* · basamak sıralaması *radix sort* · LSD / MSD basamak sıralaması *LSD / MSD radix sort*

düzenli ifade *regular expression* · niceleyici *quantifier* · ileri / geri bakış *lookahead / lookbehind* · sonlu durum makinesi *finite state machine (FSM)* · deterministik sonlu otomat *deterministic finite automaton (DFA)* · alt küme kurulumu *subset construction* · tuzak durum *trap state* · düzenleme uzaklığı *edit distance* · Levenshtein uzaklığı *Levenshtein distance* · dinamik programlama *dynamic programming* · geri okuma *traceback* · en uzun ortak alt dizi *longest common subsequence (LCS)* · en uzun ortak alt dizgi *longest common substring* · sonek dizisi *suffix array* · önek ikiye katlama *prefix doubling* · Burrows–Wheeler dönüşümü *Burrows–Wheeler transform (BWT)* · öne taşı *move-to-front (MTF)*

5.2.12 Kendini Dene

1. `{"kedi", "Kedi", "kediler", "ked", "kafa", "çay"}` dizisi `Arrays.sort` ile sıralanınca sonuç nedir? Türkçe sözlük sırasından farkı nerededir?

   Cevap

   Sonuç `Kedi, kafa, ked, kedi, kediler, çay`'dır. 'K' (75) bütün küçük harflerden önce gelir. “kafa”, ikinci karakterde a < e ile öne geçer. “ked”, “kedi”nin, “kedi” de “kediler”in önekidir. 'ç' (231) z'den bile büyük olduğu için “çay” en sona düşer. Türkçe sözlükte ise “çay” k ile başlayanlardan öncedir ve “Kedi” ile “kedi” yan yana gelir.

2. `dog, cat, dot, cab, cog` dizgilerine LSD basamak sıralaması uygulanırsa her geçişten sonra dizi nasıl olur?

   Cevap

   3. karaktere göre: `cab, dog, cog, cat, dot` (b < g < t; dog ve cog giriş sırasını korur). 2. karaktere göre: `cab, cat, dog, cog, dot` (a kovasında cab, cat; o kovasında dog, cog, dot önceki sırayla). 1. karaktere göre: `cab, cat, cog, dog, dot`.

3. `^[A-Z]\d{2,3}[a-z]?$` ifadesi “A12”, “B1234”, “c123”, “Z999x” ve “Q7y” dizgilerinden hangileriyle `matches` anlamında eşleşir?

   Cevap

   Yalnız “A12” ve “Z999x”. İfade bir büyük harf, iki ya da üç rakam ve isteğe bağlı tek bir küçük harf ister. “B1234”te dört rakam vardır. “c123” küçük harfle başlar. “Q7y”de tek rakam vardır.

4. `(?<=a)b` örüntüsü “abcabb” metninde hangi konumlarda eşleşir? “abbb” metninde `b+` ve `b+?`

ile `find` döngüsü hangi parçaları bulur?

Cevap

Geri bakış, önünde a bulunan b'leri seçer: 1 ve 4 konumları. 5'teki b'nin önünde b vardır. Bakış karakter tüketmediği için iki eşleşmenin metni de yalnız “b”dir. “abbb” metninde açgözlü `b+` tek eşleşme olarak “bbb”yi bulur. Tembel `b+?` her seferinde tek b alır ve üç ayrı “b” eşleşmesi verir.

5. “aab” örüntüsü için örüntü eşleme otomatının geçiş tablosu (alfabe {a, b}) nasıldır? “aaab” metni okunurken durumlar nasıl değişir ve örüntü nerede bulunur?

Cevap

Durum 0: a → 1, b → 0. Durum 1: a → 2, b → 0. Durum 2: a → 2, b → 3. Durum 3: a → 1, b → 0. Durum 2'de a gelirse okunanların sonu “aaa” olur ve en uzun uyan önek yine “aa”dır; bu yüzden 2'de kalınır. “aaab” için durumlar 1, 2, 2, 3'tür. Durum 3'e 3 numaralı karakterde ulaşılır; örüntü $3 - 3 + 1 = 1$ konumunda başlar.

6. `levenshtein("kitap", "katip")` kaçtır? Tablonun son satırı nedir ve hangi düzenleme dizisi bu uzaklığı verir?

Cevap

Uzaklık 2'dir. Son satır (p) *5, 4, 3, 3, 3, 2*'dir. k, t ve p iki dizgide aynı konumlardadır; yalnız 2. ve 4. harfler farklıdır. Bir çözüm iki değiştirmedir: i → a (katap), a → i (katip).

7. “ABCBDAB” ile “BDCABA” için LCS uzunluğu, yalnız ekleme ve silmeyle uzaklık ve Levenshtein uzaklığı kaçtır?

Cevap

LCS uzunluğu 4'tür (ör. BCBA). Yalnız ekleme ve silmeyle uzaklık $7 + 6 - 2 \cdot 4 = 5$'tir. Levenshtein uzaklığı da 5'tir; bu çiftte değiştirmeler daha kısa bir yol sağlamaz.

8. “abaab” dizgisinin sonek dizisi nedir? “ab” ile başlayan sonekler hangi sıralardadır ve örüntü metinde hangi konumlarda geçer?

Cevap

Sıralı sonekler aab, ab, abaab, b, baab'dır; $SA = [2, 3, 0, 4, 1]$. “ab” ile başlayanlar sıra 1 (ab) ve sıra 2'dir (abaab). SA değerleri 3 ve 0 olduğu için “ab” metinde 0 ve 3 konumlarında geçer.

9. “abab$” dizgisinin BWT'si nedir ve LF eşlemesiyle nasıl geri çevrilir?

   Cevap

   Sıralı döndürmeler $abab, ab$ab, abab$, b$aba, bab$a'dır; *L = “bb$aa”*. *F = “$aabb”*, *C[$] = 0*, *C[a] = 1*, *C[b] = 3*; sıra sayıları *0, 1, 0, 0, 1*'dir. Yürüyüş 0'dan başlar: *L[0] = b*, *LF = 3*; *L[3] = a*, *LF = 1*; *L[1] = b*, *LF = 4*; *L[4] = a*, *LF = 2*. 2 numaralı satırın L karakteri *$*'dır ve yürüyüş durur. Okunanlar b, a, b, a'dır; başa eklenerek “abab” elde edilir.

10. İkili dizgilerde “00” ile bitenleri kabul eden bir DFA nasıl kurulur? “10100” girdisi hangi durumlardan geçer?

    Cevap

    Üç durum yeter: q_0 (son sembol 0 değil ya da girdi boş; başlangıç), q_1 (son sembol tek bir 0) ve q_2 (son iki sembol 00; kabul). Geçişler: q_0: $0 \to q_1$, $1 \to q_0$; q_1: $0 \to q_2$, $1 \to q_0$; q_2: $0 \to q_2$, $1 \to q_0$. “10100” için yol $q_0 \to q_0 \to q_1 \to q_0 \to q_1 \to q_2$'dir ve dizgi kabul edilir.

5.2.13 İleri okuma

- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Bölüm 5: dizgi sıralama (LSD, MSD, üç yollu hızlı sıralama), alt dizgi arama, düzenli ifadeler ve veri sıkıştırmayı Java koduyla ve deneysel ölçümlerle anlatır.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (Cambridge University Press, 1997): sonek ağaçları, düzenleme uzaklığı ve hizalama algoritmalarını hesaplamalı biyoloji örnekleriyle derinlemesine işler.
- J. E. Hopcroft, R. Motwani ve J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (3. baskı, 2006): DFA, NFA, alt küme kurulumu ve Kleene teoreminin ispatlı anlatımı.
- R. Cox, “Regular Expression Matching Can Be Simple And Fast” (2007, swtch.com): Thompson yapısıyla otomat tabanlı eşlemeyi geri izlemeli motorlarla karşılaştıran, üstel yavaşlamayı ölçümlerle gösteren makale.

5.2.14 Kaynaklar

1. M. Burrows, D. J. Wheeler, “A Block-sorting Lossless Data Compression Algorithm”, DEC SRC Research Report 124, 1994.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (dizgi eşleştirme otomatları).
3. R. Cox, “Regular Expression Matching Can Be Simple And Fast”, 2007, https://swtch.com/~rsc/regexp/regexp1.html.

4. J. E. F. Friedl, *Mastering Regular Expressions*, 3. baskı, O'Reilly, 2006; Java `java.util.regex.Pattern` belgeleri.
5. D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
6. J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3. baskı, 2006.
7. V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals", *Soviet Physics Doklady* 10(8), 1966.
8. U. Manber, G. Myers, "Suffix Arrays: A New Method for On-Line String Searches", *SIAM Journal on Computing* 22(5), 1993.
9. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (5.1 String Sorts, 5.4 Regular Expressions).
10. R. A. Wagner, M. J. Fischer, "The String-to-String Correction Problem", *Journal of the ACM* 21(1), 1974.

KISIM II · BÖLÜM 5.3

Trie Veri Yapısı

Bir arama kutusuna “an” yazıldığında altında “and”, “ant”, “anahtar” gibi öneriler belirir. Bu önerileri milisaniyeler içinde bulmak için sözlükteki her kelimeyi tek tek denemek gerekmez. Kelimeler, ortak başlangıçlarını paylaşan bir ağaçta saklanırsa “an” ile başlayan bütün kelimeler tek bir düğümün altında toplanır. Bu ağacın adı **trie**, Türkçesiyle önek ağacıdır. Not önce trie'nin yapısını ve Java'daki düğüm sınıfını tanıtır. Ardından ekleme, arama, önek sorgusu, otomatik tamamlama ve silme işlemleri kodlarıyla ve adım adım izlemeleriyle anlatılır. Joker karakterli arama, yazım denetimi ve en uzun ortak önek gibi uygulamalar bu temel işlemlerin üzerine kurulur. Analiz kesimi işlemlerin süresini ve trie'nin bellek bedelini sayısal örneklerle hesaplar; trie'yi hash tablosu ve ikili arama ağacıyla karşılaştırır. Not, trie ailesinin öteki üyeleriyle biter: ikili trie, üçlü arama trie'si, sıkıştırılmış trie, sonek trie'si ve Aho–Corasick.

ÖĞRENME ÇIKTILARI

Yapı
Trie'nin kök, düğüm, kenar ve kelime sonu işaretinden oluşan yapısını açıklamak; bir kelime kümesinin trie'sini çizmek, düğümlerini ve boş referanslarını saymak.

İşlemler
Ekleme, arama, önek sorgusu, otomatik tamamlama ve silme işlemlerini Java ile yazmak ve adım adım izlemek. Önek sayacı, joker karakterli arama ve en uzun ortak önek gibi türev işlemleri bu temel üzerine kurmak.

Analiz
İşlemlerin $O(L)$ süresini ve $O(N \cdot M)$ belleğini gerekçelendirmek; çocukları saklamanın yollarını tartmak; trie'yi hash tablosu ve dengeli ikili arama ağacıyla karşılaştırmak.

Türevler
İkili trie, üçlü arama trie'si, sıkıştırılmış trie, sonek trie'si ve Aho–Corasick algoritmasının hangi soruyu çözdüğünü açıklamak ve küçük örneklerde izlemek.

5.3.1 Trie kavramı

Dizgilerle (*string*) çalışan programların büyük bölümü aynı soruyu tekrar tekrar sorar: bu kelime sözlükte var mı? Yazım denetleyicisi her kelime için bu soruyu sorar. Derleyici her tanımlayıcıyı anahtar sözcük tablosunda arar. Bir yönlendirici her paketin adresini rota tablosunda arar. Bu soruların çoğu için hash tablosu iyi bir cevaptır. Ama bazı sorular tam kelimeyle değil, kelimenin başıyla ilgilidir: "an ile başlayan kelimeler hangileri?", "bu adresin en uzun eşleşen öneki hangisi?" Bu tür sorularda kelimelerin ortak başlangıçlarını bir arada tutan bir yapı gerekir.

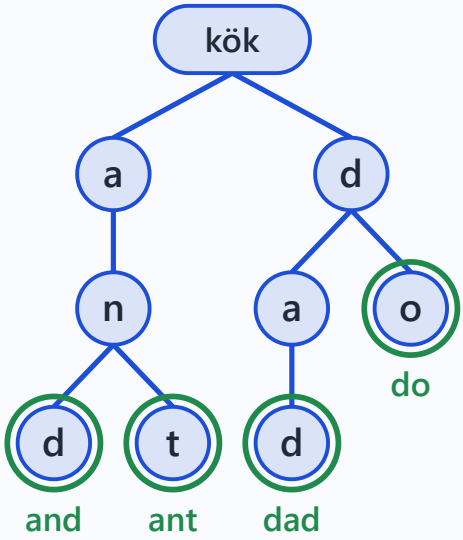
Notta sık kullanılan iki terim baştan netleşmelidir. Bir dizginin **öneki** (*prefix*), dizginin başından alınan bir parçadır: "and" dizgisinin önekleri "", "a", "an" ve "and"dir. **Sonek** (*suffix*) ise dizginin sonundan alınan parçadır: "and"in sonekleri "and", "nd", "d" ve ""dir. Dizginin içinden ardışık karakterlerle alınan her parça bir **alt dizgidir**. Her alt dizgi, bir sonekin önekidir. Bu gözlem sonek trie'sinin temelidir.

5.3.1.1 Tanım

Bir sözlük düşünülsün: and, ant, dad, do. Bu dört kelimeden ikisi "a", ikisi "d" ile başlar. "and" ile "ant" ayrıca "an" önekini de paylaşır. Trie bu paylaşımı açıkça gösterir. Kökten aşağı her adım bir karakter okur. Aynı karakterle başlayan kelimeler aynı dala girer ve ancak ilk farklı karakterde ayrılır.

> **Tanım 5.3.1**
>
> **Trie** (*trie, prefix tree*), bir dizgi kümesini köklü bir ağaçta saklayan veri yapısıdır. Ağacın her kenarı bir karakter taşır; bir düğümün çocuklarına giden kenarların karakterleri birbirinden farklıdır. Kök boş dizgiyi temsil eder. Her düğüm, kökten o düğüme inen yoldaki karakterlerin birleşimini, yani bir öneki temsil eder. Kümedeki bir kelimenin bittiği düğüm ayrıca **kelime sonu** olarak işaretlenir. Bir kelime ancak kökten başlayan yolu varsa ve yolun son düğümü işaretliyse kümededir.

Tanımın son cümlesi önemlidir. Yolun var olması tek başına yetmez. "an" yolu trie'de vardır, ama "an" kümede değildir; "an" düğümü işaretsizdir. Yol, yalnızca kümedeki bir kelimenin önekidir. Trie'nin adı, bilgi geri getirme anlamındaki İngilizce *retrieval* sözcüğünden gelir. Adı E. Fredkin 1960'ta "Trie Memory" makalesinde önermiştir. Türkçede "önek ağacı" adı da kullanılır; bu notta iki ad aynı yapıyı anlatır.



Şekil 5.3.1: and, ant, dad ve do kelimelerinin trie'si. Çift çemberli düğümlerde bir kelime biter; kelime düğümün altında yazılıdır.

Şekil 5.3.1'de kökten çıkan iki kenar vardır: “a” ve “d”. “a” düğümünün tek çocuğu “n”dir. “an” düğümünde yol ikiye ayrılır: “d” ile “and”e, “t” ile “ant”e gidilir. Sağ tarafta “d” düğümü de ikiye ayrılır: “a” ile “da”ya, “o” ile “do”ya. “da”nın tek çocuğu “d”dir ve “dad” orada biter. Dört kelimenin toplam karakter sayısı *3 + 3 + 3 + 2 = 11*'dir, ama kök dışında yalnız 8 düğüm vardır. Aradaki 3 karakter ortak öneklerin paylaşılmasıyla kazanılmıştır: “and” ile “ant” iki karakter (“a”, “n”), “dad” ile “do” bir karakter (“d”) paylaşır.

Trie'nin kullanım alanları bu yapıdan doğar. Otomatik tamamlamada yazılan önekin düğümüne inilir ve altındaki kelimeler önerilir. Yazım denetiminde kelime trie'de aranır; bulunamazsa ona yakın kelimeler yine trie üzerinde aranır. IP yönlendirmede adresler bit dizisi olarak saklanır ve bir paketin adresine en uzun önekle eşleşen rota bulunur.

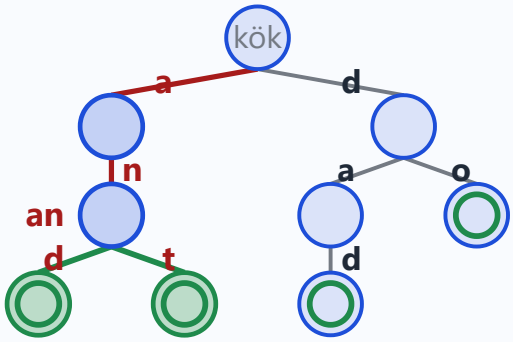
5.3.1.2 Neden trie?

Soru şudur: bir sözlükte “an” ile başlayan bütün kelimeler bulunacak. Sözlükte n kelime vardır ve kelimelerin uzunluğu en çok L'dir. Farklı veri yapıları bu soruya farklı bedellerle cevap verir.

Dizi ya da liste. Her kelime sırayla alınır ve ilk iki karakteri “an” ile karşılaştırılır. Bir karşılaştırma en çok L karakter okur, toplam iş $O(n \cdot L)$'dir. Sözlük büyüdükçe süre doğrusal olarak artar. Dizi sıralıysa ikili aramayla ilk eşleşen kelime $O(L \cdot \log n)$'de bulunabilir, ama yeni kelime eklemek dizideki kaydırma yüzünden pahalı olur.

Hash tablosu. Tam bir kelimeyi ortalama $O(L)$ sürede bulur: önce kelimenin L karakterinden hash değeri hesaplanır, sonra ilgili kovaya bakılır. Ancak hash tablosu önek sorusuna cevap veremez. “an” ile “and”in hash değerleri arasında hiçbir ilişki yoktur. İyi bir hash fonksiyonu zaten benzer anahtarları tablonun uzak yerlerine dağıtmak için tasarlanır. Bu yüzden “an ile başlayan kelimeler” sorusu ancak bütün tablo taranarak cevaplanır.

Trie. Kökten önce “a” dalına, sonra “n” dalına inilir; iki adım yeter. Ulaşılan düğümün altındaki her kelime “an” ile başlar ve bu alt ağacın dışında “an” ile başlayan kelime yoktur. Önek düğümünü bulmanın maliyeti sözlükteki kelime sayısına değil, önekin uzunluğuna bağlıdır.



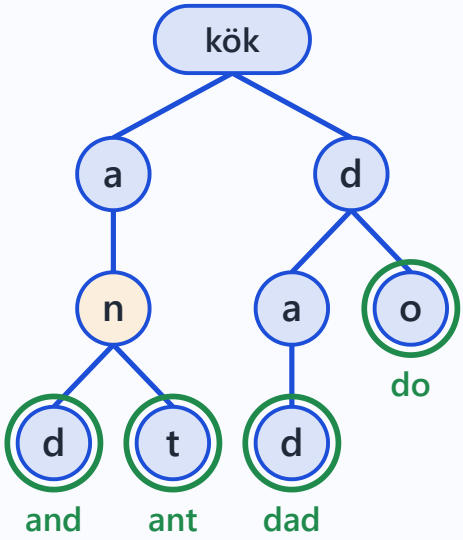
Şekil 5.3.2: “an” önekinin düğümü. Bu düğümün alt ağacındaki her kelime (and, ant) “an” ile başlar; kalan dallara hiç bakılmaz.

Trie'nin asıl katkısı bu gözlemdir: işlem süresi n'ye değil L'ye bağlıdır. Sözlükte dört kelime de olsa, bir milyon kelime de olsa “dad” en çok üç adımda bulunur. Dengeli bir ikili arama ağacında ise $log_2 10^6 \approx 20$ karşılaştırma gerekir ve her karşılaştırma kelimenin karakterlerini yeniden okur.

5.3.1.3 Temel bileşenler

Trie dört bileşenden oluşur. Bu bileşenlerin her biri koddaki bir öğeye karşılık gelir.

- **Kök** (*root*): boş dizgiyi temsil eden düğümdür; hiçbir karaktere karşılık gelmez. Her işlem kökten başlar. Boş bir trie yalnız kökten oluşur.
- **Düğümler** (*nodes*): kökten bir düğüme kadarki yol bir önektir. Her düğüm çocuklarına giden bağlantıları tutar. Bir düğümün en çok M çocuğu olabilir; M alfabenin boyudur. Küçük İngilizce harflerde $M = 26$'dır.
- **Kenarlar** (*edges*): bir düğümü çocuğuna bağlar ve bir karakter taşır. Aynı düğümden çıkan iki kenar aynı karakteri taşıyamaz; bu kural her önekin trie'de en çok bir düğümü olmasını sağlar.
- **Kelime sonu işareti** (*end of word*): koddaki adı `kelimeSonu`'dur. Düğümde bir kelimenin bittiğini belirtir. Şekillerde çift çemberle gösterilir.

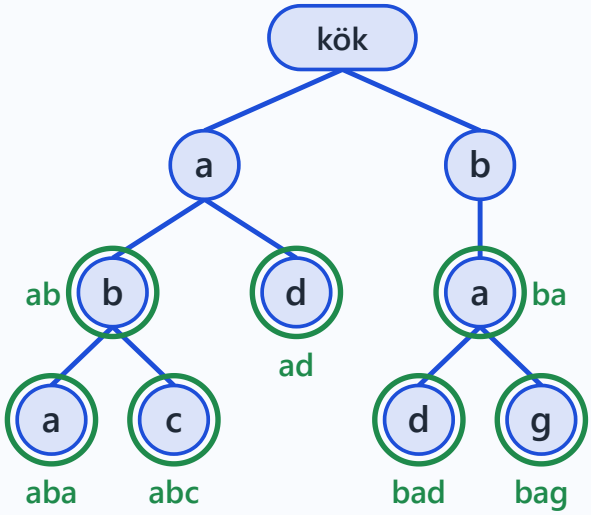


Şekil 5.3.3: Trie'nin bileşenleri. Renkli düğüm "an" önekini temsil eder. "an" bir kelime değildir (işaretsiz), ama "and" ile "ant"in önekidir.

Bileşenler arasındaki ilişki şöyle özetlenebilir. Düğümle önek arasında birebir bir eşleme vardır: her önek en çok bir düğümle temsil edilir. Kenarlar karakterleri taşır, düğümler ise önekleri. Kelime sonu işareti, hangi öneklerin aynı zamanda kümedeki kelimeler olduğunu söyler. Yaprak düğümler her zaman işaretlidir: işaretsiz bir yaprak, hiçbir kelimenin yolu üzerinde olmayan gereksiz bir düğüm olurdu. Silme işlemi de bu yüzden işaretsiz yaprak bırakmaz.

5.3.1.4 Örnek: yedi kelimelik trie

Biraz daha büyük bir örnek kelime sonu işaretinin neden şart olduğunu gösterir. Kelimeler ab, aba, abc, ad, ba, bad ve bag'dir. Yedi kelimenin toplam uzunluğu *2 + 3 + 3 + 2 + 2 + 3 + 3 = 18* karakterdir.



Şekil 5.3.4: ab, aba, abc, ad, ba, bad, bag kelimelerinin trie'si. "ab" ve "ba" düğümleri hem kelime sonudur hem

de başka kelimelerin yolu üzerindedir.

Örnek 5.3.1

Soru. Yedi kelimelik trie'de kaç düğüm vardır? Hangi düğümler işaretlidir?

Çözüm. Düğümler kelimelerin bütün farklı önekleridir. "a" ile başlayanlar: a, ab, aba, abc, ad (5 düğüm). "b" ile başlayanlar: b, ba, bad, bag (4 düğüm). Toplam 9 düğüm, kökle birlikte 10 düğüm vardır. 18 karakter 9 düğüme sığmıştır. İşaretli düğümler kelimelerin kendileridir: ab, aba, abc, ad, ba, bad, bag. İşaretsiz iki düğüm kalır: a ve b. Bunlar yalnız önektir.

"ab" düğümüne yalnız şekle bakarak karar verilemez. Düğümün iki çocuğu (a ve c) vardır; yani "ab" başka kelimelerin önekidir. Aynı zamanda kendisi de bir kelimedir. Yapı bu iki durumu ayırt edemez; ayrımı yalnız `kelimeSonu` işareti yapar. İşaret olmasaydı "ab" ile "a" aynı türden düğümler olurdu ve "ab" sözlükte mi sorusunun cevabı kaybolurdu.

Sık yapılan hata

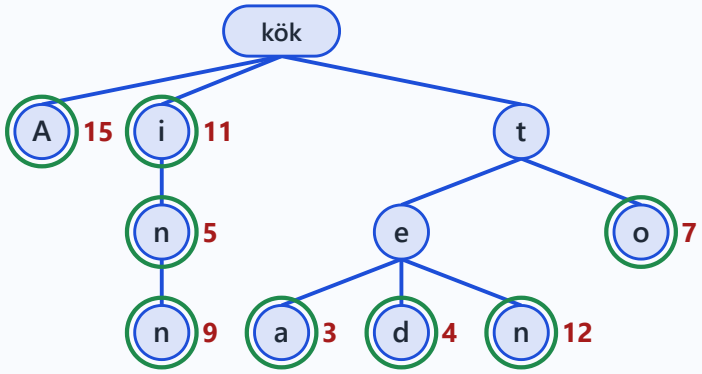
Kelime sonunu yaprak olmakla karıştırmak. "Bir kelime yaprakta biter" kuralı yanlıştır. "ab" ve "ba" iç düğümdür, ama kelimedir. Doğru kural ters yöndedir: her yaprak bir kelime sonudur, ama her kelime sonu yaprak değildir.

5.3.1.5 Değer taşıyan trie

Trie yalnız bir kelime kümesini değil, bir eşlemeyi (*map*) de saklayabilir. Bunun için `kelimeSonu` bayrağının yerine düğümde bir değer tutulur. Değer `null` ise düğüm bir anahtarın sonu değildir; değilse o düğümde biten anahtarın değeridir. Böylece trie, anahtarları dizgi olan bir sözlük gibi çalışır.

Anahtar	Değer	Anahtar	Değer
A	15	tea	3
i	11	ted	4
in	5	ten	12
inn	9	to	7

Tablo 5.3.1: Değer taşıyan trie'nin anahtarları ve değerleri.



Şekil 5.3.5: Değer taşıyan trie. Sayılar düğümde saklanan değerlerdir. “t” ve “te” düğümleri değer taşımaz; yalnız önektir.

Şekilde sekiz anahtar vardır ve her biri bir düğümde biter. “i”, “in” ve “inn” zincir hâlinde birbirinin önekidir; üçü de değer taşır. “t” düğümü ise iki dala ayrılır: “to” ve “te”. “te”nin üç çocuğu vardır: tea, ted ve ten. Anahtar sayısı 8, kök dışındaki düğüm sayısı 10'dur: A, i, in, inn, t, te, tea, ted, ten, to. Değer taşımayan iki düğüm (t ve te) yalnız yolu oluşturur.

Bu örnekte bir ayrıntı daha vardır: “A” büyük harftir. Çocukları 26 öğeli bir dizide tutan olağan trie yalnız küçük harfleri saklayabilir. Büyük harfli bir anahtar için ya daha geniş bir alfabe gerekir ya da çocuklar bir `HashMap` içinde tutulur. Alfabe büyüdükçe düğüm başına bellek de büyür; bu bedel trie'nin en zayıf yanıdır.

5.3.2 Düğüm sınıfı

5.3.2.1 TrieDugumu sınıfı

Trie'yi Java'da kurmanın en doğrudan yolu, her düğümde alfabenin her harfi için bir çocuk referansı tutmaktır. Alfabe küçük İngilizce harflerden oluşuyorsa her düğümde 26 öğeli bir dizi yeterlidir. Dizinin *i*. hücresi, düğümden *i*. harfle inilen çocuğu gösterir; o harfle başlayan bir devam yoksa hücre `null`'dır.

```
public class TrieDugumu {
  TrieDugumu[] cocuk;  // 'a'..'z'
  boolean kelimeSonu;  // kelime biter mi?

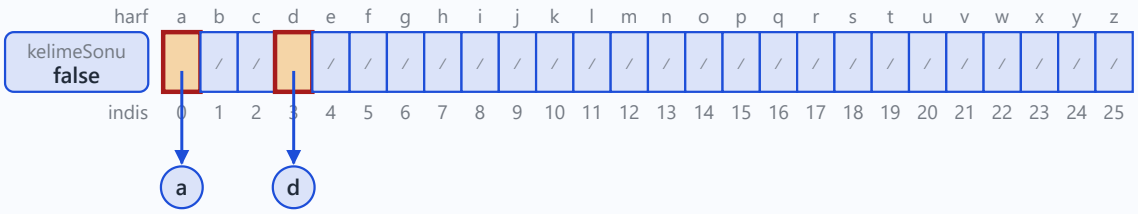
  public TrieDugumu() {
    kelimeSonu = false;
    cocuk = new TrieDugumu[26];
  }
}
```

Sınıfın iki alanı vardır. `cocuk` dizisi 26 referans tutar; Java yeni bir dizinin bütün referans hücrelerini `null` ile başlatır. `kelimeSonu`, kökten bu düğüme gelen yolun kümedeki bir kelime olup olmadığını söyler; yapıcıda `false` yapılır. Java bu değeri de varsayılan olarak verir, ama açıkça yazmak niyeti belli eder.

Sınıfta karakterin kendisi saklanmaz. Bir düğümün hangi karakteri temsil ettiği, ebeveyninin dizisinde hangi indiste durduğundan anlaşılır. Kökün `cocuk[0]` hücresindeki düğüm "a"dır, `cocuk[3]` hücresindeki düğüm "d"dir. Karakteri ayrıca saklamak bellek harcamaktan başka bir şey kazandırmaz. Ancak bazı türevlerde, örneğin üçlü arama trie'sinde, karakter düğümün içinde tutulmak zorundadır.

Trie'nin kendisi, kökü tutan bir sınıftır. Notta verilen bütün metotlar bu sınıfın içinde yazılmış sayılır ve `kok` alanını kullanır:

```
public class Trie {
  private final TrieDugumu kok = new TrieDugumu();
  // ekle, ara, onekVarMi, tamamla, sil ... bu sınıfın metotlarıdır
}
```



Şekil 5.3.6: and, ant, dad ve do eklendikten sonra kökün `cocuk` dizisi. 26 hücreden yalnız 0 (a) ve 3 (d) doludur; / işareti `null`'dır.

Şekil 5.3.6 kökün dizisini açık hâliyle gösterir. 26 hücrenin yalnız ikisi doludur. Bu tablo trie'nin bellek sorununun ilk işaretidir. Kökün altındaki düğümlerde durum daha da seyrektir: "an" düğümünde 2, "a" düğümünde yalnız 1 hücre doludur. Yapraklarda dizinin 26 hücresinin hepsi boştur, ama dizi yine de ayrılmıştır.

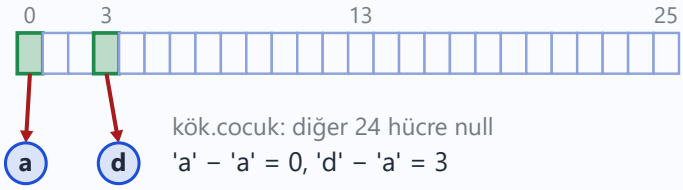
5.3.2.2 Karakterden indise

Bir karakterin dizideki indisini bulmak için Java'nın `char` türünün bir sayı olduğu kullanılır. `char`, karakterin UTF-16 kod değerini saklar ve aritmetik işlemlerde `int`'e dönüşür. Küçük harflerin kodları ardışıktır: 'a' 97'dir, 'z' 122'dir. Bu yüzden `ch - 'a'` ifadesi 'a'yı 0'a, 'z'yi 25'e eşler.

`ch`	**Unicode**	`ch - 'a'`
'a'	97	0
'd'	100	3
'n'	110	13

`ch`	Unicode	`ch - 'a'`
'o'	111	14
't'	116	19
'z'	122	25

Tablo 5.3.2: Bazı karakterlerin kodları ve dizideki indisleri.



Şekil 5.3.7: `kok.cocuk` dizisinde 'a' − 'a' = 0 ve 'd' − 'a' = 3 hücreleri doludur; diğer 24 hücre `null`'dır.

Çocuğa erişim bir dizi indisiyle yapıldığı için *O(1)*'dir: `dugum.cocuk[ch - 'a']`. "and" kelimesi için sırasıyla `cocuk[0]`, `cocuk[13]` ve `cocuk[3]` kullanılır. Bu hız, dizi gösteriminin en büyük üstünlüğüdür. Çocuk bulmak için ne karşılaştırma ne de hash hesabı gerekir.

Aynı formül alfabe dışındaki karakterlerde çalışmaz. Büyük 'A' harfinin kodu 65'tir ve `'A' - 'a'` = *65 − 97 = −32* olur. Türkçe 'ç' harfinin kodu 231'dir: `'ç' - 'a'` = *231 − 97 = 134*. 'ş' için sonuç *351 − 97 = 254*'tür. Bu indislerin hiçbiri 0–25 aralığında değildir ve dizi erişimi `ArrayIndexOutOfBoundsException` fırlatır. Çözüm iki türlüdür: girdi önceden küçük harfe ve a–z alfabesine dönüştürülür ya da çocuklar karakterle anahtarlanan bir `HashMap`'te tutulur.

Sık yapılan hata

Girdiyi denetlemeden `ch - 'a'` kullanmak. "Ankara" ya da "çay" gibi bir kelime ilk karakterde istisnaya yol açar. Daha sinsi olanı, ekleme yarıda kaldığında trie'de yarım bir yolun kalmasıdır: "kaç" eklenirken 'k' ve 'a' düğümleri oluşur, 'ç'de istisna fırlar. Trie'de işaretsiz bir "ka" yolu kalır. Girdi işlenmeden önce denetlenmeli, ya da alfabe baştan geniş seçilmelidir.

5.3.3 Ekleme

5.3.3.1 Algoritma ve kod

Bir kelimeyi trie'ye eklemek, kelimenin yolunu kökten başlayarak yürümek ve eksik kalan kısmı tamamlamaktır. Yolun başı başka kelimelerle paylaşılıyorsa o kısım zaten vardır; ekleme ilk eksik karakterden sonra yeni düğümler açar. Yol bitince son düğüm işaretlenir.

1. **Kökten** başlanır.
2. Kelimenin her karakteri için ilgili çocuğa bakılır: `cocuk[ch - 'a']`.
3. Çocuk `null` ise **yeni düğüm** oluşturulur; varsa (ortak önek) mevcut düğüm kullanılır.
4. O çocuğa ilerlenir.
5. Karakterler bitince son düğümde `kelimeSonu = true` yapılır.

```
public void ekle(String kelime) {
  TrieDugumu dugum = kok;
  for (char ch : kelime.toCharArray()) {
    int i = ch - 'a';
    if (dugum.cocuk[i] == null)
      dugum.cocuk[i] = new TrieDugumu();
    dugum = dugum.cocuk[i];
  }
  dugum.kelimeSonu = true;
}
```

Kod bu beş adımı satır satır izler. `dugum` değişkeni kökle başlar ve döngünün her turunda bir düzey aşağı iner. `toCharArray()` kelimeyi karakter dizisine çevirir; for-each döngüsü karakterleri soldan sağa verir. Her karakter için `i = ch - 'a'` hesaplanır. `dugum.cocuk[i] == null` ise o karakterle devam eden bir yol henüz yoktur; yeni bir `TrieDugumu` oluşturulur ve hücreye bağlanır. Koşul yanlışsa yol zaten vardır ve hiçbir şey oluşturulmaz. İki durumda da `dugum = dugum.cocuk[i]` ile çocuğa inilir. Döngü bittiğinde `dugum`, kelimenin son karakterinin düğümüdür ve işaretlenir.

Kodun bir özelliği dikkat çeker: ekleme hiçbir zaman var olan bir düğümü değiştirmez, yalnız yeni düğüm bağlar ve son düğümün işaretini `true` yapar. Aynı kelime ikinci kez eklenirse döngü bütün düğümleri hazır bulur, yeni düğüm açmaz ve zaten `true` olan işareti yeniden `true` yapar. Yapı değişmez. Trie bu yüzden bir küme gibi davranır: tekrar eden kelimeler bir kez saklanır.

5.3.3.2 Adım adım: and, ant, dad, do

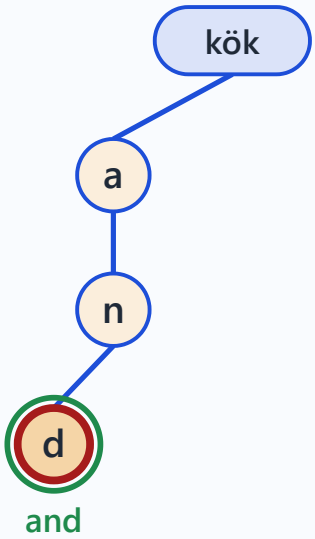
Boş trie'ye sırayla and, ant, dad ve do eklensin. Başlangıçta trie yalnız kökten oluşur: kökün 26 çocuğunun hepsi `null`'dır ve `kelimeSonu = false`'tur. Tablo 5.3.3, her karakterde ilgili hücrenin durumunu ve yapılan işi verir.

Kelime	`ch`	`i`	`cocuk[i]`	İşlem	Düğüm
and	'a'	0	`null`	yeni düğüm	a
	'n'	13	`null`	yeni düğüm	an
	'd'	3	`null`	yeni düğüm	and
	—	—	—	`kelimeSonu = true`	and ✓

Kelime	`ch`	`i`	`cocuk[i]`	İşlem	Düğüm
ant	'a'	0	dolu	ilerlenir (ortak önek)	a
	'n'	13	dolu	ilerlenir (ortak önek)	an
	't'	19	`null`	yeni düğüm	ant
	—	—	—	`kelimeSonu = true`	ant ✓
dad	'd'	3	`null`	yeni düğüm	d
	'a'	0	`null`	yeni düğüm	da
	'd'	3	`null`	yeni düğüm	dad
	—	—	—	`kelimeSonu = true`	dad ✓
do	'd'	3	dolu	ilerlenir (ortak önek)	d
	'o'	14	`null`	yeni düğüm	do
	—	—	—	`kelimeSonu = true`	do ✓

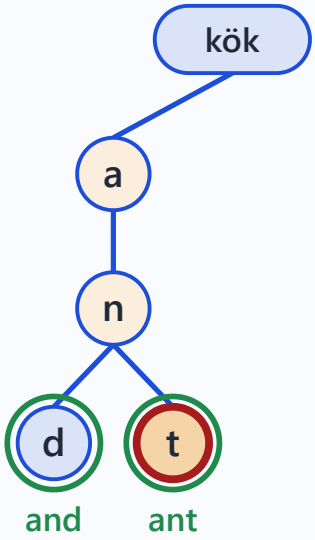
Tablo 5.3.3: and, ant, dad, do kelimelerinin sırayla eklenmesi.
"Düğüm" sütunu işlemden sonra `dugum`'un gösterdiği öneki verir.

"and" eklenirken trie boştur; üç karakterin üçü de yeni düğüm açar ve son düğüm işaretlenir. Kökten itibaren `cocuk[0] → cocuk[13] → cocuk[3]` yolu kurulmuştur.



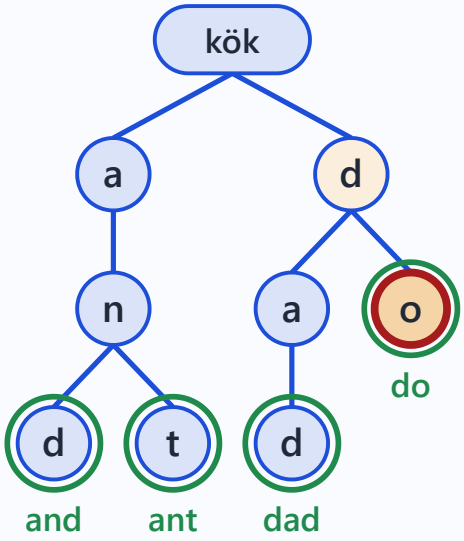
Şekil 5.3.8: "and" eklendikten sonra trie: kök, a, an ve işaretli and düğümü.

"ant" eklenirken ilk iki karakter hazır bir yol bulur. 'a' için `cocuk[0]`, 'n' için `cocuk[13]` doludur; döngü yalnız ilerler. 't' için `cocuk[19]` boştur ve tek bir yeni düğüm açılır. "an" öneki iki kelime arasında paylaşılmıştır. Bu paylaşım trie'nin bellek kazancının kaynağıdır.



Şekil 5.3.9: “ant” eklendi. Yalnız 1 yeni düğüm (t) gerekti; “an” öneki paylaşıldı.

“dad” kökte boş bir hücreye (`cocuk[3]`) rastlar ve üç yeni düğümle kendi dalını kurar. Dikkat edilirse “dad” içindeki iki 'd' farklı düğümlerdir: biri kökün çocuğu, öbürü “da”nın çocuğudur. Aynı karakter farklı öneklerde farklı düğümlere karşılık gelir. Son olarak “do”, 'd' düğümünü paylaşır ve 'o' için tek bir yeni düğüm açar.



Şekil 5.3.10: Son durum: 4 kelime, 11 karakter, kök hariç 8 düğüm.

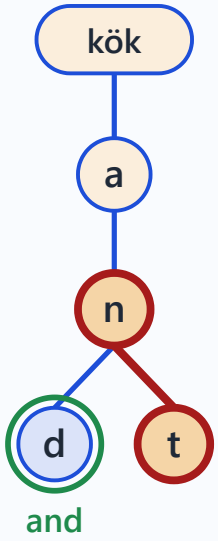
Dört eklemede açılan yeni düğümlerin sayısı sırasıyla 3, 1, 3 ve 1'dir; toplam 8. Kelimelerin toplam uzunluğu 11 olduğuna göre üç karakterlik bir paylaşım sağlanmıştır. Kelimeler başka bir sırayla eklenseydi ara durumlar değişirdi, ama son trie aynı olurdu. Trie'nin şekli yalnız kelime kümesine bağlıdır, ekleme sırasına bağlı değildir. Bu özellik ikili arama ağacından önemli bir farktır: orada ekleme sırası ağacın yüksekliğini belirler.

5.3.3.3 Kod izleme: ekle("ant")

Kodun satır düzeyinde nasıl çalıştığını görmek için trie'de yalnız "and" varken `ekle("ant")` çağrısı izlensin. Tablo 5.3.4 her adımda değişkenlerin değerini verir.

Adım	Satır	`ch`	`i`	Koşul / işlem	`dugum`
1	`dugum = kok`	—	—	başlangıç	kök
2	döngü	'a'	0	`cocuk[0] == null` yanlış; yeni düğüm yok	a
3	döngü	'n'	13	`cocuk[13] == null` yanlış	an
4	döngü	't'	19	`cocuk[19] == null` doğru → yeni düğüm bağlanır	ant
5	döngü sonrası	—	—	`dugum.kelimeSonu = true`	ant ✓

Tablo 5.3.4: `ekle("ant")` izlemesi; başlangıçta trie'de yalnız "and" vardır.



Şekil 5.3.11: Dördüncü adım: `cocuk[19]` boş bulundu ve yeni "t" düğümü (renkli) bağlandı; `dugum` henüz "an"dadır.

İzleme üç noktayı açıkça gösterir. Birincisi, koşul yalnız bir kez doğru olur; yani yalnız bir düğüm oluşturulur. İkincisi, yeni düğüm oluşturulduğu anda `kelimeSonu` değeri `false`'tur; işaret ancak döngü bittikten sonra ve yalnız son düğüme konur. Ara düğümler ("a", "an") işaretsiz kalır. Üçüncüsü, döngü her zaman kelimenin bütün karakterleri üzerinde gezinir; ekleme erken bitmez. Bu yüzden ekleme her durumda tam L tur sürer.

5.3.3.4 Eklemenin maliyeti

Döngü kelimenin her karakteri için bir kez döner. Her turda bir indis hesabı, bir dizi erişimi, gerekirse bir düğüm oluşturma ve bir atama yapılır. Düğüm oluşturmak 26 hücrelik bir dizi ayırmak demektir. Bu iş alfabe boyu kadar sürer, ama alfabe sabit kabul edildiği için sabit bir maliyettir. Toplam süre $O(L)$'dir; L kelimenin uzunluğudur. Süre trie'deki kelime sayısından tamamen bağımsızdır.

Bellek açısından bir ekleme en çok L yeni düğüm açar; bu en kötü durum, kelimenin hiçbir öneki trie'de yokken ortaya çıkar. Her düğüm 26 referans ve bir mantıksal değer taşır, yani bir ekleme en çok $O(L \cdot M)$ bellek ayırır. Kelime tamamen var olan bir yolun üzerindeyse (örneğin "kalem" varken "ka" eklenirse) hiç düğüm açılmaz; yalnız işaret değişir.

Örnek 5.3.2

Soru. Trie'de yalnız "kalem" varken `ekle("ka")` çağrılıyor. Kaç düğüm oluşur? Ardından `ara("ka")` ne döndürür?

Çözüm. 'k' için kökün `cocuk[10]` hücresi doludur; döngü ilerler. 'a' için "k" düğümünün `cocuk[0]` hücresi de doludur. Döngü biter ve "ka" düğümü işaretlenir. Hiç düğüm oluşmaz. Önceden "ka" yolu vardı ama kelime değildi; şimdi işaretlidir ve `ara("ka")` `true` döndürür. "kalem" bu işlemden etkilenmez.

Sık yapılan hata

İşaretlemeyi döngünün içine koymak. `dugum.kelimeSonu = true` satırı döngü gövdesine taşınırsa "and" eklenirken "a" ve "an" de işaretlenir. Sonra `ara("an")` `true` döndürür ve trie, eklenmemiş kelimeleri "bulur". İşaret yalnız son düğüme konmalıdır.

5.3.4 Arama ve önek sorguları

5.3.4.1 Arama algoritması

Arama, eklemenin okuma yapan karşılığıdır. Aynı yol yürünür; fark, eksik bir dalla karşılaşıldığında düğüm oluşturulmaması, aramanın hemen bitirilmesidir.

1. Kökten başlanır.
2. Her karakter için ilgili dala (çocuğa) gidilir.
3. Dal yoksa (`null`) kelime trie'de yoktur.
4. Kelimenin sonuna gelince `kelimeSonu` denetlenir: `true` ise kelime bulunmuştur; `false` ise kelime yoktur, yalnız var olan bir kelimenin önekidir.

```
public boolean ara(String kelime) {
  TrieDugumu dugum = kok;
  for (char ch : kelime.toCharArray()) {
    int i = ch - 'a';
    if (dugum.cocuk[i] == null)
      return false;
    dugum = dugum.cocuk[i];
  }
  return dugum.kelimeSonu;
}
```

Kod eklemeyle neredeyse aynıdır. Farklı olan iki satırdır. Birincisi: çocuk `null` ise yeni düğüm oluşturulmaz, `return false` ile çıkılır. İkincisi: döngü bittikten sonra işaret konmaz, okunur: `return dugum.kelimeSonu`. Bu son satır aramanın en önemli satırıdır. Yolun bütün karakterleri bulunmuş olsa bile, son düğüm işaretli değilse kelime trie'de yoktur.

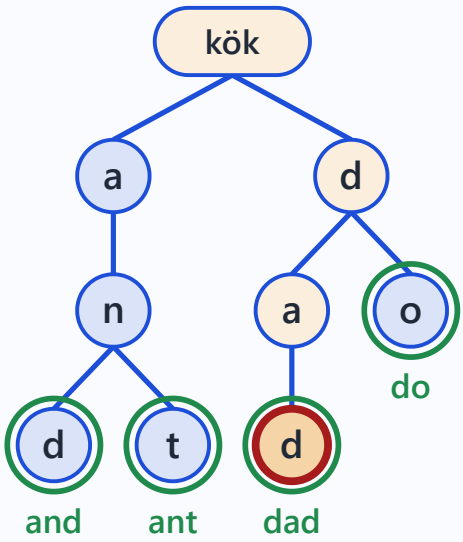
Arama da *O(L)* sürer. Kelime trie'de yoksa döngü daha erken bitebilir: yol *k*. karakterde kesiliyorsa yalnız *k* tur döner. Trie'de bir milyon kelime de olsa "dad" en çok 3 adımda bulunur ya da bulunamaz.

5.3.4.2 Kod izleme: ara("dad")

Trie and, ant, dad ve do kelimelerinden oluşsun. `ara("dad")` çağrısının izlemesi Tablo 5.3.5'tedir.

Tur	`ch`	`i`	`cocuk[i]`	`dugum` (sonra)
—	—	—	—	kök
1	'd'	3	dolu, devam	d
2	'a'	0	dolu, devam	da
3	'd'	3	dolu, devam	dad
son	kelime bitti; `dugum.kelimeSonu` = `true`			sonuç `true`

Tablo 5.3.5: `ara("dad")` izlemesi; trie: and, ant, dad, do.

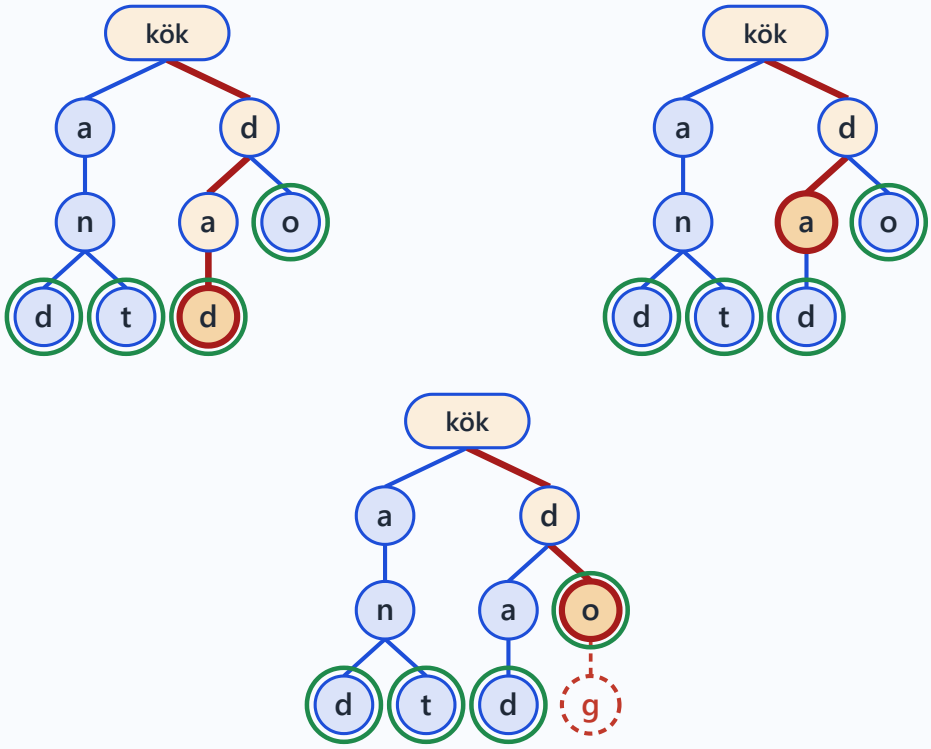


Şekil 5.3.12: `ara("dad")` son adımı: `dugum` "dad" düğümündedir ve düğüm işaretlidir; metot `true` döndürür.

Birinci ve üçüncü turda aynı indis (3) kullanılır, ama farklı düğümlerin dizilerinde: ilki kökün, ikincisi "da"nın dizisidir. Aramanın her turu bir düzey aşağıda çalışır. Üç tur sonunda yol tamamdır ve son düğümün işareti okunur.

5.3.4.3 Aramanın üç sonucu

Bir arama üç farklı biçimde sonuçlanabilir. Üçünü ayırt etmek, arama ile önek sorgusu arasındaki farkı anlamanın da yoludur.



Şekil 5.3.13: Soldan sağa `ara("dad")`, `ara("da")` ve `ara("dog")`. Kesikli kırmızı çember, aranan ama trie'de bulunmayan düğümdür.

- `ara("dad")` → `true`. Bütün karakterler bulunur ve son düğüm işaretlidir.
- `ara("da")` → `false`. Yol vardır, ama "da" düğümünde `kelimeSonu = false`'tur. "da" yalnız "dad"in önekidir. Döngü iki turu tamamlar ve son satır `false` döndürür.
- `ara("dog")` → `false`. "do" düğümünün `cocuk[6]` hücresi ('g' − 'a' = 6) `null`'dır. Dal yoktur; döngü üçüncü turda erken çıkar. "do" bir kelime olsa bile bu sonucu değiştirmez.

İkinci ve üçüncü durum aynı değeri (`false`) döndürür, ama nedenleri farklıdır. İkincisinde yol tamdır, eksik olan işarettir. Üçüncüsünde yol kesilmiştir. Bu fark önek sorgusunda belirleyici olur: önek sorgusu ikinci durumda `true`, üçüncüsünde `false` döndürür.

5.3.4.4 Önek sorgusu

Önek sorgusu (*prefix query*), trie'de verilen önekle başlayan en az bir kelime olup olmadığını sorar. Java'nın `String` sınıfındaki `startsWith` metodunun bir küme üzerindeki karşılığıdır. Trie'de bu soru, önekin yolunun var olup olmadığı sorusuyla aynıdır. Trie'deki her düğüm, en az bir kelimenin yolu üzerindedir; çünkü düğümler yalnız ekleme sırasında açılır ve silme işaretsiz yaprak bırakmaz. Bu yüzden yol varsa o önekle başlayan bir kelime de vardır.

```
public boolean onekVarMi(String onek) {
  TrieDugumu dugum = kok;
  for (char ch : onek.toCharArray()) {
    int i = ch - 'a';
    if (dugum.cocuk[i] == null)
      return false;
    dugum = dugum.cocuk[i];
  }
  return true;  // ara: dugum.kelimeSonu
}
```

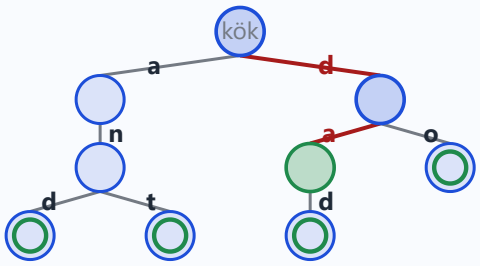
Kod `ara` ile aynıdır; tek fark son satırdır. `onekVarMi` döngüyü yolu kesilmeden bitirirse `true` döndürür ve son düğümün işaretine bakmaz. Yorum satırı farkı açıkça yazar: `ara` bu noktada `dugum.kelimeSonu` döndürür. Süre *O(P)*'dir; *P* önekin uzunluğudur.

Sorgu	`ara`	`onekVarMi`	Açıklama
"dad"	`true`	`true`	yol tam, son düğüm işaretli
"da"	`false`	`true`	yol tam, son düğüm işaretsiz
"an"	`false`	`true`	yol tam, son düğüm işaretsiz
"dog"	`false`	`false`	“do”dan sonra 'g' dalı yok

Tablo 5.3.6: `ara` ve `onekVarMi` sonuçları; trie: and, ant, dad, do.

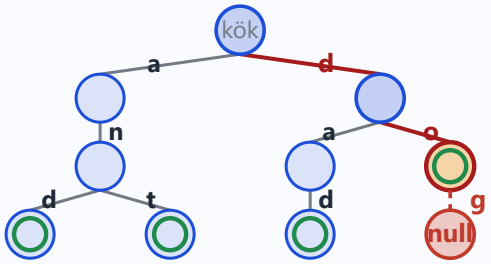
Tablodaki sonuçlar iki metodun tek farkını gösterir: işaretsiz bir düğümde biten sorgularda `ara` `false`, `onekVarMi` `true` döndürür. Yolun kesildiği sorgularda ikisi de `false` döndürür.

Kod izlemesi için iki çağrı alınsın: `onekVarMi("da")` ve `onekVarMi("dog")`. İlk çağrıda `dugum` köke konur. 'd' için `i = 3`, `cocuk[3]` doludur ve “d” düğümüne inilir. 'a' için `i = 0`, `cocuk[0]` doludur ve “da”ya inilir. Önek bitmiştir ve yol kesilmemiştir; metot `true` döndürür. “da” düğümü işaretsiz olduğu için `ara("da")` burada `false` verirdi.



Şekil 5.3.14: `onekVarMi("da")` : yol kesilmeden “da” düğümüne inildi; sonuç `true`.

İkinci çağrıda `dugum` yeniden köke konur. 'd' (`i = 3`) ve 'o' (`i = 14`) için hücreler doludur; “do” düğümüne inilir. 'g' için `i = 6` 'dır ve “do” düğümünün `cocuk[6]` hücresi `null` 'dır. Metot `false` döndürür. “do” bir kelimedir, ama “dog” ile başlayan hiçbir kelime yoktur.



Şekil 5.3.15: `onekVarMi("dog")` : “do” düğümünün 'g' dalı yoktur; sonuç `false` .

Sık yapılan hata

`ara` ile `onekVarMi` 'yi karıştırmak. `ara` 'nın son satırına `return true` yazılırsa metot sessizce önek sorgusuna dönüşür. Dal bulunamayınca yine `false` döner, ama yolu olan her önek kelime sayılır. Bu hata küçük testlerde görünmeyebilir: “dad”, “and” gibi gerçek kelimelerle yapılan testler doğru sonuç verir. Hata ancak “da” gibi işaretsiz bir önekle sınanınca ortaya çıkar. Arama testlerine her zaman en az bir işaretsiz önek konmalıdır.

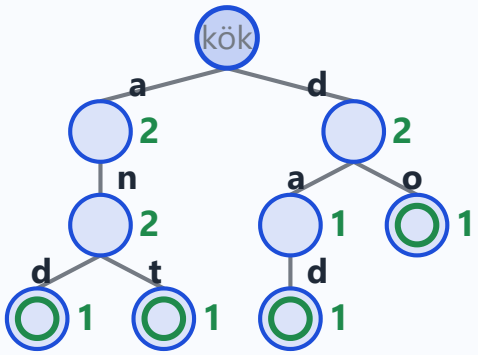
5.3.4.5 Önek sayacı

`onekVarMi` bir önekle başlayan kelime olup olmadığını söyler, ama kaç tane olduğunu söylemez. “an ile başlayan kaç kelime var?” sorusu, önek düğümünün alt ağacındaki işaretli düğümleri saymayı gerektirir. Bu sayım alt ağacın bütün düğümlerinde gezinir. Alt ağaçta S düğüm varsa ve her düğümde 26 hücre denetleniyorsa sorgu $O(P + S \cdot M)$ sürer. Sözlük büyükse ve önek kısaysa (örneğin tek harf) bu, sözlüğün büyük bir bölümünü gezmek demektir.

Daha iyi bir yol, sayımı ekleme sırasında yapmaktır. `TrieDugumu` sınıfına bir `int gecen` alanı eklenir. Bu alan, düğümün yolundan geçen kelime sayısını tutar. Ekleme kökten başlayıp her geçtiği düğümün sayacını bir artırır. Sorgu sırasında önek düğümüne inilir ve sayaç okunur.

```
void ekle(String kelime) {
  TrieDugumu d = kok;
  d.gecen++;
  for (char ch : kelime.toCharArray()) {
    int i = ch - 'a';
    if (d.cocuk[i] == null) d.cocuk[i] = new TrieDugumu();
    d = d.cocuk[i];
    d.gecen++;          // bir kelime daha geçti
  }
  d.kelimeSonu = true;
}
int onekSayisi(String onek) {
  TrieDugumu d = kok;
  for (char ch : onek.toCharArray()) {
    d = d.cocuk[ch - 'a'];
    if (d == null) return 0;
  }
  return d.gecen;
}
```

Eklemede kökün sayacı döngüden önce artırılır; böylece kökte trie'deki toplam kelime sayısı tutulur. Döngü içindeki `d.gecen++` satırı, yeni oluşturulmuş ya da var olan her düğümün sayacını artırır. `onekSayisi` metodunda yol kesilirse 0 döner; yol bitince önek düğümünün sayacı döner. Döngü önce çocuğa iner, sonra `null` denetimi yapar; bu sıra `ara`'dakinden farklıdır ama aynı işi görür.



Şekil 5.3.16: and, ant, dad, do trie'sinde `gecen` sayaçları (yeşil). Kökte 4, “a” ve “an”de 2, “d”de 2, “da” ve “do”da 1'dir.

Sorgu	Sonuç	Gerekçe
`onekSayisi("an")`	2	and ve ant “an” düğümünden geçer
`onekSayisi("da")`	1	yalnız dad geçer
`onekSayisi("x")`	0	kökün `cocuk[23]` hücresi boş

Tablo 5.3.7: `onekSayisi` sorguları; trie: and, ant, dad, do.

Sayaç yöntemi sorguyu *O(P)*'ye indirir. Bedeli düğüm başına bir `int`'tir; 26 referanslık bir düğümde bu ek bellek çok küçüktür. Silme işleminde ise yol boyunca bütün sayaçlar bir azaltılmalıdır. Azaltma unutulursa sayaçlar silinmiş kelimeleri saymaya devam eder. Ayrıca aynı kelime iki kez eklenirse sayaçlar iki kez artar; kümede tekrar istenmiyorsa ekleme öncesinde `ara` ile kelimenin varlığı denetlenmelidir.

> **Sık yapılan hata**
>
> Sayacı yalnız kelime sonunda artırmak. `d.gecen++` satırı döngüden çıkarılıp yalnız son düğümde çalıştırılırsa sayaç, "bu düğümde biten kelime sayısı"na dönüşür. "an" düğümünde değer 0 olur ve `onekSayisi("an")` yanlış sonuç verir. Sayaç yolun her düğümünde artırılmalıdır.

5.3.4.6 Joker karakterli arama

Bulmaca çözen ya da kelime oyunu oynayan bir program şu soruyu sorabilir: "d, bilinmeyen bir harf, d" biçiminde bir kelime var mı? Desende nokta (`.`) herhangi bir harf yerine geçer. Bu tür sorguya **joker karakterli arama** (*wildcard search*) denir. Normal harf aramayı tek bir yola indirir. Nokta ise o düzeydeki her dolu dalın denenmesini gerektirir. Arama bu yüzden özyinelemeli yazılır.

```
boolean joker(TrieDugumu d, String s, int j) {
  if (d == null) return false;
  if (j == s.length()) return d.kelimeSonu;
  char ch = s.charAt(j);
  if (ch != '.')
    return joker(d.cocuk[ch - 'a'], s, j + 1);
  for (int i = 0; i < 26; i++)
    if (joker(d.cocuk[i], s, j + 1)) return true;
  return false;
}
```

Metot üç parametre alır: şu anki düğüm `d`, desen `s` ve desende sıradaki karakterin indisi `j`. İlk çağrı `joker(kok, desen, 0)` biçimindedir. Düğüm `null` ise bu dal yoktur ve `false` döner. Desen bitmişse (`j == s.length()`) sonuç düğümün işaretidir; tıpkı `ara`'nın son satırında olduğu gibi. Sıradaki karakter bir harfse yalnız o harfin çocuğuna inilir. Nokta ise 26 çocuğun her biri sırayla denenir; birinde eşleşme bulunursa `true` hemen yukarı taşınır ve kalan dallar denenmez. Boş bir çocuk için yapılan çağrı ilk satırda hemen döner.

Desen	Sonuç	Eşleşen / gerekçe
"d.d"	`true`	dad
".n."	`true`	and (ant'a hiç bakılmaz)
".o"	`true`	do

Desen	Sonuç	Eşleşen / gerekçe
"b.."	`false`	kökün 'b' dalı yok
"...."	`false`	4 harfli kelime yok

Tablo 5.3.8: Joker karakterli arama sonuçları; trie: and, ant, dad, do.

Örnek 5.3.3

Soru. `joker(kok, "d.d", 0)` çağrısı hangi düğümlerden geçer?

Çözüm. `j = 0`, karakter 'd' bir harftir; yalnız `kok.cocuk[3]` 'e, yani "d" düğümüne inilir. `j = 1`, karakter noktadır; "d"nin 26 çocuğu sırayla denenir. `i = 0` için "da" düğümü vardır ve `joker(da, "d.d", 2)` çağrılır. Bu çağrıda karakter 'd'dir ve "dad" düğümüne inilir. `j = 3` desenin uzunluğuna eşittir ve "dad" işaretli olduğu için `true` döner. Değer yukarı taşınır; "d" düğümündeki döngü `i = 1` 'e hiç geçmez. "do" dalı (`i = 14`) denenmez.

Maliyet desene ve trie'ye bağlıdır. Desende k nokta varsa en kötü durumda 26^k dal denenir. Ancak boş dallar hemen döndüğü için gerçek maliyet, trie'de gerçekten var olan yollarla sınırlıdır. "...." deseni bütün trie'yi en çok dördüncü düzeye kadar gezer ve 4 harfli kelime bulamaz. Hash tablosu bu sorguyu ancak bütün kelimeleri tek tek deneyerek cevaplayabilir: deseni bilinen bir anahtara çevirmenin yolu yoktur.

5.3.4.7 Yazım denetimi: trie üzerinde düzenleme uzaklığı

Yazım denetleyicisi sözlükte bulunmayan bir kelimeye öneriler sunar. Öneriler, kelimeye "yakın" sözlük kelimeleridir. Yakınlığın ölçüsü olarak çoğunlukla **düzenleme uzaklığı** (*edit distance*) kullanılır: bir dizgiyi ötekine çevirmek için gereken en az tek karakterlik ekleme, silme ve değiştirme sayısı. Bu ölçünün öteki adı Levenshtein uzaklığıdır. "dod" ile "dad" arasındaki uzaklık 1'dir (o → a). "dod" ile "do" arasındaki uzaklık da 1'dir (son d silinir).

İki dizgi arasındaki uzaklık dinamik programlamayla bir tablo doldurularak hesaplanır. Sorgu dizgisi q, sözlük kelimesi w olsun. Tablonun i. satırı, w'nun ilk i karakteri ile q'nun bütün önekleri arasındaki uzaklıkları tutar. Satır, bir önceki satırdan şu kuralla hesaplanır:

$$S_i[0] = i,\quad S_i[j] = \min(\, S_i[j-1] + 1,\ \ S_{i-1}[j] + 1,\ \ S_{i-1}[j-1] + (w_i \neq q_j\ ?\ 1 : 0)\,)$$

Üç terim sırasıyla ekleme, silme ve değiştirmenin (ya da karakterler eşitse ücretsiz geçişin) maliyetidir. Son satırın son hücresi iki dizgi arasındaki uzaklıktır.

Saf yöntem, sözlükteki her kelime için ayrı bir tablo doldurur. Ama tablonun i. satırı yalnız kelimenin ilk i karakterine bağlıdır. Ortak öneki olan kelimeler bu satırları paylaşır. Trie tam olarak bu paylaşımı sağlar: her düğüm bir önektir ve düğümün satırı ebeveyninin satırından tek adımda hesaplanır. Trie

derinlik öncelikli gezilir; her düğümde bir satır üretilir. “d” düğümünün satırı bir kez hesaplanır ve hem “dad” hem “do” için kullanılır.

Gezinmeye bir de budama eklenir. Bir satırın en küçük değeri *k*'yi aşıyorsa, bu düğümün altındaki hiçbir kelime sorguya *k* uzaklıkta olamaz. Yeni satırın her hücresi, doğrudan ya da soldaki komşusu üzerinden, önceki satırın bir hücresine 0 ya da daha fazla eklenerek elde edilir. Bu yüzden satırın en küçük değeri derinlik arttıkça azalmaz. Böyle bir düğümün alt ağacına inilmez.

Düğüm	“” d o d	En küçük	Son	Karar
(kök)	0 1 2 3	0	—	başlangıç satırı
a	1 1 2 3	1	—	devam
an	2 2 2 3	2	—	buda (en küçük 2 > 1)
d	1 0 1 2	0	—	devam
da	2 1 1 2	1	—	devam
dad	3 2 2 1	1	1	öner
do	2 1 0 1	0	1	öner

Tablo 5.3.9: Sorgu “dod”, *k = 1*; trie: and, ant, dad, do. Satır sütunları sırasıyla “”, d, o, d önekleridir. “Son” sütunu yalnız kelime düğümlerinde anlamlıdır.

Örnek 5.3.4

Soru. “a” ve “dad” satırları kökün satırından başlayarak nasıl hesaplanır?

Çözüm. Kökün satırı boş önekle “dod”un önekleri arasındaki uzaklıklardır: 0, 1, 2, 3. “a” satırında ilk hücre 1'dir. İkinci hücre *min(1 + 1, 1 + 1, 0 + [a ≠ d]) = min(2, 2, 1) = 1*. Üçüncü hücre *min(1 + 1, 2 + 1, 1 + [a ≠ o]) = 2*. Dördüncü hücre *min(2 + 1, 3 + 1, 2 + [a ≠ d]) = 3*. Satır 1, 1, 2, 3'tür. “dad” satırı “da” satırından (2, 1, 1, 2) hesaplanır: ilk hücre 3; sonra *min(4, 2, 2 + 0) = 2*; *min(3, 2, 1 + 1) = 2*; *min(3, 3, 1 + 0) = 1*. Satır 3, 2, 2, 1'dir ve son hücre 1, yani “dad” ile “dod” arasındaki uzaklıktır.

Gezinti sonunda öneriler dad (o → a) ve do (son d silinir) olur. “an” satırının en küçüğü 2'dir ve 1'i aşar; “and” ve “ant” düğümlerine hiç inilmez. Gerçekten de “and” ile “dod” arasındaki uzaklık 2, “ant” ile 3'tür. Ziyaret edilen düğümler a, an, d, da, dad ve do'dur. Her satır *O(|q|)* iş tutar. Budama sayesinde büyük bir sözlüğün çoğu hiç ziyaret edilmez. Aynı gezinti, klavyede yan yana harfleri daha ucuz sayan ağırlıklı bir uzaklıkla da çalışır. Ancak *k* büyüdükçe budama zayıflar ve ziyaret edilen düğüm sayısı hızla artar; uygulamalarda *k* çoğunlukla 1 ya da 2 seçilir.

5.3.5 Otomatik tamamlama ve en uzun ortak önek

5.3.5.1 tamamla ve topla

Otomatik tamamlama (*autocomplete*), yazılan öneke göre olası kelimeleri önerir. Trie'de bu iş iki aşamada yapılır. Önce önek düğümü bulunur; bu adım önek sorgusuyla aynıdır. Ardından bu düğümün alt ağacı derinlik öncelikli gezilir ve işaretli her düğümün kelimesi listeye eklenir.

```
List<String> tamamla(String onek) {
  List<String> sonuc = new ArrayList<>();
  TrieDugumu d = dugumBul(onek);
  if (d != null)
    topla(d, new StringBuilder(onek), sonuc);
  return sonuc;
}
void topla(TrieDugumu d, StringBuilder yol,
           List<String> sonuc) {
  if (d.kelimeSonu) sonuc.add(yol.toString());
  for (int i = 0; i < 26; i++)
    if (d.cocuk[i] != null) {
      yol.append((char) ('a' + i));
      topla(d.cocuk[i], yol, sonuc);
      yol.setLength(yol.length() - 1);
    }
}
```

`tamamla` önce `dugumBul` yardımcısıyla önek düğümünü arar. Yardımcı, `onekVarMi`'nin `boolean` yerine düğümün kendisini döndüren sürümüdür:

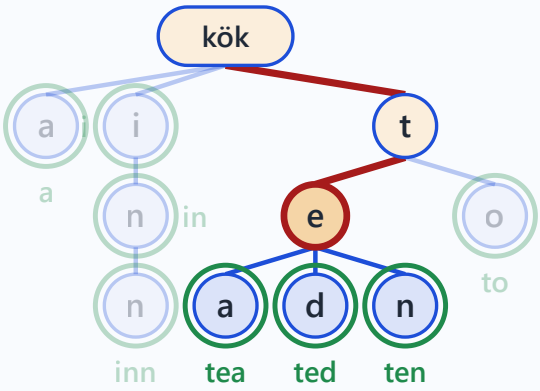
```
TrieDugumu dugumBul(String onek) {
  TrieDugumu d = kok;
  for (char ch : onek.toCharArray()) {
    d = d.cocuk[ch - 'a'];
    if (d == null) return null;  // yol kesildi
  }
  return d;
}
```

Önek düğümü yoksa `dugumBul` `null` döndürür ve `tamamla` boş bir liste verir. Düğüm varsa `topla` çağrılır. `topla`'nın `yol` parametresi, kökten şu anki düğüme kadarki dizgiyi tutan bir `StringBuilder`'dır; başlangıçta önekin kendisidir. Metot önce düğümün işaretine bakar: düğüm bir kelime sonuysa `yol`'un o anki değeri sonuca eklenir. Sonra 26 indis üzerinde sırayla gezinilir. Dolu her çocuk için üç iş yapılır: çocuğun karakteri `yol`'a eklenir, çocuk için `topla` özyinelemeli çağrılır, çağrı dönünce eklenen karakter silinir.

Son adım, `yol.setLength(yol.length() - 1)`, bir **geri izleme** (*backtracking*) adımıdır. Bütün çağrılar aynı `StringBuilder` nesnesini paylaşır. Bir alt ağaçtan dönülürken o alt ağaç için eklenen karakter silinmezse, bir sonraki kardeşin kelimesi yanlış bir önekle kurulur. Her çağrı `yol`'u, kendisine verildiği hâliyle geri bırakır.

5.3.5.2 Kod izleme: tamamla("te")

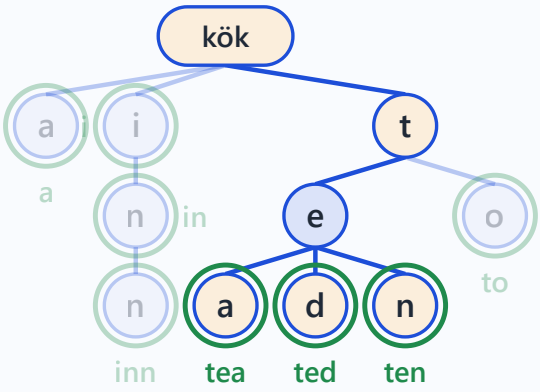
Trie a, to, tea, ted, ten, i, in ve inn kelimelerinden oluşsun. Bu, değer taşıyan trie örneğinin küçük harfli biçimidir. `tamamla("te")` çağrısı izlensin.



Şekil 5.3.17: `dugumBul("te")` kökten t → e inerek "te" düğümünü bulur.

Adım	Çağrı / işlem	`yol`	`sonuc`
1	`dugumBul("te")` → "te" düğümü	te	[]
2	`topla(te)` : işaretsiz, eklenmez	te	[]
3	`i = 0` ('a') dolu → append, `topla(tea)`	tea	[]
4	"tea" işaretli → eklenir; çocuk yok	tea	[tea]
5	dönüş, son harf silinir	te	[tea]
6	`i = 3` ('d') dolu → `topla(ted)`	ted	[tea, ted]
7	dönüş, son harf silinir	te	[tea, ted]
8	`i = 13` ('n') dolu → `topla(ten)`	ten	[tea, ted, ten]
9	dönüş, son harf silinir; döngü biter	te	[tea, ted, ten]

Tablo 5.3.10: `tamamla("te")` izlemesi. Girinti, özyineleme derinliğini gösterir.



Şekil 5.3.18: `tamamla("te")` sonucu: [tea, ted, ten]. Yalnız "te" düğümünün alt ağacı gezildi; soluk düğümlere hiç bakılmadı.

Sonuç `[tea, ted, ten]`'dir ve alfabetik sıradadır. Bu sıra rastlantı değildir: döngü indisler üzerinde 0'dan 25'e gezinir ve indis sırası harf sırasıdır (a < d < n). Ayrıca bir düğüm, çocuklarından önce listeye eklenir; bu yüzden bir önek, onu uzatan kelimelerden önce gelir. Kelimeler trie'ye hangi sırayla eklenmiş olursa olsun çıktı aynıdır.

Maliyet iki parçalıdır. Önek düğümünü bulmak $O(P)$ sürer. Alt ağaçta S düğüm varsa her biri bir kez ziyaret edilir ve her ziyarette 26 hücre denetlenir: $O(S \cdot M)$. Toplam $O(P + S \cdot M)$'dir. Ayrıca her bulunan kelime için `toString()` bir kopya üretir; bu kopyaların toplam uzunluğu çıktının boyudur. Önek kısaysa alt ağaç büyük olabilir. Uygulamalar bu yüzden çoğunlukla ilk birkaç öneriyle yetinir. En sık kullanılan kelimeleri önce göstermek için düğümlerde kullanım sıklığı da saklanır.

Örnek 5.3.5

Soru. Kelimeler sırasıyla bal, ba, bas, bat ve baba olarak eklenmiş. `tamamla("ba")`, `tamamla("bx")` ve `tamamla("bas")` ne döndürür?

Çözüm. "ba" işaretlidir ve listeye ilk olarak eklenir. Ardından çocuklar indis sırasıyla gezilir: b (1), l (11), s (18), t (19). Sonuç `[ba, baba, bal, bas, bat]`'tir. "bx" düğümü yoktur, `dugumBul` `null` döndürür ve sonuç boş listedir. "bas" işaretlidir ve çocuğu yoktur: `[bas]`.

Sık yapılan hata

Geri izlemeyi unutmak. `yol.setLength(...)` satırı silinirse "te" düğümünde 'a', 'd', 'n' karakterleri birikir. Sonuç `[tea, tead, teadn]` olur. Her özyinelemeli çağrıdan sonra paylaşılan durum, çağrıdan önceki hâline getirilmelidir.

5.3.5.3 En uzun ortak önek

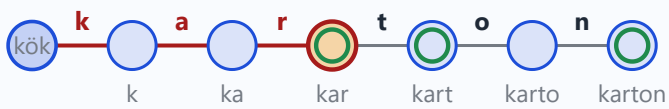
Bir dizgi kümesinin **en uzun ortak öneki** (*longest common prefix*), kümedeki bütün dizgilerin paylaştığı en uzun baş kısımdır. Trie'de bu önek kökten başlayan tek dallı bir zincirdir. Kökten inilir; düğümün tek çocuğu varsa ve düğüm bir kelimenin sonu değilse ilerlenir. Geçilen karakterler cevaptır.

```
String enUzunOrtakOnek() {
  StringBuilder sb = new StringBuilder();
  TrieDugumu d = kok;
  while (!d.kelimeSonu) {
    int tek = -1, sayi = 0;
    for (int i = 0; i < 26; i++)
      if (d.cocuk[i] != null) { sayi++; tek = i; }
    if (sayi != 1) break;  // dal ya da yaprak
    sb.append((char) ('a' + tek));
    d = d.cocuk[tek];
  }
  return sb.toString();
}
```

Döngü koşulu `!d.kelimeSonu`'dur: işaretli bir düğüme gelindiğinde durulur. Döngü gövdesinde 26 hücre sayılır; `sayi` dolu hücre sayısını, `tek` son görülen dolu indisi tutar. `sayi` 1 değilse (0 ise yaprağa, 2 ya da daha fazlaysa dallanmaya gelinmiştir) döngü kırılır. Tek çocuk varsa karakteri eklenir ve çocuğa inilir.

Düğüm `d`	`kelimeSonu`	`sayi`, `tek`	İşlem	`sb`
kök	`false`	1, 10 ('k')	'k' eklenir	k
k	`false`	1, 0 ('a')	'a' eklenir	ka
ka	`false`	1, 17 ('r')	'r' eklenir	kar
kar	`true`	—	döngü biter	kar

Tablo 5.3.11: `enUzunOrtakOnek()` izlemesi; trie: kar, kart, karton.



Şekil 5.3.19: kar, kart, karton trie'si tek bir zincirdir. İniş "kar" düğümünde durur, çünkü "kar" bir kelimedir.

"kar" düğümünün tek çocuğu (t) vardır, ama iniş orada durur. Neden? Ortak önek kümedeki *her* kelimenin öneki olmalıdır. "kar" kelimesinin kendisi "kar"dan uzun bir önek taşıyamaz. Koşulda işaret denetimi olmasaydı iniş "karton"a kadar sürerdi ve yanlış olarak "karton" döndürülürdü. Trie'de yalnız

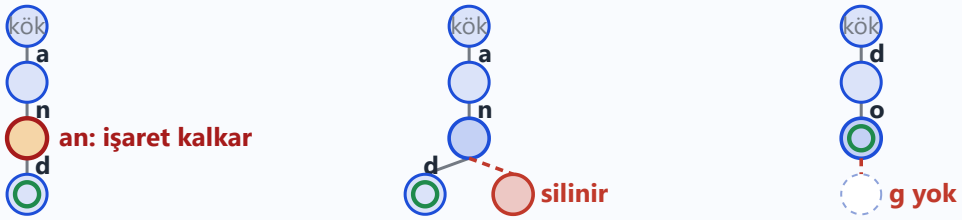
kart ve karton olsaydı iniş “kart”a kadar devam ederdi ve orada `kelimeSonu` durdururdu: cevap “kart” olurdu.

Maliyet, cevabın uzunluğu P ise $O(P \cdot M)$'dir: her düzeyde 26 hücre sayılır. Kökün birden fazla çocuğu varsa döngü ilk turda kırılır ve cevap boş dizgidir; örneğin and, ant, dad, do kümesinin ortak öneki yoktur.

5.3.6 Silme

5.3.6.1 Üç durum

Silme, trie işlemlerinin en dikkat isteyenidir. Bir kelimenin yolu başka kelimelerle paylaşılıyor olabilir. Yolu baştan sona silmek, öneki paylaşan öteki kelimeleri de yok eder. Bu yüzden silme yalnız kelimeye *özgü* düğümleri kaldırmalıdır. Kelimenin trie'deki konumuna göre üç durum ortaya çıkar.



Şekil 5.3.20: Silmenin üç durumu. Solda “an” silinir: yalnız işaret kalkar. Ortada “ant” silinir: t düğümü kaldırılır, “an” önekinde durulur. Sağda “dog” aranır: kelime trie'de yoktur.

1. **Kelime başka bir kelimenin önekidir.** Son düğümün çocuğu vardır. Yalnız `kelimeSonu = false` yapılır; düğüm silinmez, çünkü başka kelimelerin yolu üzerindedir.
2. **Kelimenin kendine ait bir kuyruğu vardır.** Sondan başa doğru, çocuğu kalmayan ve işaretsiz düğümler silinir. Paylaşılan bir öneke ya da başka bir kelimenin sonuna gelince durulur.
3. **Kelime trie'de yoktur.** Dal bulunamaz ya da son düğüm işaretsizdir. Hiçbir şey değişmez.

İkinci durumdaki “kuyruk”, kelimenin son düğümünden yukarı doğru, yalnız bu kelimeye hizmet eden düğümler zinciridir. Bir düğüm iki koşul birlikte sağlanırsa silinebilir: çocuğu yoktur ve işaretsizdir. Çocuğu varsa başka bir kelimenin yolundadır. İşaretliyse kendisi bir kelimedir. Kuyruk, bu iki koşuldan birini bozan ilk düğümde biter.

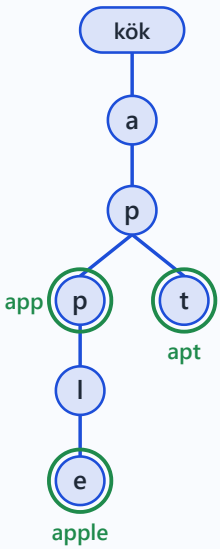
Örnek 5.3.6

Soru. Trie'de “apple” ve “app” var. “app” silinince yapı nasıl değişir?

Çözüm. “app”in son düğümü olan ikinci 'p' düğümünün bir çocuğu vardır: 'l'. “apple” bu düğümün altından devam eder. Düğüm silinemez; yalnız işareti kaldırılır (durum 1). Sonrasında `ara("app")` `false`, `ara("apple")` `true` döndürür. Düğüm sayısı değişmez.

5.3.6.2 Adım adım: app, apple, apt

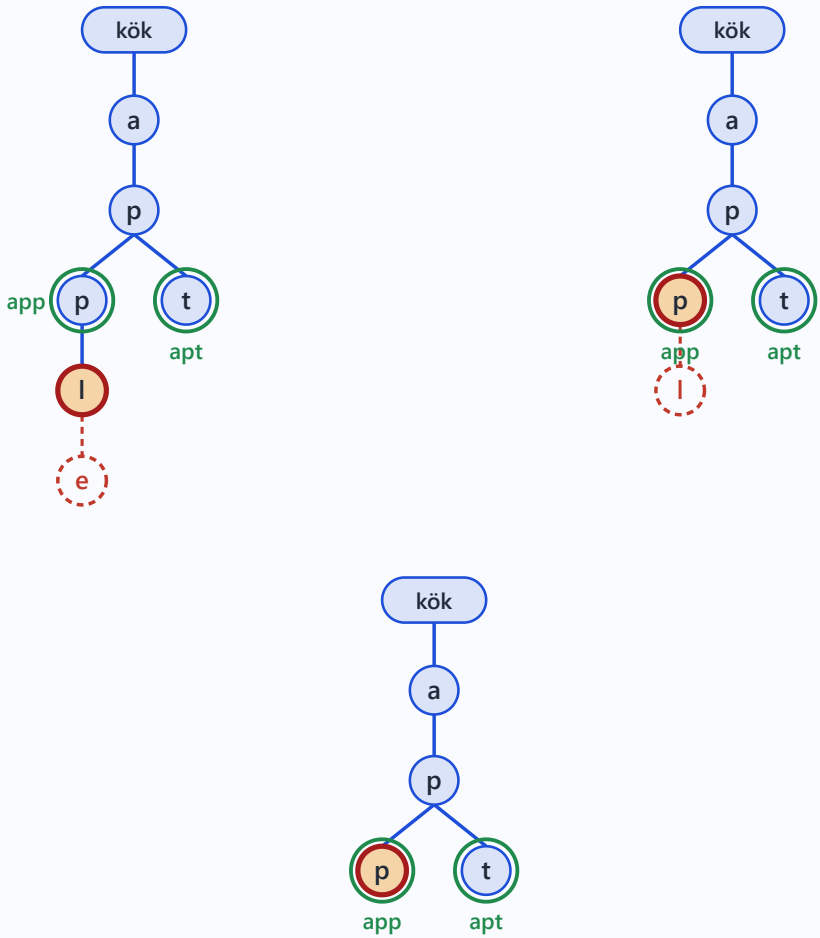
Trie app, apple ve apt kelimelerinden oluşsun. Kök dışında 6 düğüm vardır: a, ap, app, appl, apple, apt. Aşağıdaki iki senaryonun ikisi de bu ilk trie'den başlar.



Şekil 5.3.21: app, apple ve apt kelimelerinin trie'si.

Senaryo 1: `sil("app")`. a → p → p yolu izlenir. “app” düğümü işaretlidir, yani kelime vardır. İşaret kaldırılır: `kelimeSonu = false`. Düğümün 'l' çocuğu olduğu için silinmez. Bu durum 1'dir. “apple” hâlâ bulunur; düğüm sayısı 6 olarak kalır.

Senaryo 2: `sil("apple")`. Yol a-p-p-l-e izlenir ve “apple” bulunur. 'e' düğümünün işareti kalkar. Bu düğümün çocuğu yoktur; ebeveyni onu siler (`cocuk[4] = null`). Sonra 'l' düğümüne bakılır: artık çocuksuz ve işaretsizdir, o da silinir. Ardından “app” düğümüne gelinir. Bu düğüm işaretlidir, çünkü “app” başka bir kelimedir. Silme burada durur.



Şekil 5.3.22: `sil("apple")` : önce 'e', sonra 'l' düğümü silinir; işaretli "app" düğümünde durulur. Sonuç: app ve apt, kök hariç 4 düğüm.

İki senaryo, silme kararlarının *aşağıdan yukarı* verildiğini gösterir. Bir düğümün silinip silinmeyeceği, çocuklarının silinmesine bağlıdır. 'l' düğümü başlangıçta silinemezdi, çünkü 'e' çocuğu vardı. 'e' silindikten sonra 'l' silinebilir hâle geldi. Bu bağımlılık, silmenin doğal olarak özyinelemeyle yazılmasının nedenidir: önce kelimenin sonuna inilir, kararlar dönüş yolunda verilir.

5.3.6.3 Özyinelemeli kod

```
// true: ebeveyn d'yi silebilir
boolean sil(TrieDugumu d, String k, int j) {
  if (j == k.length()) {        // kelime bitti
    if (!d.kelimeSonu) return false;
    d.kelimeSonu = false;
    return bosMu(d);            // çocuksuz mu?
  }
  int i = k.charAt(j) - 'a';
  TrieDugumu c = d.cocuk[i];
  if (c == null) return false;  // kelime yok
  if (sil(c, k, j + 1)) {
    d.cocuk[i] = null;          // çocuğu sil
    return !d.kelimeSonu && bosMu(d);
  }
  return false;
}
```

Metodun dönüş değeri ebeveyne iletilen bir karardır: `true`, ebeveynin bu düğümü silebileceği anlamına gelir. Çağrı `sil(kok, kelime, 0)` biçimindedir; kökün dönüş değeri yok sayılır, çünkü kök hiçbir zaman silinmez. `j`, kelimede sıradaki karakterin indisidir.

Metot iki kısımdan oluşur. İlk kısım taban durumdur: `j == k.length()` ise kelimenin son düğümüne gelinmiştir. Düğüm işaretsizse kelime trie'de yoktur ve `false` döner (durum 3). İşaretliyse işaret kaldırılır ve düğümün çocuğu olup olmadığına bakılır. Çocuğu yoksa `true` döner: ebeveyn bu düğümü silebilir. Çocuğu varsa `false` döner (durum 1).

İkinci kısım özyinelemeli durumdur. Sıradaki karakterin çocuğu `c` bulunur. `c` `null` ise kelime trie'de yoktur ve `false` döner (durum 3). Varsa `sil(c, k, j + 1)` çağrılır. Çağrı `true` döndürürse çocuk silinebilir demektir: `d.cocuk[i] = null` ile bağ koparılır. Ardından aynı soru bu düğüm için sorulur: düğüm işaretsiz ve artık çocuksuz mu? Cevap bir üst düzeye iletilir. Çağrı `false` döndürürse hiçbir şey silinmez ve `false` yukarı taşınır. Bir düzeyde zincir kırılınca üstteki hiçbir düğüm silinmez.

Kod, `bosMu` yardımcısını kullanır. Yardımcı 26 hücrenin hepsinin `null` olup olmadığını denetler:

```
boolean bosMu(TrieDugumu d) {
  for (int i = 0; i < 26; i++)
    if (d.cocuk[i] != null) return false;
  return true;
}
```

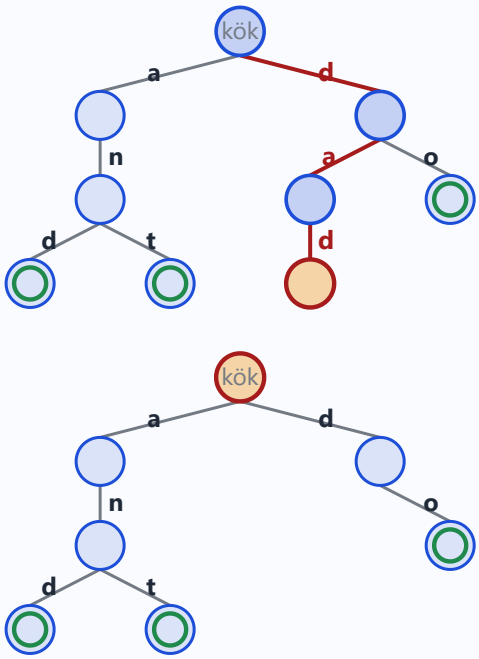
Java'da bir düğümü “silmek”, ona giden son referansı `null` yapmaktır. Kopan düğüm artık erişilebilir değildir ve çöp toplayıcı belleği geri alır. Bu yüzden kod yalnız ebeveyndeki hücreyi boşaltır.

5.3.6.4 Kod izleme: sil("dad")

Trie and, ant, dad ve do kelimelerinden oluşsun. `sil(kok, "dad", 0)` çağrısı izlensin. Özyineleme önce üç düzey iner, sonra dönüş yolunda kararlar verilir. Tablo 5.3.12 iniş ve dönüş adımlarını sırayla verir.

Adım	Çerçeve	`j`	İşlem	Dönüş
1	kök	0	'd' → `i = 3`, `c` = “d”; `sil(d, "dad", 1)`	—
2	d	1	'a' → `i = 0`, `c` = “da”; `sil(da, "dad", 2)`	—
3	da	2	'd' → `i = 3`, `c` = “dad”; `sil(dad, "dad", 3)`	—
4	dad	3	`j == k.length()`; işaretli → `kelimeSonu = false`; `bosMu(dad)` = `true`	`true`
5	da	2	`cocuk[3] = null`; “da” işaretsiz ve çocuksuz	`true`
6	d	1	`cocuk[0] = null`; “d” işaretsiz ama 'o' çocuğu var	`false`
7	kök	0	çağrı `false` döndü; kök bir şey silmez	`false`

Tablo 5.3.12: `sil(kok, "dad", 0)` izlemesi; trie: and, ant, dad, do. Çerçeve, çağrının düğümünü gösterir.



Şekil 5.3.23: Solda dördüncü adım: “dad” düğümünün işareti kaldırıldı. Sağda sonuç: “dad” ve “da” düğümleri koparıldı; kalan kelimeler and, ant, do.

İzlemede iki düğüm silinir: “dad” ve “da”. “d” düğümü işaretsizdir, ama “do” için 'o' çocuğu vardır; `bosMu` `false` verir ve zincir orada kırılır. Kökteki çağrı `false` alır ve kökün dönüş değeri yok sayılır. Kalan trie and, ant ve do kelimelerini içerir; kök dışında 6 düğüm vardır.

Silmenin maliyeti şöyle hesaplanır. İniş L düzey sürer. Dönüşte her düzeyde en çok bir `bosMu` çağrısı yapılır ve her biri 26 hücreyi tarar. Toplam $O(L \cdot M)$'dir. Düğümde dolu çocuk sayısını tutan bir sayaç olsaydı `bosMu` sabit sürede cevap verirdi ve silme $O(L)$'ye inerdi. Bu sayaç, eklemede bir hücre doldurulurken artırılır, silmede bir hücre boşaltılırken azaltılır.

Sık yapılan hata

Silmede ortak düğümü silmek. Silme bazen "kelimenin yolundaki bütün düğümler kökten itibaren `null` yapılır" diye yazılır. Bu sürüm, app, apple ve apt bulunan bir trie'de `sil("app")` çağrısında kökün 'a' hücresini boşaltır. a ile başlayan üç kelimenin üçü de kaybolur. Doğru silme yalnız işareti kaldırır ve aşağıdan yukarı, çocuksuz ve işaretsiz düğümleri siler. "app"in 'l' çocuğu olduğu için hiçbir düğüm silinmez.

Sık yapılan hata

Dönüşte işaret denetimini atlamak. Özyinelemeli kısımdaki `!d.kelimeSonu &&` koşulu silinirse `sil("apple")` çağrısı "app" düğümünü de siler, çünkü 'l' koparıldıktan sonra "app" çocuksuz kalır. Böylece "app" kelimesi kaybolur. Bir düğüm ancak çocuksuz *ve* işaretsizse silinebilir.

5.3.7 Analiz

5.3.7.1 Karmaşıklık

Trie işlemlerinin maliyeti iki büyüklüğe bağlıdır: işlenen dizginin uzunluğu ve alfabenin boyu. L kelimenin, P önekin uzunluğudur. M alfabenin boyudur; küçük İngilizce harflerde 26'dır. S, önek düğümünün alt ağacındaki düğüm sayısıdır.

İşlem	Zaman	Açıklama
`ekle(kelime)`	$O(L)$	her karakter için bir adım; en çok L yeni düğüm
`ara(kelime)`	$O(L)$	dal yoksa daha erken biter
`onekVarMi(onek)`	$O(P)$	yolun varlığı yeterli
`onekSayisi(onek)`	$O(P)$	`gecen` sayacıyla
`tamamla(onek)`	$O(P + S \cdot M)$	önek düğümünden derinlik öncelikli gezinme
`sil(kelime)`	$O(L \cdot M)$	`bosMu` 26 hücreyi tarar; çocuk sayacıyla $O(L)$

Tablo 5.3.13: Dizi gösterimli trie'de işlemlerin süresi.

Tablonun en önemli özelliği bir sütunun eksik olmasıdır: hiçbir satırda n, yani saklanan kelime sayısı yoktur. İkili arama ağacındaki $\log n$ çarpanı trie'de görünmez. Trie bir milyon kelime de saklasa "dad" aramasının adım sayısı 3'tür. Bu özellik, trie işlemlerinin maliyetini $O(\log n)$ ya da $O(n)$ diye ifade et-

menin neden yanlış olduğunu da açıklar.

Burada L'nin de bir sınırı olduğu unutulmamalıdır. Bir sözlükte n farklı kelime varsa ve alfabe M harfliyse kelimelerin en az bir kısmının uzunluğu $\log_M n$ mertebesinde olmak zorundadır. Kısa bir alfabeyle çok sayıda farklı kelime ancak uzun kelimelerle kurulabilir. Yine de gerçek sözlüklerde L küçüktür (Türkçe ve İngilizce kelimelerin çoğu 15 harften kısadır) ve trie'nin üstünlüğü pratikte geçerlidir.

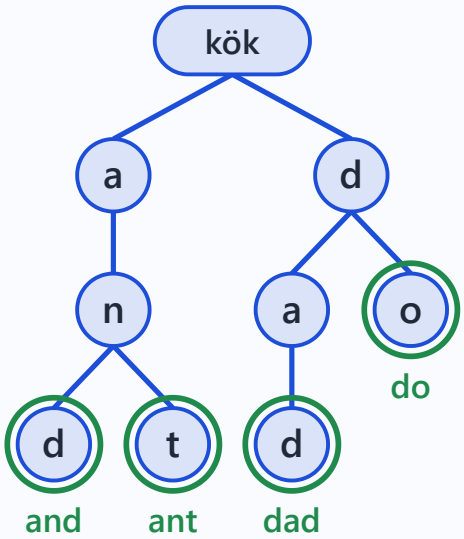
Bellek gereksinimi $O(N \cdot M)$'dir. N düğüm sayısıdır ve en çok bütün kelimelerin toplam uzunluğu artı 1'dir (kök). Bu sınıra, hiçbir kelime başka bir kelimeyle önek paylaşmıyorsa ulaşılır. Her düğüm M referanslık bir dizi taşıdığı için toplam bellek $N \cdot M$ ile orantılıdır.

5.3.7.2 Bellek: sayısal örnek

and, ant, dad ve do kelimelerinin trie'si üzerinden bellek tek tek sayılsın.

Büyüklük	Değer
Toplam karakter	*3 + 3 + 3 + 2 = 11*
Düğüm sayısı N	*1 (kök) + 8 = 9*
Çocuk referansı	*9 × 26 = 234*
Dolu referans	8 (kenar sayısı)
Boş (`null`) referans	226 (≈ %97)

Tablo 5.3.14: and, ant, dad, do trie'sinin bellek dökümü.



Şekil 5.3.24: and, ant, dad, do trie'si: kök ve 8 düğüm, 8 kenar.

Dolu referans sayısı kenar sayısına eşittir, çünkü her kenar tam olarak bir dolu hücreyle temsil edilir. Ağaçta kökten başka her düğümün tek bir ebeveyni olduğu için kenar sayısı $N - 1 = 8$'dir. Geri kalan $234 - 8 = 226$ hücre boştur; oran $226 / 234 \approx 0{,}966$, yani yaklaşık %97'dir. Ortak önekler 11 karakteri

8 düğüme indirmiştir, ama 26 öğeli diziler belleğin neredeyse tamamını `null` ile doldurur.

Büyük ölçekte tablo daha da çarpıcıdır. Bir milyon düğümlü bir trie 26 milyon referans tutar. 64 bitlik bir Java sanal makinesinde bir referans 8 bayttır; sıkıştırılmış referanslar açıksa 4 bayttır. Yalnız referans dizileri $26 \cdot 10^6 \cdot 8 \approx 208 \cdot 10^6$ bayt, yani yaklaşık 200 MB tutar; 4 baytlık referanslarla yaklaşık 100 MB. Buna her nesnenin başlık bilgisi eklenir. Alfabe büyüdükçe sorun ağırlaşır. Unicode'un temel düzlemi 65.536 karakterdir; her düğümde bu boyda bir dizi ayırmak açıkça olanaksızdır.

5.3.7.3 Güçlü ve zayıf yanlar

Trie'nin güçlü yanları şunlardır:

- **Hızlı önek tabanlı arama.** Önek sorgusu, önek sayacı ve otomatik tamamlama doğal işlemlerdir.
- **Kelime araması *O(L)* sürer** ve süre kelime sayısından bağımsızdır.
- **Ortak önekler bir kez saklanır.** Önek paylaşımı yüksek kümelerde (sözlükler, URL'ler, dosya yolları) düğüm sayısı toplam karakter sayısının çok altında kalır.
- **Sıra bilgisi korunur.** Derinlik öncelikli gezinme kelimeleri sözlüksel sırada verir. Hash çakışması diye bir sorun yoktur; en kötü durum da $O(L)$'dir.

Zayıf yanları da açıktır:

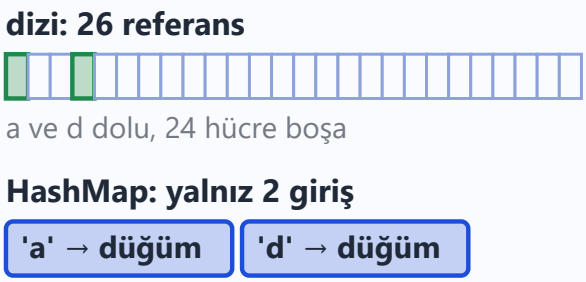
- **Yüksek bellek tüketimi,** özellikle büyük alfabelerde. Alan karmaşıklığı $O(N \cdot M)$'dir.
- **Ortak öneki az olan kümelerde kazanç yoktur.** Rastgele anahtarlarda neredeyse her karakter yeni bir düğüm açar.
- **Çok sayıda küçük nesne.** Bir arama, bellekte birbirinden uzak düğümlere atlar. İşlemcinin önbelleği (*cache*) bu erişim düzenini iyi karşılamaz. Bu yüzden aynı mertebede iki yapı arasında trie pratikte daha yavaş kalabilir.

5.3.7.4 Çocukları saklamanın yolları

Trie'nin mantığı, çocukların nasıl saklandığından bağımsızdır. Değişen yalnız "*ch* karakterli çocuğu ver" işleminin nasıl yapıldığıdır. Dört yaygın seçenek vardır.

Gösterim	Çocuğa erişim	Düğüm başına bellek	Not
`TrieDugumu[26]` dizi	$O(1)$	M referans	basit, hızlı; seyrek düğümlerde israf
`HashMap<Character, TrieDugumu>`	$O(1)$ ort.	çocuk sayısı kadar	her alfabe (Türkçe, Unicode) çalışır; sıra korunmaz
`TreeMap` / sıralı liste	$O(\log M)$	çocuk sayısı kadar	alfabetik gezinme korunur
Üçlü arama trie'si (TST)	$O(\log M)$ ort.	3 referans	her düğümde küçük / eşit / büyük bağlantısı

Tablo 5.3.15: Çocukları saklamanın yolları.



Şekil 5.3.25: Dizi gösterimi her düğümde 26 hücre ayırır; `HashMap` yalnız var olan çocukları saklar.

`HashMap` gösterimi Türkçe karakter sorununu da çözer. Anahtar, karakterin kendisidir; indis hesabı yoktur. Ekleme kodu şöyle kısalır:

```
class HaritaDugumu {
  Map<Character, HaritaDugumu> cocuk = new HashMap<>();
  boolean kelimeSonu;
}

void ekle(String kelime) {
  HaritaDugumu d = kok;
  for (char ch : kelime.toCharArray())
    d = d.cocuk.computeIfAbsent(ch, x -> new HaritaDugumu());
  d.kelimeSonu = true;
}
```

`computeIfAbsent`, anahtar haritada yoksa verilen fonksiyonla yeni bir değer oluşturup ekler ve değeri döndürür; anahtar varsa mevcut değeri döndürür. Böylece dizi sürümündeki "`null` ise oluştur, sonra in" iki adımı tek çağrıya iner. Bu trie'ye "çay", "çam", "şeker" ve "Ankara" eklenebilir; kökün haritasında 'A', 'ç' ve 'ş' anahtarları bulunur. Bedeli, her düğümde bir harita nesnesi ve hash hesabıdır. Ayrıca `HashMap` anahtarları sırasız tuttuğu için otomatik tamamlamanın alfabetik çıktısı kaybolur. Sıralı çıktı gerekiyorsa `TreeMap` kullanılır; erişim *O(log M)*'ye çıkar.

Seçim üç soruya göre yapılır: alfabe ne kadar büyük, bellek ne kadar kısıtlı, sıralı çıktı gerekli mi? Küçük ve sabit bir alfabede dizi en hızlısıdır. Büyük ya da önceden bilinmeyen alfabede harita gerekir. Bellek en önemli kısıtsa üçlü arama trie'si ya da sıkıştırılmış trie tercih edilir.

5.3.7.5 Trie, hash tablosu ve dengeli BST

Dizgi anahtarları saklamak için üç yapı sık sık karşılaştırılır. Tablo 5.3.16, n kelimelik ve kelime uzunluğu L olan bir küme için tipik maliyetleri verir.

İşlem	Trie	Hash tablosu	Dengeli BST
Tam kelime arama	*O(L)*	*O(L)* ortalama	*O(L · log n)*

İşlem	Trie	Hash tablosu	Dengeli BST
Önek sorgusu	$O(P)$	desteklenmez	$O(P \cdot \log n)$
Önekle başlayan k kelime	$O(P + S \cdot M)$	bütün tabloyu tarama	$O(P \cdot \log n + k \cdot L)$
Sıralı listeleme	DFS, $O(N \cdot M)$	sıralama gerekir	ortada-kök (*inorder*) gezinme, $O(n)$
En kötü durum	$O(L)$	$O(n \cdot L)$ (çakışma)	$O(L \cdot \log n)$
Bellek	yüksek	düşük	düşük

Tablo 5.3.16: Trie, hash tablosu ve dengeli ikili arama ağacı (BST). S: önek düğümünün alt ağacındaki düğüm sayısı.

Tam kelime aramasında trie ile hash tablosu aynı mertebededir. Bu ilk bakışta şaşırtıcı görünebilir, çünkü hash tablosu sık sık "$O(1)$" diye anılır. Ama o $O(1)$, anahtarın hash değeri hesaplanmış kabul edildiğinde geçerlidir. Bir dizginin hash değerini hesaplamak için L karakterin hepsi okunur. Eşitlik denetimi de L karakteri karşılaştırır. Dolayısıyla hash tablosu da $O(L)$'dir. Dengeli BST'de ise $\log n$ düğüm ziyaret edilir ve her ziyarette iki dizgi karşılaştırılır; bu karşılaştırma en çok L karakter okur.

Trie'nin asıl farkı önek ve sıra sorgularındadır. Hash tablosu öneki hiç desteklemez. BST sıralı olduğu için önekle başlayan kelimeleri ardışık bir aralıkta tutar ve aralığın başını $O(P \cdot \log n)$'de bulur. Trie aynı soruyu $\log n$ çarpanı olmadan cevaplar. Bunun karşılığında trie bellekte en pahalı yapıdır. Seçim, yapılacak sorguların türüne göre yapılır: yalnız tam eşleşme gerekiyorsa hash tablosu yeterlidir; önek ve sıra sorguları sıksa trie öne çıkar.

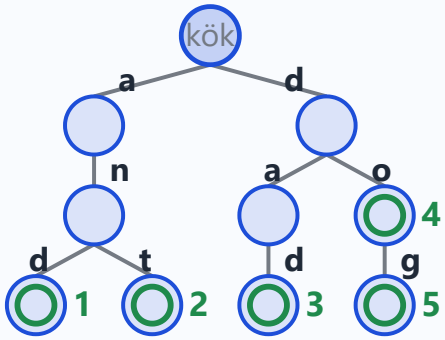
Örnek 5.3.7

Soru. Bir trie'de $n = 1.000.000$ kelime var; aranan kelimenin uzunluğu $L = 8$. `ara` en kötü durumda kaç düğüm adımı atar? Dengeli BST'de durum nedir?

Çözüm. Trie'de her karakter için bir çocuğa inilir; en çok 8 adım atılır. Dengeli BST'de $\log_2 10^6 \approx 19{,}9$, yani yaklaşık 20 düğüm ziyaret edilir. Her ziyarette en çok 8 karakter karşılaştırılır; en kötü durumda yaklaşık $20 \cdot 8 = 160$ karakter okunur. Düz bir listede ise kelimeler tek tek denenir ve iş $O(n \cdot L)$ olur.

5.3.7.6 Trie ile sıralama

Trie'nin kelimeleri sözlüksel sırada verme özelliği bir sıralama algoritmasına dönüştürülebilir. Girdi sırası dog, ant, do, and, dad olsun. Bütün kelimeler trie'ye eklenir. Sonra kökten **önce-kök** (*preorder*) gezinme yapılır: önce düğümün kendisi işlenir, sonra çocukları 'a'dan 'z'ye sırayla ziyaret edilir. İşaretli her düğümde kelime yazılır. Bu gezinme, `topla` metodunun kökten çağrılmasıyla aynıdır.



Şekil 5.3.26: dog, ant, do, and, dad kelimelerinin trie'si. Yeşil sayılar, önce-kök gezinmede kelimelerin yazılma sırasıdır.

Çıktı and, ant, dad, do, dog olur. İki kural bu sırayı garanti eder. Birincisi, kardeşler indis sırasıyla, yani alfabetik sırayla ziyaret edilir: “a” dalındaki bütün kelimeler “d” dalındakilerden önce gelir. İkincisi, düğüm çocuklarından önce yazılır: “do”, “dog”dan önce gelir. Bu, sözlüksel sıranın tanımıyla örtüşür: bir önek, onu uzatan her dizgiden küçüktür.

Maliyet hesabında toplam karakter sayısı N ile gösterilsin. Ekleme toplam $O(N)$ sürer. Gezinme her düğümde 26 hücreyi denetler: $O(N \cdot M)$. Sabit alfabede bu $O(N)$'dir. Karşılaştırmalı sıralama ise $O(n \log n)$ karşılaştırma yapar ve her karşılaştırma $O(L)$ sürebilir; ortak önekleri uzun kelimelerde aynı karakterler tekrar tekrar okunur. Trie bu tekrarı ortadan kaldırır. Aynı fikir “burst sort” algoritmasının ve MSD basamak sıralamasının dizgi sürümlerinin temelidir. MSD basamak sıralaması da kelimeleri ilk karaktere göre kovalara ayırır, her kovayı ikinci karaktere göre böler; bu bölünme ağacı aslında bir trie'dir.

5.3.8 Uygulamalar ve türevler

5.3.8.1 Kullanım alanları

Trie'nin temel işlemleri birçok gerçek uygulamanın çekirdeğidir. Dört örnek alan öne çıkar.

- **Otomatik tamamlama.** Arama kutuları ve tümleşik geliştirme ortamları (IDE) yazılan önekin düğümüne iner ve alt ağaçtaki kelimeleri önerir. Düğümlerde kullanım sıklığı da saklanırsa sık kullanılan kelimeler önce gösterilir.
- **Yazım denetimi.** Sözlük trie'de tutulur. Kelime bulunamazsa trie üzerinde küçük düzenlemelerle (ekleme, silme, değiştirme) yakın kelimeler aranır; düzenleme uzaklığı satırları trie gezilirken hesaplanır.
- **IP yönlendirme.** Alfabesi *{0, 1}* olan bir **ikili trie** (*binary trie*) ile en uzun önek eşleşmesi yapılır. Bir paketin adresi, rota tablosundaki öneklerden en uzun eşleşene yönlendirilir.

- **Kelime oyunları ve T9.** Boggle ya da kelime bulmacası çözen bir program harflerden yollar kurar. Kurulan yol hiçbir kelimenin öneki değilse o yöndeki arama hemen kesilir; bu budama `onekVarMi` ile yapılır. Eski telefonlardaki T9 sistemi de tuş dizisinden kelime önermek için trie benzeri bir yapı kullanır.

5.3.8.2 İkili trie ve en uzun önek eşleşmesi

İnternet yönlendiricileri bir paketi nereye göndereceklerine, paketin hedef adresine bakarak karar verir. Rota tablosu tek tek adresleri değil, adres öneklerini saklar. “0*” biçimindeki bir rota 0 ile başlayan bütün adresleri, “011*” ise 011 ile başlayanları kapsar. Bir adres birden fazla önekle eşleşebilir; kural, **en uzun önek eşleşmesinin** (*longest prefix match*) seçilmesidir. Uzun önek daha özel bir rotadır ve genel rotaya göre önceliklidir.

Rotalar $0^* \to A$, $01^* \to B$, $011^* \to C$, $1^* \to D$ ve $110^* \to E$ olsun. Her rota, ikili trie'de öneki temsil eden düğümde saklanır. Düğüm sınıfı iki çocuklu bir dizi ve bir rota alanı taşır; rota ekleme, olağan eklemeyle aynıdır:

```
class IkiliDugum {
  IkiliDugum[] cocuk = new IkiliDugum[2];  // '0' ve '1'
  String rota;                             // null: rota yok
}

void rotaEkle(String onek, String rota) {
  IkiliDugum d = kok;
  for (char b : onek.toCharArray()) {
    int i = b - '0';
    if (d.cocuk[i] == null) d.cocuk[i] = new IkiliDugum();
    d = d.cocuk[i];
  }
  d.rota = rota;
}
```

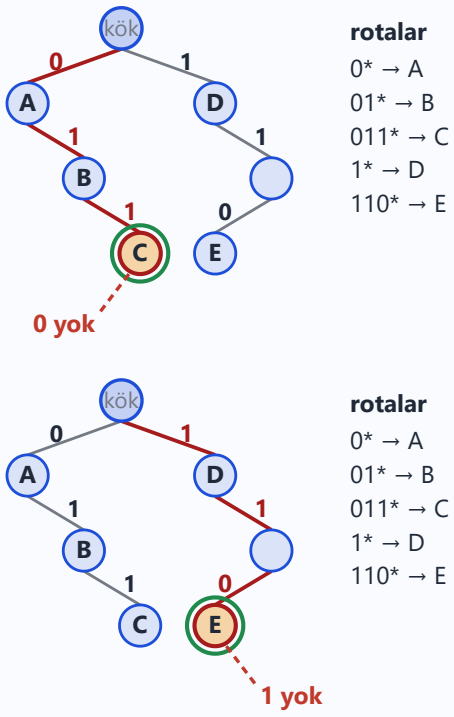
Adres için en uzun eşleşme şöyle bulunur. Kökten başlanır ve adresin bitleri sırayla izlenir. Yol boyunca rota taşıyan her düğümde rota hatırlanır. Yol kesildiğinde ya da adres bittiğinde, hatırlanan son rota cevaptır. Son görülen rota, yolda görülenler içinde en derindeki, yani en uzun önektir.

```
String enUzunEslesme(String adres) {
  IkiliDugum d = kok;
  String enIyi = null;
  for (int k = 0; k < adres.length(); k++) {
    if (d.rota != null) enIyi = d.rota;  // daha uzun önek
    d = d.cocuk[adres.charAt(k) - '0'];
    if (d == null) return enIyi;
  }
  return d.rota != null ? d.rota : enIyi;
}
```

Döngünün her turunda önce şu anki düğümün rotası denetlenir; varsa `enIyi` güncellenir. Sonra adresin *k*. bitine göre çocuğa inilir: `adres.charAt(k) - '0'`, '0' karakterini 0'a, '1'i 1'e çevirir. Çocuk yoksa hatırlanan en iyi rota döner. Döngü bütün bitleri işlerse son düğümün rotası da denetlenir: varsa o, yoksa `enIyi` döner.

Adres	İzlenen düğümler (rota)	Durma nedeni	Sonuç
0110	kök, 0 (A), 01 (B), 011 (C)	`k = 3` : '0' dalı yok	C
1001	kök, 1 (D)	`k = 1` : '0' dalı yok	D
1101	kök, 1 (D), 11 (—), 110 (E)	`k = 3` : '1' dalı yok	E

Tablo 5.3.17: `enUzunEslesme` izlemesi; rotalar 0* → A, 01* → B, 011* → C, 1* → D, 110* → E.



Şekil 5.3.27: Solda 0110 adresi: 0 → 01 → 011 inilir, '0' dalı yoktur; son görülen rota C'dir. Sağda 1101 adresi: "11" ara düğümdür, rotası yoktur; "110"daki E seçilir.

0110 adresinde `enIyi` sırasıyla A, B ve C olur. Her yeni rota bir öncekinden daha uzun bir önektir ve onun yerini alır. 1001 adresinde yalnız D görülür; özel bir rota olmadığı için genel rota kullanılır. 1101 adresinde "11" düğümü yalnız yol üzerindedir ve `enIyi` 'yi değiştirmez. Adım sayısı adresin uzunluğuyla sınırlıdır: 32 bitlik bir IPv4 adresi için en çok 32 adım atılır, rota tablosunda kaç rota olduğu fark etmez.

Gerçek yönlendiriciler bu fikri sıkıştırılmış ikili trie'lerle (Patricia) ve donanım desteğiyle hızlandırır, ama temel ilke aynıdır. Önek tablosu hash tablosunda tutulsaydı her olası önek uzunluğu için ayrı bir arama gerekirdi.

5.3.8.3 En büyük XOR çifti

İkili trie'nin daha az bilinen bir kullanımı bit işlemleriyle ilgilidir. Soru şudur: bir sayı dizisinde $a \oplus b$ değerini en büyük yapan çift hangisidir? $\oplus$ bit düzeyinde dışlamalı veya (XOR) işlemidir: iki bit farklıysa sonuç 1, aynıysa 0'dır. Dizi [3, 10, 5, 25, 2, 8] olsun. Kaba kuvvet bütün çiftleri dener: n sayı için $n(n-1)/2$ çift.

Trie yöntemi sayıları en anlamlı bitten başlayarak sabit uzunlukta (burada 5 bit) bir ikili trie'ye ekler. Sonra her x için kökten inilir ve her bitte, x'in o bitinin *tersi* olan dal seçilir; dal yoksa mecburen aynı bit seçilir. Ters bit seçildiği her düzeyde sonucun o biti 1 olur.

```
int enBuyukXor(int[] a, int B) {
  IkiliDugum kok = new IkiliDugum();
  for (int x : a) {                       // her sayıyı B bitle ekle
    IkiliDugum d = kok;
    for (int b = B - 1; b >= 0; b--) {
      int bit = (x >> b) & 1;
      if (d.cocuk[bit] == null) d.cocuk[bit] = new IkiliDugum();
      d = d.cocuk[bit];
    }
  }
  int enIyi = 0;
  for (int x : a) {                       // her sayı için ters bitleri ara
    IkiliDugum d = kok;
    int y = 0;
    for (int b = B - 1; b >= 0; b--) {
      int bit = (x >> b) & 1;
      if (d.cocuk[1 - bit] != null) { d = d.cocuk[1 - bit]; y |= (1 - bit) << b; }
      else { d = d.cocuk[bit]; y |= bit << b; }
    }
    enIyi = Math.max(enIyi, x ^ y);
  }
  return enIyi;
}
```

Açgözlü seçimin doğruluğu ikili sayıların bir özelliğine dayanır. Bir sayıda üst bitteki 1, alttaki bütün bitlerin toplamından değerlidir: $2^b > 2^{b-1} + \ldots + 2 + 1 = 2^b - 1$. Bu yüzden üst bitte 1 elde etmek, alt bitlerde ne kaybedilirse kaybedilsin kârlıdır. Her düzeyde mümkünse ters dal seçmek doğru karardır.

sayı	ikili (5 bit)
3	**00011**
10	**01010**
5	**00101**
25	**11001**
2	**00010**
8	**01000**
5 ⊕ 25	**11100 = 28**

Şekil 5.3.28: Altı sayının 5 bitlik gösterimi. 5 ile 25'in XOR'u 11100 = 28'dir ve dizideki en büyük değerdir.

Örnek 5.3.8

Soru. *x = 5 = 00101* için trie'de hangi yol izlenir ve hangi sayıya varılır?

Çözüm. İstenen bitler 5'in tersidir: 11010. Bit 4'te 1 istenir; yalnız 25 (11001) bu dala sahiptir ve inilir. Bit 3'te 1 istenir; 25'in biti 1'dir, var. Bit 2'de 0 istenir; 25'in biti 0'dır, var. Bit 1'de 1 istenir; yalnız 25 kalmıştır ve biti 0'dır, mecburen 0'a inilir. Bit 0'da 0 istenir; 25'in biti 1'dir, mecburen 1'e inilir. Varılan sayı 11001 = 25'tir ve $5 \oplus 25 = 00101 \oplus 11001 = 11100 = 28$. Bütün sayılar için aynı iş yapılınca en büyük değer 28 bulunur.

Yöntem	Zaman	İş miktarı
Bütün çiftler	$O(n^2)$	$n(n-1)/2 \approx 5 \cdot 10^9$ işlem
İkili trie	$O(n \cdot B)$	ekleme ve sorgu, $2 \cdot n \cdot B \approx 6 \cdot 10^6$ adım

Tablo 5.3.18: En büyük XOR için iki yöntemin maliyeti; $n = 10^5$, $B = 30$ bit.

Trie yöntemi her sayı için iki kez B düzey boyunca gezinir: bir kez eklerken, bir kez sorgularken. Fark yaklaşık bin kattır. Burada trie'nin "uzunluk" parametresi, sayının bit sayısı B'dir.

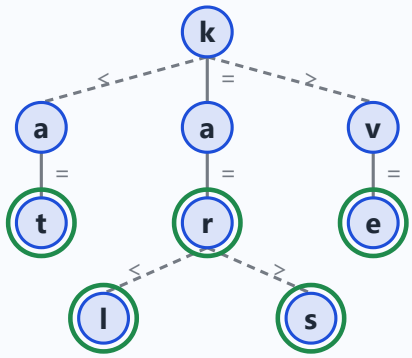
5.3.8.4 Üçlü arama trie'si

Dizi gösteriminin bellek sorununa bir çözüm **üçlü arama trie'sidir** (*ternary search trie*, TST). TST'de her düğüm bir karakter ve üç bağlantı taşır: `sol` (küçük), `orta` (eşit, sonraki karaktere geç) ve `sag` (büyük). Bir düğümün sol ve sağ zinciri, olağan trie'deki kardeş çocuklar için küçük bir ikili arama ağacıdır. Orta bağlantı ise trie'deki "bir düzey aşağı in" adımına karşılık gelir. Kelime sonu düğümde işaretlenir.

```
class TstDugum {
  char c;                    // düğümün karakteri
  boolean son;               // kelime burada biter mi?
  TstDugum sol, orta, sag;
  TstDugum(char c) { this.c = c; }
}

TstDugum ekle(TstDugum d, String s, int i) {
  char ch = s.charAt(i);
  if (d == null) d = new TstDugum(ch);
  if (ch < d.c) d.sol = ekle(d.sol, s, i);
  else if (ch > d.c) d.sag = ekle(d.sag, s, i);
  else if (i < s.length() - 1) d.orta = ekle(d.orta, s, i + 1);
  else d.son = true;
  return d;
}
```

Ekleme özyinelemelidir. Düğüm yoksa sıradaki karakterle yeni bir düğüm oluşturulur. Karakter düğümünkinden küçükse sola, büyükse sağa gidilir; bu iki durumda `i` değişmez, çünkü aynı karakter için kardeş aranır. Karakterler eşitse ve kelime bitmemişse ortaya gidilir ve `i` bir artar. Eşitse ve son karakterse düğüm işaretlenir.



Şekil 5.3.29: kar, kal, at, ve, kas kelimelerinin üçlü arama trie'si. Düz çizgi orta (=), kesikli çizgiler sol (<) ve sağ (>) bağlantılarıdır.

Şekildeki TST, kelimeler kar, kal, at, ve, kas sırasıyla eklenerek kurulmuştur. İlk kelime “kar” k → a → r omurgasını oluşturur. “kal” bu omurgayı “ka”ya kadar izler; 'l' < 'r' olduğu için 'r'nin soluna eklenir. “at”, kökte 'a' < 'k' olduğu için kökün soluna, “ve” ise 'v' > 'k' olduğu için sağına gider. “kas” 'r'ye kadar iner ve 's' > 'r' olduğu için sağa eklenir. Ekleme sırası şekli değiştirir: önce eklenen kelime omurga olur.

Yapı	Düğüm	Bağlantı
26 öğeli dizili trie (kök dahil)	10	*10 × 26 = 260*

Yapı	Düğüm	Bağlantı
TST	9	*9 × 3 = 27*

Tablo 5.3.19: kar, kal, at, ve, kas için iki gösterimin bellek karşılaştırması.

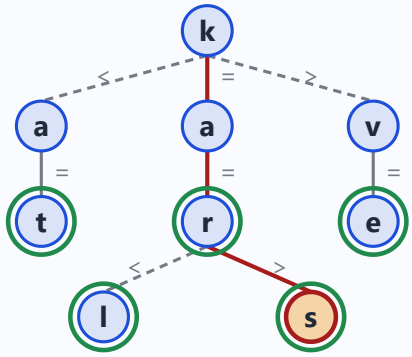
Dizili trie'de kök ve k, ka, kar, kal, kas, a, at, v, ve düğümleri vardır; 10 düğüm 260 referans ayırır. TST'de boş kök yoktur; her karakter bir düğümdür ve 9 düğüm yalnız 27 bağlantı taşır. Fark yaklaşık on katır.

```
boolean ara(String s) {
  TstDugum d = kok;
  int i = 0;
  while (d != null) {
    char ch = s.charAt(i);
    if (ch < d.c) d = d.sol;
    else if (ch > d.c) d = d.sag;
    else if (i == s.length() - 1) return d.son;
    else { d = d.orta; i++; }
  }
  return false;
}
```

Arama yinelemeli yazılmıştır. `i`, kelimede sıradaki karakterin indisidir. Karakter düğümünkinden küçükse sola, büyükse sağa gidilir ve `i` değişmez. Eşitse iki durum vardır: son karakterse düğümün işareti döner, değilse ortaya inilir ve `i` artar. Bağlantı `null` olunca döngü biter ve `false` döner.

Çağrı	`d.c`	`ch, i`	Karar
`ara("kas")`	k	k, 0	eşit, son değil → orta, `i = 1`
	a	a, 1	eşit → orta, `i = 2`
	r	s, 2	's' > 'r' → sağ, `i` değişmez
	s	s, 2	eşit ve son indis → `d.son` = `true`
`ara("al")`	k	a, 0	'a' < 'k' → sol
	a	a, 0	eşit, son değil → orta, `i = 1`
	t	l, 1	'l' < 't' → sol; bağlantı `null` → `false`

Tablo 5.3.20: TST'de `ara("kas")` ve `ara("al")` izlemesi.

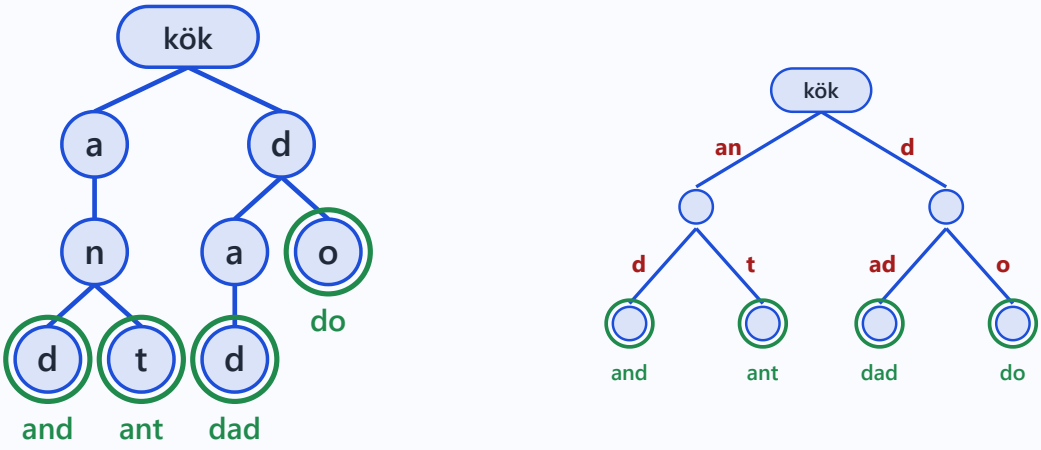


Şekil 5.3.30: `ara("kas")` son adımı: k → a → r → s; 3 karakter için 4 düğüm ziyaret edildi.

“kas” aramasında 3 karakter için 4 düğüm ziyaret edilir: üçü orta adım, biri “r”nin kardeşlerinden 's'yi bulmak için atılan sağ adımdır. Genel olarak TST'de arama, L orta adım ile her düzeyde kardeşler arasında ikili arama adımlarının toplamıdır. Kelimeler rastgele sırada eklenirse ortalama maliyet $O(L + \log n)$'dir. Kelimeler sıralı eklenirse kardeş ağaçları bozulur: at, kal, kar, kas, ve sırasıyla eklenen TST'de 'a'nın sağına 'k', 'k'nin sağına 'v' dizilir ve 'l', 'r', 's' kardeşleri bir zincir oluşturur. Bu durum, ikili arama ağacına sıralı ekleme sorununun aynısıdır. Bentley ve Sedgewick 1997'deki çalışmalarında TST'yi büyük alfabeler (örneğin Unicode) için bellek ve hız dengesini koruyan bir yapı olarak ayrıntılı incelemiştir.

5.3.8.5 Sıkıştırılmış trie

Standart trie'de çoğu düğümün tek çocuğu vardır. “karton” kelimesinin “kar”dan sonraki kısmı, her biri tek çocuklu “t”, “o”, “n” düğümlerinden oluşan bir zincirdir. Bu zincirler hiçbir karar içermez; yalnız karakterleri sırayla taşır. **Sıkıştırılmış trie** (*compressed trie, radix tree*), tek çocuklu ve işaretsiz düğüm zincirlerini tek bir kenarda birleştirir. Kenar artık tek bir karakter değil, bir alt dizgi taşır.



Şekil 5.3.31: and, ant, dad, do kelimeleri. Solda standart trie: kök hariç 8 düğüm. Sağda sıkıştırılmış trie: 6 düğüm; kenarlar “an”, “ad” gibi alt dizgiler taşır.

Şekilde “a” düğümünün tek çocuğu vardır ve işaretsizdir; “a” ile “an” tek bir “an” kenarında birleşir. “an” düğümü iki çocuklu olduğu için kalır. “da” düğümü de tek çocuklu ve işaretsizdir; “d”den “dad”e giden yol “ad” kenarı olur. “d” düğümü iki çocuklu olduğu için kalır. Sonuçta kök dışında an, and, ant, d, dad ve do düğümleri kalır.

Teorem 5.3.1 (Sıkıştırılmış trie'nin boyu)

n kelimelik sıkıştırılmış bir trie'de kök hariç en çok $2n - 1$ düğüm vardır.

İspat. Sıkıştırmadan sonra kökten başka her düğüm ya işaretlidir ya da en az iki çocukludur; tek çocuklu ve işaretsiz düğümler birleştirilmiştir. İşaretli düğüm sayısı kelime sayısına eşittir: n. Her yaprak işaretlidir; bu yüzden yaprak sayısı en çok n'dir. Herhangi bir köklü ağaçta en az iki çocuklu düğümlerin sayısı yaprak sayısından küçüktür: her böyle düğüm yaprak sayısına en az bir ekler. Dolayısıyla işaretsiz ve en az iki çocuklu düğümler en çok $n - 1$ tanedir. Toplam en çok $n + (n - 1) = 2n - 1$'dir. □

Teorem, düğüm sayısının kelimelerin uzunluğundan bağımsız olarak $O(n)$ olduğunu söyler. Standart trie'de ise düğüm sayısı toplam karakter sayısıyla orantılıdır. and, ant, dad, do örneğinde $n = 4$ için sınır 7'dir ve gerçek değer 6'dır. Kenar etiketleri yine de toplam olarak karakterleri taşır; ama bu etiketler ayrı dizgiler olarak değil, özgün kelimelere işaret eden (başlangıç, uzunluk) çiftleri olarak saklanabilir. Böylece bellek de $O(n)$ düğüm ve kelimelerin kendisiyle sınırlı kalır.

Örnek 5.3.9

Soru. car, cart, care, cat ve dog kelimelerinin standart trie'sinde kök hariç kaç düğüm vardır? Sıkıştırılmış trie'de kaç düğüm kalır?

Çözüm. Standart trie'nin düğümleri c, ca, car, cart, care, cat, d, do, dog'dur: 9 düğüm. Sıkıştırmada “c” tek çocuklu ve işaretsizdir; “c” ile “ca” birleşir ve “ca” kenarı oluşur. “ca” iki çocuklu (r, t) olduğu için kalır. “car” işaretlidir, kalır. cart, care ve cat yapraktır. “d” ve “do” tek çocuklu ve işaretsizdir; “dog” kenarı tek düğüme iner. Kalan düğümler ca, car, cart, care, cat, dog'dur: kök hariç 6, kök dahil 7 düğüm.

Morrison 1968'de bu fikrin ikili alfabe sürümünü **Patricia** adıyla yayımladı. Patricia trie'si, IP yönlendirme tablolarında ve dosya sistemlerinde kullanılır. Sıkıştırmanın bedeli, ekleme ve silmenin karmaşıklaşmasıdır. Yeni bir kelime bir kenarın ortasında ayrılıyorsa kenar ikiye bölünür ve araya yeni bir düğüm girer. Silmede ise tek çocuklu kalan bir düğüm ebeveyn kenarıyla yeniden birleştirilir.

5.3.8.6 Sonek trie'si

Şimdiye kadarki örneklerde trie bir kelime kümesini saklıyordu. Bir metnin içinde arama yapmak için farklı bir küme seçilir: metnin bütün sonekleri. T = "abab" metninin sonekleri abab, bab, ab ve b'dir. Bu dört sonek bir trie'ye eklenirse **sonek trie'si** (*suffix trie*) elde edilir.

Sonek trie'sinin gücü bir eşdeğerlik zincirinden gelir:

> *P, T'nin alt dizgisidir $\Leftrightarrow$ P, T'nin bir sonekinin önekidir $\Leftrightarrow$ P, sonek trie'sinde kökten başlayan bir yoldur.*

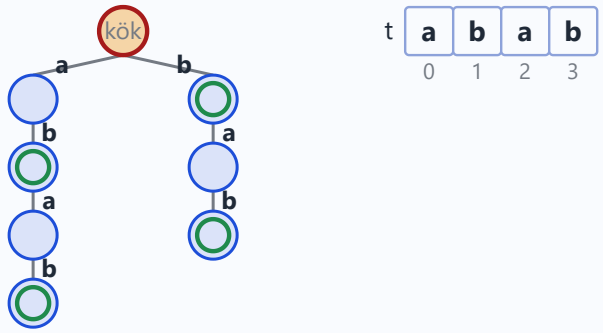
İlk eşdeğerlik alt dizginin tanımından gelir: P, T'de i. konumda başlıyorsa i. konumdan başlayan sonekin önekidir. İkinci eşdeğerlik, trie'de her önekin bir yol olmasındandır. Sonuç olarak "P, T'de geçiyor mu?" sorusu `onekVarMi(P)` ile cevaplanır. Süre $O(m)$'dir; $m = |P|$'dir ve metnin uzunluğundan bağımsızdır. Metin bir kez işlenir, ardından istenen sayıda örüntü hızla aranır.

```
TrieDugumu sonekTrie(String t) {
  TrieDugumu kok = new TrieDugumu();
  for (int s = 0; s < t.length(); s++) {
    TrieDugumu d = kok;
    for (int j = s; j < t.length(); j++) {
      int i = t.charAt(j) - 'a';
      if (d.cocuk[i] == null)
        d.cocuk[i] = new TrieDugumu();
      d = d.cocuk[i];
    }
    d.kelimeSonu = true;  // sonek bitti
  }
  return kok;
}
```

Kod dış döngüde her sonekin başlangıcı (`s`) üzerinde gezinir. İç döngü, `s` 'den başlayarak sonekin karakterlerini olağan ekleme gibi trie'ye ekler. Sonek bitince son düğüm işaretlenir. İşaret alt dizgi sorgusu için gerekli değildir; ama bir dizginin *sonek* olup olmadığını sormak için kullanılabilir.

`s`	Sonek	Açılan düğümler	İşaretlenen
0	abab	a, ab, aba, abab (4)	abab
1	bab	b, ba, bab (3)	bab
2	ab	yok; yol zaten var	ab
3	b	yok; yol zaten var	b

Tablo 5.3.21: `sonekTrie("abab")` izlemesi.



Şekil 5.3.32: “abab” metninin sonek trie'si: kök dahil 8 düğüm. abab, bab, ab ve b sonekleri işaretlidir.

İlk iki sonek yeni dallar açar; son ikisi ise önceki soneklerin yolları üzerinde kalır ve yalnız işaret alır. Toplam 7 düğüm açılmıştır; kökle birlikte 8 düğüm vardır. Sorgular şöyle cevaplanır: “ba” kökten bir yoldur (b → a), dolayısıyla “abab”ın alt dizgisidir. “bb” için “b”den sonra 'b' dalı yoktur; “bb” alt dizgi değildir. “bab” üç adımda bulunur, “bb” iki adımda reddedilir.

Sonek trie'sinin bedeli boyudur. n uzunluklu bir metnin en çok $n(n + 1)/2$ farklı boş olmayan alt dizgisi vardır ve sonek trie'sinin her düğümü farklı bir alt dizgidir. Bütün karakterleri farklı “abcd” metninde bu sınıra ulaşılır: 10 düğüm. “banana” metninin 21 olası alt dizgisinden 15'i farklıdır ve sonek trie'sinde kök dışında 15 düğüm vardır. Genel olarak bellek $O(n^2)$'dir; bir milyon karakterlik bir metin için bu kabul edilemez.

Çözüm yine sıkıştırmadır. Sıkıştırılmış sonek trie'sine **sonek ağacı** (*suffix tree*) denir. Sonek ağacında n sonek olduğu için sıkıştırılmış trie teoremine göre $O(n)$ düğüm vardır. Kenar etiketleri metindeki (başlangıç, bitiş) konum çiftleriyle saklanır. Ukkonen'in 1995'te yayımladığı algoritma sonek ağacını $O(n)$ sürede kurar. Daha az bellek isteyen bir seçenek ise **sonek dizisidir** (*suffix array*): soneklerin sözlüksel sıradaki başlangıç indislerini tutan bir dizi. Sonek dizisinde alt dizgi araması ikili aramayla $O(m \log n)$ sürer.

5.3.8.7 Aho–Corasick

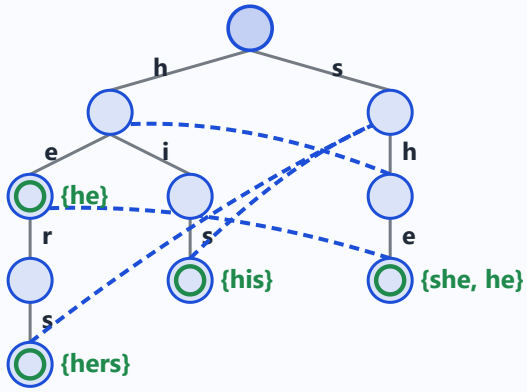
Bir virüs tarayıcısı bir dosyada binlerce imzayı aynı anda arar. Her imza için metni ayrı ayrı taramak, örüntü sayısı kadar tarama demektir. **Aho–Corasick algoritması** bütün örüntüleri bir trie'ye yerleştirir ve metni yalnız bir kez soldan sağa okur. Okuma sırasında trie'de bir düğümden ötekine geçilir. Bu yaklaşım, tek örüntülü KMP algoritmasının çok örüntülü genellemesidir.

Örüntüler he, she, his ve hers olsun. Trie'de h → he → her → hers, h → hi → his ve s → sh → she yolları vardır. Trie'nin yanında her düğüm için bir **hata bağlantısı** (*failure link*) hesaplanır.

Tanım 5.3.2

Örüntü trie'sinde bir u düğümünün **hata bağlantısı**, u'nun temsil ettiği dizginin trie'de düğümü bulunan en uzun *özel* sonekine (dizginin kendisinden kısa sonekine) giden bağlantıdır. Böyle bir sonek yoksa bağlantı köke gider.

Örnekte “she”nin sonekleri “he” ve “e”dir; “he” trie'de vardır, hata bağlantısı “he”ye gider. “sh”nin soneki “h” trie'dedir. “hers”in sonekleri “ers”, “rs” ve “s”dir; yalnız “s” trie'dedir. “his”in hata bağlantısı da “s”ye gider. Geri kalan düğümlerin (h, s, he, hi, her) bağlantıları köke gider.



Şekil 5.3.33: he, she, his, hers örüntülerinin trie'si. Kesikli çizgiler hata bağlantılarıdır; süslü parantezler düğümün çıktı kümesidir.

Metin okunurken şu anki düğümden okunan karakterin dalı varsa o dala geçilir. Dal yoksa hata bağlantısıyla geri çekilinir ve aynı karakter yeni düğümde denenir; bu, KMP'de önek tablosuyla geri çekilmenin karşılığıdır. Kökte de dal yoksa kökte kalınır. Her düğümde o düğümün **çıktı kümesi** bildirilir. Çıktı kümesi, düğümün kendi örüntüsüyle hata zinciri boyunca ulaşılan örüntülerin birleşimidir. “she” düğümünün çıktısı {she, he}'dir, çünkü hata bağlantısı “he”ye gider ve “he” bir örüntüdür.

İndis	Karakter	Durum (düğüm)	Bulunan
0	u	kök ('u' dalı yok)	—
1	s	s	—
2	h	sh	—
3	e	she	she, he
4	r	she → (hata) he → her	—
5	s	hers	hers

Tablo 5.3.22: “ushers” metninin Aho–Corasick ile taranması.

4. indisteki karakterde (r) “she” düğümünün 'r' dalı yoktur. Hata bağlantısıyla “he”ye çekilinir; “he”nin 'r' dalı vardır ve “her” düğümüne geçilir. Bu geri çekilme, “she” ile okunan son iki karakterin (h, e) boşa gitmesini önler: “he” zaten eşleşmiş bir öneki temsil eder. Metin tek geçişte okunur ve üç eşleşme bulunur: 3. indiste biten she ve he, 5. indiste biten hers.

Hata bağlantıları trie üzerinde genişlik öncelikli gezinmeyle hesaplanır: bir düğümün bağlantısı, ebeveyninin bağlantısından başlayarak aynı karakterin dalı aranarak bulunur. Bu ön işlem örüntü uzunluklarının toplamı m ile orantılıdır. Tarama ise n uzunluklu metin için $O(n)$ geçiş ve z eşleşmenin bildirimi kadar sürer. Toplam süre $O(n + m + z)$'dir (Aho ve Corasick, 1975). Algoritma virüs imzası taramada, ağ saldırı tespit sistemlerinde ve yasaklı kelime süzgeçlerinde kullanılır. Unix'teki `fgrep` komutunun ilk sürümleri de bu algoritmaya dayanıyordu.

5.3.8.8 Trie ailesi

Trie türevleri ortak bir fikrin farklı biçimleridir. Tablo 5.3.23 hangi sorunun hangi yapıyla en iyi cevaplandığını özetler.

Yapı	Sakladığı	Güçlü olduğu soru	Sorgu
Standart trie	kelime kümesi	tam kelime, önek, otomatik tamamlama	$O(L)$
Önek sayaçlı trie	+ düğümden geçen kelime sayısı	“bu önekle kaç kelime var?”	$O(P)$
Sıkıştırılmış trie	tek çocuklu zincirler birleşik	uzun ortak önekli anahtarlar, bellek	$O(L)$
Üçlü arama trie'si	düğüm başına 3 bağlantı	büyük alfabe, az bellek	$O(L + \log n)$ ort.
İkili trie	bit dizileri (IP, tam sayı)	en uzun önek eşleşmesi, en büyük XOR	$O(B)$
Sonek trie'si / dizisi	bütün sonekler (dizi: $O(n)$ bellek)	“P, T'nin alt dizgisi mi?”	$O(m)$ / $O(m \log n)$
Aho–Corasick	örüntü trie'si + hata bağlantıları	birçok örüntüyü metinde aynı anda bulmak	$O(n + z)$

Tablo 5.3.23: Trie ailesi: hangi soruya hangi yapı?

Ortak fikir hep aynıdır: anahtar karakter karakter (ya da bit bit) okunur ve ortak önekler paylaşılır. Farklar, düğümde neyin saklandığında ve çocuklara nasıl erişildiğindedir. Yapı seçiminde önce sorunun türü belirleyicidir: tam eşleşme mi, önek mi, alt dizgi mi, çok örüntü mü? Ardından alfabenin büyüklüğü ve bellek bütçesi dikkate alınır.

5.3.9 Sık yapılan hatalar

Trie kodu kısadır, ama birkaç ince nokta sık sık gözden kaçar. Aşağıdaki hataların çoğu küçük testlerde görünmez ve ancak özel girdilerle ortaya çıkar.

Sık yapılan hata

`kelimeSonu`**'nu unutmak.** Aramada yalnız yolun varlığına bakmak her öneki kelime sayar. and, ant, dad, do trie'sinde "da" bulunmuş gibi görünür. Kelime sonu işareti olmadan trie bir kelime kümesini değil, kümenin bütün öneklerini saklar.

Sık yapılan hata

`ara` **ile** `onekVarMi` **'yi karıştırmak.** Önek sorgusu son düğümün işaretine bakmaz; tam kelime araması bakmak zorundadır. İki metot yalnız son satırda ayrılır ve bu satırın yanlış yazılması derleme hatası vermez.

Sık yapılan hata

Alfabe dışı karakter. `'A' - 'a'` = *–32*, `'ş' - 'a'` = 254'tür; dizi sınırı aşılır. Girdi küçük harfe ve a–z alfabesine çevrilmeli ya da çocuklar `HashMap` 'te tutulmalıdır. Türkçe metinlerde küçük harfe çevirirken de dikkat gerekir: Java'da `"I".toLowerCase()` İngilizce yerel ayarda "i", Türkçe yerel ayarda "ı" verir; parametresiz çağrı makinenin yerel ayarına bağlıdır.

Sık yapılan hata

Silmede ortak düğümü silmek. Yolu baştan sona silmek, öneki paylaşan öteki kelimeleri de yok eder. Yalnız çocuksuz ve işaretsiz düğümler, aşağıdan yukarı silinir.

Sık yapılan hata

Karmaşıklığı *n* ile ifade etmek. Trie işlemleri *O(log n)* ya da *O(n)* değil, *O(L)*'dir: kelime sayısından bağımsızdır. Buna karşılık hash tablosundaki aramayı "*O(1)*, trie'den hızlı" saymak da yanlıştır; hash değerini hesaplamak da *L* karakteri okur.

Sık yapılan hata

Belleği hafife almak. Her düğüm 26 referans taşır; milyonlarca düğümde yüzlerce MB tutar. Seyrek bir trie'de `HashMap` , üçlü arama trie'si ya da sıkıştırma gerekir.

Sık yapılan hata

Geri izlemede paylaşılan durumu geri almamak. `topla` gibi özyinelemeli gezintilerde ortak bir `StringBuilder` kullanılıyorsa her çağrıdan sonra eklenen karakter silinmelidir. Aksi hâlde sonraki kardeşlerin kelimeleri bozuk çıkar.

5.3.10 Özet

- **Yapı:** trie bir dizgi kümesini köklü bir ağaçta saklar. Kök boş dizgidir, her kenar bir karakter taşır, her düğüm bir önektir. Kelime, kökten başlayan ve son düğümü işaretli bir yoldur. Ortak önekler tek kez saklanır; trie'nin şekli ekleme sırasına bağlı değildir.
- **Düğüm:** `TrieDugumu[26] cocuk` ve `boolean kelimeSonu`. Karakter düğümde saklanmaz; indis `ch - 'a'` ile bulunur. Alfabe dışı karakterler dizi sınırını aşar.
- **Ekleme:** çocuk yoksa oluşturulur, varsa ilerlenir; sonda son düğüm işaretlenir. $O(L)$; en çok L yeni düğüm.
- **Arama ve önek sorgusu:** dal yoksa `false` ; arama sonda `kelimeSonu` 'nu, önek sorgusu `true` 'yu döndürür. $O(L)$ ve $O(P)$. `gecen` sayacı önekle başlayan kelime sayısını $O(P)$'de verir.
- **Joker ve yazım denetimi:** joker karakter her dolu dalı dener. Düzenleme uzaklığı satırları trie üzerinde ebeveynden çocuğa hesaplanır; ortak öneklerin satırı bir kez hesaplanır, satırın en küçüğü sınırı aşan dallar budanır.
- **Otomatik tamamlama:** önek düğümü bulunur, alt ağaç derinlik öncelikli gezilir; sonuç alfabetik sıradadır. $O(P + S \cdot M)$. En uzun ortak önek, kökten tek çocuklu ve işaretsiz düğümler boyunca inilerek bulunur.
- **Silme:** işaret kaldırılır; çocuksuz ve işaretsiz düğümler aşağıdan yukarı silinir. Üç durum: kelime önekse yalnız işaret kalkar, kuyruğu varsa kuyruk silinir, kelime yoksa hiçbir şey değişmez. $O(L \cdot M)$, çocuk sayacıyla $O(L)$.
- **Bellek:** $O(N \cdot M)$; and, ant, dad, do trie'sinde 234 referansın 226'sı `null` 'dır. Seyrek düğümlerde `HashMap` , `TreeMap` , üçlü arama trie'si ya da sıkıştırılmış trie kullanılır.
- **Karşılaştırma:** tam aramada trie ve hash tablosu $O(L)$'dir; trie'nin farkı önek ve sıra sorgularındadır. Önce-kök gezinme kelimeleri sözlüksel sırada verir.
- **Türevler:** ikili trie (en uzun önek eşleşmesi, en büyük XOR), üçlü arama trie'si (düğüm başına 3 bağlantı), sıkıştırılmış trie ve Patricia (n kelime için en çok $2n - 1$ düğüm), sonek trie'si → sonek ağacı ve sonek dizisi, Aho–Corasick (hata bağlantılarıyla çok örüntülü arama, $O(n + m + z)$).

5.3.11 Kavram listesi

trie (önek ağacı) *trie, prefix tree* · önek *prefix* · sonek *suffix* · alt dizgi *substring* · kelime sonu işareti *end of word* · alfabe *alphabet* · referans *reference (child pointer)*

önek sorgusu *prefix query* | joker karakterli arama *wildcard search* | düzenleme uzaklığı *edit distance*
otomatik tamamlama *autocomplete* | geri izleme *backtracking* | derinlik öncelikli arama *depth-first search (DFS)*
en uzun ortak önek *longest common prefix* | sözlüksel sıra *lexicographic order* | hash tablosu *hash table*
basamak sıralaması *radix sort* | ikili trie *binary trie* | en uzun önek eşleşmesi *longest prefix match*
üçlü arama trie'si *ternary search trie (TST)* | sıkıştırılmış trie *compressed trie (radix tree)*
sonek trie'si / sonek ağacı *suffix trie / suffix tree* | sonek dizisi *suffix array*
Aho–Corasick algoritması *Aho–Corasick algorithm* | hata bağlantısı *failure link*

5.3.12 Kendini Dene

Sorular 1–7 aynı trie'yi kullanır: sel, selam, sen, ses ve su kelimeleri, 26 öğeli dizili trie'ye eklenmiştir.

1. Trie'de kök dahil kaç düğüm vardır? Toplam kaç çocuk referansı `null`'dır?

   Cevap

   Düğümler s, se, sel, sela, selam, sen, ses ve su'dur: 8 düğüm, kökle 9. Referans sayısı *9 × 26 = 234*'tür. Dolu referans kenar sayısı kadardır (8); *234 − 8 = 226* referans `null`'dır.

2. `ara("se")`, `onekVarMi("se")`, `ara("sela")` ve `onekVarMi("sela")` ne döndürür?

   Cevap

   Sırasıyla `false`, `true`, `false`, `true`. "se" ve "sela" yolları vardır, ama iki düğüm de işaretsizdir; ikisi de yalnız öneklerdir.

3. `tamamla("se")` hangi listeyi döndürür? Liste neden bu sıradadır?

   Cevap

   `[sel, selam, sen, ses]`. "se" işaretsiz olduğu için eklenmez. Çocuklar indis sırasıyla gezilir: l (11), n (13), s (18). "sel" düğümü kendisi işaretli olduğu için "selam"dan önce eklenir.

4. Kelimeler önek sayaçlı `ekle` ile eklenmiş olsun. Kökte, "se" ve "sel" düğümlerinde `gecen` değeri kaçtır?

   Cevap

   Kökte 5, "se"de 4 (su hariç hepsi), "sel"de 2 (sel ve selam). `onekSayisi("se")` 4 döndürür.

5. Her silme ilk trie'ye ayrı ayrı uygulanırsa `sil("sel")`, `sil("selam")` ve `sil("su")` kaçar düğüm siler?

Cevap

`sil("sel")`: 0; “sel”in 'a' çocuğu vardır, yalnız işaret kalkar. `sil("selam")`: 2; “selam” ve “sela” silinir, işaretli “sel” düğümünde durulur. `sil("su")`: 1; “su” silinir, “s” düğümünün 'e' çocuğu olduğu için durulur.

6. `enUzunOrtakOnek()` ne döndürür? “su” kelimesi trie'de olmasaydı sonuç ne olurdu?

Cevap

“s”. Kökün tek çocuğu 's'dir, ama “s” düğümünün iki çocuğu (e, u) vardır ve iniş durur. “su” olmasaydı “s”nin tek çocuğu 'e' olurdu; “se” düğümünün üç çocuğu (l, n, s) vardır ve iniş orada durur: “se”.

7. Sorgu “sez” ve $k = 1$ için trie üzerinde düzenleme uzaklığı araması hangi kelimeleri önerir? Hangi düğümde budama yapılır?

Cevap

Satırlar (“”, s, e, z sütunlarıyla): s → 1 0 1 2, se → 2 1 0 1, sel → 3 2 1 1, sen ve ses → 3 2 1 1, su → 2 1 1 2, sela → 4 3 2 2. Son hücresi 1 olan kelimeler sel, sen ve ses'tir; üçü de önerilir. “su”nun son hücresi 2'dir, önerilmez. “sela” satırının en küçüğü 2'dir ve budanır; “selam”a inilmez.

8. Aynı beş kelimenin sıkıştırılmış trie'sinde kök hariç kaç düğüm vardır? Teoremdeki üst sınır kaçtır?

Cevap

s (iki çocuklu), se (üç çocuklu), sel, selam (“am” kenarıyla), sen, ses ve su: 7 düğüm. Standart trie'deki “sela” düğümü “am” kenarına katılmıştır. Sınır $2 \cdot 5 - 1 = 9$'dur.

9. [2, 9, 12, 5] dizisinde XOR değeri en büyük çift hangisidir? 4 bitlik ikili trie'de 2 için hangi sayıya varılır?

Cevap

Gösterimler 0010, 1001, 1100, 0101'dir. 2 için istenen 1101'dir: bit 3'te 1 (9 ve 12 var), bit 2'de 1 (yalnız 12), bit 1'de 0 (12'nin biti 0), bit 0'da 1 istenir ama 12'nin biti 0'dır. Varılan sayı 12'dir ve $2 \oplus 12 = 1110 = 14$. En büyük değer 14'tür (2 ile 12).

10. “aab” metninin sonek trie'sinde kök hariç kaç düğüm vardır? “ba” ve “ab” alt dizgi midir?

Cevap

Sonekler aab, ab, b'dir. Düğümler a, aa, aab, ab, b: 5 düğüm; bu sayı metnin farklı alt dizgilerinin sayısıdır (“a” iki kez geçer, bir kez sayılır). “ab” kökten a → b yoludur, alt dizgidir. “ba” için “b”nin hiç çocuğu yoktur; alt dizgi değildir.

5.3.13 İleri okuma

- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 5.2: R'li trie'ler ve üçlü arama trie'leri için Java kodunu, silmeyi ve bellek ölçümlerini ayrıntılı verir.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (1997): sonek ağaçlarını, Ukkonen algoritmasını ve Aho–Corasick'i biyolojik dizi uygulamalarıyla birlikte işler.
- J. L. Bentley ve R. Sedgewick, “Fast Algorithms for Sorting and Searching Strings”, *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1997: üçlü arama trie'sinin ve ona dayalı üç yollu dizgi hızlı sıralamasının analizini yapar.
- A. V. Aho ve M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, *Communications of the ACM* 18(6), 1975: hata bağlantılı çok örüntülü eşleme algoritmasının özgün makalesidir.

5.3.14 Kaynaklar

1. A. V. Aho, M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, *Communications of the ACM* 18(6), 1975.
2. J. L. Bentley, R. Sedgewick, “Fast Algorithms for Sorting and Searching Strings”, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
4. E. Fredkin, “Trie Memory”, *Communications of the ACM* 3(9), 1960.
5. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
6. U. Manber, G. Myers, “Suffix Arrays: A New Method for On-Line String Searches”, *SIAM Journal on Computing* 22(5), 1993.
7. D. R. Morrison, “PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric”, *Journal of the ACM* 15(4), 1968.
8. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 5.2: Tries).
9. E. Ukkonen, “On-line Construction of Suffix Trees”, *Algorithmica* 14(3), 1995.

KISIM III

Tasarım Paradigmaları

Bölüm 6–8

KISIM III · BÖLÜM 6

Böl ve Fethet Yaklaşımı

Büyük bir problemi doğrudan çözmek çoğu zaman zordur. Aynı problemin yarısı ise daha kolaydır; dörtte biri daha da kolaydır. Bir noktada problem o kadar küçülür ki çözümü apaçık olur. Böl ve fethet yaklaşımı bu gözleme dayanır: problem aynı türden küçük parçalara bölünür, parçalar aynı yöntemle çözülür ve çözümler birleştirilerek asıl soru yanıtlanır. Sıralama ve aramadan tanıdık birkaç algoritma da bu kalıba uyar. Bu not önce yaklaşımın adımlarını, genel şablonunu ve çalışma süresinin yineleme bağıntısıyla nasıl hesaplandığını anlatır. Ardından tanıdık algoritmalara paradigma gözüyle kısaca bakar. Asıl ağırlık dört yeni algoritmadadır: Quickhull ile dışbükey örtü, şerit yöntemiyle en yakın nokta çifti, Karatsuba tam sayı çarpımı ve Strassen matris çarpımı. Her algoritma için kod, adım adım izleme ve maliyet hesabı verilir.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum06/src`

ÖĞRENME ÇIKTILARI

Paradigma
Böl ve fethet yaklaşımının adımlarını açıklamak; bir problemin bu yaklaşıma uygun olup olmadığına, özellikle alt problemlerin bağımsız mı yoksa örtüşen mi olduğuna bakarak karar vermek.

Analiz
Bir böl ve fethet algoritmasının yineleme bağıntısını kurmak; bağıntıyı ana teoremle ya da özyineleme ağacıyla çözmek; ana teoremin uygulanamadığı durumları tanımak.

Geometrik algoritmalar
Quickhull ile dışbükey örtüyü ve şerit yöntemiyle en yakın nokta çiftini adım adım bulmak; yönelim testini yalnız tam sayı çarpımıyla hesaplamak; şeritte en çok yedi karşılaştırmanın neden yettiğini gerekçelendirmek.

Hızlı çarpma
Karatsuba (tam sayı) ve Strassen (matris) çarpımının, özyinelemeli çağrı sayısını bir azaltarak süreyi nasıl düşürdüğünü göstermek; iki algoritmayı küçük örneklerde elle çalıştırmak.

6.1 Böl ve fethet paradigması

Bir algoritma tasarlarken iki şey düşünülür: problemin kendisi ve onu çözmek için izlenecek genel strateji. Strateji düzeyindeki kalıplara **paradigma** denir. Kaba kuvvet, açgözlü yaklaşım, dinamik programlama ve geri izleme böyle kalıplardır. Böl ve fethet bunların en eskilerinden ve en verimlilerinden biridir. Adı askerî bir deyimden gelir: büyük bir ordu bütün hâlinde yenilemiyorsa küçük birliklere ayrılır ve birlikler tek tek yenilir. Algoritmalarda "ordu" büyük problemdir, "birlikler" de onun küçük kopyalarıdır.

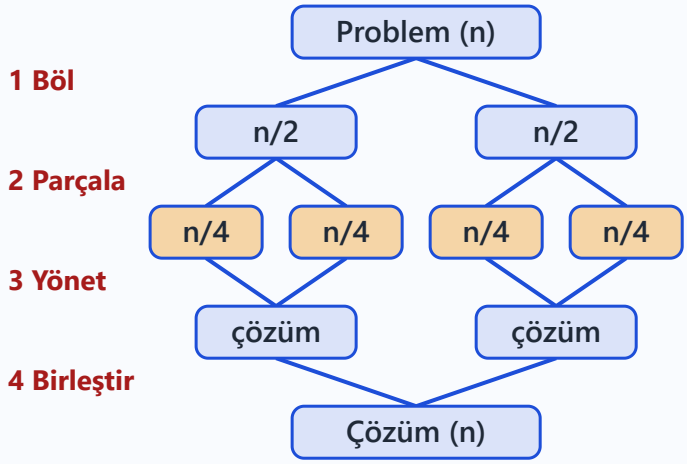
6.1.1 Böl, parçala, yönet, birleştir

Tanım 6.1

Böl ve fethet (*divide and conquer*), bir problemi aynı türden, daha küçük ve birbirinden bağımsız alt problemlere ayıran bir algoritma tasarım yaklaşımıdır. Alt problemler aynı yöntemle özyinelemeli olarak çözülür; bu çözümler birleştirilerek asıl problemin çözümü kurulur. Alt problem doğrudan çözülebilecek kadar küçüldüğünde bölme durur; bu duruma **taban durum** (*base case*) denir.

Yaklaşım dört adımla anlatılır. Ders kitaplarının çoğu yalnız üç adım sayar (böl, yönet, birleştir). Dört adımlı anlatım, bölmenin tek seferde değil, taban duruma kadar tekrar tekrar yapıldığını ayrıca vurgular:

1. **Böl:** Karmaşık problem daha küçük, yönetilebilir alt problemlere bölünür. Örneğin n öğeli bir dizi, $n/2$ öğeli iki yarıya ayrılır.
2. **Parçala:** Alt problemler, doğrudan çözülebilecek kadar küçülene dek bölünmeye devam eder. Bu tekrar özyinelemeyle sağlanır: alt probleme aynı fonksiyon uygulanır ve o fonksiyon da kendi girdisini böler.
3. **Yönet:** Her alt problem ayrı ayrı çözülür. Taban durumdaki alt problemler doğrudan, ötekiler özyinelemeli çağrıyla çözülür.
4. **Birleştir:** Alt problemlerin çözümleri birleştirilerek asıl problemin çözümü kurulur.



Şekil 6.1: Böl ve fethetin genel görünümü. *n* boyutlu problem önce iki *n/2*'lik, sonra dört *n/4*'lük alt probleme bölünür. En küçük alt problemler doğrudan çözülür. Çözümler aşağıdan yukarıya birleştirilerek asıl çözüm elde edilir.

Şekil 6.1 bu akışı bir ağaç olarak gösterir. Ağacın kökü asıl problemdir. Her iç düğüm bir alt problemdir ve çocukları onun bölündüğü parçalardır. Yapraklar taban durumlardır. Algoritma yukarıdan aşağıya bölerek iner, aşağıdan yukarıya birleştirerek çıkar. Programda bu ağacın karşılığı **özyineleme ağacıdır**: her düğüm bir fonksiyon çağrısıdır.

Yaklaşımın işlemesi için alt problemlerin iki özellik taşıması gerekir. Birincisi, alt problem asıl problemle *aynı türdendir*; yalnız daha küçüktür. Bir diziyi sıralamak, yarısını sıralamakla aynı türden bir iştir. Bu yüzden aynı fonksiyon kendini çağırabilir ve ayrı bir "yarım dizi sıralama" fonksiyonu gerekmez. İkincisi, alt problemler *birbirinden bağımsızdır*. Sol yarıyı sıralamak sağ yarıyı sıralamaktan hiçbir bilgi istemez. Bu bağımsızlık bozulursa, yani alt problemler ortak alt alt problemler içerirse, aynı iş tekrar tekrar yapılır. Fibonacci özyinelemesi bunun en bilinen örneğidir.

6.1.2 Neden böl ve fethet?

Böl ve fethet ilk bakışta yalnız bir düzenleme yöntemi gibi görünür: işi parçalara ayırıp parçaları tek tek yapmak. Oysa yaklaşımın asıl gücü, uygun problemlerde çalışma süresini mertebe olarak düşürebilmesidir. Bu notta ele alınan algoritmaların hemen hepsi, aynı problemi çözen saf yöntemden asimptotik olarak daha hızlıdır. Örnekler: sıralamada $O(n^2)$ yerine $O(n \log n)$, sıralı dizide aramada $O(n)$ yerine $O(\log n)$, tam sayı çarpımında $O(n^2)$ yerine yaklaşık $O(n^{1{,}585})$.

Yaklaşımın başka yararları da vardır:

- **Anlaşılırlık ve yeniden kullanım.** Karmaşık problem küçük, yönetilebilir parçalara ayrılınca daha anlaşılır olur. Alt problemlerin çözüm yöntemi her düzeyde aynı fonksiyondur. Kod kısa kalır ve her düzeyde yeniden kullanılır.
- **Paralellik.** Alt problemler birbirinden bağımsız olduğu için farklı işlemci çekirdeklerinde aynı anda çözülebilir. Sol ve sağ yarının sıralanması birbirini beklemez.

- **Bellek yerelliği.** Alt problemler küçüldükçe önbelleğe sığar. Küçük bir alt dizi önbelleğe bir kez yüklenir ve üzerindeki bütün iş orada biter. Bu, büyük veride gözle görülür bir hız kazancı sağlar.
- **Kolay doğruluk kanıtı.** Özyinelemeli yapı, doğruluğun tümevarımla kanıtlanmasını doğal kılar: taban durum doğruysa ve birleştirme, doğru alt çözümlerden doğru çözüm kuruyorsa algoritma doğrudur.

Yaklaşımın bedelleri de bilinmelidir:

- **Özyineleme yükü.** Her çağrı çağrı yığınında bir çerçeve açar ve bunun zaman ve bellek maliyeti vardır. Bazı algoritmalar ek bellek de ister; örneğin birleştirme sıralaması $O(n)$ ek dizi kullanır.
- **Örtüşen alt problemler.** Alt problemler bağımsız değilse aynı iş tekrar tekrar yapılır ve süre üstel olabilir.
- **Küçük girdilerde sabit çarpanlar.** Asimptotik kazanç büyük n'de görünür. Küçük n'de bölme, birleştirme ve çağrı yükü baskın çıkar. Bu yüzden pratikte bir **eşik** belirlenir; alt problem eşiğin altına düşünce basit bir yönteme geçilir. Karatsuba ve Strassen uygulamaları böyle çalışır.

Böl ve fethetin kullanıldığı alanlar geniştir. Sıralama ve arama, hesaplamalı geometri, büyük sayı ve matris aritmetiği bunların başında gelir. Hızlı Fourier dönüşümü ve oyun ağaçlarında arama da aynı kalıbı izler.

6.1.3 Genel şablon

Böl ve fethet algoritmalarının hepsi aynı iskeleti izler. Aşağıdaki Java kodu bu iskeleti problemden bağımsız bir biçimde yazar. `Problem` ve `Sonuc` türleri, `bol()`, `dogrudanCoz()` ve `birlestir()` metotları her algoritmada başka bir anlam kazanır.

```
Sonuc bolVeFethet(Problem p) {
  if (p.boyut() <= ESIK)
    return dogrudanCoz(p);          // taban durumu
  Problem[] alt = bol(p);           // a alt problem, boyut n/b
  Sonuc[] cozum = new Sonuc[alt.length];
  for (int i = 0; i < alt.length; i++)
    cozum[i] = bolVeFethet(alt[i]); // yönet: özyineleme
  return birlestir(cozum);          // birleştir
}
```

Şablonun satırları paradigmanın adımlarıyla birebir eşleşir:

- `if (p.boyut() <= ESIK) return dogrudanCoz(p);` taban durumdur. Problem eşikten büyük değilse bölme durur ve problem doğrudan çözülür. Eşik çoğu zaman 1'dir: tek öğeli dizi zaten sıralıdır, tek öğenin en büyüğü kendisidir. Eşik daha büyük de seçilebilir; en yakın nokta çiftinde eşik 3'tür.
- `Problem[] alt = bol(p);` böl adımıdır. Problem a tane alt probleme ayrılır; her birinin boyutu n/b civarındadır.

- `for` döngüsü yönet adımıdır. Her alt problem için şablonun kendisi çağrılır. Parçala adımı ayrı bir satır değildir; bu özyinelemeli çağrının içinde, alt problemin yeniden bölünmesiyle gerçekleşir.
- `return birlestir(cozum);` birleştir adımıdır. Alt çözümler asıl çözüme dönüştürülür.

Şablonun çalışma süresi de aynı yapıdan okunur. *T(n)*, *n* boyutlu problemi çözmenin süresi olsun. Algoritma *a* tane *n/b* boyutlu alt problemi özyinelemeyle çözer; bunların toplam süresi $a \cdot T(n/b)$'dir. Bölme ve birleştirme için harcanan süre *f(n)* ile gösterilir. Böylece

$$T(n) = a \cdot T(n/b) + f(n)$$

bağıntısı elde edilir. Bu bağıntıda *a* alt problem sayısı, *n/b* alt problem boyutu, *f(n)* de bölme ve birleştirme maliyetidir. Bağıntı, bir böl ve fethet algoritmasının maliyetini üç sayıyla özetler. Algoritmayı hızlandırmanın yolları da bu üç sayıdan okunur:

- **Taban durum olmadan özyineleme bitmez.** Bağıntı yalnız büyük *n* için geçerlidir; küçük *n*'de *T(n)* sabit bir değerdir. Taban durum unutulursa program sonsuz özyinelemeye girer ve `StackOverflowError` ile durur.
- **Bölme dengeli olmalıdır.** *n/2 + n/2* biçimindeki bölme iyidir: problem her düzeyde yarıya iner ve yaklaşık $\log_2 n$ düzeyde taban duruma varılır. *1 + (n − 1)* biçimindeki bölme kötüdür: problem her düzeyde yalnız bir öğe küçülür ve *n* düzey gerekir.
- **Hız kazancı çoğu zaman *a*'yı azaltmaktan ya da *f(n)*'yi küçültmekten gelir.** Karatsuba ve Strassen algoritmalarının bütün kazancı *a*'yı bir azaltmaktan gelir. Birleştirme maliyetini *O(n log n)*'den *O(n)*'ye indirmek de en yakın nokta çiftinde belirleyici olur.

6.1.4 Şablonun uygulaması: en büyük öğe

Şablonu somut bir örnekte görmek için çok basit bir problem seçilir: bir dizinin en büyük öğesini bulmak. Bu problem tek bir döngüyle de çözülür; böl ve fethet burada süreyi kısaltmaz. Örneğin amacı, şablonun her parçasının koddaki karşılığını göstermektir.

Böl ve fethet düşüncesiyle problem şöyle kurulur. Dizinin en büyük öğesi, sol yarının en büyüğü ile sağ yarının en büyüğünden büyük olanıdır. Bu cümle problemi aynı türden iki küçük probleme indirger. Tek öğeli bir aralıkta en büyük öğe o öğenin kendisidir; bu da taban durumdur.

```
int enBuyuk(int[] a, int alt, int ust) {
  if (alt == ust) return a[alt];    // taban
  int orta = (alt + ust) / 2;
  int sol = enBuyuk(a, alt, orta);
  int sag = enBuyuk(a, orta + 1, ust);
  return Math.max(sol, sag);        // birleştir
}
```

Metot dizinin `a[alt..ust]` aralığıyla çalışır. Bütün dizi için `enBuyuk(a, 0, n - 1)` çağrılır. Satırların görevi şöyledir:

- `if (alt == ust) return a[alt];` taban durumdur. Aralıkta tek öğe vardır ve en büyük öğe odur.
- `int orta = (alt + ust) / 2;` böl adımıdır. Aralık ortadan ikiye ayrılır: `[alt..orta]` ve `[orta+1..ust]`. Tam sayı bölmesi aşağı yuvarladığı için öğe sayısı tekse sol yarı bir öğe fazla alır.
- İki özyinelemeli çağrı yönet adımıdır: `sol` sol yarının, `sag` sağ yarının en büyüğüdür.
- `return Math.max(sol, sag);` birleştir adımıdır ve tek bir karşılaştırmadır.

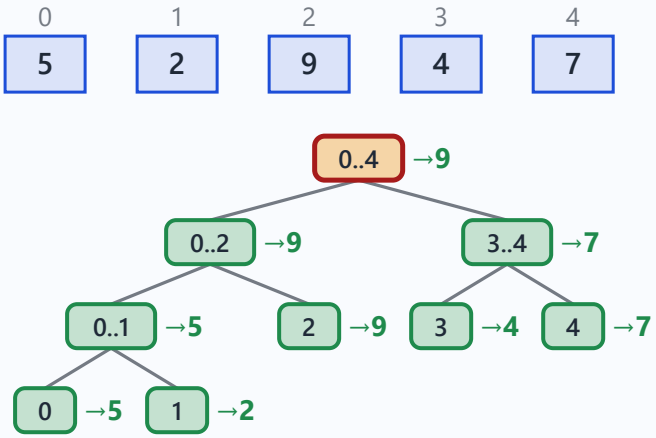
Bölme yeni dizi oluşturmaz; yalnız iki indis hesaplar. Bu yüzden bölme ve birleştirme birlikte sabit sürelidir: $f(n) = O(1)$. Bağıntı $T(n) = 2T(n/2) + O(1)$ olur ve çözümü $\Theta(n)$'dir.

6.1.5 Kod izleme: enBuyuk(a, 0, 4)

İzlemede dizi `a = [5, 2, 9, 4, 7]`'dir ve `enBuyuk(a, 0, 4)` çağrılır. Aralıkta beş öğe vardır. Orta indis $(0 + 4)/2 = 2$'dir; sol yarı `[0..2]`, sağ yarı `[3..4]` olur. Sol yarı üç öğe, sağ yarı iki öğe alır. Tablo 6.1 çağrıların sırasını ve her çağrının döndürdüğü değeri verir.

Adım	Olay	Çağrı yığını	Dönüş
1	`(0, 4)`: orta = 2, önce sol yarı `[0..2]`	(0,4)	—
2	`(0, 2)`: orta = 1, önce `[0..1]`, sonra `[2..2]`	(0,4), (0,2)	—
3	`(0, 1)`: orta = 0, önce `[0..0]`, sonra `[1..1]`	(0,4), (0,2), (0,1)	—
4	`(0, 0)`: tek öğe, taban durum	…, (0,1), (0,0)	5
5	`(1, 1)`: tek öğe, taban durum	…, (0,1), (1,1)	2
6	`(0, 1)`: sol = 5, sağ = 2 → max	(0,4), (0,2), (0,1)	5
7	`(2, 2)`: tek öğe, taban durum	(0,4), (0,2), (2,2)	9
8	`(0, 2)`: sol = 5, sağ = 9 → max	(0,4), (0,2)	9
9	`(3, 4)`: orta = 3, önce `[3..3]`, sonra `[4..4]`	(0,4), (3,4)	—
10	`(3, 3)`: tek öğe, taban durum	(0,4), (3,4), (3,3)	4
11	`(4, 4)`: tek öğe, taban durum	(0,4), (3,4), (4,4)	7
12	`(3, 4)`: sol = 4, sağ = 7 → max	(0,4), (3,4)	7
13	`(0, 4)`: sol = 9, sağ = 7 → max	(0,4)	9

Tablo 6.1: `enBuyuk(a, 0, 4)` izlemesi, `a = [5, 2, 9, 4, 7]`. Çağrılar `(alt, ust)` çiftiyle gösterilir; çağrı yığını soldan sağa büyür.



Şekil 6.2: `enBuyuk(a, 0, 4)` çağrısının özyineleme ağacı. Her düğüm bir `alt..ust` aralığıdır; okun yanındaki sayı o çağrının döndürdüğü değerdir. Beş yaprak taban durumdur; dört iç düğümün her biri bir karşılaştırma yapar.

İzlemede çağrıların sırası dikkat ister. Bir çağrı, sol yarının işi tamamen bitmeden sağ yarıya geçmez. 3. adımda `(0, 1)` çağrısı başlar; 4. ve 5. adımda iki yaprağı çözer ve ancak 6. adımda sonucunu döndürür. Sağ yarı `(3, 4)` ise 9. adıma kadar hiç başlamaz. Bu sıra, özyineleme ağacının *derinlik öncelikli* gezilmesidir. Çağrı yığını hiçbir zaman ağacın düzey sayısından fazla çerçeve tutmaz; burada en çok dört çerçeve vardır (`(0,4)` , `(0,2)` , `(0,1)` ve bir yaprak).

Sonuç 9'dur. Toplam çağrı sayısı 9'dur: 5 yaprak ve 4 iç düğüm. Yapılan karşılaştırma sayısı iç düğüm sayısına, yani 4'e eşittir. Genel olarak n öğeli dizide özyineleme ağacının n yaprağı vardır. Her iç düğümün tam iki çocuğu olduğu için iç düğüm sayısı $n - 1$'dir. Algoritma $n - 1$ karşılaştırma ve $2n - 1$ çağrı yapar.

Basit bir döngü de diziyi baştan sona gezerek $n - 1$ karşılaştırmayla en büyük öğeyi bulur. Böl ve fethet karşılaştırma sayısını azaltmamıştır; üstelik çağrı yükü eklemiştir. O hâlde bu sürümün yararı nedir? Yanıt paralelliktir. İki yarı birbirinden bağımsızdır ve aynı anda çözülebilir. Sınırsız sayıda işlemci varsa ağacın her düzeyi tek adımda biter. Ağacın yüksekliği $\lceil \log_2 n \rceil$ olduğundan sonuç $O(\log n)$ paralel adımda bulunur. Döngüyle yazılan sürümde her karşılaştırma bir öncekinin sonucunu bekler ve bu zincir kısaltılamaz.

Sık yapılan hata

Böl ve fethetin her problemde süreyi kısalttığını sanmak. En büyük öğe probleminde iki yarının da bütünüyle incelenmesi gerekir; hiçbir öğe atlanamaz. Bu yüzden toplam iş n'den aşağı inemez. Kazanç üç durumda ortaya çıkar. Alt problemlerden biri atılabilir (ikili arama). Alt problem sayısı azaltılabilir (Karatsuba, Strassen). Ya da birleştirme saf yöntemden ucuza yapılabilir (birleştirme sıralaması, en yakın nokta çifti).

6.1.6 Doğruluk ve tümevarım

Böl ve fethet algoritmalarının doğruluğu, özyinelemenin yapısını izleyen bir tümevarımla kanıtlanır. Tümevarım problem boyutu üzerinden yapılır. Önce taban durumda algoritmanın doğru olduğu gösterilir. Sonra daha küçük bütün girdilerde doğru olduğu kabul edilir ve birleştirme adımının doğru sonuç ürettiği gösterilir. `enBuyuk` için bu şöyle yapılır.

> **Teorem 6.1 (enBuyuk doğrudur)**
>
> *alt ≤ ust* olan her aralık için `enBuyuk(a, alt, ust)`, `a[alt..ust]` aralığının en büyük öğesini döndürür.

İspat. Tümevarım aralıktaki öğe sayısı *k = ust − alt + 1* üzerinden yapılır. *k = 1* ise *alt = ust*'tür ve metot `a[alt]`'ı döndürür; tek öğeli aralığın en büyüğü odur. *k ≥ 2* olsun ve metodun *k*'den az öğeli her aralıkta doğru çalıştığı kabul edilsin. `orta = ⌊(alt + ust)/2⌋` için *alt ≤ orta < ust*'tür. Bu yüzden iki yarı da boş değildir ve her biri *k*'den az öğe içerir. Tümevarım varsayımına göre `sol` sol yarının, `sag` sağ yarının en büyüğüdür. Aralığın her öğesi iki yarıdan birindedir; o hâlde aralığın en büyüğü `max(sol, sag)`'dir. ■

Kanıtta kullanılan *alt ≤ orta < ust* eşitsizliği önemlidir. Bölme, iki yarıdan birini boş ya da aralığın tamamı kadar bırakırsa tümevarım varsayımı kullanılamaz ve program sonsuz özyinelemeye girebilir. Örneğin `orta` yukarı yuvarlanıp sol yarı `[alt..orta]`, sağ yarı `[orta+1..ust]` alınsaydı, iki öğeli aralıkta `orta = ust` olur ve sağ yarı boş kalırdı. Her böl ve fethet algoritmasında aynı soru sorulmalıdır: her alt problem gerçekten daha küçük müdür ve boş alt problem nasıl ele alınır?

6.2 Yineleme bağıntıları

Özyinelemeli bir algoritmanın süresi kendi cinsinden yazılır: *n* boyutlu problemin süresi, daha küçük problemlerin süreleri ve bölme ile birleştirme maliyeti cinsinden ifade edilir. Bu tür denkleme **yineleme bağıntısı** (*recurrence relation*) denir. Bu kesim, bağıntıların kurulması ve ana teoremle çözülmesi hakkında böl ve fethet için gerekenleri kısaca hatırlatır.

6.2.1 Bağıntıyı kurmak

Bağıntı, algoritmanın kodundan üç soru sorularak okunur:

1. Kaç özyinelemeli çağrı yapılıyor? Bu sayı *a*'dır. Burada yalnız gerçekten çalışan çağrılar sayılır. İkili arama iki ayrı çağrı satırı içerir, ama her yürütmede yalnız biri çalışır; bu yüzden *a = 1*'dir.
2. Her çağrının girdisi ne kadar? Girdi *n/b* ise *b* okunur. Girdi *n − 1* gibi sabit miktarda küçülüyorsa bağıntı ana teoremin biçimine uymaz.
3. Çağrılar dışında ne kadar iş yapılıyor? Bölme ve birleştirmenin toplam maliyeti *f(n)*'dir. Döngüler, dizi kopyalama, alt liste oluşturma bu maliyete girer.

Örnek 6.1

Aşağıdaki algoritmaların bağıntıları kodlarına bakılarak yazılır.

- `enBuyuk`: iki çağrı, her biri yarım aralıkla; dışarıda bir karşılaştırma: $T(n) = 2T(n/2) + O(1)$.
- Birleştirme sıralaması: iki çağrı, her biri yarım diziyle; dışarıda iki yarıyı birleştiren doğrusal bir döngü: $T(n) = 2T(n/2) + O(n)$.
- İkili arama: tek çağrı (sol ya da sağ yarı), sabit sayıda karşılaştırma: $T(n) = T(n/2) + O(1)$.
- Hızlı üs alma: `us(x, n)` bir kez `us(x, n/2)` 'yi çağırır ve en çok iki çarpma yapar: $T(n) = T(n/2) + O(1)$. Burada n girdinin uzunluğu değil, üssün değeridir.
- Saf özyinelemeli Fibonacci: iki çağrı, ama girdiler $n - 1$ ve $n - 2$'dir: $T(n) = T(n-1) + T(n-2) + O(1)$. Bu bağıntı ana teoremin biçiminde değildir.

6.2.2 Ana teorem

Ana teorem, $T(n) = aT(n/b) + f(n)$ biçimindeki bağıntıları hesap yapmadan çözmeye yarar. Teorem, kökteki işi temsil eden $f(n)$'yi, yaprakların toplam işini temsil eden $n^{\log_b a}$ ile karşılaştırır. Bu ikinci değere yaprak işi, üsteki $\log_b a$'ya da **kritik üs** denir: özyineleme ağacında $\log_b n$ düzey vardır ve her düzeyde düğüm sayısı a katına çıkar; yaprak sayısı $a^{\log_b n} = n^{\log_b a}$'dır.

Teorem 6.2 (Ana teorem)

$a \geq 1$ ve $b > 1$ sabitler, $f(n)$ asimptotik olarak pozitif bir fonksiyon olsun ve $T(n) = aT(n/b) + f(n)$ verilsin. $c = \log_b a$ denirse:

1. Bir $\varepsilon > 0$ için $f(n) = O(n^{c-\varepsilon})$ ise $T(n) = \Theta(n^c)$'dir. (Durum 1: yapraklar baskın.)
2. $f(n) = \Theta(n^c)$ ise $T(n) = \Theta(n^c \log n)$'dir. (Durum 2: her düzey eşit.)
3. Bir $\varepsilon > 0$ için $f(n) = \Omega(n^{c+\varepsilon})$ ise ve bir $k < 1$ sabiti ile yeterince büyük her n için $a \cdot f(n/b) \leq k \cdot f(n)$ düzenlilik koşulu sağlanıyorsa $T(n) = \Theta(f(n))$'dir. (Durum 3: kök baskın.)

Teoremin dili ilk bakışta ağırdır, ama kullanımı kısadır: $f(n)$ ile $n^{\log_b a}$ karşılaştırılır; büyük olan kazanır; ikisi aynı mertebedeyse sonuca bir $\log n$ çarpanı eklenir. “Polinom olarak küçük” ifadesi, aradaki farkın en az n^{ε} gibi bir kuvvet olması demektir. Yalnız bir $\log n$ çarpanı kadar fark, polinom farkı sayılmaz.

Örnek 6.2

Üç bağıntı ana teoremle çözülür.

1. $T(n) = 4T(n/2) + n$: $a = 4$, $b = 2$, $n^{\log_2 4} = n^2$. $f(n) = n$, n^2'den polinom olarak küçüktür ($\varepsilon = 1$). Durum 1: $T(n) = \Theta(n^2)$.
2. $T(n) = 2T(n/2) + n^2$: $n^{\log_2 2} = n$. $f(n) = n^2$ polinom olarak büyüktür. Düzenlilik koşulu: $2 \cdot (n/2)^2 = n^2/2 \le \tfrac{1}{2} \cdot n^2$; $k = \tfrac{1}{2}$ ile sağlanır. Durum 3: $T(n) = \Theta(n^2)$.
3. $T(n) = 2T(n/2) + 1$ (`enBuyuk`): $n^{\log_2 2} = n$, $f(n) = 1 = n^0$. Fark n^1'dir. Durum 1: $T(n) = \Theta(n)$. Bu sonuç, $n - 1$ karşılaştırma ve $2n - 1$ çağrı sayımıyla uyumludur.

6.2.3 Özyineleme ağacında düzey toplamları

Ana teoremin üç durumu, özyineleme ağacının düzeylerindeki toplam işe bakılınca kendiliğinden anlaşılır. i. düzeyde a^i alt problem vardır ve her birinin boyutu n/b^i'dir. Bu düzeyde yapılan toplam iş $a^i \cdot f(n/b^i)$'dir. Algoritmanın toplam süresi, bütün düzeylerin toplamıdır. Düzey toplamları kökten yapraklara doğru ya büyür, ya sabit kalır, ya da küçülür. Tablo 6.2 $n = 16$ için beş bağıntının düzey toplamlarını verir.

Bağıntı	0	1	2	3	4	Toplam	Eğilim
$2T(n/2) + n$	16	16	16	16	16	80	sabit (durum 2)
$3T(n/2) + n$	16	24	36	54	81	211	×1,5 (durum 1)
$4T(n/2) + n$	16	32	64	128	256	496	×2 (durum 1)
$7T(n/2) + n^2$	256	448	784	1372	2401	5261	×1,75 (durum 1)
$2T(n/2) + n^2$	256	128	64	32	16	496	×0,5 (durum 3)

Tablo 6.2: $n = 16$ için özyineleme ağacının düzey toplamları $a^i \cdot f(16/2^i)$. Düzey 0 kök, düzey 4 yapraklardır.

Tablo üç davranışı yan yana gösterir. İlk satırda her düzey aynı işi yapar: 16. Düzey sayısı $\log_2 n + 1$ olduğundan toplam $n \log n$ mertebesindedir; bu durum 2'dir ve birleştirme sıralamasının bağıntısıdır. İkinci, üçüncü ve dördüncü satırlarda düzey toplamları geometrik olarak büyür. Geometrik dizide toplam, son terimin sabit katıdır; son terim de yaprak düzeyidir. $3T(n/2) + n$ için yaprak düzeyi $3^4 = 81 = 16^{\log_2 3}$'tür. Karatsuba bağıntısı budur ve süresi $\Theta(n^{\log_2 3})$'tür. Son satırda düzey toplamları yarıya iner. Toplam bu kez ilk terimin, yani kökün sabit katıdır: $\Theta(n^2)$.

Bu bakış, böl ve fethette hızlanmanın nereden geldiğini de açıklar. $4T(n/2) + n$ ile $3T(n/2) + n$ arasındaki tek fark, her düzeyde bir alt problemin eksik olmasıdır. Ama bu fark her düzeyde katlanır: $n = 16$'da yaprak düzeyinde 256 yerine 81 iş kalır. n büyüdükçe oran $(4/3)^{\log_2 n} = n^{2 - 1{,}585}$ gibi büyür.

6.2.4 Bu bölümün algoritmaları ve ana teorem

Tablo 6.3 bu notta ele alınan algoritmaların bağıntılarını ana teoremle çözer. Hesapların ortak yanı, a, b ve $f(n)$'nin koddan okunması ve kritik üssün $f(n)$'nin üssüyle karşılaştırılmasıdır.

Algoritma	a	b	f(n)	$n^{\log_b a}$	Durum	T(n)
İkili arama, hızlı üs alma	1	2	1	$n^0 = 1$	2	$\Theta(\log n)$
Birleştirme sıralaması, en yakın çift, dengeli hızlı sıralama	2	2	n	n	2	$\Theta(n \log n)$
En büyük öğe	2	2	1	n	1	$\Theta(n)$
Karatsuba	3	2	n	$n^{1,585}$	1	$\Theta(n^{1,585})$
Saf blok matris çarpımı	8	2	n^2	n^3	1	$\Theta(n^3)$
Strassen	7	2	n^2	$n^{2,807}$	1	$\Theta(n^{2,807})$

Tablo 6.3: Böl ve fethet algoritmalarının bağıntıları ve ana teoremle çözümleri.

Tablonun son üç satırı ortak bir ders taşır. Karatsuba ve Strassen'in bütün kazancı a'yı bir azaltmaktan gelir: 4'ten 3'e ve 8'den 7'ye. Bölme ve birleştirme maliyeti $f(n)$ değişmez, hatta biraz artar. Durum 1'de süre yalnız kritik üsse bağlıdır ve kritik üs $\log_b a$'dır. a küçülünce üs de küçülür: $\log_2 4 = 2$ iken $\log_2 3 \approx 1{,}585$; $\log_2 8 = 3$ iken $\log_2 7 \approx 2{,}807$.

6.2.5 Ana teoremin uygulanamadığı bağıntılar

Ana teorem güçlüdür ama dar bir kalıba uyar: bütün alt problemler aynı boyutta olmalı ve boyut bir *oranla* küçülmelidir. Böl ve fethet algoritmalarının bir kısmı bu kalıba uymaz:

- **Sabit miktarda küçülme.** $T(n) = T(n-1) + n$ bağıntısında b yoktur. Dengesiz hızlı sıralama ve en kötü durumdaki Quickhull böyle davranır. Bağıntı açılınca $n + (n-1) + \ldots + 1 = n(n+1)/2 = \Theta(n^2)$ bulunur.
- **Eşit olmayan bölmeler.** Quickhull'un bağıntısı $T(n) = T(n_1) + T(n_2) + O(n)$'dir ve n_1, n_2 girdiye göre değişir. $T(n) = T(n/3) + T(2n/3) + n$ gibi sabit oranlı ama eşit olmayan bölmeler de teoremin dışında kalır. Bunlar özyineleme ağacıyla çözülür: son örnekte her düzeyin toplamı en çok n'dir ve en uzun dal $\log_{3/2} n$ düzeydir; sonuç $\Theta(n \log n)$'dir.
- **Durumlar arasında kalan $f(n)$.** $T(n) = 2T(n/2) + n/\log n$'de $f(n) = n/\log n$, n'den küçüktür ama polinom olarak küçük değildir: fark yalnız bir $\log n$ çarpanıdır. Teoremin hiçbir durumu uygulanmaz. Benzer biçimde en yakın nokta çiftinin hatalı sürümündeki $2T(n/2) + n \log n$ bağıntısı da ana teoremin üç durumunun dışındadır; çözümü $\Theta(n \log^2 n)$'dir.

> **Sık yapılan hata**
>
> Ana teoremi kalıba uymayan bir bağıntıya zorla uygulamak. $T(n) = T(n-1) + n$ için "$a = 1$, $f(n) = n$, durum 3, sonuç $\Theta(n)$" demek yanlıştır; doğru sonuç $\Theta(n^2)$'dir. Teorem uygulanmadan önce bağıntının $aT(n/b) + f(n)$ biçiminde olduğu, $b > 1$ olduğu ve $f(n)$'nin üç durumdan birine girdiği denetlenmelidir.

6.3 Temel örnekler

Birleştirme sıralaması, hızlı sıralama ve ikili arama tanıdık algoritmalardır. Bu kesim onları yeniden anlatmaz; yalnız aynı paradigmanın üç farklı yüzü olarak yan yana koyar. Ardından iki yeni örnek gelir: üs almayı logaritmik zamana indiren hızlı üs alma ve böl ve fethetin *kötü* sonuç verdiği Fibonacci özyinelemesi.

6.3.1 Klasik böl ve fethet algoritmaları

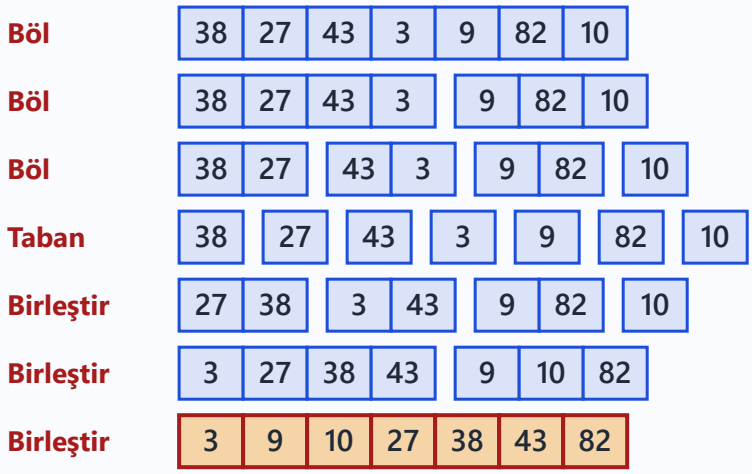
Tablo 6.4 bu notta adı geçen algoritmaların böl ve birleştir adımlarını ve bağıntılarını tek tabloda toplar. Tabloya satır satır değil, sütun sütun bakmak öğreticidir. “Birleştir” sütununda iki algoritmanın karşısında “yok” yazar: ikili aramada ve hızlı sıralamada iş ya bölme anında yapılır ya da sonuç tek bir alt problemden doğrudan gelir.

Algoritma	Böl	Birleştir	Bağıntı	Süre
İkili arama	ortadaki öğeye bakılır, bir yarı atılır	yok	$T(n/2) + O(1)$	$O(\log n)$
Birleştirme sıralaması	dizi ortadan ikiye ayrılır	sıralı yarılar birleştirilir	$2T(n/2) + O(n)$	$O(n \log n)$
Hızlı sıralama	pivota göre bölümlenir	yok (yerinde)	$T(k) + T(n-k-1) + O(n)$	ort. $O(n \log n)$
Hızlı üs alma	$x^n = (x^{\lfloor n/2 \rfloor})^2 \cdot (x)$	kare alınır	$T(n/2) + O(1)$	$O(\log n)$
Quickhull	en uzak noktayla üçgen kurulur	örtü parçaları eklenir	$T(n_1) + T(n_2) + O(n)$	ort. $O(n \log n)$
En yakın nokta çifti	x'e göre ortadan bölünür	orta şerit taranır	$2T(n/2) + O(n)$	$O(n \log n)$
Karatsuba	sayı iki yarıya bölünür	3 çarpım toplanır	$3T(n/2) + O(n)$	$O(n^{1,585})$
Strassen	matris 4 bloğa bölünür	7 çarpım toplanır	$7T(n/2) + O(n^2)$	$O(n^{2,807})$

Tablo 6.4: Klasik böl ve fethet algoritmaları.

6.3.2 Birleştirme sıralaması: iş birleştirmede

Birleştirme sıralaması böl ve fethetin en düz örneğidir. **Böl** adımı diziyi ortadan ikiye ayırır ve yalnız bir orta indis hesapladığı için $O(1)$ sürer. **Yönet** adımında iki yarı özyinelemeyle sıralanır. **Birleştir** adımında iki sıralı yarı, her adımda iki yarının başındaki küçük öğe alınarak tek geçişte birleştirilir; bu adım $\Theta(n)$ sürer.



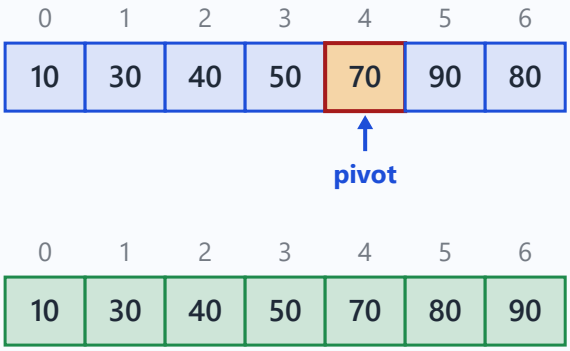
Şekil 6.3: `[38, 27, 43, 3, 9, 82, 10]` dizisinin birleştirme sıralaması. Üstteki üç satır bölmeyi, "Taban" satırı tek öğeli dizileri, alttaki üç satır birleştirmeleri gösterir. Son satır sıralı dizidir: `[3, 9, 10, 27, 38, 43, 82]`.

Şekil 6.3 yedi öğeli bir dizinin sıralanmasını düzey düzey gösterir. Önce *orta = (0 + 6)/2 = 3* ile dizi `[38 27 43 3]` ve `[9 82 10]` yarılarına ayrılır. Yarılar yeniden bölünür: `[38 27]`, `[43 3]`, `[9 82]`, `[10]`. Tek öğeli diziler zaten sıralıdır ve taban durumdur. Birleştirme aşağıdan yukarıya çalışır. Önce ikililer `[27 38]`, `[3 43]`, `[9 82]` olur. Sonra `[3 27 38 43]` ve `[9 10 82]` elde edilir; en son bu ikisi `[3 9 10 27 38 43 82]`'yi verir. Her birleştirme, iki yarının uzunlukları toplamı kadar adım sürer. Bir düzeydeki bütün birleştirmeler birlikte en çok n öğeye dokunur. Düzey sayısı $\lceil \log_2 7 \rceil = 3$'tür.

Bağıntı $T(n) = 2T(n/2) + \Theta(n)$'dir ve ana teoremin 2. durumuna girer: $\Theta(n \log n)$. Bölme girdiye bakmadan hep ortadan yapıldığı için bu süre en iyi, ortalama ve en kötü durumda aynıdır. Paradigma açısından en önemli gözlem şudur: **asıl iş birleştirmede yapılır**. Karşılaştırmaların hepsi birleştirme döngüsündedir; bölme hiçbir öğeye bakmaz.

6.3.3 Hızlı sıralama: iş bölmede

Hızlı sıralama aynı problemi ters yönden çözer. **Böl** adımı, bir pivot öğe seçip diziyi pivottan küçükler ve büyükler olarak iki tarafa ayırır. Bu işleme **bölümleme** (*partition*) denir ve $\Theta(n)$ sürer. **Yönet** adımında iki parça özyinelemeyle sıralanır. **Birleştir** adımı yoktur: parçalar yerinde sıralandığı için iki sıralı parça zaten doğru sırada yan yana durur. Birleştirme sıralamasının tersine **asıl iş bölmede yapılır**.



Şekil 6.4: `[10, 80, 30, 90, 40, 50, 70]` dizisinde Lomuto bölümlemesiyle hızlı sıralama. Solda ilk bölümlemeden sonraki durum: pivot 70 indis 4'e yerleşmiştir. Sağda sıralı dizi.

Şekil 6.4'teki örnekte Lomuto bölümlemesi kullanılır: aralığın son öğesi pivottur. Tablo 6.5 çağrıların sırasını verir.

Çağrı	Pivot	Yeni indis	Sol parça	Sağ parça	Dizi
`bolumle(dizi, 0, 6)`	70	4	[10 30 40 50]	[90 80]	10 30 40 50 70 90 80
`bolumle(dizi, 0, 3)`	50	3	[10 30 40]	[—]	10 30 40 50 70 90 80
`bolumle(dizi, 0, 2)`	40	2	[10 30]	[—]	10 30 40 50 70 90 80
`bolumle(dizi, 0, 1)`	30	1	[10]	[—]	10 30 40 50 70 90 80
`bolumle(dizi, 5, 6)`	80	5	[—]	[90]	10 30 40 50 70 80 90

Tablo 6.5: `[10, 80, 30, 90, 40, 50, 70]` için bölümleme çağrıları (pivot: aralığın son öğesi).

İlk bölümleme dengeli sayılabilir: dört ve iki öğelik parçalar oluşur. Ama sol parça `[10 30 40 50]` zaten sıralıdır. Son öğe pivot olduğu için her bölümleme yalnız bir öğeyi ayırır. Pivot 50, 40 ve 30 sırayla sona yerleşir; sağ parça her seferinde boş kalır. Bu, böl ve fethetin en kötü bölmesidir: *1 + (n − 1)*.

Hızlı sıralamanın süresi bölmenin dengesine bağlıdır. Pivot ortanca civarındaysa bağıntı $T(n) = 2T(n/2) + O(n)$ olur ve süre $O(n \log n)$'dir; ortalama durum budur. Sıralı girişte son öğe pivot seçilirse her bölümleme bir öğe ayırır; bağıntı $T(n) = T(n - 1) + O(n)$, süre $O(n^2)$ olur. Çare, pivotu girdiden bağımsız seçmektir: rastgele pivot ya da ilk, orta ve son öğenin ortancası.

İki sıralama algoritmasının karşılaştırması paradigmanın esnekliğini gösterir. İşin bir yerde yapılması kaçınılmazdır. Hızlı sıralama önce akıllıca böler ve birleştirme bedavaya gelir. Birleştirme sıralaması körü körüne böler ve akıllıca birleştirir. Birleştirme sıralaması dengeyi garanti eder ama ek dizi ister; hızlı sıralama yerinde çalışır ama dengeyi pivot seçimine bırakır.

6.3.4 İkili arama: azalt ve fethet

Sıralı bir dizide ikili arama, ortadaki öğeyi aranan değerle karşılaştırır ve dizinin bir yarısını tamamen atar. Arama yalnız kalan yarıda sürer. Bağıntı $T(n) = T(n/2) + O(1)$'dir; $a = 1$, $b = 2$, $n^{\log_2 1} = n^0 = 1$ ve $f(n) = 1$ aynı mertebededir. Durum 2'den $\Theta(\log n)$ çıkar. Bir milyon öğeli dizide en çok 20 yoklama yeter.

İkili aramada tek bir alt problem vardır ve birleştirme yoktur. Bu yüzden bazı kaynaklar ona böl ve fethet değil, **azalt ve fethet** (*decrease and conquer*) adını verir: problem bölünmez, yalnız küçültülür. Ayrım terminolojiktir; analiz aynı bağıntıyla yapılır. Asıl gözlem şudur: alt problemlerden biri atılabiliyorsa kazanç büyüktür. Doğrusal arama $O(n)$, ikili arama $O(\log n)$ sürer.

6.3.5 Hızlı üs alma

x^n'yi hesaplamanın saf yolu x'i kendisiyle $n - 1$ kez çarpmaktır. $n = 10^6$ için bu 999.999 çarpım demektir. Böl ve fethet ile çok daha az çarpım yeter. Gözlem basittir:

> $$x^n = (x^{n/2})^2 \ (n \text{ çift}), \quad x^n = x \cdot (x^{\lfloor n/2 \rfloor})^2 \ (n \text{ tek}).$$

Örneğin $3^{13} = 3 \cdot (3^6)^2$, $3^6 = (3^3)^2$, $3^3 = 3 \cdot (3^1)^2$'dir. x^n problemi, üssü yarı olan tek bir alt probleme indirgenir. Alt problemin sonucu bir kez hesaplanır ve kendisiyle çarpılır. Taban durum $x^0 = 1$'dir.

```
long us(long x, int n) {
  if (n == 0) return 1;         // taban
  long y = us(x, n / 2);
  if (n % 2 == 0) return y * y; // çift
  return x * y * y;             // tek
}
```

Kodun en önemli satırı `long y = us(x, n / 2);`'dir. Yarım üssün değeri bir değişkende saklanır ve iki kez kullanılır: çift n'de `y * y`, tek n'de `x * y * y` döndürülür. Böylece her çağrı yalnız *bir* özyinelemeli çağrı yapar ve en çok iki çarpma ekler. Bağıntı $T(n) = T(n/2) + O(1)$'dir ve sonuç $O(\log n)$'dir. Çağrı sayısı, n'nin ikili yazımındaki basamak sayısından bir fazladır; çarpım sayısı en çok $2(\lfloor \log_2 n \rfloor + 1)$'dir.

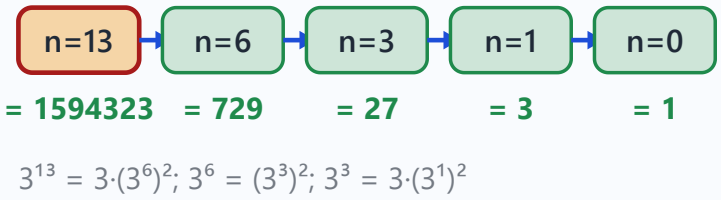
6.3.6 Kod izleme: us(3, 13)

`us(3, 13)` çağrıldığında üs her çağrıda yarıya iner: $13 \to 6 \to 3 \to 1 \to 0$. Çağrılar önce bu zincir boyunca iner, sonra değerler geri dönerken çarpmalar yapılır. Tablo 6.6 dönüş sırasını verir.

Çağrı	`y`	n çift mi?	Dönen değer	Çarpım
`us(3, 0)`	—	taban	1	0
`us(3, 1)`	1	tek	$3 \cdot 1 \cdot 1 = 3$	2

Çağrı	`y`	n çift mi?	Dönen değer	Çarpım
`us(3, 3)`	3	tek	$3 \cdot 3 \cdot 3 = 27$	2
`us(3, 6)`	27	çift	$27 \cdot 27 = 729$	1
`us(3, 13)`	729	tek	$3 \cdot 729 \cdot 729 = 1.594.323$	2

Tablo 6.6: `us(3, 13)` izlemesi. Satırlar dönüş sırasındadır: en derin çağrı önce döner.



Şekil 6.5: `us(3, 13)` çağrı zinciri ve dönüş değerleri. Zincir dallanmaz: her çağrı tek bir alt çağrı yapar. Alt satırdaki eşitlikler $3^{13} = 3 \cdot (3^6)^2$ ayrışmasını gösterir.

Toplam çarpım sayısı $2 + 2 + 1 + 2 = 7$'dir; saf yöntem 12 çarpım yapardı. Fark küçük üslerde azdır, büyük üslerde çarpıcıdır. $n = 10^6$'da hızlı üs alma 21 çağrı ve 27 çarpım yapar; saf yöntem 999.999 çarpım yapar.

Bir uyarı: `long` türü en çok $2^{63} - 1 \approx 9{,}22 \cdot 10^{18}$ değerini tutar. $3^{39} \approx 4{,}05 \cdot 10^{18}$ sığar, ama $3^{40} \approx 1{,}22 \cdot 10^{19}$ sığmaz ve Java hata vermeden taşar. Gerçek uygulamalarda hızlı üs alma ya `BigInteger` ile ya da her çarpımdan sonra bir modüle göre kalan alınarak ($x^n \bmod m$) kullanılır. İkinci biçim, açık anahtarlı şifrelemede çok büyük üslerin hesaplanmasının temelidir.

Sık yapılan hata

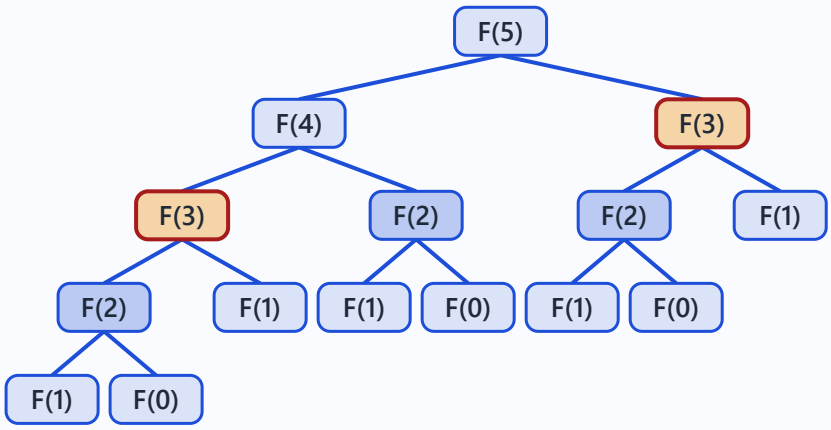
`y` değişkenini kullanmayıp yarım üssü her kullanımda yeniden hesaplamak, örneğin çift n için `return us(x, n / 2) * us(x, n / 2);`, tek n için `return x * us(x, n / 2) * us(x, n / 2);` yazmak. Kod yine doğru sonucu verir, ama her çağrı artık iki özyinelemeli çağrı yapar. Bağıntı $T(n) = 2T(n/2) + O(1)$ olur ve ana teoremin 1. durumundan $\Theta(n)$ çıkar: logaritmik kazanç tamamen kaybolur. Bu sürüm `us(3, 11)` için 31, `us(2, 16)` için 63 çağrı yapar; doğru sürüm 5 ve 6 çağrı yapar. Aynı alt problemi iki kez çözmek, böl ve fethette pahalı bir alışkanlıktır.

6.3.7 Uygun olmayan durum: örtüşen alt problemler

Fibonacci sayıları $F(0) = 0$, $F(1) = 1$ ve $F(n) = F(n - 1) + F(n - 2)$ ile tanımlanır. Tanım özyinelemelidir ve doğrudan koda dökülebilir:

```
long fib(int n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}
```

Kod böl ve fethet biçimindedir: problem iki küçük alt probleme ayrılır, alt problemler aynı fonksiyonla çözülür ve sonuçlar toplanarak birleştirilir. Sonuç da doğrudur. Ama bu kötü bir örnektir, çünkü alt problemler birbirinden bağımsız değildir. *F(n − 1)*'i hesaplamak için *F(n − 2)* zaten hesaplanır; sonra *F(n − 2)* ikinci çağrıda baştan hesaplanır.



Şekil 6.6: `fib(5)` çağrı ağacı. Vurgulu düğümler aynı alt problemin tekrarlarıdır: *F(3)* iki kez, *F(2)* üç kez hesaplanır.

Şekil 6.6'daki ağaçta `fib(5)` için 15 çağrı yapılır. *F(3)* iki kez, *F(2)* üç kez, *F(1)* beş kez, *F(0)* üç kez hesaplanır. Çağrı sayısı $C(n) = 2F(n + 1) − 1$ formülüyle büyür: `fib(30)` için 2.692.537, `fib(40)` için 331.160.281 çağrı yapılır. Bağıntı $T(n) = T(n − 1) + T(n − 2) + O(1)$'dir ve çözümü $O(\varphi^n)$'dir; burada $\varphi = (1 + \sqrt{5})/2 \approx 1{,}618$ altın orandır.

Çözüm, her alt problemin sonucunu ilk hesaplandığında saklamak ve ikinci kez gerektiğinde tablodan okumaktır. Bu yaklaşımın adı **dinamik programlamadır**; Fibonacci için süreyi $O(n)$'ye indirir. Paradigma seçiminde sorulacak soru şudur: alt problemler birbirinden bağımsız mı, yoksa ortak alt alt problemler mi içeriyor? Bağımsızsa böl ve fethet, örtüşüyorsa dinamik programlama uygundur. Her adımda yerel olarak en iyi seçim küresel en iyiye götürüyorsa açgözlü yaklaşım daha da basit bir çözüm verir.

Sık yapılan hata

Bir problemin özyinelemeli tanımı olduğu için doğrudan böl ve fethetle çözülebileceğini sanmak. Özyinelemeli tanım, doğru bir algoritma verir; hızlı bir algoritma vermeyebilir. Çağrı ağacında aynı argümanla birden çok çağrı görülüyorsa alt problemler örtüşüyordur ve sonuçlar

saklanmalıdır.

6.4 Dışbükey örtü ve Quickhull

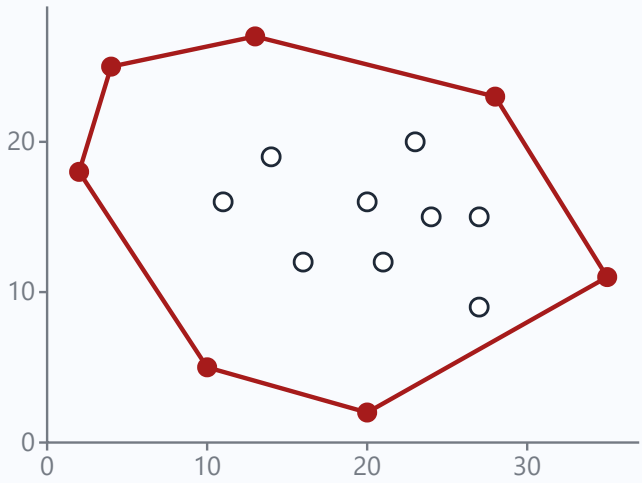
Hesaplamalı geometri, noktalar, doğrular ve çokgenler üzerinde çalışan algoritmaları inceler. Bu alanın en temel problemlerinden biri dışbükey örtüdür. Problemin böl ve fethetle çözümü olan Quickhull, adını hızlı sıralamaya benzerliğinden alır: bir "pivot" seçilir, noktalar bu pivota göre bölümlenir ve parçalar özyinelemeyle işlenir.

6.4.1 Dışbükey örtü problemi

Tanım 6.2

Bir çokgen, içindeki herhangi iki noktayı birleştiren doğru parçası tamamen çokgenin içinde kalıyorsa **dışbükeydir**. Düzlemde n noktalı bir S kümesinin **dışbükey örtüsü** (*convex hull*), S'nin bütün noktalarını içeren en küçük dışbükey çokgendir.

Tanımı akılda tutmanın kolay bir yolu vardır. Noktaların yerine bir tahtaya çiviler çakıldığı ve bütün çivilerin çevresine bir lastik bant geçirildiği düşünülsün. Bant bırakılınca dıştaki çivilere sarılır ve gerilir. Bandın aldığı şekil dışbükey örtüdür. Bu benzetme iki önemli gerçeği de gösterir. Birincisi, örtünün köşeleri S'nin noktalarıdır; bant yalnız çivilere değer. İkincisi, içteki çiviler bandın biçimini hiç etkilemez.



Şekil 6.7: 16 noktalı bir küme ve 7 köşeli dışbükey örtüsü. Dolu noktalar örtünün köşeleridir; içi boş noktalar örtünün içinde kalır.

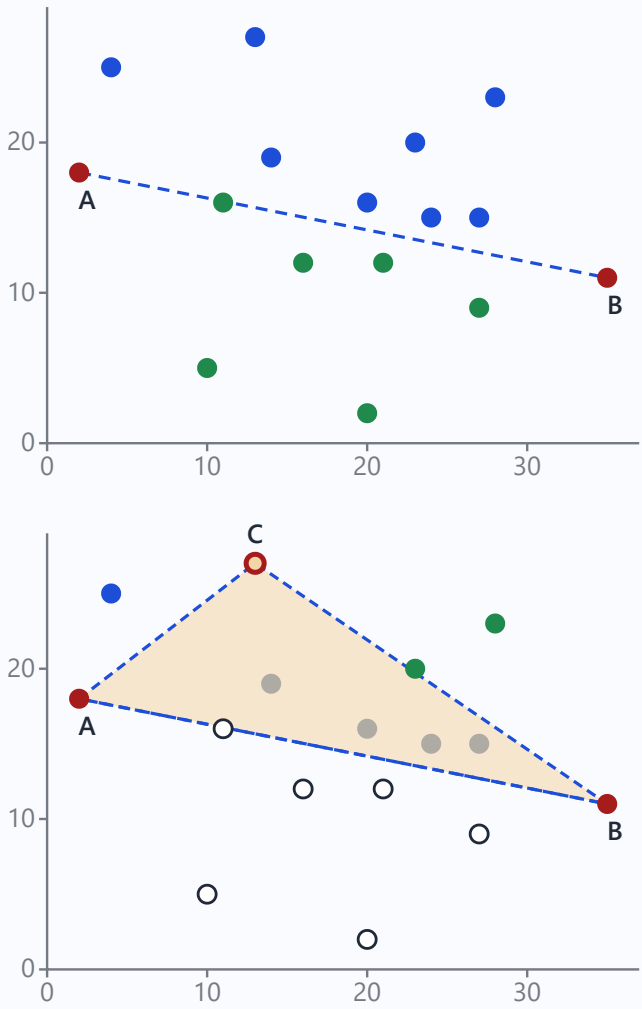
Algoritmanın çıktısı, örtünün köşelerinin sıralı listesidir; geleneksel olarak saat yönünün tersine sıralanır. Yalnız “hangi noktalar örtüdedir?” sorusunu yanıtlamak yetmez, çünkü çokgeni çizmek ya da üzerinde hesap yapmak için köşelerin sırası gerekir. Şekil 6.7'deki kümenin örtüsü yedi köşelidir; kalan dokuz nokta içeridedir.

Dışbükey örtü pek çok uygulamanın ilk adımıdır. Oyunlarda ve robotikte çarpışma algılama, karmaşık bir nesnenin yerine onun örtüsünü kullanarak hızlanır: iki örtü kesişmiyorsa nesneler de kesişmez. Coğrafi bilgi sistemleri bir bölgedeki gözlem noktalarının sınırını örtüyle çizer. Görüntü işlemede bir şeklin örtüsü onun biçimini özetler. Bir kümenin birbirinden en uzak iki noktası mutlaka örtünün köşeleridir; bu gözlem en uzak nokta çifti problemini küçültür. Kümelemede her kümenin sınırı örtüyle gösterilir.

6.4.2 Quickhull algoritması

Quickhull şu gözleme dayanır: bir doğruya en uzak nokta, o doğrunun o tarafındaki noktaların örtüsünde mutlaka yer alır. Algoritma şu adımlarla çalışır:

1. x koordinatı en küçük nokta A ve en büyük nokta B bulunur. İkisi de örtüdedir: en soldaki ve en sağdaki noktalar hiçbir zaman başka noktaların arasında kalamaz.
2. AB doğrusu noktaları iki alt kümeye ayırır: doğrunun üstü (A'dan B'ye bakarken solu) ve altı (sağı).
3. Bir alt kümede AB'ye en uzak C noktası bulunur. C örtüdedir.
4. ACB üçgeninin içindeki noktalar elenir: üçgenin köşeleri örtüde olduğuna göre üçgenin içi de örtünün içindedir ve bu noktalar örtünün köşesi olamaz.
5. AC'nin dışında kalanlar ve CB'nin dışında kalanlar için aynı işlem özyinelemeyle yapılır.
6. Bir kenarın dışında hiç nokta kalmayınca o kenar bir örtü kenarıdır ve özyineleme durur. Bulunan noktalar örtüyü oluşturur.



Şekil 6.8: Quickhull'un ilk adımları. Solda: en soldaki *A* ve en sağdaki *B* bulunmuş, *AB* doğrusu noktaları üst ve alt iki kümeye ayırmıştır. Sağda: üst kümede *AB*'ye en uzak nokta *C = (13, 27)*'dir; *ACB* üçgeninin içindeki dört nokta elenir.

Hızlı sıralamayla benzerlik açıktır. Hızlı sıralamada pivot diziyi iki parçaya ayırır; Quickhull'da en uzak nokta *C*, üçgenin iki kenarına göre kümeyi iki parçaya ayırır. Bir fark vardır: Quickhull'da bölümlemede bazı noktalar hiçbir parçaya girmez. Üçgenin içindeki noktalar tamamen atılır ve bu atma, algoritmanın pratikte hızlı olmasının asıl nedenidir.

Algoritmada birleştirme adımı neredeyse yoktur. Her özyinelemeli çağrı bulduğu noktayı örtü listesine ekler; iki alt çağrının sonuçları ayrıca birleştirilmez. İş, hızlı sıralamada olduğu gibi bölme anında yapılır: en uzak noktayı bulmak ve noktaları üçgenin kenarlarına göre ayırmak.

6.4.3 Yönelim testi ve uzaklık

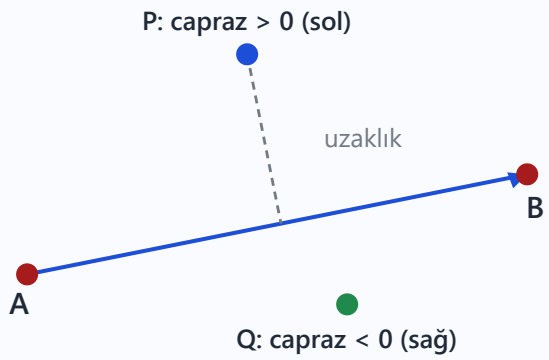
Algoritma iki geometrik soruyu sürekli sorar: bir *P* noktası *A* → *B* doğrusunun solunda mı, sağında mı? Ve *P*'nin *AB*'ye uzaklığı ne kadar? İki soru da tek bir formülle yanıtlanır. Formül yalnız tam sayı çarpımı ve çıkarması kullanır; açı, karekök ya da bölme gerekmez.

```
long capraz(Nokta A, Nokta B, Nokta P) {
  return (long)(B.x - A.x) * (P.y - A.y)
       - (long)(B.y - A.y) * (P.x - A.x);
}
```

`capraz` metodu, $A \to B$ ve $A \to P$ vektörlerinin çapraz çarpımının (vektörel çarpımının) z bileşenini hesaplar. `(long)` dönüşümleri, büyük koordinatlarda `int` çarpımının taşmasını önler. Değerin işareti yönü verir:

- `capraz > 0`: P, $A \to B$'nin solundadır (saat yönünün tersine dönüş).
- `capraz < 0`: P, $A \to B$'nin sağındadır (saat yönünde dönüş).
- `capraz = 0`: A, B, P aynı doğru üzerindedir.

Değerin mutlak büyüklüğü de anlam taşır. $|capraz|$, A, B ve P köşeli üçgenin alanının iki katıdır. Üçgenin alanı ayrıca $\tfrac{1}{2} \cdot |AB| \cdot h$'dir; burada h, P'nin AB doğrusuna dik uzaklığıdır. Buradan $|capraz| = |AB| \cdot h$ çıkar. Aynı A ve B için $|AB|$ sabittir. Bu yüzden doğruya en uzak nokta, `capraz` değeri en büyük olan noktadır ve uzaklığı ayrıca hesaplamaya gerek yoktur.



Şekil 6.9: Yönelim testi. P, $A \to B$'nin solundadır ve `capraz` pozitiftir; Q sağındadır ve `capraz` negatiftir. Kesikli çizgi P'nin doğruya dik uzaklığıdır.

Örnek 6.3

$A = (2, 18)$ ve $B = (35, 11)$ olsun. $B - A = (33, -7)$'dir.

- $C = (13, 27)$ için $C - A = (11, 9)$: $capraz = 33 \cdot 9 - (-7) \cdot 11 = 297 + 77 = 374$.
- $(28, 23)$ için $(26, 5)$: $capraz = 33 \cdot 5 - (-7) \cdot 26 = 165 + 182 = 347$.
- $(10, 5)$ için $(8, -13)$: $capraz = 33 \cdot (-13) - (-7) \cdot 8 = -429 + 56 = -373$; nokta doğrunun altındadır.

İlk iki nokta doğrunun üstündedir. 374, 347'den büyük olduğu için $C = (13, 27)$ doğruya daha uzaktır. Gerçek uzaklık $374 / |AB| = 374 / \sqrt{(33^2 + 7^2)} = 374 / \sqrt{1138} \approx 11{,}09$'dur, ama karşılaştırma için bu bölme gereksizdir.

Sık yapılan hata

Yönelim testinde nokta sırasını karıştırmak. `capraz(A, B, P)` ile `capraz(B, A, P)` ters işaretlidir. Quickhull üst kümeyi `solda(P, A, B)`, alt kümeyi `solda(P, B, A)` ile bulur: aynı test, doğrunun yönü ters çevrilerek öbür tarafa uygulanır. Sıra karışırsa bir küme iki kez işlenir, öbürü hiç işlenmez.

6.4.4 Kod: quickHull() ve bul()

```
void quickHull(List<Nokta> P) {
  Nokta A = enSol(P), B = enSag(P);
  ortu.add(A); ortu.add(B);
  bul(solda(P, A, B), A, B); // üst küme
  bul(solda(P, B, A), B, A); // alt küme
}
void bul(List<Nokta> S, Nokta A, Nokta B) {
  if (S.isEmpty()) return;   // AB örtü kenarı
  Nokta C = enUzak(S, A, B); // max capraz
  ortu.add(C);
  bul(solda(S, A, C), A, C); // AC'nin dışı
  bul(solda(S, C, B), C, B); // CB'nin dışı
}  // ACB üçgeninin içi elenir
```

Kod iki metottan oluşur. `quickHull` başlatıcıdır: en sol ve en sağ noktayı bulur, onları örtüye ekler ve iki yarı için `bul`'u çağırır. `bul(S, A, B)`, $A \to B$ doğrusunun solunda kalan S kümesiyle çalışır ve bu kümedeki örtü köşelerini bulur:

- `if (S.isEmpty()) return;` taban durumdur. Doğrunun dışında hiç nokta yoksa AB bir örtü kenarıdır.
- `Nokta C = enUzak(S, A, B);` en büyük `capraz` değerli noktayı seçer ve `ortu.add(C);` onu örtüye ekler.
- İki özyinelemeli çağrı, $A \to C$'nin solunda ve $C \to B$'nin solunda kalan noktalarla çalışır. Bu iki kümeden hiçbirine girmeyen noktalar üçgenin içindedir ve kendiliğinden elenir; onlar için ayrı bir kod yoktur.

Alt kümenin $B \to A$ yönüyle işlenmesi bir incelik içerir. Doğrunun altı, $A \to B$'nin sağı, yani $B \to A$'nın soludur. Bu sayede `bul` her zaman "doğrunun solundaki noktalar" sözleşmesiyle çalışır ve tek bir metot iki tarafa da yeter. Kodda kullanılan yardımcı metotlar aşağıdadır. `enSol` ve `enSag` listeyi bir

kez tarayıp x'i en küçük ve en büyük noktayı döndürür.

```
List<Nokta> solda(List<Nokta> S, Nokta A, Nokta B) {
  List<Nokta> sonuc = new ArrayList<>();
  for (Nokta p : S)
    if (capraz(A, B, p) > 0) sonuc.add(p); // A→B'nin solunda
  return sonuc;
}
Nokta enUzak(List<Nokta> S, Nokta A, Nokta B) {
  Nokta enIyi = null;
  long enBuyukDeger = -1;
  for (Nokta p : S) {
    long c = capraz(A, B, p);
    if (c > enBuyukDeger) { enBuyukDeger = c; enIyi = p; }
  }
  return enIyi;
}
```

`solda` ve `enUzak` listeyi birer kez tarar; ikisi de $O(|S|)$ sürer. Bir `bul` çağrısı bu yüzden kendi kümesinin boyuyla orantılı iş yapar: bir `enUzak` ve iki `solda` taraması.

`ortu` listesi köşeleri bulunma sırasıyla tutar ve bu sıra çokgenin çevresindeki sıra değildir. Sıralı çıktı için `ortu.add(C)` satırını iki özyinelemeli çağrının arasına taşımak yeter. Bu durumda üst kümede köşeler A'dan B'ye, alt kümede B'den A'ya doğru eklenir; A en başa, B iki kümenin arasına yazılırsa köşeler saat yönünde sıralanır. Liste ters çevrilince saat yönünün tersine sıra elde edilir.

6.4.5 Kod izleme: 16 nokta

İzlemede Şekil 6.7'deki 16 nokta kullanılır: (2,18), (4,25), (10,5), (11,16), (13,27), (14,19), (16,12), (20,16), (20,2), (21,12), (23,20), (24,15), (27,9), (27,15), (28,23), (35,11). En sol nokta $A = (2, 18)$, en sağ nokta $B = (35, 11)$'dir ve ikisi örtüye eklenir. Kalan 14 noktanın 8'i $A \rightarrow B$'nin solunda (üst küme), 6'sı sağındadır (alt küme). Tablo 6.7 `bul` çağrılarını sırayla verir.

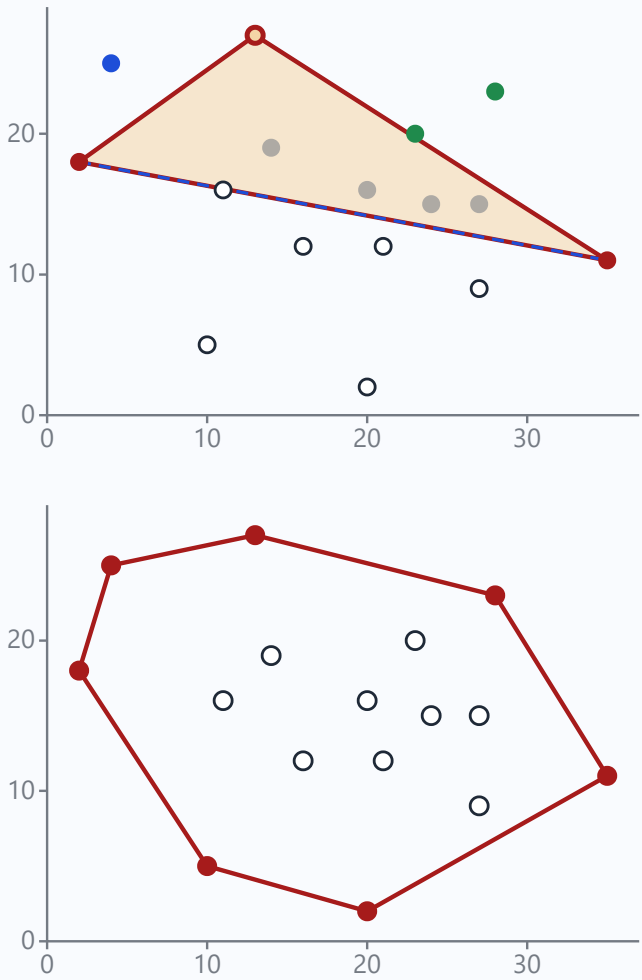
Çağrı	Doğru	$\|S\|$	En uzak C	Elenen	AC dışı	CB dışı
1	(2,18) → (35,11)	8	(13,27)	4	1	2
2	(2,18) → (13,27)	1	(4,25)	0	0	0
3	(2,18) → (4,25)	0	S boş: örtü kenarı			
4	(4,25) → (13,27)	0	S boş: örtü kenarı			
5	(13,27) → (35,11)	2	(28,23)	1	0	0
6	(13,27) → (28,23)	0	S boş: örtü kenarı			
7	(28,23) → (35,11)	0	S boş: örtü kenarı			
8	(35,11) → (2,18)	6	(20,2)	4	0	1
9	(35,11) → (20,2)	0	S boş: örtü kenarı			

Çağrı	Doğru	|S|	En uzak C	Elenen	AC dışı	CB dışı
10	(20,2) → (2,18)	1	(10,5)	0	0	0
11	(20,2) → (10,5)	0	S boş: örtü kenarı			
12	(10,5) → (2,18)	0	S boş: örtü kenarı			

Tablo 6.7: 16 noktada Quickhull izlemesi. "Elenen", *ACB* üçgeninin içinde kalan nokta sayısıdır.

Tablonun ilk satırı algoritmanın en verimli anıdır. Üst kümenin 8 noktasından en uzak olanı *C = (13, 27)*'dir; `capraz` değeri 374'tür. Üçgenin içinde kalan dört nokta, (14,19), (20,16), (24,15) ve (27,15), tek hamlede elenir. *AC*'nin dışında yalnız (4,25), *CB*'nin dışında (23,20) ve (28,23) kalır. Sayılar tutarlıdır: *8 = 1 + 4 + 1 + 2* (C, elenenler ve iki dış küme).

2. çağrıda tek noktalı küme işlenir. Tek nokta doğal olarak en uzak noktadır: (4,25) örtüye girer ve iki alt çağrı boş kümeyle döner. 5. çağrıda iki nokta vardır: *(13,27) → (35,11)* doğrusuna (28,23) daha uzaktır (`capraz` = 152); (23,20) üçgenin içinde kalır ve elenir. Alt kümede 8. çağrı *C = (20, 2)*'yi seçer (`capraz` = 402) ve (11,16), (16,12), (21,12), (27,9) noktalarını eler. Kalan tek nokta (10,5), 10. çağrıda örtüye girer.



Şekil 6.10: Quickhull izlemesinin iki anı. Solda 1. çağrı: üst kümede en uzak nokta (13,27), taralı üçgenin içindeki dört nokta elenir. Sağda izlemenin sonu: 7 köşeli örtü.

İzleme sonunda örtü yedi köşelidir: (2,18), (35,11), (13,27), (4,25), (28,23), (20,2), (10,5). Dokuz nokta elenmiştir ve toplam 12 `bul` çağrısı yapılmıştır. Çağrı sayısı yapı gereği bellidir: örtüye `bul` içinde eklenen her köşe iki alt çağrı açar ve her örtü kenarı bir boş çağrıyla biter. Burada 5 köşe `bul` içinde bulunmuş (5 dolu çağrı) ve örtünün 7 kenarı 7 boş çağrı vermiştir.

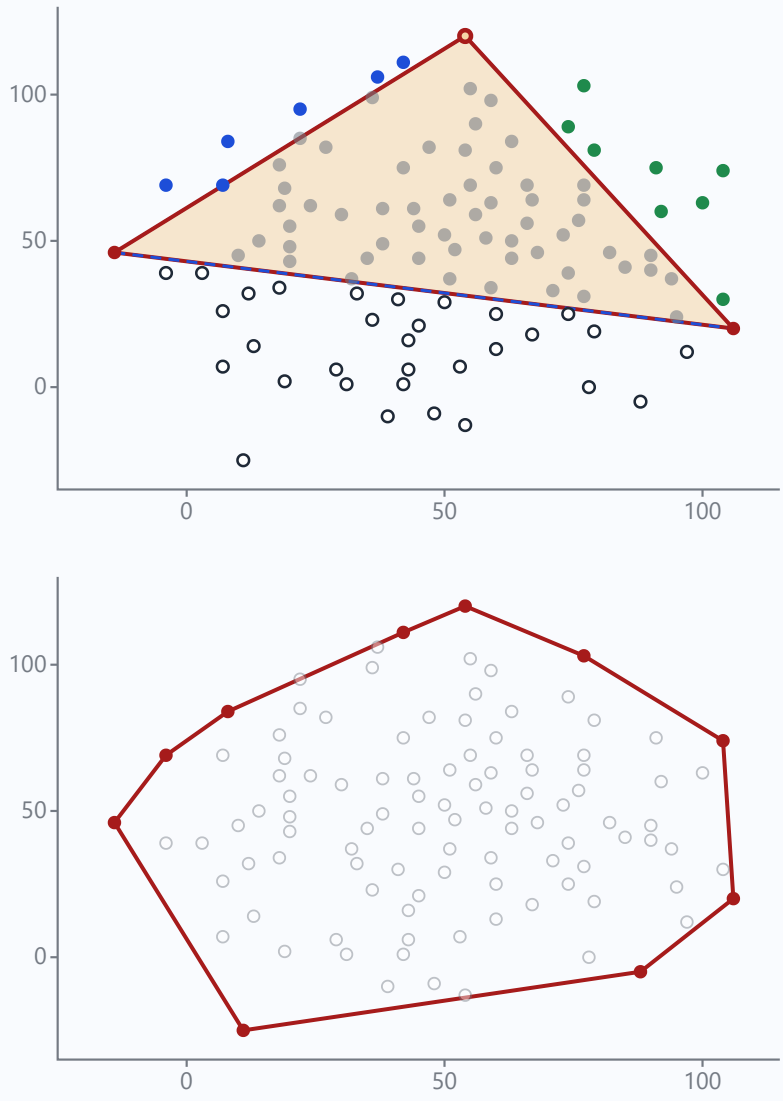
6.4.6 104 noktalı örnek

Algoritmanın gücü büyük kümelerde daha açık görünür. 104 noktalı bir kümede en sol nokta (−14, 46), en sağ nokta (106, 20)'dir. Bu iki noktayı birleştiren doğrunun üstünde 71, altında 31 nokta vardır. Tablo 6.8 yalnız dolu çağrıları, yani örtüye nokta ekleyen çağrıları verir. “Aday” sütunu, henüz ne örtüye eklenmiş ne de elenmiş nokta sayısıdır; başta *71 + 31 = 102*'dir.

Doğru	$\|S\|$	En uzak	Elenen	Kalan aday
(−14,46) → (106,20)	71	(54,120)	56	45
(−14,46) → (54,120)	6	(8,84)	2	42

Doğru	$\|S\|$	En uzak	Elenen	Kalan aday
(−14,46) → (8,84)	1	(−4,69)	0	41
(8,84) → (54,120)	2	(42,111)	1	39
(54,120) → (106,20)	8	(104,74)	6	32
(54,120) → (104,74)	1	(77,103)	0	31
(106,20) → (−14,46)	31	(11,−25)	25	5
(106,20) → (11,−25)	5	(88,−5)	4	0

Tablo 6.8: 104 noktalı kümede Quickhull'un dolu çağrıları.



Şekil 6.11: 104 noktalı örnek. Üstte ilk dolu çağrı: üst kümede en uzak nokta (54,120); büyük üçgenin içindeki 56 nokta tek adımda elenir. Altta sonuç: 10 köşeli örtü.

İlk çağrı tek başına 56 noktayı, alt kümedeki ilk çağrı 25 noktayı eler. Elenen 94 noktanın 81'i bu iki çağrıdan gelir. Sonraki çağrıların kümeleri küçüktür: 6, 1, 2, 8, 1, 5 nokta. Toplam 8 dolu çağrıyla örtünün 10 köşesi bulunur ve 94 nokta elenir. Her adımda noktaların çoğu elendiği için iş hızla azalır. Düzgün dağılmış rastgele noktalarda bu durum tipiktir: örtünün köşe sayısı n'ye göre çok küçüktür ve noktaların büyük çoğunluğu ilk birkaç üçgenin içine düşer.

6.4.7 Analiz ve diğer örtü algoritmaları

Her `bul` çağrısı kendi kümesini sabit sayıda tarar: en uzak noktayı bulmak için bir kez, iki dış kümeyi ayırmak için iki kez. Çağrının işi $O(|S|)$'dir. Bağıntı

$$T(n) = T(n_1) + T(n_2) + O(n), \quad n_1 + n_2 \le n - 1$$

biçimindedir. n_1 ve n_2 girdiye bağlıdır; bu yüzden ana teorem uygulanamaz ve durumlar ayrı ayrı incelenir:

- **Ortalama durum.** Bölmeler dengeliyse ve her adımda çok nokta eleniyorsa süre $O(n \log n)$'dir. Düzlemde düzgün dağılmış rastgele noktalarda algoritma çoğu zaman bundan da hızlıdır, çünkü elenen noktalar özyinelemenin derinlerine hiç inmez.
- **En kötü durum.** Her adımda en uzak nokta yalnız bir noktayı ayırabilir. Örneğin noktalar dışbükey bir eğri üzerinde dizilmiş ve en uzak nokta hep bir uca komşu düşüyor olabilir. Bu durumda bağıntı $T(n) = T(n-1) + O(n)$ olur ve süre $O(n^2)$'dir. Bu girdide hiçbir nokta elenmez; bütün noktalar örtüdedir.
- **Çıktıya duyarlı sınır.** Örtünün köşe sayısı h olsun. Özyineleme ağacının bir düzeyindeki kümeler ayrıktır; bu yüzden bir düzeyin toplam işi en çok n'dir. Kökten yaprağa giden her yoldaki dolu çağrılar farklı köşeler bulduğu için ağacın derinliği h'yi aşmaz. Toplam iş $O(n \cdot h)$'dir. Köşe sayısı küçükse bu sınır $n \log n$'den bile iyidir.

Dışbükey örtü için başka algoritmalar da vardır. Tablo 6.9 bunları süreleriyle karşılaştırır.

Algoritma	Süre	Yaklaşım
Kaba kuvvet (her kenar adayı)	$O(n^3)$	her nokta çifti için öteki noktaların hepsi aynı taraftaysa çift bir örtü kenarıdır
Jarvis yürüyüşü (*gift wrapping*)	$O(n \cdot h)$	bir köşeden başlayıp her adımda en dıştaki sonraki köşeye sarılır
Graham taraması	$O(n \log n)$	noktalar açıya göre sıralanır, yığınla sağa dönüşler atılır
Andrew monoton zincir	$O(n \log n)$	noktalar x'e göre sıralanır, üst ve alt zincir ayrı kurulur
Quickhull	ort. $O(n \log n)$, en kötü $O(n^2)$	böl ve fethet; en uzak nokta ve üçgen içi eleme
Chan	$O(n \log h)$	küçük grupların örtüleri Graham ile, birleşim Jarvis ile bulunur

Tablo 6.9: Dışbükey örtü algoritmaları (h: örtünün köşe sayısı).

Bu sürelerin altına inmek mümkün değildir. Dışbükey örtü bulan bir algoritma sıralama problemini de çözebilir: sıralanacak $x_1, \ldots, x_n$ sayıları $y = x^2$ parabolü üzerine (x_i, x_i^2) noktaları olarak yerleştirilir. Parabol dışbükey olduğu için bütün noktalar örtünün köşesidir ve örtünün sıralı köşe listesi, sayıları sıralı olarak verir. Karşılaştırmalı modelde sıralama $\Omega(n \log n)$ zaman gerektirdiğine göre, dışbükey örtü de en az bu kadar zaman gerektirir. Graham ve Andrew algoritmaları bu anlamda en iyidir. Chan algoritmasının $O(n \log h)$ süresi bu alt sınırla çelişmez, çünkü parabol örneğinde $h = n$'dir.

6.5 En yakın nokta çifti

Düzlemdeki noktalar arasında birbirine en yakın iki noktayı bulmak, hesaplamalı geometrinin en eski problemlerinden biridir. Kaba kuvvet çözümü bütün çiftlere bakar ve karesel süre alır. Böl ve fethet çözümü ise süreyi $O(n \log n)$'ye indirir. Çözümün ilginç yanı birleştirme adımıdır: iki yarının sınırında kalan çiftleri kaçırmamak için geometrik bir gözlem gerekir.

6.5.1 Problem ve kaba kuvvet

Tanım 6.3

Düzlemde $n \geq 2$ noktalı bir küme verilsin. **En yakın nokta çifti** (*closest pair of points*) problemi, Öklid uzaklığı

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

en küçük olan $p \neq q$ çiftini (ya da yalnız bu en küçük uzaklığı) bulmaktır.

Problem pek çok uygulamada ortaya çıkar. Hava trafiği kontrolünde birbirine en çok yaklaşan iki uçak sürekli izlenir. Kümeleme algoritmaları en yakın iki noktayı ya da kümeyi birleştirerek ilerler. Çarpışma algılamada birbirine en yakın iki nesne ilk aday çifttir. Coğrafi verilerde birbirine yakın tesisler bulunur.

En doğrudan çözüm bütün çiftlerin uzaklığını hesaplayıp en küçüğünü almaktır:

```
double kabaKuvvet(List<Nokta> p) {
  double enAz = Double.MAX_VALUE;
  for (int i = 0; i < p.size(); i++)
    for (int j = i + 1; j < p.size(); j++)
      enAz = Math.min(enAz, mesafe(p.get(i), p.get(j)));
  return enAz;
}
```

İç içe iki döngü her çifti bir kez dener: j her zaman i'den büyüktür, böylece hem (p, q) hem (q, p) denenmez, bir nokta kendisiyle de karşılaştırılmaz. Çift sayısı $n(n-1)/2$'dir ve süre $O(n^2)$'dir. 10 nokta için 45 uzaklık hesaplanır; $n = 10^6$ için yaklaşık $5 \cdot 10^{11}$ uzaklık gerekir. Modern bir bilgisayarda bu, dakikalar süren bir iştir.



Şekil 6.12: Sekiz noktada kaba kuvvet. Tek bir noktadan yedi uzaklık hesaplanır; bütün noktalar için toplam 28 çift denenir. En yakın çift vurgulanmıştır.

Tek boyutta problem kolaydır. Sayılar sıralanır ve yalnız ardışık sayılar karşılaştırılır: en yakın iki sayı, sıralı dizide mutlaka yan yanadır. Süre sıralamanın süresidir: $O(n \log n)$. İki boyutta bu fikir doğrudan çalışmaz. Noktalar x'e göre sıralandığında x'leri yakın iki nokta, y'leri çok farklı olduğu için birbirinden uzak olabilir. x'e göre sıralı dizide yan yana olmayan iki nokta da en yakın çift olabilir. Böl ve fethet bu güçlüğü aşar.

6.5.2 Böl ve fethet: orta şerit

Algoritma şu adımlarla çalışır:

1. Noktalar bir kez x'e, bir kez y'ye göre sıralanır. Bu iki sıralı liste X ve Y olarak adlandırılır. Sıralama yalnız başta, özyinelemeden önce yapılır.
2. **Böl:** X'in ortasındaki noktanın x koordinatından geçen dikey L doğrusu, noktaları sol ve sağ iki yarıya ayırır.
3. **Yönet:** İki yarıda en yakın uzaklık özyinelemeyle bulunur: δ_L ve δ_R. İkisinin küçüğü $\delta = min(\delta_L, \delta_R)$'dir.
4. **Birleştir:** Daha yakın bir çift hâlâ olabilir: bir noktası solda, öbürü sağda olan bir çift. Böyle bir çift varsa L'yi keser ve iki noktası da L'ye δ'dan yakındır. Bu yüzden yalnız $|x - L| < \delta$ olan noktalara, yani 2δ genişliğindeki **şerit** noktalarına bakılır.
5. Şerit y sırasıyla taranır. Her nokta yalnız kendisinden sonra gelen ve y farkı δ'dan küçük olan noktalarla karşılaştırılır.

Birleştirme adımının gerekçesi şudur. İki noktası da aynı yarıda olan çiftler özyinelemede zaten incelenmiştir ve hiçbiri δ'dan yakın değildir. Geriye yalnız L'yi kesen çiftler kalır. Böyle bir çiftin uzaklığı δ'dan küçükse x farkı da δ'dan küçüktür; bu yüzden iki noktası da L'nin en çok δ uzağındadır. Benzer biçimde y farkı da δ'dan küçük olmalıdır. Şeridi y sırasıyla taramak, bu ikinci koşulu ucuza denetlemenin yoludur: y farkı δ'yı aştığı anda sonraki noktalar daha da uzaktır ve iç döngü durur.

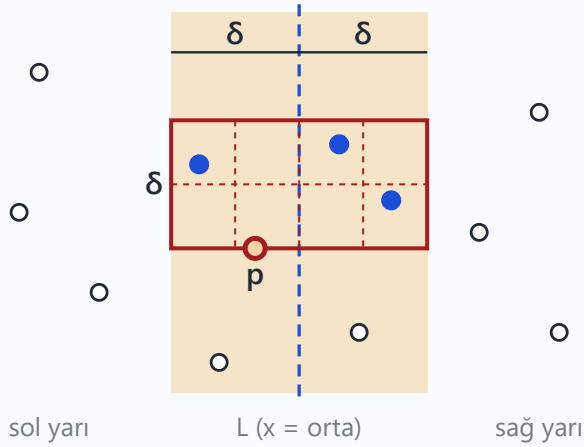
6.5.3 Şeritte neden en çok yedi komşu?

Şerit kalabalık olabilir. Kötü bir girdide noktaların hepsi şeride düşebilir. Şerit noktalarının her çiftini denemek yine karesel süre alırdı. Algoritmayı hızlı yapan gözlem, şeritteki her noktanın yalnız sabit sayıda komşusuyla karşılaştırılmasının yetmesidir.

Teorem 6.3 (Şerit komşuluğu)

Şerit noktaları y'ye göre sıralı olsun. Bir p noktasından sonra gelen ve y farkı δ'dan küçük olan şerit noktası sayısı en çok 7'dir.

İspat. p'den sonra gelip y farkı δ'dan küçük olan noktalar, alt kenarı p'nin y düzeyinde olan 2δ genişliğinde, δ yüksekliğinde bir dikdörtgenin içindedir. Dikdörtgen L ile ikiye ayrılır ve her yarısı $\delta \times \delta$'lık bir karedir. Her kare, kenarı $\delta/2$ olan dört küçük kareye bölünür; toplam 8 küçük kare oluşur. Bir küçük kare tamamen bir yarının içindedir. Aynı küçük karede iki nokta olsaydı ikisi aynı yarıda olurdu ve aralarındaki uzaklık en çok karenin köşegeni, yani $\delta/\sqrt{2} < \delta$ olurdu. Oysa o yarının en yakın uzaklığı en az δ'dır; çelişki. O hâlde her küçük karede en çok bir nokta vardır. Bu noktalardan biri p'nin kendisidir; geriye en çok 7 nokta kalır. ■



Şekil 6.13: L doğrusu çevresinde 2δ genişliğinde şerit. p'nin üstündeki $\delta \times 2\delta$ dikdörtgen sekiz küçük kareye bölünmüştür; her küçük karede en çok bir nokta bulunabilir.

Teorem sınırı verir; kod bu sınırı açıkça yazmaz. İç döngü y farkı δ'yı aşınca durur. Teorem, bu koşul altında iç döngünün en çok 7 tur döndüğünü garanti eder. Bazı kaynaklar sınırı 6 ya da 5'e kadar indirir; sabitin değeri süre mertebesini değiştirmez. Önemli olan, şerit taramasının $O(7n) = O(n)$ sürmesidir.

Sık yapılan hata

Şeritte toplam nokta sayısının sınırlı olduğunu sanmak. Şerit n noktanın hepsini içerebilir; sınırlı olan, her noktanın karşılaştırıldığı komşu sayısıdır. Bir başka hata da şerit noktalarını x sırasıyla taramaktır. Komşuluk sınırı yalnız y sırasında geçerlidir; x sırasında taranan şeritte bir nokta çok sayıda noktayla karşılaştırılmak zorunda kalabilir.

6.5.4 Kod: enYakin()

```
double enYakin(List<Nokta> X, List<Nokta> Y) {
  int n = X.size(), orta = n / 2;
  if (n <= 3) return kabaKuvvet(X);
  Nokta m = X.get(orta); // Ysol/Ysag: Y'nin yarıları
  double dSol = enYakin(X.subList(0, orta), Ysol);
  double dSag = enYakin(X.subList(orta, n), Ysag);
  double d = Math.min(dSol, dSag);
  List<Nokta> s = new ArrayList<>();  // şerit
  for (Nokta p : Y)                   // y sırası
    if (Math.abs(p.x - m.x) < d) s.add(p);
  for (int i = 0; i < s.size(); i++) {
    Nokta p = s.get(i);
    for (int j = i + 1; j < s.size()
         && s.get(j).y - p.y < d; j++)
      d = Math.min(d, mesafe(p, s.get(j)));
  }
  return d;
}
```

Metot iki liste alır: `X`, noktaların x'e göre sıralı listesi, `Y` de aynı noktaların y'ye göre sıralı listesidir. İlk çağrıdan önce bütün noktalar bir kez iki ayrı ölçüte göre sıralanır. Satırlar şöyle çalışır:

- `if (n <= 3) return kabaKuvvet(X);` taban durumdur. Üç ya da daha az nokta için kaba kuvvet en çok üç uzaklık hesaplar; bölmeye değmez. Eşiğin 3 olmasının bir nedeni daha vardır: tek noktalı bir yarıda en yakın çift tanımsızdır.
- `Nokta m = X.get(orta);` bölme doğrusunu belirler: L, $x = m.x$ doğrusudur. Sol yarı `X.subList(0, orta)`, sağ yarı `X.subList(orta, n)` 'dir.
- `Ysol` ve `Ysag`, `Y` 'nin iki yarıya düşen noktalarıdır ve y sırasını korur. Kod bunların kurulmasını bir açıklama satırıyla geçer; bir yolu aşağıda verilmiştir.
- İki özyinelemeli çağrı `dSol` ve `dSag` 'ı verir; `d` bunların küçüğüdür.
- İlk `for` döngüsü şeridi kurar. `Y` üzerinde gezildiği için şerit kendiliğinden y'ye göre sıralıdır; ayrıca sıralamaya gerek yoktur.
- İç içe döngüler şeridi tarar. İç döngünün koşulu `s.get(j).y - p.y < d` 'dir; `d` küçüldükçe döngü daha erken durur. Daha yakın bir çift bulunursa `d` güncellenir.

```
List<Nokta> Ysol = new ArrayList<>(), Ysag = new ArrayList<>();
Set<Nokta> solYari = new HashSet<>(X.subList(0, orta));
for (Nokta p : Y)                  // y sırası korunur
  if (solYari.contains(p)) Ysol.add(p);
  else Ysag.add(p);
```

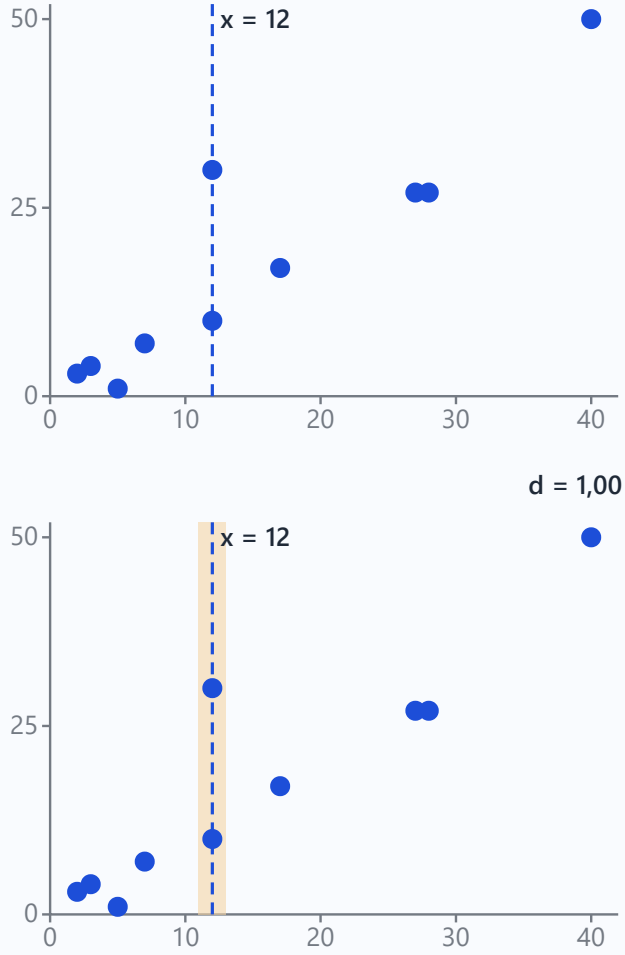
Bu parça `Ysol` ve `Ysag`'ı tek geçişte kurar. Sol yarının noktaları bir `HashSet`'e konur; `Y` sırayla gezilir ve her nokta, sol yarıda olup olmamasına göre iki listeden birine eklenir. `Nokta` sınıfı `equals` metodunu yeniden tanımlamadığı için küme nesnelerin kimliğiyle çalışır; aynı koordinatlı iki farklı nokta bile karışmaz. Geçiş beklenen *O(n)* sürer ve listeler *y* sırasını korur. Yeniden sıralama yapılmaz.

6.5.5 Kod izleme: 10 nokta

İzlemede on nokta kullanılır: (2,3), (12,30), (40,50), (5,1), (12,10), (3,4), (7,7), (17,17), (27,27), (28,27). *x*'e göre sıralı liste `X = [(2,3), (3,4), (5,1), (7,7), (12,30), (12,10), (17,17), (27,27), (28,27), (40,50)]`'dir. Aynı *x*'e sahip (12,30) ve (12,10) noktaları, sıralama kararlı olduğu için girdideki sıralarını korur. *y*'ye göre sıralı liste `Y = [(5,1), (2,3), (3,4), (7,7), (12,10), (17,17), (27,27), (28,27), (12,30), (40,50)]`'dir.

Adım	Çağrı	Olay	Sonuç
1	`X[0..9]`	*n = 10 > 3*; orta = 5, *m = (12, 10)*; sol 5, sağ 5 nokta	—
2	`X[0..4]`	*n = 5*; orta = 2, *m = (5, 1)*; sol 2, sağ 3 nokta	—
3	`X[0..1]`	kaba kuvvet: (2,3), (3,4)	1,41
4	`X[2..4]`	kaba kuvvet: (5,1), (7,7), (12,30); en yakın (5,1)–(7,7)	6,32
5	`X[0..4]`	*d = min(1,41; 6,32) = 1,41*	—
6	`X[0..4]`	şerit $\lvert x - 5\rvert < 1{,}41$: [(5,1)]; karşılaştırma yok	1,41
7	`X[5..9]`	*n = 5*; orta = 2, *m = (27, 27)*; sol 2, sağ 3 nokta	—
8	`X[5..6]`	kaba kuvvet: (12,10), (17,17)	8,60
9	`X[7..9]`	kaba kuvvet: (27,27), (28,27), (40,50); en yakın (27,27)–(28,27)	1,00
10	`X[5..9]`	*d = min(8,60; 1,00) = 1,00*	—
11	`X[5..9]`	şerit $\lvert x - 27\rvert < 1$: [(27,27)]; karşılaştırma yok	1,00
12	`X[0..9]`	*d = min(1,41; 1,00) = 1,00*	—
13	`X[0..9]`	şerit $\lvert x - 12\rvert < 1$: [(12,10), (12,30)]; *y* farkı 20 ≥ 1, karşılaştırma yok	1,00

Tablo 6.10: `enYakin(X, Y)` izlemesi. Uzaklıklar iki ondalık basamağa yuvarlanmıştır.



Şekil 6.14: On noktalı izleme. Solda ilk bölme: L doğrusu $x = 12$'dedir. Sağda son adım: $d = 1$ genişliğindeki dar şeritte yalnız (12,10) ve (12,30) kalır.

Sonuç $d = 1{,}00$'dır; en yakın çift (27,27)–(28,27)'dir. Bu çift sağ yarının sağ yarısında, tabandaki kaba kuvvetle bulunmuştur. Bu örnekte şeritler hiçbir çift karşılaştırması yapmaz. Bunun nedeni, δ'nın erkenden küçük çıkması ve şeritlerin çok daralmasıdır. Algoritmanın toplam işi 8 uzaklık hesabıdır (3. adımda 1, 4. adımda 3, 8. adımda 1, 9. adımda 3). Kaba kuvvet aynı girdide 45 uzaklık hesaplardı.

Şeridin gerçekten iş yaptığı bir örnek de görülmelidir. L doğrusu $x = 10$'da olsun ve iki yarıdan $\delta_L = 5$, $\delta_R = 4$ dönsün. `Y` listesi (8,1), (13,2), (11,4), (7,6), (12,9), (9,10), (15,12) olsun.

Örnek 6.4

1. $d = \min(5, 4) = 4$. Şerit $|x - 10| < 4$ koşuluyla kurulur: (15,12) dışarıda kalır; şeritte y sırasıyla (8,1), (13,2), (11,4), (7,6), (12,9), (9,10) vardır.
2. (8,1): (13,2) ile uzaklık $\sqrt{26} \approx 5{,}10$, d değişmez. (11,4) ile $\sqrt{18} \approx 4{,}24$, değişmez. (7,6)'nın y farkı $5 \geq 4$; iç döngü durur.
3. (13,2): (11,4) ile $\sqrt{8} \approx 2{,}83$; $d = 2{,}83$ olur. (7,6)'nın y farkı $4 \geq 2{,}83$; durur.

4. (11,4): (7,6) ile $\sqrt{20} \approx 4{,}47$, değişmez. (12,9)'un y farkı 5; durur.
5. (7,6): (12,9)'un y farkı 3 ≥ 2,83; hiç karşılaştırma yapılmaz.
6. (12,9): (9,10) ile $\sqrt{10} \approx 3{,}16$, değişmez. Şeridin sonu.

Toplam 5 uzaklık hesaplanır. Sonuç $d \approx 2{,}83$'tür ve çift (13,2)–(11,4)'tür. Bu çift L'yi keser: bir noktası sağda, öbürü soldadır. İki yarıda ayrı ayrı aranınca bulunamazdı; birleştirme adımı onu yakalamıştır.

6.5.6 Analiz

Adım	Maliyet	Not
x ve y'ye göre ön sıralama	$O(n \log n)$	yalnız bir kez, özyinelemeden önce
Bölme (`X` ikiye, `Y` sırayı koruyarak `Ysol` / `Ysag` 'a)	$O(n)$	yeniden sıralama yok
Şeridi kurma	$O(n)$	`Y` zaten y'ye göre sıralı
Şeridi tarama	$O(7n) = O(n)$	her nokta en çok 7 komşu
Toplam	$O(n \log n)$	$T(n) = 2T(n/2) + O(n)$

Tablo 6.11: En yakın çift algoritmasının adım adım maliyeti.

Özyinelemenin her düzeyinde bölme, şerit kurma ve şerit tarama birlikte $O(n)$ sürer. Bağıntı birleştirme sıralamasınınkiyle aynıdır: $T(n) = 2T(n/2) + O(n)$ ve sonuç $O(n \log n)$. Başlangıçtaki iki sıralama da $O(n \log n)$ sürdüğü için toplam süre $O(n \log n)$'dir. Bu süre en iyisidir. Cebirsel karar ağacı modelinde, noktaların hepsinin farklı olup olmadığını sınamak bile $\Omega(n \log n)$ gerektirir. En yakın uzaklığın sıfır olup olmadığı da bu soruyu yanıtlar.

Sık yapılan hata

Şeridi her çağrıda yeniden sıralamak. `Y` listesi yerine şerit noktaları x sırasından alınıp her çağrıda y'ye göre sıralanırsa birleştirme maliyeti $O(n \log n)$ olur. Bağıntı $T(n) = 2T(n/2) + O(n \log n)$'ye dönüşür ve sonuç $O(n \log^2 n)$ olur. Fark küçük görünür, ama $n = 2^{20}$ için $\log_2 n = 20$ kat fazladan iş anlamına gelir. Doğrusu y sırasını bir kez hesaplamak ve bölerken korumaktır.

Taban durumun seçimi de algoritmayı etkiler. Eşik $n \le 3$ yerine $n \le 1$ olsaydı tek noktalı yarı `Double.MAX_VALUE`, yani pratikte sonsuz döndürürdü. Sonuç yine doğru olurdu. İki noktalı bir aralık iki tek noktaya bölünür ve d sonsuz kalır. Şerit iki noktayı da içerir ve şerit taraması kaçan çifti bulur. Ama çağrı sayısı artar ve küçük girdilerde gereksiz özyineleme yükü oluşur.

6.6 Karatsuba çarpımı

İki tam sayıyı çarpmak, bilgisayarın tek komutla yaptığı bir iş gibi görünür. Bu yalnız sayılar bir makine sözcüğüne, örneğin 64 bite sığdığı sürece doğrudur. Şifrelemede, bilimsel hesapta ve bilgisayar cebirinde yüzlerce ya da binlerce basamaklı sayılar çarpılır. Bu sayılar basamak dizileri olarak tutulur ve çarpmanın süresi basamak sayısına bağlıdır. 1960'ta Kolmogorov, iki n basamaklı sayının çarpımının $\Omega(n^2)$ işlem gerektirdiğini öne sürdü. Henüz öğrenci olan Anatoli Karatsuba kısa sürede bu savın yanlış olduğunu gösteren algoritmayı buldu.

6.6.1 Okul yöntemi ve saf bölme

İlkokulda öğrenilen çarpma yönteminde ikinci sayının her basamağı birinci sayının her basamağıyla çarpılır ve kaydırılmış ara sonuçlar toplanır. İki n basamaklı sayı için n^2 tek basamaklı çarpım yapılır: süre $O(n^2)$'dir.

Böl ve fethetin ilk denemesi sayıları ikiye bölmektir. n basamaklı x ve y, $m = \lfloor n/2 \rfloor$ noktasından bölünür:

$$x = a \cdot 10^m + b, \quad y = c \cdot 10^m + d$$

Burada a ve c sayıların üst yarıları, b ve d alt yarılarıdır. Örneğin $1234 = 12 \cdot 100 + 34$'tür. Çarpım açılınca

$$x \cdot y = ac \cdot 10^{2m} + (ad + bc) \cdot 10^m + bd$$

bulunur. 10^m ile çarpmak yalnız sola m sıfır eklemektir ve $O(n)$ sürer; toplamalar da $O(n)$ sürer. Asıl iş dört yarım boyutlu çarpımdır: ac, ad, bc, bd. Bağıntı $T(n) = 4T(n/2) + O(n)$ olur. Ana teoremde $n^{\log_2 4} = n^2$ baskındır ve sonuç $O(n^2)$'dir. Saf bölme hiçbir kazanç sağlamaz: dört çarpımın her biri $n^2/4$ sürer ve toplam yine n^2'dir.

6.6.2 Dört çarpım yerine üç

Karatsuba'nın gözlemi, ortadaki $ad + bc$ teriminin iki ayrı çarpım gerektirmediğidir. Şu eşitlik vardır:

$$(a + b)(c + d) = ac + (ad + bc) + bd \quad \Rightarrow \quad ad + bc = (a + b)(c + d) - ac - bd$$

ac ve bd zaten hesaplanacaktır. Bir çarpım daha, $(a + b)(c + d)$, yapılınca orta terim iki çıkarmayla elde edilir. Üç çarpım şöyle adlandırılır:

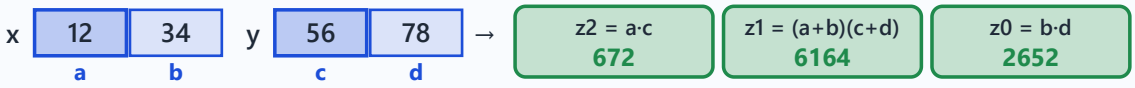
$$z_2 = ac, \quad z_0 = bd, \quad z_1 = (a + b)(c + d), \quad x \cdot y = z_2 \cdot 10^{2m} + (z_1 - z_2 - z_0) \cdot 10^m + z_0$$

Bir çarpımdan kurtulmanın bedeli birkaç fazladan toplama ve çıkarmadır. Toplamalar $O(n)$ sürer; çarpımlar ise yarım boyutlu birer problemdir ve özyinelemeyle çözülür. Bağıntı $T(n) = 3T(n/2) + O(n)$ olur.

Örnek 6.5

$1234 \cdot 5678$ hesaplansın. $n = 4$, $m = 2$, $10^m = 100$'dür.

1. Bölme: $a = 12$, $b = 34$, $c = 56$, $d = 78$.
2. $z_2 = 12 \cdot 56 = 672$, $z_0 = 34 \cdot 78 = 2652$.
3. $z_1 = (12 + 34)(56 + 78) = 46 \cdot 134 = 6164$.
4. Orta terim: $z_1 - z_2 - z_0 = 6164 - 672 - 2652 = 2840$. Denetim: $ad + bc = 12 \cdot 78 + 34 \cdot 56 = 936 + 1904 = 2840$.
5. Sonuç: $672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = 6.720.000 + 284.000 + 2652 = 7.006.652$.



Şekil 6.15: Karatsuba'nın ilk düzeyi: 1234 ve 5678 ikiye bölünür; üç çarpım $z_2 = a \cdot c$, $z_1 = (a + b)(c + d)$, $z_0 = b \cdot d$ hesaplanır.

6.6.3 Kod: karatsuba()

```
long karatsuba(long x, long y) {
  if (x < 10 || y < 10) return x * y; // taban
  int n = Math.max(basamak(x), basamak(y));
  long p = (long) Math.pow(10, n / 2);
  long a = x / p, b = x % p;   // x = a·p + b
  long c = y / p, d = y % p;   // y = c·p + d
  long z0 = karatsuba(b, d);
  long z1 = karatsuba(a + b, c + d);
  long z2 = karatsuba(a, c);
  return z2 * p * p + (z1 - z2 - z0) * p + z0;
}
```

Kod formülün doğrudan karşılığıdır:

- `if (x < 10 || y < 10) return x * y;` taban durumdur. Çarpanlardan biri tek basamaklıysa çarpım doğrudan yapılır. Koşulun "ya da" ile kurulması önemlidir: $7 \cdot 15$ gibi bir çarpımda bölme anlamsızdır, çünkü 7'nin üst yarısı yoktur.

- `n` iki sayının basamak sayılarının büyüğüdür; `p = 10^(n/2)` bölme noktasıdır. `Math.pow` bir `double` döndürür; 10'un küçük kuvvetleri `double`'da tam gösterildiği için `(long)` dönüşümü güvenlidir.
- `a = x / p` ve `b = x % p` üst ve alt yarıdır; `c` ve `d` de `y` için aynıdır.
- Üç özyinelemeli çağrı sırasıyla `z0`, `z1`, `z2`'yi hesaplar. Son satır sonuçları birleştirir.

Basamak sayısını veren yardımcı metot basittir:

```
int basamak(long x) {
  return String.valueOf(Math.abs(x)).length();
}
```

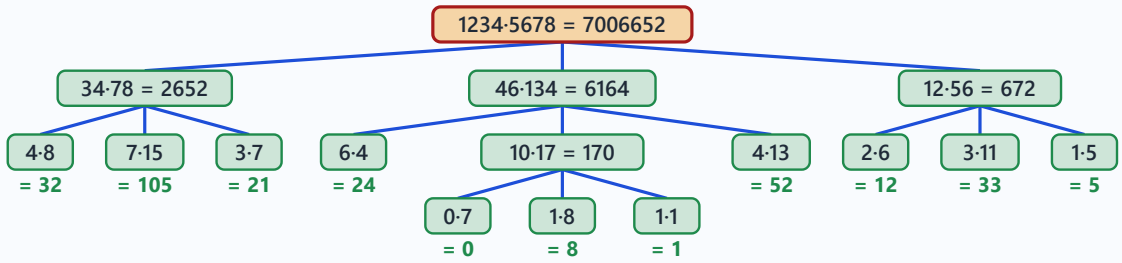
Bu Java kodu algoritmayı göstermek içindir. `long` türü en çok 19 basamaklı sayıları tuttuğu için büyük sayılarda kullanılamaz. Gerçek uygulamalar sayıları basamak dizileri olarak tutar; Java'da bu iş `BigInteger` sınıfıyla yapılır. Kodun doğruluğu yine de denetlenebilir: rastgele seçilen 100.000 sayı çiftinde `karatsuba(x, y)` ile `x * y` aynı sonucu vermiştir.

6.6.4 Kod izleme: 1234 · 5678

`karatsuba(1234, 5678)` çağrısında $n = 4$, $p = 100$'dür. Çağrı, *(34, 78)*, *(46, 134)* ve *(12, 56)* için üç alt çağrı yapar. Tablo 6.12 bütün çağrıları çağrılma sırasıyla verir.

Çağrı	*n*, *p*	a, b, c, d	Birleştirme	Dönüş
(1234, 5678)	4, 100	12, 34, 56, 78	$672 \cdot 10^4 + (6164 - 672 - 2652) \cdot 100 + 2652$	7.006.652
(34, 78)	2, 10	3, 4, 7, 8	$21 \cdot 100 + (105 - 21 - 32) \cdot 10 + 32$	2652
4 · 8, 7 · 15, 3 · 7	taban		doğrudan çarpım	32, 105, 21
(46, 134)	3, 10	4, 6, 13, 4	$52 \cdot 100 + (170 - 52 - 24) \cdot 10 + 24$	6164
6 · 4	taban		doğrudan çarpım	24
(10, 17)	2, 10	1, 0, 1, 7	$1 \cdot 100 + (8 - 1 - 0) \cdot 10 + 0$	170
0 · 7, 1 · 8, 1 · 1	taban		doğrudan çarpım	0, 8, 1
4 · 13	taban		doğrudan çarpım	52
(12, 56)	2, 10	1, 2, 5, 6	$5 \cdot 100 + (33 - 5 - 12) \cdot 10 + 12$	672
2 · 6, 3 · 11, 1 · 5	taban		doğrudan çarpım	12, 33, 5

Tablo 6.12: `karatsuba(1234, 5678)` izlemesi. Girintiler çağrı derinliğini gösterir.



Şekil 6.16: `karatsuba(1234, 5678)` çağrı ağacı. Her düğüm bir çarpımdır; altındaki sayı döndürdüğü değerdir. Yeşil düğümler taban durumdur.

İzlemenin ilginç noktası *(46, 134)* çağrısıdır. z_1 için yarıların toplamı alındığında sayılar bir basamak büyüyebilir: *12 + 34 = 46* iki basamaklıdır, ama *56 + 78 = 134* üç basamaklıdır. Kod bunu kendiliğinden karşılar: n iki sayının basamak sayısının büyüğüdür ve $p = 10^{\lfloor 3/2 \rfloor} = 10$ seçilir. Bölme $46 = 4 \cdot 10 + 6$ ve $134 = 13 \cdot 10 + 4$ olur. Bu bir basamaklık büyüme süre mertebesini değiştirmez.

Toplam 16 çağrı yapılır: 1 kök, 3 çocuk, 9 torun ve *(10, 17)*'nin 3 çocuğu. 11 çağrı taban durumdur. Okul yöntemi aynı çarpım için 16 tek basamaklı çarpım yapar. Bu küçük örnekte kazanç belirgin değildir, çünkü taban çarpımlarından bazıları ($7 \cdot 15$, $4 \cdot 13$, $3 \cdot 11$) tek basamaklı çarpımdan büyüktür. Kazanç büyük sayılarda ortaya çıkar.

6.6.5 Analiz ve pratik

Bağıntı $T(n) = 3T(n/2) + O(n)$'dir. $n^{\log_2 3} \approx n^{1,585}$, $f(n) = n$'den polinom olarak büyüktür. Ana teoremin 1. durumundan $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1,585})$ çıkar. Aynı sonuç sayarak da bulunur. $n = 2^k$ basamaklı sayılarda her düzeyde çarpım sayısı 3 katına çıkar ve k düzey sonra tek basamaklı çarpımlara inilir. Tek basamaklı çarpım sayısı $3^k = 3^{\log_2 n} = n^{\log_2 3}$'tür. Örneğin $n = 8$ için $3^3 = 27$ çarpım yapılır; okul yöntemi 64 çarpım yapar.

n (basamak)	n^2	$n^{1,585}$	Oran
10	100	38	2,6×
100	10.000	1.479	6,8×
1.000	$1{,}0 \cdot 10^6$	56.871	17,6×
10.000	$1{,}0 \cdot 10^8$	$2{,}2 \cdot 10^6$	45,7×
$1{,}0 \cdot 10^6$	$1{,}0 \cdot 10^{12}$	$3{,}2 \cdot 10^9$	309,2×

Tablo 6.13: n^2 ile $n^{1,585}$'in karşılaştırılması (sabit çarpanlar yok sayılmıştır).

Oran $n^{0,415}$ gibi büyür: bin basamakta yaklaşık 18, bir milyon basamakta yaklaşık 300 kat. Ancak tablo sabit çarpanları göstermez. Karatsuba her düzeyde fazladan toplama, çıkarma ve geçici sayılar kullanır. Küçük sayılarda bu yük kazancı aşar ve Karatsuba okul yönteminden yavaştır. Bu yüzden uygulamalar bir eşik kullanır: sayı belirli bir basamak sayısının altına inince okul yöntemine geçilir.

Java'nın `BigInteger.multiply` metodu bu yaklaşımı izler: küçük sayılarda okul yöntemini, büyük sayılarda Karatsuba'yı, daha büyüklerinde Toom-Cook yöntemini kullanır. Toom-Cook, Karatsuba fikrinin genellemesidir: sayıyı üç parçaya bölüp dokuz yerine beş çarpım yapar ve $O(n^{\log_3 5}) \approx O(n^{1{,}465})$ süre verir. Daha da hızlı algoritmalar hızlı Fourier dönüşümüne dayanır. Schönhage–Strassen algoritması $O(n \log n \log\log n)$ sürer. Harvey ve van der Hoeven'in 2019'da duyurduğu algoritma $O(n \log n)$ sürer. Bu son algoritma kuramsal öneme sahiptir; sabitleri çok büyük olduğu için pratikte kullanılmaz.

6.7 Strassen matris çarpımı

Karatsuba'nın fikri matrislere de uygulanır. 1969'da Volker Strassen, iki matrisin çarpımının klasik $\Theta(n^3)$ yöntemden hızlı yapılabileceğini gösterdi. Yöntem yine bir özyinelemeli çağrıyı eksiltmeye dayanır: sekiz blok çarpımı yerine yedi.

6.7.1 Klasik ve blok matris çarpımı

$n \times n$ boyutlu A ve B matrislerinin çarpımı $C = A \cdot B$'nin her hücresi, A'nın bir satırıyla B'nin bir sütununun iç çarpımıdır: $C[i][j] = \Sigma_k A[i][k] \cdot B[k][j]$. Klasik yöntem bu tanımı üç iç içe döngüyle yazar:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```

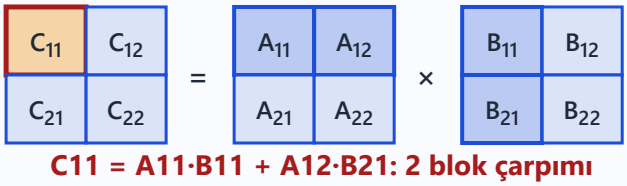
n^2 hücrenin her biri için n çarpım yapılır: toplam n^3 çarpım. Kodda `C`'nin başlangıçta sıfırlarla dolu olduğu varsayılır; Java'da yeni oluşturulan dizi zaten sıfırlarla doludur.

Böl ve fethetin ilk denemesi, her matrisi $n/2 \times n/2$'lik dört bloğa bölmektir:

> $$A = [A_{11}\, A_{12};\, A_{21}\, A_{22}],\quad B = [B_{11}\, B_{12};\, B_{21}\, B_{22}],\quad C = [C_{11}\, C_{12};\, C_{21}\, C_{22}]$$

Bloklar sayı gibi çarpılıp toplanabilir; yalnız çarpımın sırası değiştirilemez. Sonuç blokları şöyledir:

- $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
- $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
- $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
- $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$



Şekil 6.17: Blok matris çarpımı. C_{11}, A'nın ilk blok satırı ile B'nin ilk blok sütunundan hesaplanır: iki blok çarpımı ve bir blok toplamı.

Sekiz blok çarpımı ve dört blok toplamı gerekir. Bir blok toplamı $(n/2)^2$ hücreyi toplar ve $\Theta(n^2)$ sürer. Bağıntı $T(n) = 8T(n/2) + \Theta(n^2)$ olur; $n^{\log_2 8} = n^3$ baskındır ve sonuç $\Theta(n^3)$'tür. Karatsuba'daki saf bölmede olduğu gibi yalnız bölmek yetmez; çarpım sayısı azaltılmalıdır.

6.7.2 Sekiz yerine yedi çarpım

Strassen, yedi ara çarpımın uygun toplamlarıyla dört sonuç bloğunun da elde edilebileceğini gösterdi. Tablo 6.14 bu çarpımları ve sonuç bloklarını verir.

Çarpım	Tanım	Sonuç bloğu
P_1	$A_{11} \cdot (B_{12} - B_{22})$	$C_{11} = P_5 + P_4 - P_2 + P_6$
P_2	$(A_{11} + A_{12}) \cdot B_{22}$	$C_{12} = P_1 + P_2$
P_3	$(A_{21} + A_{22}) \cdot B_{11}$	$C_{21} = P_3 + P_4$
P_4	$A_{22} \cdot (B_{21} - B_{11})$	$C_{22} = P_5 + P_1 - P_3 - P_7$
P_5	$(A_{11} + A_{22}) \cdot (B_{11} + B_{22})$	
P_6	$(A_{12} - A_{22}) \cdot (B_{21} + B_{22})$	
P_7	$(A_{11} - A_{21}) \cdot (B_{11} + B_{12})$	

Tablo 6.14: Strassen'in yedi çarpımı ve sonuç blokları.

Formüllerin doğruluğu açılarak denetlenir. C_{12} en kolay olanıdır: $P_1 + P_2 = A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{22} + A_{12}B_{22} = A_{11}B_{12} + A_{12}B_{22}$. C_{11} için P_5, P_4, P_2 ve P_6 açılınca on iki terim yazılır. Bunlardan onu birbirini götürür ve geriye $A_{11}B_{11} + A_{12}B_{21}$ kalır. Burada çarpımların sırası korunmalıdır: matrislerde $A_{11}B_{11}$ ile $B_{11}A_{11}$ genellikle farklıdır. Strassen'in formülleri hiçbir yerde çarpımın değişme özelliğini kullanmaz; bu yüzden bloklar için de geçerlidir.

Yedi çarpımın yanında 18 blok toplama ya da çıkarma yapılır: P'lerin tanımında 10, sonuç bloklarında 8. Bloklar da Strassen ile özyinelemeli çarpılır; taban durum 1×1 matristir, yani tek bir sayı çarpımıdır.

6.7.3 Adım adım: 2 × 2 örnek

Örnek 6.6

$A = [[1, 3], [7, 5]]$ ve $B = [[6, 8], [4, 2]]$ olsun. Bloklar 1×1 olduğundan her blok tek bir sayıdır: $A_{11} = 1, A_{12} = 3, A_{21} = 7, A_{22} = 5$; $B_{11} = 6, B_{12} = 8, B_{21} = 4, B_{22} = 2$.

1. $P_1 = 1 \cdot (8 - 2) = 6$
2. $P_2 = (1 + 3) \cdot 2 = 8$
3. $P_3 = (7 + 5) \cdot 6 = 72$
4. $P_4 = 5 \cdot (4 - 6) = -10$
5. $P_5 = (1 + 5) \cdot (6 + 2) = 48$
6. $P_6 = (3 - 5) \cdot (4 + 2) = -12$
7. $P_7 = (1 - 7) \cdot (6 + 8) = -84$
8. $C_{11} = 48 + (-10) - 8 + (-12) = 18$, $C_{12} = 6 + 8 = 14$, $C_{21} = 72 + (-10) = 62$, $C_{22} = 48 + 6 - 72 - (-84) = 66$.

Sonuç $C = [[18, 14], [62, 66]]$'dır. Klasik yöntemle denetim: $1 \cdot 6 + 3 \cdot 4 = 18$, $1 \cdot 8 + 3 \cdot 2 = 14$, $7 \cdot 6 + 5 \cdot 4 = 62$, $7 \cdot 8 + 5 \cdot 2 = 66$. Strassen 7 çarpım kullanmıştır; klasik yöntem 8 çarpım kullanır.

Sık yapılan hata

Strassen formüllerinde işaret hatası yapmak. $C_{22} = P_5 + P_1 - P_3 - P_7$'dir; P_7 çıkarılır. "$+ P_7$" yazılırsa program hata vermeden yanlış matris üretir: yukarıdaki örnekte C_{22} 66 yerine $48 + 6 - 72 + (-84) = -102$ çıkar. Bu tür hata derleyicinin ya da çalışma zamanının yakalayabileceği bir hata değildir. Uygulamayı küçük rastgele matrislerde klasik yöntemle karşılaştırmak bu hatayı hemen gösterir.

6.7.4 Analiz ve pratik

Bağıntı $T(n) = 7T(n/2) + \Theta(n^2)$'dir. $n^{\log_2 7} \approx n^{2{,}807}$, n^2'den polinom olarak büyüktür ve ana teoremin 1. durumundan $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2{,}807})$ çıkar. Sayarak da görülür: $n = 2^k$ için skaler çarpım sayısı 7^k'dir; klasik yöntem $8^k = n^3$ çarpım yapar. 16×16 matrislerde Strassen 2401, klasik yöntem 4096 çarpım yapar. $n = 1024 = 2^{10}$ için $7^{10} \approx 2{,}8 \cdot 10^8$ ve $8^{10} \approx 1{,}07 \cdot 10^9$'dur; Strassen yaklaşık 3,8 kat daha az çarpım yapar.

Algoritmanın pratikte bazı sorunları vardır:

- **Boyut.** n 2'nin kuvveti değilse matris sıfırlarla doldurularak en yakın kuvvete büyütülür (*padding*). Sıfırlar sonucu değiştirmez, ama işi artırır.

- **Eşik.** 18 ek toplama ve geçici matrisler yüzünden küçük n'de Strassen klasik yöntemden yavaştır. Kütüphaneler alt matris boyu 64–128 civarına inince klasik yönteme geçer.
- **Sayısal kararlılık.** Kayan noktalı sayılarda çıkarma işlemleri yuvarlama hatasını büyütebilir. Klasik yöntem bu açıdan daha güvenilirdir.
- **Bellek.** Her düzeyde geçici bloklar oluşturulur; ek bellek $O(n^2)$'dir.

Strassen'den sonra matris çarpımının üssü ω için daha küçük üst sınırlar bulundu. 2024 itibarıyla bilinen sınır $\omega < 2{,}372$'dir. Bu algoritmaların sabitleri o kadar büyüktür ki pratikte kullanılmazlar. Strassen'in asıl dersi kuramsaldır: çarpımı toplamalarla takas etmek, a'yı bir azaltarak üssü düşürür. Aynı fikir Karatsuba'da 4'ten 3'e, Strassen'de 8'den 7'ye inişte görülür.

6.8 Değerlendirme

Tablo 6.15 bu notta ele alınan problemlerde saf yöntem ile böl ve fethet çözümünü yan yana koyar.

Problem	Saf yöntem	Böl ve fethet	Bağıntı
Sıralama	$O(n^2)$ (kabarcık)	$O(n \log n)$	$2T(n/2) + n$
Sıralı dizide arama	$O(n)$	$O(\log n)$	$T(n/2) + 1$
Üs alma x^n	$O(n)$ çarpım	$O(\log n)$	$T(n/2) + 1$
Dışbükey örtü	$O(n^3)$	ort. $O(n \log n)$	$T(n_1) + T(n_2) + n$
En yakın nokta çifti	$O(n^2)$	$O(n \log n)$	$2T(n/2) + n$
Tam sayı çarpımı	$O(n^2)$	$O(n^{1{,}585})$	$3T(n/2) + n$
Matris çarpımı	$O(n^3)$	$O(n^{2{,}807})$	$7T(n/2) + n^2$

Tablo 6.15: Saf yöntem ve böl ve fethet.

Kazancın kaynağı üç türlüdür. Birincisi, iş dengeli bölünür ve her düzey $O(n)$'de birleştirilir: sıralama ve en yakın nokta çifti. İkincisi, alt problem sayısı azaltılır: Karatsuba ve Strassen. Üçüncüsü, alt problemlerden biri tamamen atılır ya da tek bir alt problemin sonucu yeniden kullanılır: ikili arama ve hızlı üs alma. Quickhull ise birinci ve üçüncü türün bir karışımıdır: üçgenin içindeki noktalar atılır ve kalan iş iki dengesiz parçaya bölünür.

Bir problemi böl ve fethetle çözmeye karar verirken sorulacak sorular şunlardır: Problem aynı türden küçük parçalara bölünebiliyor mu? Parçalar birbirinden bağımsız mı? Parçaların çözümleri, saf yöntemden daha ucuza birleştirilebiliyor mu? İlk iki sorunun yanıtı evet, üçüncüsünün yanıtı hayırsa böl ve fethet doğru ama yavaş bir algoritma verir; en büyük öğe örneğinde olduğu gibi. Parçalar örtüşüyorsa dinamik programlama, her adımda yerel en iyi seçim yetiyorsa açgözlü yaklaşım daha uygundur.

6.9 Sık yapılan hatalar

- **Taban durumu unutmak** ya da hiç ulaşılamayan bir taban durum yazmak. Sonuç sonsuz özyineleme ve `StackOverflowError`'dır. Karatsuba'da taban durum `x < 10 || y < 10`'dur. Bu satır silinirse tek basamaklı sayılarda $p = 10^0 = 1$ olur. `karatsuba(4, 8)` ilk alt çağrıda `karatsuba(0, 0)`'ı çağırır; o da kendini aynı argümanlarla yeniden çağırır ve özyineleme hiç bitmez.
- **Dengesiz bölme.** Sıralı dizide son öğenin pivot seçilmesi ya da Quickhull'da her adımda yalnız bir noktanın ayrılması $T(n) = T(n-1) + n = O(n^2)$ verir.
- **Örtüşen alt problemleri fark etmemek.** Fibonacci'yi düz özyinelemeyle hesaplamak üstel süre alır; sonuçları saklayan dinamik programlama gerekir.
- **Aynı alt problemi iki kez çağırmak.** Hızlı üs almada `us(x, n / 2)`'yi iki kez çağırmak $\Theta(\log n)$'yi $\Theta(n)$'ye çıkarır.
- **Birleştirme maliyetini küçümsemek.** En yakın nokta çiftinde şeridi her çağrıda sıralamak süreyi $O(n \log^2 n)$ yapar; $2T(n/2) + n \log n \neq \Theta(n \log n)$.
- **Ana teoremi yanlış yerde kullanmak.** $T(n) = T(n-1) + n$, eşit olmayan bölmeler ve $f(n) = n/\log n$ gibi durumlar arasında kalan fonksiyonlar teoremin kapsamı dışındadır.
- **İşaret ve yön hataları.** Strassen'de $C_{22} = P_5 + P_1 - P_3 - P_7$'deki tek bir işaret hatası hata vermeden yanlış matris üretir. Quickhull'da `capraz`'a verilen nokta sırası karışırsa üst ve alt küme yer değiştirir. Küçük bir örnekte saf yöntemle karşılaştırma bu hataları yakalar.
- **Böl ve fethetin her zaman hızlandırdığını sanmak.** En büyük öğe probleminde böl ve fethet de $n - 1$ karşılaştırma yapar; kazanç süre değil, paralelliktir.

6.10 Özet

- Böl ve fethet, problemi aynı türden, küçük ve bağımsız alt problemlere böler, alt problemleri özyinelemeyle çözer ve sonuçları birleştirir. Taban durum şarttır.
- Süre $T(n) = aT(n/b) + f(n)$ bağıntısıyla ifade edilir. Ana teorem $f(n)$'yi $n^{\log_b a}$ ile karşılaştırır: yapraklar baskınsa $\Theta(n^{\log_b a})$, eşitse bir $\log n$ çarpanı eklenir, kök baskınsa $\Theta(f(n))$.
- Eşit olmayan ya da sabit miktarda küçülen bölmelerde ana teorem uygulanmaz; özyineleme ağacı kullanılır.
- Birleştirme sıralamasında asıl iş birleştirmede, hızlı sıralamada bölümlemededir. İkili arama ve hızlı üs alma tek alt problemle $O(\log n)$ sürer.
- Alt problemler örtüşüyorsa (Fibonacci) böl ve fethet üstel süre alır; dinamik programlama gerekir.
- Quickhull en sol ve en sağ noktadan başlar, her adımda doğruya en uzak noktayı (en büyük `capraz`) örtüye ekler ve üçgenin içini eler. Ortalama $O(n \log n)$, en kötü $O(n^2)$, çıktıya duyarlı $O(n \cdot h)$ sürer.
- En yakın nokta çiftinde noktalar x'e göre ikiye bölünür, $\delta = \min(\delta_L, \delta_R)$ alınır ve 2δ genişliğindeki şerit y sırasıyla taranır. Her nokta en çok 7 komşuyla karşılaştırılır; süre $O(n \log n)$'dir.

- Karatsuba $ad + bc = (a + b)(c + d) - ac - bd$ eşitliğiyle dört çarpım yerine üç yapar: $\Theta(n^{1,585})$.
- Strassen sekiz blok çarpımı yerine yedi yapar: $\Theta(n^{2,807})$. Her iki algoritmada kazanç, a'yı bir azaltmaktan gelir; küçük girdilerde bir eşiğin altında saf yönteme geçilir.

6.11 Kavram listesi

böl ve fethet *divide and conquer* · taban durum *base case* · alt problem *subproblem* · özyineleme *recursion* · yineleme bağıntısı *recurrence (relation)* · özyineleme ağacı *recursion tree* · ana teorem *master theorem* · birleştirme sıralaması *merge sort* · hızlı sıralama *quick sort* · bölümleme *partition* · ikili arama *binary search* · örtüşen alt problemler *overlapping subproblems* · dinamik programlama *dynamic programming* · dışbükey örtü *convex hull* · en yakın nokta çifti *closest pair of points* · kaba kuvvet *brute force* · alt sınır *lower bound* · Karatsuba çarpımı *Karatsuba multiplication* · Strassen algoritması *Strassen's algorithm*

6.12 Kendini Dene

1. Şu bağıntıların ana teoreme göre çözümü nedir? (a) $T(n) = 3T(n/3) + n$, (b) $T(n) = T(n/2) + n$, (c) $T(n) = 16T(n/4) + n^2$.

   Cevap

   (a) $n^{\log_3 3} = n = f(n)$; durum 2: $\Theta(n \log n)$. (b) $n^{\log_2 1} = 1$, $f(n) = n$ polinom olarak büyük; düzenlilik $f(n/2) = n/2 \le \tfrac{1}{2} \cdot n$ sağlanır; durum 3: $\Theta(n)$. (c) $n^{\log_4 16} = n^2 = f(n)$; durum 2: $\Theta(n^2 \log n)$.

2. `enBuyuk` metodu `[3, 8, 1, 9, 4, 6]` dizisinde `enBuyuk(a, 0, 5)` ile çağrılıyor. Toplam kaç çağrı ve kaç karşılaştırma yapılır, sonuç nedir?

   Cevap

   Altı öğe için özyineleme ağacında 6 yaprak ve 5 iç düğüm vardır: 11 çağrı, 5 karşılaştırma. İlk bölme `[0..2]` ve `[3..5]`'tir; sol yarının en büyüğü 8, sağ yarınınki 9'dur. Sonuç 9'dur.

3. `us(2, 25)` kaç özyinelemeli çağrı ve kaç çarpım yapar? Saf yöntem kaç çarpım yapar?

   Cevap

   Üs dizisi $25 \to 12 \to 6 \to 3 \to 1 \to 0$'dır: 6 çağrı. Çarpımlar: $n = 1$ (tek) 2, $n = 3$ (tek) 2, $n = 6$ (çift) 1, $n = 12$ (çift) 1, $n = 25$ (tek) 2; toplam 8. Sonuç $2^{25} = 33.554.432$'dir. Saf yöntem 24 çarpım yapar.

4. $A = (0, 0)$, $B = (10, 0)$ ve üst kümede (2,3), (5,6), (8,2), (5,2) noktaları olsun. Quickhull hangi noktayı C seçer? Hangi noktalar elenir, hangisiyle özyineleme sürer?

   Cevap

   $capraz(A, B, P) = 10 \cdot P_y$: değerler 30, 60, 20, 20'dir; $C = (5, 6)$. $capraz(A, C, (2,3)) = 5 \cdot 3 - 6 \cdot 2 = 3 > 0$: (2,3), AC'nin dışındadır. (8,2) için $capraz(C, B, \cdot) = 5 \cdot (-4) - (-6) \cdot 3 = -2$, (5,2) için $capraz(A, C, \cdot) = -20$ ve $capraz(C, B, \cdot) = -20$'dir: ikisi de üçgenin içindedir ve elenir. Özyineleme AC tarafında yalnız (2,3) ile sürer; CB tarafı boştur.

5. En yakın çift algoritmasında L doğrusu $x = 10$'dadır ve iki yarıdan $\delta = 3$ dönmüştür. `Y` listesi (9,1), (11,2), (14,4), (10,5), (8,6), (12,7)'dir. Şerit hangi noktalardan oluşur, kaç uzaklık hesaplanır ve sonuç ne olur?

   Cevap

   (14,4) için $|14 - 10| = 4 \geq 3$; şerit dışındadır. Şerit: (9,1), (11,2), (10,5), (8,6), (12,7). (9,1)–(11,2) uzaklığı $\sqrt{5} \approx 2{,}24$'tür ve d 2,24 olur; (10,5)'in y farkı 4 olduğu için iç döngü durur. (11,2)'den sonra (10,5)'in y farkı 3; durur. (10,5)–(8,6) = 2,24 ve (10,5)–(12,7) ≈ 2,83 hesaplanır. (8,6)–(12,7) ≈ 4,12 hesaplanır. Toplam 4 uzaklık; sonuç $d = \sqrt{5} \approx 2{,}24$.

6. `karatsuba(56, 78)` çağrısında hangi alt çağrılar yapılır? Kaç çağrı yapılır, kaçı taban durumdur?

   Cevap

   $p = 10$; $a = 5$, $b = 6$, $c = 7$, $d = 8$. $z_0 = 6 \cdot 8 = 48$ (taban). $z_1 = karatsuba(11, 15)$: iç bölmede $1, 1, 1, 5$; iç çarpımlar $1 \cdot 5 = 5$, $2 \cdot 6 = 12$, $1 \cdot 1 = 1$; sonuç $100 + (12 - 1 - 5) \cdot 10 + 5 = 165$. $z_2 = 5 \cdot 7 = 35$ (taban). Sonuç $35 \cdot 100 + (165 - 35 - 48) \cdot 10 + 48 = 4368$. Toplam 7 çağrı, 5'i taban durum.

7. $A = [[1, 0], [2, 1]]$ ve $B = [[3, 1], [0, 2]]$ matrislerinin Strassen ile çarpımında $P_1, \ldots, P_7$ ve C nedir?

   Cevap

   $P_1 = 1 \cdot (1 - 2) = -1$, $P_2 = (1 + 0) \cdot 2 = 2$, $P_3 = (2 + 1) \cdot 3 = 9$, $P_4 = 1 \cdot (0 - 3) = -3$, $P_5 = (1 + 1)(3 + 2) = 10$, $P_6 = (0 - 1)(0 + 2) = -2$, $P_7 = (1 - 2)(3 + 1) = -4$. $C_{11} = 10 - 3 - 2 - 2 = 3$, $C_{12} = -1 + 2 = 1$, $C_{21} = 9 - 3 = 6$, $C_{22} = 10 - 1 - 9 + 4 = 4$. $C = [[3, 1], [6, 4]]$; klasik yöntem de aynı sonucu verir.

8. Karatsuba fikri genelleştirilip sayılar üç parçaya bölünüyor ve dokuz parça çarpımı yerine beş çarpım yapılıyor. Bölme ve birleştirme $O(n)$ sürüyorsa algoritmanın süresi nedir? Karatsuba'dan hızlı

mıdır?

Cevap

$T(n) = 5T(n/3) + O(n)$. $n^{\log_3 5} \approx n^{1,465}$ baskındır; durum 1: $\Theta(n^{1,465})$. Karatsuba'nın $n^{1,585}$'inden asimptotik olarak hızlıdır; sabitleri daha büyük olduğu için ancak daha büyük sayılarda öne geçer. Toom-Cook yöntemi budur.

9. Saf özyinelemeli `fib(6)` kaç çağrı yapar? Neden böl ve fethet burada uygun değildir?

Cevap

$C(n) = 1 + C(n-1) + C(n-2)$, $C(0) = C(1) = 1$: $C(5) = 15$, $C(4) = 9$, $C(6) = 25$. Alt problemler bağımsız değildir: $F(4)$ hesaplanırken $F(3)$, $F(2)$ … zaten hesaplanır ve ikinci daldan yeniden hesaplanır. Sonuçlar saklanırsa (dinamik programlama) her $F(k)$ bir kez hesaplanır ve süre $O(n)$ olur.

6.13 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 4: Strassen algoritmasını, özyineleme ağacı ve yerine koyma yöntemlerini ve ana teoremin ispatını verir. Bölüm 33 dışbükey örtü ve en yakın nokta çiftini işler.
- J. Kleinberg, É. Tardos, *Algorithm Design* (2005), Bölüm 5: böl ve fethet bağıntılarını sezgisel olarak açar; en yakın nokta çifti ve tam sayı çarpımı için okunaklı anlatımlar içerir.
- C. B. Barber, D. P. Dobkin, H. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls” (1996): Quickhull'u yüksek boyutlara genelleyen ve yaygın kullanılan Qhull yazılımının temelini oluşturan makale.
- V. Strassen, “Gaussian Elimination is not Optimal” (1969): yedi çarpım fikrini tanıtan ve yalnız birkaç sayfadan oluşan özgün makale.

6.14 Kaynaklar

1. C. B. Barber, D. P. Dobkin, H. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls”, *ACM Transactions on Mathematical Software* 22(4), 1996.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 4: Divide-and-Conquer; Bölüm 33: Computational Geometry).
3. A. Karatsuba, Yu. Ofman, “Multiplication of Multidigit Numbers on Automata”, 1962.
4. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005 (Bölüm 5: Divide and Conquer).
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
6. V. Strassen, “Gaussian Elimination is not Optimal”, *Numerische Mathematik* 13, 1969.

KISIM III · BÖLÜM 7

Açgözlü Algoritmalar

Bir kasiyer para üstünü verirken önce en büyük banknotu, sonra sığan en büyük madeni parayı uzatır. Bir salon görevlisi gün içindeki toplantıları yerleştirirken önce en erken biten toplantıyı takvime yazar. İkisi de geleceği hesaplamaz; o an en iyi görünen adımı atar ve bu kararı bir daha değiştirmez. Bu tür algoritmalara açgözlü algoritma denir. Açgözlü yaklaşım hızlıdır ve kodu kısadır, ama her problemde en iyi sonucu vermez. Not önce yaklaşımın genel kalıbını ve doğru sonuç verdiği koşulları, yani açgözlü seçim özelliğini ve optimal alt yapıyı tanıtır. Doğruluğu kanıtlamanın temel aracı olan değiştirme argümanı burada ayrıntılı işlenir. Ardından beş klasik problem ele alınır: para üstü, etkinlik seçimi, son tarihli iş sıralama, kesirli sırt çantası ve Huffman kodlama. Minimum kapsayan ağaç algoritmaları ise kesim özelliği üzerinden açgözlü bir gözle yeniden değerlendirilir. Her problem için Java kodu, adım adım izleme, maliyet hesabı ve açgözlü yaklaşımın yanıldığı durumlara karşı örnekler verilir.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum07/src`

ÖĞRENME ÇIKTILARI

Yaklaşım
Açgözlü stratejiyi tanımlamak; açgözlü seçim özelliğinin ve optimal alt yapının hangi durumda doğru sonucu garanti ettiğini açıklamak; değiştirme argümanıyla basit bir doğruluk kanıtı kurmak.

Klasik problemler
Para üstü, etkinlik seçimi, iş sıralama ve kesirli sırt çantası problemlerini açgözlü olarak çözmek; kodlarını adım adım izlemek; açgözlü yaklaşımın başarısız olduğu durumlar için karşı örnek vermek.

Kapsayan ağaçlar
Kesim özelliğinin Kruskal, Prim ve Borůvka'nın açgözlü seçimini neden doğru kıldığını açıklamak; üç algoritmayı seçim kuralı, veri yapısı ve süre bakımından karşılaştırmak.

Veri sıkıştırma
Huffman ağacını kurmak, önek-özgür kodları türetmek, bir metni kodlayıp çözmek ve kazancı bit olarak hesaplamak; Huffman kodunun en iyi olduğunu gerekçelendirmek.

7.1 Açgözlü yaklaşım

Birçok algoritma problemi bir dizi karardan oluşur. Hangi para verilecek, hangi etkinlik seçilecek, hangi iş hangi saate konacak, hangi kenar ağaca eklenecek? Bu kararların sırasıyla verilmesi gerekir ve her karar sonrakileri etkiler. Bütün karar dizilerini tek tek denemek doğru sonucu verir, ama dizilerin sayısı girdiyle birlikte üstel olarak büyür. Açgözlü yaklaşım bu ağacın yalnız tek bir dalını izler: her adımda o an en iyi görünen seçeneği alır ve yoluna devam eder.

7.1.1 Adım adım karar vermek

> **Tanım 7.1**
>
> **Açgözlü algoritma** (*greedy algorithm*), çözümü adım adım kuran ve her adımda bir ölçüte göre *o an* en iyi görünen seçeneği seçen algoritmadır. Verilen bir karar sonradan geri alınmaz; algoritma geri izleme (*backtracking*) yapmaz.

Açgözlü bir algoritma bir define avcısına benzer. Avcı her yol ayrımında en çok altın görünen yöne sapar. Haritanın bütününe bakmaz; yalnız önündeki seçeneklere odaklanır. Yolculuk planlarken her kavşakta hedefe en kısa görünen yolu seçmek de aynı davranıştır. Bazen bu kısa görünen yol bir çıkmaza ya da uzun bir dolambaca götürür. Algoritma geri dönmediği için bu yanlış adımı düzeltemez.

Yaklaşımın iki yüzü bu örnekte açıkça görülür. Bir yanda hız ve sadelik vardır. Her adımda yalnız bir seçenek seçilir, seçilmeyenler bir daha incelenmez. Çoğu açgözlü algoritma adayları bir kez sıralar ve sonra tek geçişte karar verir. Öbür yanda ise garanti eksikliği vardır. O an en iyi görünen seçim, bütünün en iyi çözümüne her zaman götürmez. Açgözlü algoritma birçok problemde iyi bir yaklaşık çözüm üretir, ama en iyi (*optimal*) çözümü bulduğu ancak kanıtla gösterilebilir.

Bu notta “en iyi çözüm”, “optimal çözüm” ve “optimum” aynı anlamda kullanılır. Bir problemin birden çok en iyi çözümü olabilir. Örneğin aynı sayıda etkinliği içeren iki farklı etkinlik kümesi, ikisi de en büyük sayıya ulaşıyorsa ikisi de en iyidir. Açgözlü algoritmanın bunlardan herhangi birini bulması yeterlidir.

7.1.2 Genel şablon

Açgözlü algoritmaların çoğu aynı iskeleti izler. Önce hangi seçimlerin kısıtları bozmadığı, yani **uygun** (olurlu) çözümlerin ne olduğu belirlenir. Ardından bir **açgözlü ölçüt** seçilir: en ucuz, en erken biten, en kârlı, en sık geçen gibi. Adaylar bu ölçüte göre sıralanır. Sonra her aday sırayla incelenir. Aday mevcut çözüme eklendiğinde kısıtlar bozulmuyorsa eklenir, bozuluyorsa atlanır. Çözüm tamamlanınca algoritma durur. Sunumdaki şablon kod bu iskeleti Java biçiminde gösterir:

```
Cozum acgozlu(Aday[] adaylar) {
  Cozum s = new Cozum();   // boş çözüm
  sirala(adaylar);         // ölçüte göre
  for (Aday a : adaylar) {
    if (s.uygunMu(a))      // kısıt bozulmaz
      s.ekle(a);           // geri alınmaz
  }
  return s;
}
```

Koddaki `Cozum` ve `Aday` soyut adlardır; her problemde başka bir anlam taşırlar. Para üstünde aday bir madeni para, çözüm verilen paraların listesidir. Etkinlik seçiminde aday bir etkinlik, çözüm seçilmiş etkinliklerin kümesidir. `sirala` çağrısı adayları açgözlü ölçüte göre dizer. `uygunMu` yeni adayın kısıtları bozup bozmadığını denetler. Döngü her adayı bir kez ele alır ve `ekle` ile çözüme katılan aday bir daha çıkarılmaz. Koddaki yorumlar bu iki özelliği vurgular: denetim kısıtları korur, ekleme ise geri alınmaz.



Şekil 7.1: Açgözlü şablonun tek geçişi. Adaylar ölçüte göre sıralanmıştır; her aday bir kez incelenir. Uygun olanlar (a1, a2, a4, a6) çözüme eklenir, kısıtı bozanlar (a3, a5) atlanır. Hiçbir karar geri alınmaz.

Şekil 7.1 bu tek geçişi gösterir. Altı aday sıralanmış olarak gelir. a1 ve a2 kısıtları bozmaz ve eklenir. a3, o ana kadar seçilenlerle çatışır ve atlanır. a4 eklenir, a5 atlanır, a6 eklenir. Sonuç *{a1, a2, a4, a6}* kümesidir. a3'ün atlanması kalıcıdır: a4 ve a6 eklendikten sonra a3'e geri dönülüp "şimdi uygun mu?" diye sorulmaz.

Şablonun maliyeti iki kısımdan oluşur. *n* adayın sıralanması *O(n log n)* sürer. Ardından gelen tarama her aday için bir `uygunMu` denetimi yapar. Denetim sabit sürede bitiyorsa tarama *O(n)* sürer ve toplam maliyeti sıralama belirler: *O(n log n)*. Bazı problemlerde adaylar baştan sıralanmaz; her adımda en iyi aday bir öncelikli kuyruktan alınır. Huffman kodlamada ve Prim algoritmasında durum budur. Kuyruk işlemlerinin her biri *O(log n)* sürdüğü için toplam maliyet yine çoğunlukla *O(n log n)* düzeyindedir. Denetim pahalıysa, örneğin iş sıralamadaki gibi bir boş zaman dilimi aranıyorsa, maliyetin baskın kısmı denetime geçer.

Sık yapılan hata

Şablonu doğru uygulamanın algoritmayı doğru yaptığını sanmak. Şablon yalnız bir iskelettir; sonucu ölçüt belirler. Aynı iskelet, etkinlik seçiminde "en erken biten" ölçütüyle en iyi sonucu verir, "en erken başlayan" ölçütüyle ise çok kötü bir sonuç verebilir. Her yeni problemde seçi-

len ölçütün doğru olduğu ayrıca kanıtlanmalıdır.

7.1.3 Açgözlü seçim özelliği ve optimal alt yapı

Açgözlü bir algoritmanın en iyi sonucu vermesi için problemin iki özelliği birlikte taşıması gerekir. Birincisi ilk adımla ilgilidir: algoritmanın yaptığı ilk seçim, en iyi çözüme giden yolu kapatmamalıdır. İkincisi geri kalan işle ilgilidir. İlk seçimden sonra kalan problem aynı türden, daha küçük bir problem olmalıdır. Onun en iyi çözümü ilk seçimle birleşince bütünün en iyi çözümü çıkmalıdır.

Tanım 7.2

Bir problemde, açgözlü ölçüte göre yerel olarak en iyi olan seçimi içeren *en az bir* en iyi çözüm varsa, problem **açgözlü seçim özelliği** (*greedy-choice property*) taşır. Başka bir deyişle, ilk açgözlü seçim hiçbir zaman "yanlış yola" sokmaz.

Tanım 7.3

Bir seçim yapıldıktan sonra kalan alt problemin en iyi çözümü, o seçimle birlikte bütün problemin en iyi çözümünü veriyorsa, problem **optimal alt yapı** (*optimal substructure*) taşır.

İki özelliğin birlikte ne sağladığı tümevarımla görülür. Açgözlü seçim özelliği, ilk seçimi içeren bir en iyi çözüm olduğunu söyler. Bu seçim yapıldıktan sonra kalan problem aynı türdendir. Optimal alt yapı, kalan problemin en iyi çözümü ile ilk seçimin birleşiminin bütünün en iyi çözümü olduğunu söyler. Kalan problemde de açgözlü seçim özelliği geçerli olduğu için algoritmanın ikinci seçimi de yanlış yola sokmaz. Bu böyle sürer; her adımda elimizdeki kısmi çözüm, bir en iyi çözümün parçası olarak kalır. Algoritma bittiğinde kısmi çözüm tamdır ve en iyidir.

Tanımdaki "en az bir" sözcükleri önemlidir. Açgözlü seçim özelliği, *her* en iyi çözümün açgözlü seçimi içerdiğini söylemez. Etkinlik seçimi bunun iyi bir örneğidir: sunumdaki on etkinlikli örnekte en büyük boyut 6'dır ve bu boyutta iki farklı küme vardır. Bunlardan yalnız biri en erken biten etkinliği içerir. Özellik için bu kadarı yeterlidir.

Optimal alt yapı yalnız açgözlü algoritmalara özgü değildir. Dinamik programlama da bu özelliğe dayanır. İki yaklaşımı ayıran, açgözlü seçim özelliğidir. Dinamik programlama bir sonraki adımda hangi seçimin en iyi olduğunu bilmez; bütün seçenekleri dener ve alt problemlerin sonuçlarını bir tabloda saklar. Açgözlü algoritma ise hangi seçimin güvenli olduğunu baştan bilir ve yalnız onu yapar. Bu yüzden açgözlü algoritma, uygulanabildiği problemlerde dinamik programlamadan çok daha hızlıdır.

İki özellik birlikte sağlanmıyorsa açgözlü algoritma yalnız bir **sezgisel** (*heuristic*) yöntemdir. Sezgisel yöntem hızlıdır ve çoğu girdide makul bir sonuç verir. Ancak sonucun en iyiden ne kadar kötü olabileceğini kendi başına söylemez. *{4, 3, 1}* paralarıyla para üstü bunun en basit örneğidir: 6 tutarı için açgözlü algoritma 3 para verir, oysa 2 para yeterlidir.

7.1.4 Değiştirme argümanı

Açgözlü seçim özelliğini kanıtlamanın en yaygın yolu **değiştirme argümanıdır** (*exchange argument*). Kanıt keyfî bir en iyi çözümle başlar. Bu çözüme O denir. O açgözlü seçimi zaten içeriyorsa söylenecek bir şey yoktur. İçermiyorsa, O'daki uygun bir öğe çıkarılır ve yerine açgözlü seçim konur. Kanıtın asıl işi iki şeyi göstermektir:

1. Değişiklikten sonra çözüm hâlâ **uygundur**; hiçbir kısıt bozulmaz.
2. Değişiklikten sonra çözüm **daha kötü olmaz**; amaç değeri ya aynı kalır ya iyileşir.

Bu iki madde gösterilince açgözlü seçimi içeren bir en iyi çözüm bulunmuş olur. Optimal alt yapı ile birlikte tümevarım uygulanır ve sonuç bütün adımlara yayılır.

Argümanın gücü, en iyi çözümün neye benzediğini bilmeye gerek bırakmamasıdır. O hakkında yalnız uygun ve en iyi olduğu bilinir. Kanıt O'yu adım adım açgözlü çözüme benzetir ve her adımda kalitenin düşmediğini gösterir. Açgözlü kanıtların çoğu bu kalıba uyar. Etkinlik seçiminde ilk etkinlik değiştirilir. Kesirli sırt çantasında bir nesnenin bir parçası başka bir nesnenin parçasıyla yer değiştirir. Kesim özelliğinde bir kenar başka bir kenarla, Huffman kodlamada iki yaprak birbiriyle yer değiştirir.

Örnek 7.1

Bir öğrenci n sınav kâğıdından en fazla k tanesini okuyabilir; her kâğıdın bir puanı vardır ve toplam puan en büyük olmalıdır. Açgözlü ölçüt açıktır: en yüksek puanlı k kâğıt seçilir. Değiştirme argümanı şöyle kurulur. En yüksek puanlı kâğıt a olsun ve O onu içermeyen bir en iyi seçim olsun. O'dan herhangi bir b kâğıdı çıkarılıp yerine a konur. Kâğıt sayısı değişmediği için seçim uygundur. $puan(a) \geq puan(b)$ olduğu için toplam azalmaz. Demek ki a'yı içeren bir en iyi seçim vardır. a seçildikten sonra kalan problem, geri kalan $n - 1$ kâğıttan $k - 1$ tanesini seçmektir ve aynı türdendir.

Örnek kolay görünür, çünkü kısıt yalnız seçilen öğe sayısıyla ilgilidir. Kısıtlar karmaşıklaştıkça değiştirmenin uygunluğu bozmadığını göstermek zorlaşır. Etkinlik seçiminde bu adım bitiş zamanlarının karşılaştırılmasını, Huffman kodlamada ise ağaçtaki derinliklerin karşılaştırılmasını gerektirir.

Sık yapılan hata

Değiştirme argümanında yalnız “daha kötü olmaz” kısmını gösterip uygunluğu atlamak. Değiştirilen çözüm uygun değilse, yani kısıtlardan birini bozuyorsa, kanıt hiçbir şey göstermez. 0/1 sırt çantasında yüksek oranlı bir nesneyi düşük oranlı bir nesnenin yerine koymak kârı artırabilir, ama yeni nesne daha ağırsa kapasite aşılır. Oran ölçütünün bu problemde yanılmasının nedeni tam olarak budur.

7.1.5 Güçlü ve zayıf yanlar

Açgözlü yaklaşımın güçlü yanları kısaca şöyle sıralanabilir:

- **Anlaşılır ve kısa kod.** Algoritmanın tamamı çoğu zaman bir sıralama ve bir döngüden oluşur. Bu notta verilen Java kodlarının hiçbiri yirmi satırı geçmez.
- **Hız.** Sıralama ve tek geçiş toplam $O(n \log n)$ sürer. Aynı problemi dinamik programlama ile çözmek çoğu zaman bir tablo doldurmayı ve daha yüksek bir maliyeti gerektirir.
- **Az bellek.** Yalnız mevcut kısmi çözüm tutulur. Alt problemlerin sonuçlarını saklayan bir tablo gerekmez.
- **Doğru problemde en iyi sonuç.** Minimum kapsayan ağaç, Huffman kodlama ve etkinlik seçimi gibi problemlerde açgözlü algoritma kanıtlanmış olarak en iyidir.
- **Zor problemlerde iyi bir başlangıç.** En iyi çözümü bulmanın çok pahalı olduğu problemlerde açgözlü algoritma hızlı bir yaklaşık çözüm ya da daha iyi yöntemler için bir başlangıç noktası verir.

Zayıf yanlar da en az bunlar kadar önemlidir:

- **Garanti yoktur.** $\{1, 3, 4\}$ paralarıyla para üstü ve 0/1 sırt çantası, açgözlü sonucun en iyiden kötü olduğu bilinen örneklerdir.
- **Her problem için ayrı kanıt gerekir.** Bir açgözlü algoritmanın bir problemde doğru olması, benzer görünen başka bir problemde doğru olacağını göstermez.
- **Ölçüt seçimi belirleyicidir.** Aynı problem için akla gelen birçok ölçütten genellikle yalnız biri doğrudur. Yanlış ölçüt, kodu tamamen doğru yazılmış bir algoritmayı yanlış yapar.
- **Hata düzeltilemez.** Bir karar verildikten sonra geri alınmadığı için erken yapılan kötü bir seçimin etkisi sonuna kadar taşınır.

7.1.6 Açgözlü yaklaşım ve böl ve fethet

Böl ve fethet de bir problemi daha küçük problemlere indirger. Bu benzerlik iki yaklaşımın karıştırılmasına yol açabilir. Aradaki fark, küçültmenin nasıl yapıldığındadır. Böl ve fethet problemi önceden belirlenmiş bir kuralla böler, örneğin diziyi ortadan ikiye. Bu bölmede bir seçim yoktur. Sonra bütün parçaları çözer ve sonuçları birleştirir. Açgözlü yaklaşım ise bir seçim yapar ve bu seçimden sonra geriye tek bir alt problem kalır. O alt problem çözülür; başka bir parça yoktur. Tablo 7.1 farkları beş başlık altında toplar.

	Açgözlü	Böl ve fethet
Karar	her adımda tek, yerel en iyi seçim; geri alınmaz	sabit bir kuralla bölünür; seçim yapılmaz
Alt problemler	seçimden sonra tek alt problem kalır	birbirinden bağımsız birkaç alt problem
Doğruluk	açgözlü seçim özelliği ve optimal alt yapı kanıtlanmalı	birleştirme adımının doğruluğuna bağlı
Maliyet	genelde $O(n \log n)$: sıralama ve tek geçiş	yineleme bağıntısı ve ana teoremle bulunur
Örnek	etkinlik seçimi, Kruskal, Huffman	birleştirme sıralaması, Karatsuba

Tablo 7.1: Açgözlü yaklaşım ile böl ve fethetin karşılaştırması.

Maliyet satırı da bu farkı yansıtır. Birleştirme sıralamasının süresi $T(n) = 2T(n/2) + O(n)$ bağıntısından ana teoremle bulunur ve $O(n \log n)$'dir. Açgözlü bir algoritmanın süresi için genellikle bir bağıntı kurmaya gerek yoktur; sıralama ve tarama maliyetleri doğrudan toplanır. Doğruluk satırı ise iki yaklaşımın en önemli farkını gösterir. Böl ve fethette alt problemler doğru çözülür ve birleştirme doğruysa sonuç kendiliğinden doğrudur. Açgözlü yaklaşımda sonucun doğru olması, yapılan seçimin güvenli olmasına bağlıdır ve bu ayrıca kanıtlanmalıdır.

7.2 Para üstü problemi

Açgözlü yaklaşımın en tanıdık örneği para üstü vermektir. Bir kasiyer 68 kuruşu en az sayıda madeni parayla ödemek isterse önce 50'lik, sonra 10'luk, sonra 5'lik ve en sonda üç tane 1'lik verir. Her adımda kalan tutara sığan en büyük parayı seçer. Bu yöntem gündelik para sistemlerinde kusursuz çalışır. Ancak para birimleri farklı seçilirse aynı yöntem en iyi sonucu kaçırabilir. Bu kesim hem yöntemin işlediği hem de yanıldığı durumları inceler.

7.2.1 Problem ve açgözlü ölçüt

Tanım 7.4

Para üstü problemi (*coin change problem*): pozitif tam sayı değerli k para türü $p_1 > p_2 > \dots > p_k = 1$ ve bir T tutarı verilir. Her türden istenen sayıda para kullanılabilir. Toplamı tam olarak T olan en az sayıda para bulunmak istenir.

En küçük paranın 1 olması, her tutarın ödenebilmesini garanti eder. Bu koşul yoksa bazı tutarlar hiç ödenemez; örneğin yalnız 3 ve 5'lik paralarla 7 ödenemez. Açgözlü ölçüt şöyledir: kalan tutardan büyük olmayan **en büyük** para seçilir; tutar sıfır olana kadar bu adım tekrarlanır. Sunumdaki Java kodu bu kuralı iki iç içe döngüyle uygular:

```
// para[] büyükten küçüğe sıralı
static int paraUstu(int[] para, int tutar) {
  int adet = 0;
  for (int p : para) {
    while (tutar >= p) {
      tutar -= p;
      adet++;
    }
  }
  return adet;
}
```

Kod, para dizisinin büyükten küçüğe sıralı geldiğini varsayar. Dış döngü her para türünü bu sırayla bir kez ele alır. İç `while` döngüsü, o para kalan tutara sığdığı sürece onu kullanır: tutardan düşer ve sayacı artırır. Para artık sığmadığında iç döngü biter ve bir sonraki, daha küçük para türüne geçilir. En küçük para 1 olduğu için son türden sonra tutar her zaman sıfırdır.

Maliyet hesabı iki kısımdan oluşur. Dış döngü *k* kez döner ve her dönüşte en az bir `while` koşulu sınanır; bu *O(k)* eder. İç döngünün gövdesi her çalıştığında bir para seçilir; gövde toplam *adet* kez çalışır. Toplam süre *O(k + adet)*'tir. Para dizisi sıralı gelmezse buna *O(k log k)*'lik bir sıralama eklenir. İç döngü bölme ve kalan işlemiyle de yazılabilir: `adet += tutar / p; tutar %= p;`. Bu sürüm para sayısından bağımsız olarak *O(k)* sürer ve büyük tutarlarda daha hızlıdır. Sonuç aynıdır.

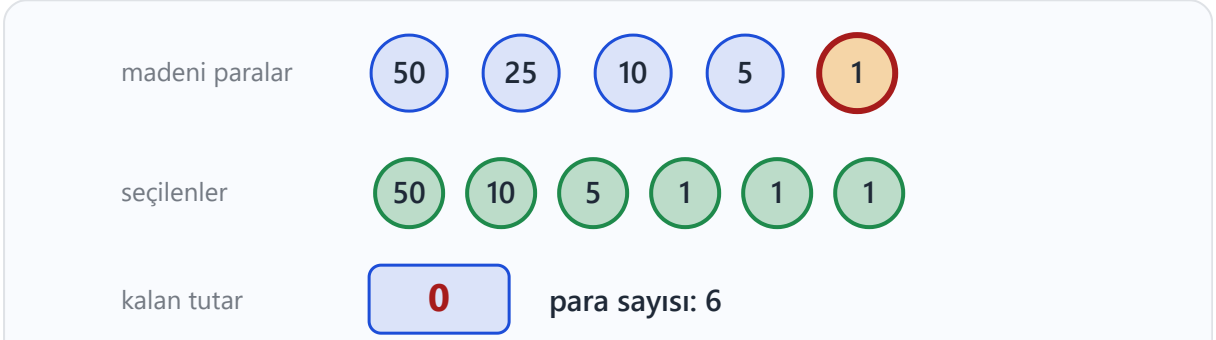
7.2.2 Kanonik bir sistemde adım adım

Sunumdaki ilk örnekte paralar *{50, 25, 10, 5, 1}*, tutar 68'dir. Tablo 7.2 her adımı verir.

Adım	Sığan en büyük para	Kalan tutar	Para sayısı
başlangıç	—	68	0
1	50	68 − 50 = 18	1
2	10 (25 sığmaz)	18 − 10 = 8	2
3	5	8 − 5 = 3	3
4	1	3 − 1 = 2	4
5	1	2 − 1 = 1	5
6	1	1 − 1 = 0	6

Tablo 7.2: *{50, 25, 10, 5, 1}* paralarıyla 68 tutarının açgözlü ödenmesi.

Sonuç *68 = 50 + 10 + 5 + 1 + 1*, yani 6 paradır. Bu, 68 için gerçekten en az para sayısıdır. Daha azının mümkün olmadığı şöyle görülebilir. 1 dışındaki bütün paralar 5'in katıdır ve 68'in 5'e bölümünden kalan 3'tür. Bu yüzden her çözümde en az üç tane 1'lik para bulunur. En fazla 5 paralık bir çözüm olsaydı, 1'likler dışındaki en fazla iki para 65'i ödemek zorunda kalırdı. Oysa iki parayla 65 ödenemez: 50 + 10 = 60, 50 + 25 = 75 ve 25 + 25 = 50'dir; 65'e eşit bir iki para toplamı yoktur. Tek parayla ya da hiç parasız da ödenemez.



Şekil 7.2: 68 tutarının açgözlü ödenmesinin son durumu. Seçilenler 50, 10, 5, 1, 1, 1; kalan tutar 0, para sayısı 6.

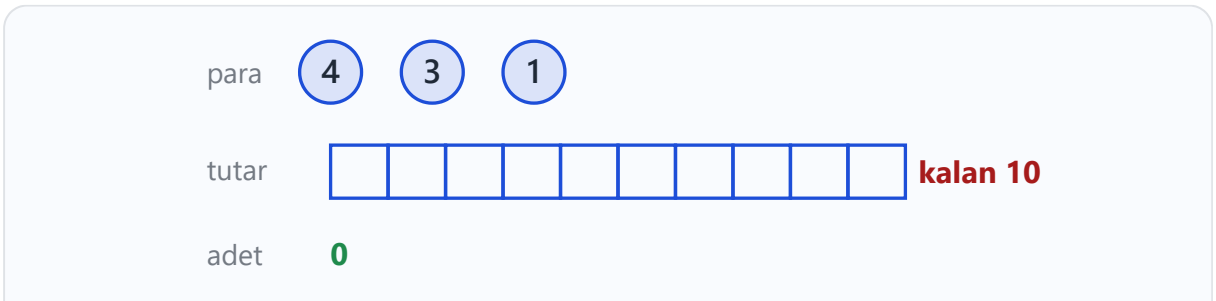
Açgözlü sonucun her tutar için en iyi olduğu para sistemlerine **kanonik** para sistemi denir. *{50, 25, 10, 5, 1}* ve *{25, 10, 5, 1}* kanonik sistemlerdir. Türk lirasının madeni paralarına karşılık gelen *{100, 50, 25, 10, 5, 1}* kuruş da kanoniktir. Euro sisteminin *{200, 100, 50, 20, 10, 5, 2, 1}* sent dizisi de öyledir. Notun doğrulama betiği bu sistemlerin her birinde 1 ile 499 arasındaki bütün tutarları dinamik programlamayla karşılaştırmış ve hiçbir farka rastlamamıştır. Gündelik para sistemlerinin kanonik olması tesadüf değildir; sistemler zaten bu özellik gözetilerek tasarlanır.

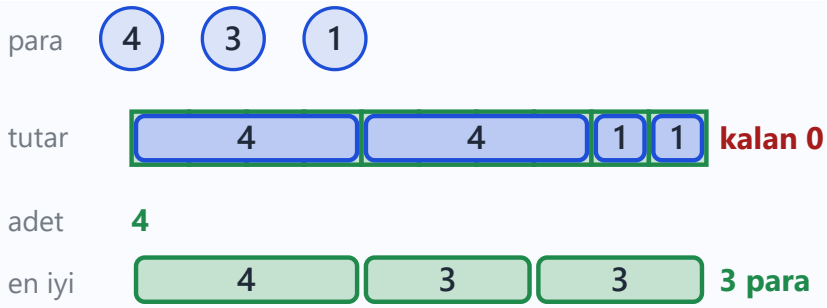
7.2.3 Kod izleme: {4, 3, 1} ve 10

Şimdi para birimleri *{4, 3, 1}* olsun ve `paraUstu` 10 tutarıyla çağrılsın. Paralar büyükten küçüğe sıralı verilmiştir. Bu sistem kanonik değildir; açgözlü sonucun en iyi olup olmayacağı baştan bilinmez. Tablo 7.3 kodun her adımında değişkenlerin değerini gösterir.

Adım	`p`	Koşul `tutar >= p`	İşlem	`tutar`	`adet`
1	4	10 ≥ 4 doğru	4 düşülür	6	1
2	4	6 ≥ 4 doğru	4 düşülür	2	2
3	4	2 ≥ 4 yanlış	`while` biter	2	2
4	3	2 ≥ 3 yanlış	3'lük hiç kullanılmaz	2	2
5	1	2 ≥ 1 doğru	1 düşülür	1	3
6	1	1 ≥ 1 doğru	1 düşülür	0	4
7	1	0 ≥ 1 yanlış	`while` biter, `return 4`	0	4

Tablo 7.3: `paraUstu({4, 3, 1}, 10)` izlemesi.





Şekil 7.3: `paraUstu({4, 3, 1}, 10)` : başlangıç durumu (üstte) ve son durum (altta). Açgözlü 4 + 4 + 1 + 1 ile 4 para kullanır; en iyi çözüm 4 + 3 + 3 ile 3 paradır.

Kod 4 döndürür: *10 = 4 + 4 + 1 + 1*. Oysa *10 = 4 + 3 + 3* yalnız 3 para kullanır. Hata ikinci adımda yapılmıştır. İlk 4'lük seçildikten sonra kalan 6 tutarı için açgözlü kural yine 4'ü seçer, çünkü 4 hâlâ sığar. Oysa 6, iki tane 3'lükle ödenebilirdi. İkinci 4'lük seçildiği anda kalan 2 için 3'lük para işe yaramaz hâle gelir ve 2, iki tane 1'likle ödenir. İzlemenin dördüncü satırı bu sonucu gösterir: 3'lük para hiç kullanılmaz.

7.2.4 Açgözlü seçim neden yanılır?

Aynı sistemde 6 tutarı daha da kısa bir karşı örnektir. Açgözlü algoritma önce 4'ü seçer, kalan 2'yi iki tane 1'likle öder: *6 = 4 + 1 + 1*, 3 para. En iyi çözüm ise *6 = 3 + 3*, 2 paradır. Şekil 7.4 bu iki sonucu gösterir.



Şekil 7.4: *{4, 3, 1}* paralarıyla 6 tutarı. Açgözlü 4 + 1 + 1 ile 3 para kullanır (üstte); en iyi çözüm 3 + 3 ile 2 paradır (altta).

Burada açgözlü seçim özelliği bozulmuştur. İlk açgözlü seçim 4'tür, ama 6'nın tek en iyi çözümü olan 3 + 3, 4'ü içermez. 4'ü içeren her çözüm en az 3 para kullanır. Değiştirme argümanı da bu sistemde işlemez: en iyi çözümdeki 3'lüklerden biri 4'lükle değiştirilirse toplam 7 olur ve çözüm uygunluğunu kaybeder.

Bir para sisteminin kanonik olup olmadığını anlamak için tek bir tutarın denenmesi yetmez. *{4, 3, 1}* sisteminde 1'den 10'a kadar olan tutarlardan yalnız 6 ve 10'da açgözlü yanılır; 1–5, 7, 8 ve 9 tutarlarında açgözlü sonuç en iyidir. 20'ye kadar hatalı tutarlar 6, 10, 14 ve 18'dir. Sunumdaki ikinci karşı örnek sistemi *{10, 7, 1}*'dir. Bu sistemde ilk hatalı tutar 14'tür: açgözlü *10 + 1 + 1 + 1 + 1* ile 5 para kullanır, en iyi çözüm *7 + 7* ile 2 paradır. 15 için açgözlü 6 para (10 + beş tane 1) verir, en iyi çözüm *7 + 7 + 1* ile 3 paradır.

Örnek 7.2

Açgözlü para üstünün yanıldığı, en büyük parası en küçük olan para sistemi hangisidir? En büyük para 2 ise sistem *{2, 1}*'dir ve her tutar için açgözlü en iyidir: önce olabildiğince 2'lik, en fazla bir tane 1'lik. En büyük para 3 ise olası sistemler *{3, 1}* ve *{3, 2, 1}*'dir. İkisinde de herhangi bir tutar $3q + r$ ($r \le 2$) biçiminde ödenir ve daha iyisi yoktur. En büyük para 4 olan sistemler *{4, 1}*, *{4, 2, 1}*, *{4, 3, 1}* ve *{4, 3, 2, 1}*'dir. Bunlardan yalnız *{4, 3, 1}* kanonik değildir ve ilk karşı örneği 6'dır. Doğrulama betiği bu sonucu bütün küçük sistemleri tarayarak bulmuştur. Demek ki *{1, 3, 4}* ve 6 tutarı, açgözlünün yanıldığı en küçük örnektir.

Bir sistemin kanonik olup olmadığı genel olarak kolayca görülmez. Komşu paraların birbirinin katı olması yeterli bir koşuldur: *{100, 50, 10, 5, 1}* gibi bir sistemde her büyük para, kendinden küçük paraların yerine geçebilir. Ancak bu koşul gerekli değildir; *{25, 10, 5, 1}* sisteminde 25, 10'un katı değildir, ama sistem yine kanoniktir. Kozen ve Zaks'ın gösterdiği gibi, kanonik olmayan bir sistemin en küçük karşı örneği en büyük iki paranın toplamından küçüktür. Bu yüzden bir sistemi denetlemek için bu sınıra kadar olan tutarları dinamik programlamayla karşılaştırmak yeterlidir. *{4, 3, 1}* için sınır 7'dir ve 6 bu aralıktadır. *{10, 7, 1}* için sınır 17'dir ve 14 bu aralıktadır.

Sık yapılan hata

Açgözlü para üstünü birkaç tutarla deneyip doğru sanmak. *{4, 3, 1}* sisteminde 1'den 5'e kadar bütün tutarlar doğru sonuç verir; ilk hata 6'da ortaya çıkar. Birkaç başarılı deneme bir kanıt değildir. Açgözlü bir algoritmanın doğruluğu ya kanıtla gösterilir ya da, para üstündeki gibi, sonlu bir aralığın tamamı en iyi yöntemle karşılaştırılarak denetlenir.

7.2.5 En iyi sonuç: dinamik programlama

Para sistemi kanonik değilse en iyi sonucu dinamik programlama verir. Fikir, her tutar için en az para sayısını küçükten büyüğe hesaplamaktır. *enAz[t]*, t tutarını ödemek için gereken en az para sayısı olsun. *enAz[0] = 0*'dır. $t > 0$ için son kullanılan para p ise geriye $t - p$ tutarı kalır ve onun da en iyi bi-

çimde ödenmesi gerekir. Bütün paralar denenir ve en iyisi seçilir:

> enAz[t] = 1 + min { enAz[t − p] : p ≤ t }

Bu bağıntı optimal alt yapının bir örneğidir: *t*'nin en iyi çözümünden son para çıkarılırsa, kalan paralar *t − p*'nin en iyi çözümünü oluşturur. Açgözlü algoritmadan farkı, son paranın hangisi olacağının bilinmemesi ve bütün seçeneklerin denenmesidir. Tablo 7.4 *{4, 3, 1}* sistemi için 0'dan 10'a kadar tabloyu açgözlü sonuçla yan yana verir.

t	0	1	2	3	4	5	6	7	8	9	10
açgözlü	0	1	2	1	1	2	**3**	2	2	3	**4**
enAz[t]	0	1	2	1	1	2	**2**	2	2	3	**3**

Tablo 7.4: *{4, 3, 1}* sisteminde açgözlü sonuç ve *enAz[t]*.

Tablonun iki hücresi ayrıntılı hesaplanırsa bağıntının işleyişi görülür. *t = 6* için üç seçenek vardır: son para 1 ise *1 + enAz[5] = 3*, son para 3 ise *1 + enAz[3] = 2*, son para 4 ise *1 + enAz[2] = 3*. En küçüğü 2'dir ve 3'lük parayla elde edilir. *t = 10* için seçenekler *1 + enAz[9] = 4*, *1 + enAz[7] = 3* ve *1 + enAz[6] = 3*'tür; en küçüğü 3'tür. Açgözlü algoritma ise yalnız 4'lük seçeneği izler ve 10 için 4 para bulur.

Dinamik programlamanın bedeli süredir. Tablo *T + 1* hücrelidir ve her hücre *k* para türünü dener; toplam *O(k · T)* zaman ve *O(T)* bellek gerekir. Açgözlü algoritma ise *O(k + adet)* sürede biter ve ek bellek kullanmaz. Sunumdaki etkileşimli slayt iki yöntemi yan yana çalıştırır: istenen para kümesi ve tutar girilir, açgözlü sonuç ile dinamik programlamanın en iyi sonucu karşılaştırılır. Hazır düğmelerden biri *{1, 7, 10}* ve 14 örneğini, öteki kanonik TL kuruş sistemini ve 68 tutarını yükler.

Sık yapılan hata

`paraUstu`'yu sırasız bir para dizisiyle çağırmak. Kod dizinin büyükten küçüğe sıralı olduğunu varsayar ve bunu denetlemez. `{1, 5, 10}` dizisiyle çağrılırsa ilk tür 1 olduğu için bütün tutar 1'liklerle ödenir; 68 için 68 para döner. Ya çağıran taraf diziyi sıralamalı ya da metodun başına bir sıralama eklenmelidir.

7.3 Etkinlik seçimi

Bir konferans salonunu kullanmak isteyen birçok etkinlik vardır. Her etkinliğin bir başlangıç ve bir bitiş saati bellidir. Salonda aynı anda yalnız bir etkinlik yapılabilir. Yönetim, birbiriyle çakışmayan olabildiğince çok etkinliği kabul etmek ister. Etkinliklerin süresi ya da önemi değil, yalnız sayısı önemlidir. Bu problem, açgözlü bir algoritmanın en iyi sonucu verdiği ve bunun kısa bir kanıtla gösterilebildiği klasik bir örnektir.

7.3.1 Problemin tanımı

Tanım 7.5

Etkinlik seçimi (*activity selection*): n etkinlik verilir; i. etkinlik $[s_i, f_i)$ yarı açık aralığını kaplar ($s_i < f_i$). İki etkinlik $[s_i, f_i)$ ve $[s_j, f_j)$, $s_j \geq f_i$ ya da $s_i \geq f_j$ ise **çakışmaz**. Birbiriyle çakışmayan en fazla sayıda etkinlikten oluşan bir küme bulunmak istenir.

Aralıkların yarı açık olması, bir etkinliğin bittiği anda bir başkasının başlayabileceği anlamına gelir. Örneğin $[3, 4)$ ile $[4, 6)$ çakışmaz. Bitiş anı etkinliğe dahil değildir. Sunumdaki örnekte on etkinlik vardır (Tablo 7.5).

Etkinlik	1	2	3	4	5	6	7	8	9	10
başlangıç s	1	2	4	6	5	8	0	3	7	10
bitiş f	3	5	6	8	7	9	2	4	9	12

Tablo 7.5: Sunumdaki on etkinlik.

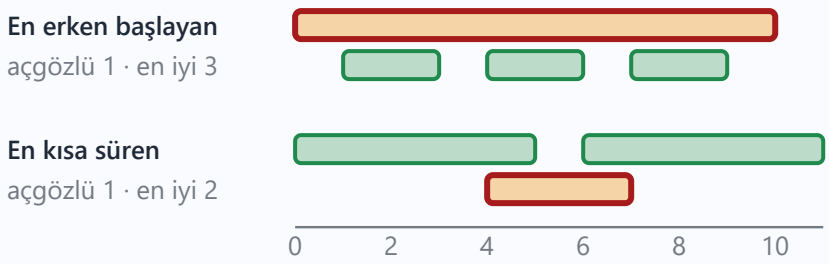
Kaba kuvvetle bütün alt kümeleri denemek 2^n küme demektir; on etkinlik için 1024 küme, kırk etkinlik için bir trilyonu aşan küme. Açgözlü algoritma ise etkinlikleri bir kez sıralar ve tek geçişte seçer.

7.3.2 Hangi ölçüt?

Açgözlü bir algoritma kurmak için önce “her adımda hangi etkinlik seçilsin?” sorusuna cevap vermek gerekir. Akla gelen dört ölçüt vardır:

1. **En erken başlayan.** Salon olabildiğince erken dolsun.
2. **En kısa süren.** Kısa etkinlikler salonu az meşgul eder.
3. **En az çakışan.** Başka etkinlikleri en az engelleyen seçilsin.
4. **En erken biten.** Salon olabildiğince erken boşalsın.

Her biri makul görünür, ama yalnız sonuncusu her girdide en iyi sonucu verir. Bir ölçütün yanlış olduğunu göstermek için tek bir karşı örnek yeter. Şekil 7.5 ilk iki ölçüt için sunumdaki karşı örnekleri gösterir.



Şekil 7.5: Yanlış ölçütlere karşı örnekler. Turuncu: açgözlünün seçtiği etkinlik; yeşil: en iyi çözüm. En erken başlayan ölçütü 1 etkinlik seçer, en iyisi 3'tür. En kısa süren ölçütü 1 etkinlik seçer, en iyisi 2'dir.

Üstteki örnekte *[0, 10)* etkinliği en erken başlar. Açgözlü algoritma onu seçer ve onunla çakışan *[1, 3)*, *[4, 6)* ve *[7, 9)* etkinliklerini atar; sonuç 1 etkinliktir. Oysa bu üç kısa etkinlik birbiriyle çakışmaz ve birlikte seçilebilir. Alttaki örnekte *[4, 7)* en kısa etkinliktir (3 birim). Açgözlü onu seçer, ama bu etkinlik hem *[0, 5)* ile hem *[6, 11)* ile çakışır. Sonuç yine 1 etkinliktir; en iyi çözüm ise iki uzun etkinliği birlikte seçer.

En az çakışan ölçütü daha ince bir karşı örnek gerektirir, çünkü küçük örneklerde çoğu zaman doğru sonuç verir. Tablo 7.6 on bir etkinlikli bir karşı örnek verir. En iyi çözüm A, B, C ve D'den oluşur ve 4 etkinlik içerir. Ancak çakışma sayısı en küçük olan etkinlik M'dir: yalnız B ve C ile çakışır. Açgözlü önce M'yi seçer ve böylece B'yi de C'yi de kaybeder. Kalan etkinliklerden en fazla iki tane daha seçilebilir; sonuç 3 etkinliktir. Doğrulama betiği eşitliklerin her türlü bozulmasını denemiş ve bu ölçütün bu girdide 3'ü geçemediğini göstermiştir.

Etkinlik	Aralık	Çakıştığı etkinlik sayısı
A	*[0, 3)*	3
B	*[4, 7)*	4
C	*[8, 11)*	4
D	*[12, 15)*	3
X_1, X_2, X_3	*[2, 5)* (üç kopya)	4'er
Y_1, Y_2, Y_3	*[10, 13)* (üç kopya)	4'er
M	*[6, 9)*	**2**

Tablo 7.6: “En az çakışan” ölçütüne karşı örnek.

En erken biten ölçütünün arkasındaki düşünce şudur: salonu en erken boşaltan etkinlik, geri kalan etkinliklere en çok zaman bırakır. Geri kalan zaman ne kadar uzunsa, oraya o kadar çok etkinlik sığabilir. Ölçüt etkinliğin süresine ya da başlangıcına değil, yalnız bitişine bakar.

7.3.3 Kod ve izleme

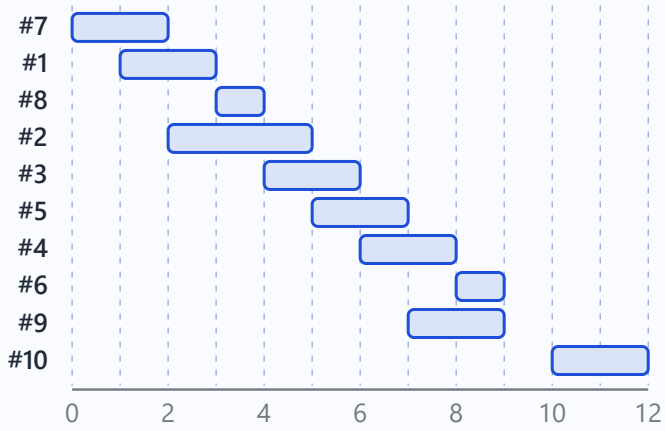
Sunumdaki `sec` metodu etkinlikleri bitiş zamanına göre sıralar, ilkini seçer ve sonra her etkinliği son seçilenin bitişiyle karşılaştırır:

```
static void sec(Etkinlik[] e) {
  Arrays.sort(e, (x, y) -> x.bitis - y.bitis);
  int i = 0;                 // son seçilen
  System.out.print(e[i].id + " ");
  for (int j = 1; j < e.length; j++) {
    if (e[j].bas >= e[i].bitis) {
      System.out.print(e[j].id + " ");
      i = j;
    }
  }
}
```

Kodda `Etkinlik` sınıfının `id`, `bas` ve `bitis` alanları vardır. `Arrays.sort` çağrısındaki `(x, y) -> x.bitis - y.bitis` ifadesi bir karşılaştırıcıdır: `x`'in bitişi daha küçükse negatif bir değer döner ve `x` öne geçer. Java'nın nesne dizileri için kullandığı sıralama **kararlıdır**; bitişi eşit iki etkinlik dizideki ilk sıralarını korur. `i` değişkeni son seçilen etkinliğin dizideki yerini tutar. Sıralı dizinin ilk öğesi en erken biten etkinliktir ve koşulsuz seçilir. Döngü `j = 1`'den başlar. `e[j].bas >= e[i].bitis` koşulu, `j`. etkinliğin son seçilenden sonra başladığını, yani onunla çakışmadığını sınar. Koşul doğruysa etkinlik yazdırılır ve `i` güncellenir.

Burada önemli bir gözlem vardır: yeni etkinlik yalnız *son* seçilenle karşılaştırılır, bütün seçilenlerle değil. Bu yeterlidir, çünkü etkinlikler bitişe göre sıralıdır. Son seçilen, seçilenlerin içinde en geç biten etkinliktir. Yeni etkinlik ondan sonra başlıyorsa, daha önce biten bütün seçilenlerden de sonra başlar.

Sıralama sonrası dizi şöyledir: *#7 [0, 2)*, *#1 [1, 3)*, *#8 [3, 4)*, *#2 [2, 5)*, *#3 [4, 6)*, *#5 [5, 7)*, *#4 [6, 8)*, *#6 [8, 9)*, *#9 [7, 9)*, *#10 [10, 12)*. #6 ile #9'un bitişi aynıdır (9); kararlı sıralama #6'yı önde tutar. Tablo 7.7 döngünün her adımını verir.



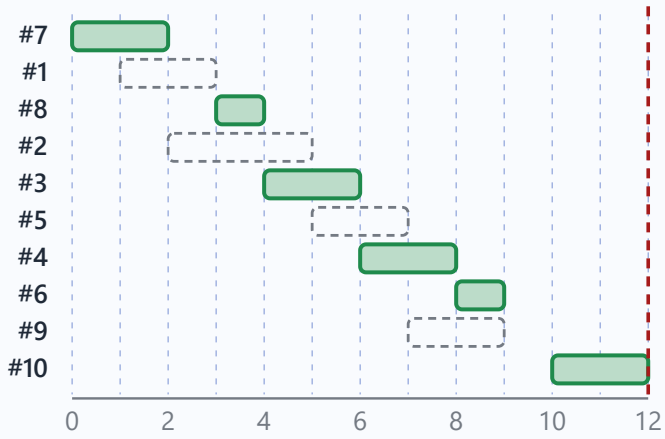
Şekil 7.6: Bitiş zamanına göre sıralanmış on etkinlik (yukarıdan aşağı). Yatay eksen zamandır.

`j`	Etkinlik	`e[j].bas`	`e[i].bitis`	Karar	Yeni `i`
0	#7 [0, 2)	0	—	ilk etkinlik, seçilir	0

`j`	**Etkinlik**	`e[j].bas`	`e[i].bitis`	**Karar**	**Yeni** `i`
1	#1 [1, 3)	1	2	1 < 2, çakışır, atlanır	0
2	#8 [3, 4)	3	2	3 ≥ 2, seçilir	2
3	#2 [2, 5)	2	4	2 < 4, atlanır	2
4	#3 [4, 6)	4	4	4 ≥ 4, seçilir	4
5	#5 [5, 7)	5	6	5 < 6, atlanır	4
6	#4 [6, 8)	6	6	6 ≥ 6, seçilir	6
7	#6 [8, 9)	8	8	8 ≥ 8, seçilir	7
8	#9 [7, 9)	7	9	7 < 9, atlanır	7
9	#10 [10, 12)	10	9	10 ≥ 9, seçilir	9

Tablo 7.7: `sec(e)` izlemesi. `e[i].bitis` son seçilenin bitişidir.

Program `7 8 3 4 6 10` yazar; 6 etkinlik seçilmiştir. Bu sayı gerçekten en büyüktür: doğrulama betiği 1024 alt kümenin hepsini denemiş ve 7 etkinlikli çakışmasız bir küme bulamamıştır. 6 etkinlikli iki küme vardır: açgözlünün bulduğu *{7, 8, 3, 4, 6, 10}* ve *{1, 8, 3, 4, 6, 10}*. İkincisi, #7 yerine #1 ile başlar. Bu durum açgözlü seçim özelliğinin "en az bir" en iyi çözümden söz ettiğini somut olarak gösterir: en erken biten #7'yi içermeyen bir en iyi çözüm de vardır, ama #7'yi içeren bir tane de vardır.



Şekil 7.7: Seçimin sonu. Yeşil etkinlikler seçilmiştir (#7, #8, #3, #4, #6, #10); kesikli çerçeveliler atlanmıştır. Dikey kesikli çizgi son seçilenin bitişidir (12).

İzlemedeki üç adım, yarı açık aralıkların önemini gösterir. #3 tam 4'te başlar ve #8 tam 4'te biter; #4 tam 6'da başlar ve #3 tam 6'da biter; #6 tam 8'de başlar ve #4 tam 8'de biter. Koşul `>=` yerine `>` yazılsaydı, son seçilenin bittiği anda başlayan etkinlikler atlanırdı. #8'den sonra #3 yerine #5 [5, 7) seçilirdi; ardından #6 [8, 9) ve #10 [10, 12) gelirdi ve sonuç 5 etkinlikle kalırdı.

Maliyet açıktır. Sıralama *O(n log n)*, tek geçiş *O(n)* sürer; toplam *O(n log n)*. Etkinlikler bitişe göre sıralı gelirse sıralama gereksizdir ve algoritma *O(n)* sürede biter. Ek bellek olarak yalnız bir `i` değişkeni tutulur.

Örnek 7.3

On bir etkinlik *(s, f)* çiftleriyle verilsin: (1, 4), (3, 5), (0, 6), (5, 7), (3, 9), (5, 9), (6, 10), (8, 11), (8, 12), (2, 14), (12, 16). Etkinlikler zaten bitişe göre sıralıdır. İlk etkinlik (1, 4) seçilir ve son bitiş 4 olur. (3, 5) ve (0, 6) 4'ten önce başlar, atlanır. (5, 7) seçilir, son bitiş 7 olur. (3, 9), (5, 9) ve (6, 10) 7'den önce başlar, atlanır. (8, 11) seçilir, son bitiş 11 olur. (8, 12) ve (2, 14) atlanır. (12, 16) seçilir. Sonuç 4 etkinliktir: (1, 4), (5, 7), (8, 11), (12, 16).

Sık yapılan hata

Yeni etkinliği son seçilenle değil, dizide bir önceki etkinlikle karşılaştırmak; yani `e[j].bas >= e[j - 1].bitis` yazmak. Önceki etkinlik atlanmış olabilir. İzlemede *j = 3*'teki #2 [2, 5) bunu gösterir: bir önceki etkinlik #8'in bitişi 4'tür ve 2 < 4 olduğu için sonuç doğru çıkar. Ama *j = 4*'teki #3 [4, 6) atlanmış #2'nin bitişi olan 5 ile karşılaştırılır; 4 < 5 olduğu için yanlışlıkla atlanır. Karşılaştırma her zaman son *seçilen* etkinlikle yapılmalıdır.

7.3.4 En erken biten neden doğrudur?

Teorem 7.1 (Etkinlik seçiminde açgözlü seçim)

a_1 en erken biten etkinlik olsun. Etkinlik seçimi probleminin a_1'i içeren bir en iyi çözümü vardır. Bu yüzden "en erken biteni seç, onunla çakışanları at, kalanlarla tekrarla" algoritması en fazla sayıda etkinlik seçer.

İspat. Kanıt değiştirme argümanıyla kurulur. $O = \{b_1, b_2, \ldots, b_k\}$ bitişe göre sıralı bir en iyi çözüm olsun. $b_1 = a_1$ ise söylenecek bir şey yoktur. Değilse $O' = (O - \{b_1\}) \cup \{a_1\}$ kümesi kurulur. a_1 en erken biten etkinlik olduğu için $f(a_1) \le f(b_1)$'dir. O çakışmasız olduğu için $f(b_1) \le s(b_2)$'dir. Bu iki eşitsizlik birleşince $f(a_1) \le s(b_2)$ bulunur. $b_2, \ldots, b_k$ etkinliklerinin hepsi b_2'den sonra başladığı için a_1 bunların hiçbiriyle çakışmaz. Demek ki O' uygundur. Eleman sayısı da değişmemiştir: $|O'| = |O| = k$. Öyleyse O' de bir en iyi çözümdür ve a_1'i içerir.

Optimal alt yapı için a_1 ve onunla çakışan etkinlikler atılır. Kalan etkinlikler, $f(a_1)$'den sonra başlayan etkinliklerdir ve aynı türden bir problem oluşturur. O''den a_1 çıkarılınca kalan küme bu küçük problemin bir çözümüdür. Küçük problemin daha büyük bir çözümü olsaydı, ona a_1 eklenerek O''den büyük bir çözüm elde edilirdi; bu, O''nin en iyi olmasıyla çelişir. Tümevarımla algoritmanın her adımdaki seçimi bir en iyi çözümün parçası olarak kalır. □

Kanıtta en erken biten etkinliğin özel bir rolü vardır: değiştirilen b_1'den *daha geç* bitmez. Bu yüzden onun yerine geçtiğinde sonraki etkinliklere engel olmaz. En erken başlayan etkinlik için aynı argüman kurulamaz: en erken başlayan etkinlik çok geç bitebilir ve b_2 ile çakışabilir. Şekil 7.5'teki *[0, 10)* tam olarak bu durumdadır.

Aynı problem ters yönden de çözülebilir. Etkinlikler başlangıca göre büyükten küçüğe sıralanır ve her adımda *en geç başlayan* etkinlik seçilir. Bu, zaman ekseni ters çevrildiğinde en erken biten ölçütünün aynısıdır ve aynı kanıtla en iyi sonucu verir. Problemin bir genellemesi olan *ağırlıklı* etkinlik seçiminde ise her etkinliğin bir değeri vardır ve toplam değer en büyük olmalıdır. Bu sürümde hiçbir basit açgözlü ölçüt doğru değildir; problem dinamik programlamayla çözülür.

7.4 Son tarihli iş sıralama

Bir atölyede tek bir makine var ve birkaç sipariş sırada bekliyor. Her sipariş makinede bir gün sürer. Her siparişin bir teslim tarihi ve bir kârı vardır; sipariş teslim tarihine kadar bitirilirse kâr kazanılır, bitirilmezse sipariş kaybedilir. Atölye hangi siparişleri hangi günlere koyarsa toplam kârı en büyük olur? Bu problem etkinlik seçimine benzer, ama burada bütün işlerin süresi aynıdır ve önemli olan sayı değil, kârdır.

7.4.1 Problemin tanımı

Tanım 7.6

Son tarihli iş sıralama (*job sequencing with deadlines*): n iş verilir. Her iş 1 birim sürer; i. işin bir son tarihi d_i ve bir kârı k_i vardır. Zaman 1, 2, 3, … diye numaralanmış birim dilimlere (*slot*) bölünmüştür ve her dilimde en fazla bir iş yapılır. İş d_i. dilimin sonuna kadar bitirilirse k_i kâr kazanılır. Toplam kârı en büyük olan iş kümesi ve dilim ataması bulunmak istenir.

Bir işin d_i son tarihi, işin 1'den d_i'ye kadar olan dilimlerden birine konabileceği anlamına gelir. Son tarihi 1 olan bir iş yalnız birinci dilime konabilir. Son tarihi 3 olan bir iş birinci, ikinci ya da üçüncü dilime konabilir. Sunumdaki örnekte altı iş vardır (Tablo 7.8).

İş	1	2	3	4	5	6
son tarih	4	1	1	1	3	2
kâr	20	10	40	30	50	30

Tablo 7.8: Sunumdaki altı iş.

İlk bakışta üç işin (2, 3 ve 4) son tarihi 1'dir ve bunlardan yalnız biri yapılabilir. Hangisinin yapılacağı ve öteki işlerin nereye konacağı birlikte düşünülmelidir.

7.4.2 Açgözlü ölçüt ve en geç boş dilim

Açgözlü algoritma iki karar verir. Birincisi işlerin hangi sırayla ele alınacağıdır: işler **kâra göre azalan** sırayla incelenir, yani en kârlı iş önce. İkincisi her işin hangi dilime konacağıdır: iş, son tarihinden önceki **en geç boş** dilime konur. Son tarihine kadar boş dilim kalmamışsa iş atlanır ve bir daha ele alın-

maz.

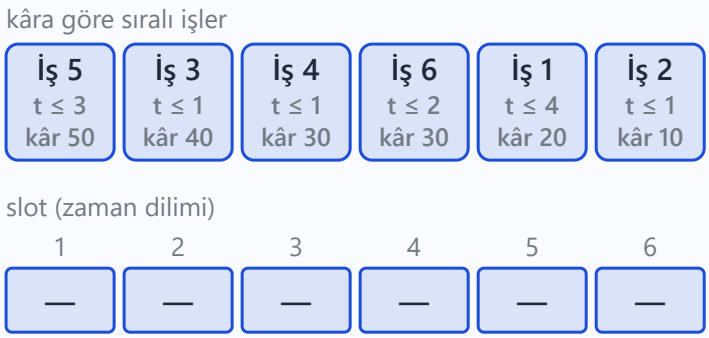
İkinci kararın gerekçesi şudur. Bir işi gereğinden erken bir dilime koymak, son tarihi daha yakın olan bir işin yerini kapatabilir. Örneğin son tarihi 3 olan bir iş birinci dilime konursa, sonradan gelecek ve son tarihi 1 olan bir işin gidebileceği tek yer dolmuş olur. Aynı iş üçüncü dilime konsaydı birinci dilim boş kalırdı. En geç boş dilim, erken dilimleri son tarihi daha yakın işlere bırakır ve esnekliği korur.

7.4.3 Kod ve izleme

Sunumdaki `isSirala` metodu bu iki kuralı uygular ve toplam kârı döndürür:

```
static int isSirala(Is[] is, int n) {
  Arrays.sort(is, (a, b) -> b.kar - a.kar);
  int[] slot = new int[n];
  Arrays.fill(slot, -1);
  int toplam = 0;
  for (int i = 0; i < n; i++) {
    int j = Math.min(n, is[i].sonTarih) - 1;
    for (; j >= 0; j--) {
      if (slot[j] == -1) {
        slot[j] = is[i].id;
        toplam += is[i].kar;
        break;
      }
    }
  }
  return toplam;
}
```

`Is` sınıfının `id`, `sonTarih` ve `kar` alanları vardır. `Arrays.sort` çağrısındaki karşılaştırıcı `b.kar - a.kar`, kârı büyük olan işi öne alır. Kararlı sıralama sayesinde eşit kârlı işler ilk sıralarını korur. `slot` dizisi n dilimi tutar: `slot[j]`, $j + 1$. dilime konan işin numarasıdır, boşsa −1'dir. Java'da diziler 0'dan başladığı için birinci dilim `slot[0]`'dadır. n iş en fazla n dilim kullanabilir; bu yüzden n'den büyük son tarihler n'ye indirilir. `Math.min(n, is[i].sonTarih) - 1` ifadesi bunu yapar ve işin konabileceği en geç dilimin indisini verir. İç döngü bu dilimden geriye doğru ilk boş dilimi arar. Boş dilim bulunursa iş yerleştirilir, kâr toplama eklenir ve `break` ile arama biter. Döngü $j = -1$'e kadar inerse uygun dilim yoktur ve iş sessizce atlanır.

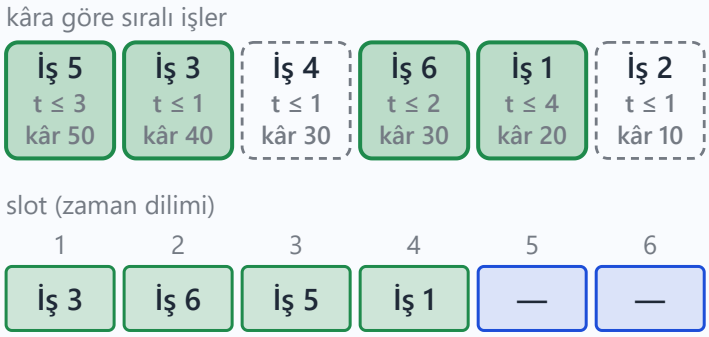


Şekil 7.8: İşler kâra göre azalan sırada; altı dilim boş. Eşit kârlı İş 4 ve İş 6 ilk sıralarını korur.

Kâra göre sıralanmış dizi İş 5 (50), İş 3 (40), İş 4 (30), İş 6 (30), İş 1 (20), İş 2 (10) şeklindedir. Tablo 7.9 her işin yerleştirilmesini adım adım verir. Tabloda dilimler 1'den numaralanır.

İş	Son tarih	Kâr	Bakılan dilimler	Karar	Dilimler 1–6	`toplam`
5	3	50	3	3. dilime	– – 5 – – –	50
3	1	40	1	1. dilime	3 – 5 – – –	90
4	1	30	1 (dolu)	atlanır	3 – 5 – – –	90
6	2	30	2	2. dilime	3 6 5 – – –	120
1	4	20	4	4. dilime	3 6 5 1 – –	140
2	1	10	1 (dolu)	atlanır	3 6 5 1 – –	140

Tablo 7.9: `isSirala(is, 6)` izlemesi.



Şekil 7.9: Sonuç: dilimler 1–4'te sırasıyla İş 3, İş 6, İş 5, İş 1; toplam kâr 140. İş 4 ve İş 2 yapılamaz.

Metot 140 döndürür. Makine birinci gün İş 3'ü, ikinci gün İş 6'yı, üçüncü gün İş 5'i, dördüncü gün İş 1'i yapar. Beşinci ve altıncı dilimler boş kalır; hiçbir işin son tarihi 4'ü geçmediği için bu dilimlerin kullanılması zaten mümkün değildir. 140'ın en büyük toplam kâr olduğu, doğrulama betiğinde altı işin bütün alt kümeleri denenerek gösterilmiştir.

İzlemenin ilginç adımı İş 4'tür. Kârı 30'dur ve İş 6 ile eşittir, ama son tarihi 1'dir ve birinci dilimi daha kârlı İş 3 almıştır. İş 4 için başka bir dilim yoktur. İş 6 ise son tarihi 2 olduğu için ikinci dilime sığar. Eşit kârlı iki işten birinin yapılıp ötekinin yapılamaması, son tarihlerin farklı olmasından kaynaklanır.

Örnek 7.4

Beş iş (numara, son tarih, kâr) olarak verilsin: (1, 2, 100), (2, 1, 19), (3, 2, 27), (4, 1, 25), (5, 3, 15). Kâra göre sıra İş 1, İş 3, İş 4, İş 2, İş 5'tir. İş 1'in son tarihi 2'dir; 2. dilim boştur ve iş oraya konur (toplam 100). İş 3'ün son tarihi de 2'dir. 2. dilim doludur, bir öncekine bakılır: 1. dilim boştur ve iş oraya konur (toplam 127). İş 4'ün ve İş 2'nin son tarihi 1'dir; 1. dilim doludur ve ikisi de atlanır. İş 5'in son tarihi 3'tür; 3. dilim boştur (toplam 142). Sonuç: makine sırasıyla İş 3, İş 1 ve İş 5'i yapar, toplam kâr 142'dir.

7.4.4 Yanlış ölçütler

İki karardan her biri yanlış seçilebilir. Önce işlerin sırası ele alınsın. İşleri son tarihe göre sıralamak doğal görünür: önce acil işler yapılsın. Ancak bu ölçüt kârı dikkate almaz. Son tarihi 1 olan iki iş düşünülsün: birinin kârı 10, ötekinin 40'tır. Yalnız biri yapılabilir. Son tarihe göre sıralamada (kararlı sıralamayla) ilk gelen iş seçilir; ilk gelen kârı 10 olan işse sonuç 10'dur. Kâra göre sıralama ise 40'lık işi seçer. Sunumdaki altı işte son tarihe göre sıralama ve en geç boş dilim kuralı 110 kâr verir; kâr sırası 140 verir.

Sonra dilim seçimi ele alınsın. İş, son tarihinden önceki *en erken* boş dilime konsun. Sunumdaki altı işte İş 5 (son tarih 3) birinci dilime gider. Ardından gelen İş 3'ün tek seçeneği birinci dilimdir, ama o dolmuştur; 40 kârlı iş kaybedilir. Algoritma sonunda İş 5, İş 6 ve İş 1'i yapar ve yalnız 100 kâr elde eder. En geç boş dilim kuralıyla aynı işler 140 kâr getirir. İki işli bir örnek daha kısadır: X'in son tarihi 2, kârı 50; Y'nin son tarihi 1, kârı 40 olsun. En erken boş dilim kuralı X'i birinci dilime koyar ve Y'yi kaybeder: 50. En geç boş dilim kuralı X'i ikinci dilime koyar, Y birinci dilime sığar: 90.

Sık yapılan hata

İşi son tarihinden önceki en erken boş dilime koymak. Bu kural, sonradan gelecek ve son tarihi yakın olan işlere yer bırakmaz. Doğru kural en geç boş dilimdir. Kodda bu, iç döngünün `j` 'yi son tarihten başlatıp geriye saymasıyla sağlanır; döngü 0'dan başlayıp ileri sayarsa algoritma yanlış olur.

7.4.5 Doğruluk ve maliyet

Algoritmanın doğruluğu iki parçada gösterilir. Birinci parça, en geç boş dilim kuralının bir işi ancak gerçekten sığmıyorsa reddettiğidir.

Teorem 7.2 (En geç boş dilim)

Algoritmanın o ana kadar kabul ettiği işler kümesi A ve sıradaki iş x olsun. En geç boş dilim kuralı x için boş bir dilim bulamıyorsa, $A \cup \{x\}$ kümesindeki işler hiçbir dilim atamasıyla son tarihlerine yetiştirilemez.

İspat. x için boş dilim bulunamadığına göre 1'den $d(x)$'e kadar bütün dilimler doludur. $t \geq d(x)$, 1'den t'ye kadar bütün dilimlerin dolu olduğu en büyük sayı olsun; $t + 1$. dilim boştur ya da $t = n$'dir. 1–t dilimlerindeki her işin son tarihi t'den küçük ya da eşittir. Bunun nedeni şudur: son tarihi t'den büyük olan bir iş yerleştirildiği anda $t + 1$. dilim boştu, çünkü dilimler hiçbir zaman boşaltılmaz. En geç boş dilim kuralı o işi $t + 1$'e ya da daha geç bir dilime koyardı, 1–t arasına koymazdı. Öyleyse 1–t dilimlerindeki t işin ve x'in, yani $t + 1$ işin hepsinin son tarihi en fazla t'dir. t dilime $t + 1$ iş sığmaz. □

İkinci parça, kâra göre azalan sıranın doğru olduğudur ve bir değiştirme argümanıyla gösterilir. En kârlı iş a olsun ve tek başına son tarihine yetişebilsin. O, a'yı içermeyen bir en iyi çözüm olsun. O'nun 1–$d(a)$ dilimlerinden biri boşsa a oraya eklenir ve kâr artar; bu, O'nun en iyi olmasıyla çelişir. Hepsi doluysa bu dilimlerden birindeki b işi çıkarılıp yerine a konur. a kendi son tarihinden önceki bir dilime konduğu için çözüm uygundur. $k(a) \geq k(b)$ olduğu için kâr azalmaz. Demek ki a'yı içeren bir en iyi çözüm vardır. Sunum bu sonucu daha genel bir dille de anar: birlikte son tarihlerine yetiştirilebilen iş kümeleri bir **matroid** oluşturur ve matroidlerde ağırlığa göre azalan açgözlü seçim her zaman en iyidir. Kruskal algoritmasının doğruluğu da aynı genel sonucun bir örneğidir.

Maliyet şöyle hesaplanır. Sıralama $O(n \log n)$ sürer. Her iş için dilim araması en fazla $\min(n, d_i)$ dilime bakar; en büyük son tarih d_{max} ise arama toplam $O(n \cdot d_{max})$ sürer. d_{max}, n'ye kadar çıkabildiği için en kötü durum $O(n^2)$'dir. Örneğin bütün işlerin son tarihi n ise i. iş i dilime bakar ve toplam $1 + 2 + \dots + n = n(n + 1)/2$ bakış yapılır. Bu maliyet ayrık kümeler (*union-find*) yapısıyla düşürülebilir. Her dolu dilim, solundaki ilk boş dilimi gösteren bir kümeye bağlanır. Bir işin en geç boş dilimi tek bir `bul` çağrısıyla bulunur; dilim dolunca soldaki kümeyle `birlestir` yapılır. Yol kısaltma ve rütbeye göre birleştirme ile her işlem neredeyse sabit sürer ve dilim araması toplamda neredeyse $O(n)$'e iner. Toplam maliyeti bu durumda sıralama belirler.

7.5 Kesirli sırt çantası

Bir yolcunun sırt çantası en fazla 50 kilogram taşıyabilir. Önünde birkaç çuval baharat vardır; her çuvalın bir ağırlığı ve bir değeri bellidir. Baharat bölünebilir: bir çuvalın yarısı alınırsa değerinin yarısı kazanılır. Yolcu çantaya ne koyarsa taşıdığı değer en büyük olur? Bu soru kesirli sırt çantası problemidir. Nesneler bölünemediğinde, yani her nesne ya bütünüyle alındığında ya hiç alınmadığında problemin adı 0/1 sırt çantasıdır. İki problem birbirine çok benzer, ama açgözlü yaklaşım yalnız birincisinde en iyi sonucu verir.

7.5.1 Problem ve oran ölçütü

Tanım 7.7

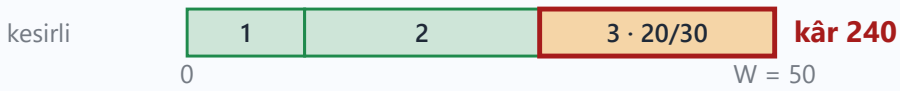
Kesirli sırt çantası (*fractional knapsack*): n nesne ve bir W kapasitesi verilir; i. nesnenin kârı k_i, ağırlığı w_i'dir. Her nesneden $0 \le x_i \le 1$ oranında bir parça alınabilir. Toplam ağırlık $\Sigma\, x_i\, w_i \le W$ koşulunu sağlayan ve toplam kârı $\Sigma\, x_i\, k_i$ en büyük yapan seçim bulunmak istenir.

Bir nesnenin ne kadar "verimli" olduğunu ağırlık başına getirdiği kâr gösterir. Bu değere nesnenin **oranı** denir: k_i / w_i. Açgözlü ölçüt, oranı en büyük nesneden başlayarak sığdığı kadarını almaktır. Bir nesnenin tamamı sığıyorsa tamamı alınır. Sığmıyorsa kalan kapasite kadar bir parçası alınır ve çanta dolar. Sunumdaki örnekte kapasite $W = 50$'dir ve üç nesne vardır (Tablo 7.10).

Nesne	Kâr	Ağırlık	Oran
1	60	10	6
2	100	20	5
3	120	30	4

Tablo 7.10: Sunumdaki üç nesne ($W = 50$).

Oranlar 6, 5 ve 4'tür; nesneler zaten oran sırasındadır. Nesne 1'in tamamı alınır (10 kg, 60 kâr). Nesne 2'nin tamamı alınır (20 kg, 100 kâr). Geriye 20 kg kapasite kalır, ama nesne 3 30 kg'dır. Nesne 3'ün $20/30$'u alınır ve $120 \cdot 20/30 = 80$ kâr kazanılır. Toplam $60 + 100 + 80 = 240$'tır.



Şekil 7.10: 50 kg'lık çantanın dolumu: nesne 1 ve 2 tümüyle, nesne 3'ün 20/30'u. Toplam kâr 240.

Dikkat edilirse oran ölçütü kârı en büyük nesneyi önce almaz. Nesne 3'ün kârı en büyüktür (120), ama ağırlığı da en büyüktür ve kilogram başına en az kârı getirir. Kâra göre azalan sırayla almak nesne 3'ün tamamını (30 kg, 120) ve nesne 2'nin tamamını (20 kg, 100) seçer ve 220'de kalır. Ağırlığa göre artan sırayla almak bu örnekte oran sırasıyla aynı sonucu verir, ama bu bir rastlantıdır: ağır ve çok kârlı bir nesne ile hafif ama değersiz bir nesne bulunan bir girdide yanılır.

7.5.2 Kod ve izleme

Sunumdaki `kesirli` metodu nesneleri orana göre azalan sıralar ve çantayı doldurur:

```
static double kesirli(Nesne[] s, int W) {
  Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
  double kar = 0;
  int kalan = W;
  for (Nesne x : s) {
    if (x.agirlik <= kalan) {        // tamamı sığar
      kalan -= x.agirlik;
      kar += x.kar;
    } else {                         // kesirli parça
      kar += x.kar * (double) kalan / x.agirlik;
      break;
    }
  }
  return kar;
}
```

`Nesne` sınıfının `kar` ve `agirlik` alanları ile `oran()` metodu vardır; `oran()`, `(double) kar / agirlik` değerini döndürür. Karşılaştırıcı `Double.compare(b.oran(), a.oran())` oranı büyük olanı öne alır. Burada çıkarma yerine `Double.compare` kullanılır, çünkü oranlar ondalıklıdır ve iki oranın farkı bir tam sayıya çevrilirse 0,5 gibi farklar kaybolur. `kalan`, çantada kalan kapasitedir. Döngü her nesne için önce tamamının sığıp sığmadığına bakar. Sığıyorsa kapasiteden ağırlığı düşer ve kârı ekler. Sığmıyorsa `else` dalında nesnenin `kalan / agirlik` kadarlık bir parçasını alır. Çanta artık tam dolmuştur ve `break` ile döngüden çıkılır. Kesirli parçanın kârı hesaplanırken `(double)` dönüşümü önemlidir: `kalan / x.agirlik` tam sayı bölmesi olarak yazılsaydı 20/30 sıfıra yuvarlanırdı.

Nesne (oran)	**Koşul** `agirlik <= kalan`	**İşlem**	`kalan`	`kar`
başlangıç	—	sıralama: 1, 2, 3	50	0
1 (6)	10 ≤ 50 doğru	tamamı alınır	40	60
2 (5)	20 ≤ 40 doğru	tamamı alınır	20	160
3 (4)	30 ≤ 20 yanlış	20/30'u alınır: *120 · 20/30 = 80*; `break`	20	240

Tablo 7.11: `kesirli(s, 50)` izlemesi.

Son satırda `kalan` 20 olarak kalır, çünkü `else` dalı kapasiteyi güncellemez; hemen ardından döngü kırıldığı için bu değer bir daha kullanılmaz. Metot `240.0` döndürür.





Şekil 7.11: Kesirli sırt çantası: başlangıçta boş çanta (üstte) ve dolmuş çanta (altta). Nesne 1 ve 2 tümüyle, nesne 3'ün 20/30'u alınmıştır; çanta dolmuştur, kâr 240.

Maliyetin baskın kısmı sıralamadır: $O(n \log n)$. Tarama en fazla n nesneye bakar ve $O(n)$ sürer. Sıralama yerine her adımda en büyük oranlı nesne bir heap'ten alınabilir; çanta erken dolarsa heap'ten yalnız birkaç nesne çıkarılır ve toplam süre $O(n + m \log n)$'ye iner (m, alınan nesne sayısıdır). Doğrusal zamanda çalışan, ortanca bulmaya dayanan bir yöntem de vardır.

Örnek 7.5

Kapasite $W = 10$, nesneler (kâr, ağırlık) olarak (10, 5), (40, 4), (30, 6), (50, 3) olsun. Oranlar sırasıyla 2, 10, 5 ve yaklaşık 16,7'dir. Oran sırası (50, 3), (40, 4), (30, 6), (10, 5) olur. (50, 3)'ün tamamı alınır: kalan 7, kâr 50. (40, 4)'ün tamamı alınır: kalan 3, kâr 90. (30, 6) sığmaz; 3/6'sı alınır ve $30 \cdot 3/6 = 15$ kâr eklenir. Toplam 105'tir. (10, 5) hiç alınmaz.

7.5.3 Oran ölçütü neden en iyidir?

Teorem 7.3 (Kesirli sırt çantasında oran ölçütü)

Nesneleri oranlarına göre azalan sırayla, sığdığı kadarıyla alan açgözlü algoritma kesirli sırt çantası probleminin en iyi çözümünü bulur.

İspat. Nesneler $k_1/w_1 \geq k_2/w_2 \geq \dots \geq k_n/w_n$ olacak biçimde numaralansın. Toplam ağırlık W'den küçükse her şey alınır ve söylenecek bir şey yoktur; bu yüzden çantanın dolduğu varsayılır. Açgözlü çözüm G, bir en iyi çözüm O olsun. O farklıysa, iki çözümün ayrıldığı ilk nesne i olsun. Açgözlü yöntem i'den sığabildiği kadar aldığı için O, i'den daha az almıştır. O'nun da çantası dolu olduğundan, bu eksiklik i'den sonra gelen bir j nesnesinden fazladan alınarak kapatılmıştır. O'da j'nin δ kilogramı çıkarılıp yerine i'nin δ kilogramı konur. Ağırlık değişmez, çözüm uygundur. Kâr $\delta \cdot (k_i/w_i - k_j/w_j)$ kadar değişir ve bu değer negatif değildir, çünkü i'nin oranı j'ninkinden küçük değildir. Değişiklik O'yu kârını düşürmeden G'ye bir adım yaklaştırır. Bu işlem sonlu kez tekrarlanınca O, G'ye dönüşür; demek ki G'nin kârı O'nunkinden az değildir. □

Kanıtın can alıcı adımı, ağırlığın küçük parçalar hâlinde değiştirilebilmesidir. δ kilogram bir nesneden çıkarılıp öbürüne eklenebilir, çünkü nesneler bölünebilir. En yüksek oranlı nesnenin her kilogramı, aynı ağırlıktaki başka her seçenekten en az onun kadar kâr getirir. Bu yüzden onu sonuna kadar alma-

mak için bir neden yoktur.

7.5.4 0/1 sırt çantasında oran ölçütü

Nesneler bölünemezse kanıttaki değiştirme adımı çöker. Aynı üç nesne ve 50 kg'lık çanta ele alınsın, ama bu kez her nesne ya bütünüyle alınsın ya hiç alınmasın. Oran ölçütü nesne 1'i ve nesne 2'yi alır (30 kg, 160 kâr). Nesne 3 kalan 20 kg'a sığmaz ve atlanır. Sonuç 160'tır. Oysa nesne 2 ile nesne 3 birlikte tam 50 kg tutar ve 220 kâr getirir. Bütün seçimler karşılaştırılınca en iyisinin bu olduğu görülür. Tek nesneler 60, 100 ve 120 getirir. İkili seçimler *1 + 2 = 160*, *1 + 3 = 180* ve *2 + 3 = 220* getirir. Üç nesne birden 60 kg tutar ve sığmaz.

Yöntem	Seçim	Ağırlık	Kâr
kesirli, oran ölçütü	1, 2 ve 3'ün 20/30'u	50	240 (en iyi)
0/1, oran ölçütü	1, 2	30	160
0/1, en iyi	2, 3	50	220

Tablo 7.12: Aynı nesnelerle kesirli ve 0/1 sırt çantası (*W = 50*).

0/1 durumunda oran ölçütü 20 kg'lık boşluğu dolduramaz. Kesirli durumda bu boşluk nesne 3'ün bir parçasıyla doldurulur. 0/1 durumunda ise parça alınamaz ve boşluk değersiz kalır. Değiştirme argümanı da işlemez: en iyi çözümdeki nesne 3 çıkarılıp yerine nesne 1 konursa ağırlık 30 kg'a düşer ama kâr 220'den 160'a iner. Ağırlıklar eşit olmadığı için değiştirme "aynı ağırlık, daha çok kâr" biçiminde yapılamaz. 0/1 sırt çantası dinamik programlamayla $O(n \cdot W)$ sürede çözülür.

Kesirli çözümün değeri 0/1 çözümünün değeri için bir üst sınırdır: 0/1 çözümü kesirli problemin de uygun bir çözümüdür, bu yüzden kesirli en iyinin değerini geçemez. Örnekte 220 ≤ 240'tır. Bu sınır, 0/1 sırt çantasını dal ve sınır yöntemiyle çözen algoritmalarda budama için kullanılır.

> **Sık yapılan hata**
>
> Kesirli problemde en iyi olan oran ölçütünü 0/1 sırt çantasına uygulamak. İki problemin girdisi aynıdır, yalnız "nesne bölünebilir mi?" sorusunun cevabı farklıdır. Bu küçük fark açgözlü seçim özelliğini ortadan kaldırır. Oran ölçütü 0/1 problemde ancak bir sezgisel olarak kullanılabilir.

7.6 Minimum kapsayan ağaçta açgözlü seçim

Minimum kapsayan ağaç, açgözlü yaklaşımın en başarılı uygulamalarından biridir. Bağlı, ağırlıklı ve yönsüz bir çizgenin **kapsayan ağacı**, çizgenin bütün düğümlerini içeren, bağlı ve döngüsüz bir alt çizgesidir; V düğümlü bir çizgede tam olarak $V - 1$ kenarı vardır. Kenar ağırlıklarının toplamı en küçük olan kapsayan ağaca **minimum kapsayan ağaç** (*minimum spanning tree*, MST) denir. Bir şehirler

ağını en az kablo ile bağlamak bu problemin tipik bir uygulamasıdır. Kruskal, Prim ve Borůvka algoritmaları MST'yi bulur ve üçü de açgözlüdür. Bu kesim algoritmaları baştan anlatmaz; üçünün açgözlü seçiminin neden güvenli olduğunu tek bir özellik üzerinden açıklar.

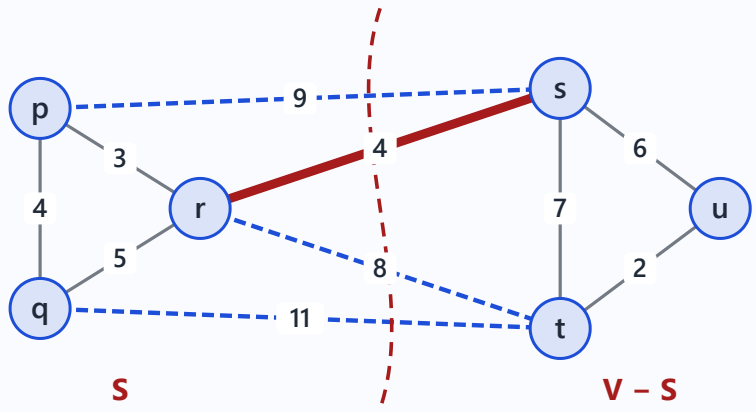
7.6.1 Kesim özelliği

Bir çizgenin düğümleri iki boş olmayan kümeye ayrılsın: S ve $V - S$. Bu ayrıma bir **kesim** (*cut*) denir. Bir ucu S'de, öteki ucu $V - S$'de olan kenarlar kesimi **geçer**. Her kapsayan ağaç kesimi geçen en az bir kenar içermek zorundadır; yoksa S ile $V - S$ arasında bağlantı kurulamaz.

Teorem 7.4 (Kesim özelliği)

Bağlı, ağırlıklı, yönsüz bir çizgenin herhangi bir kesimi alınsın. Kesimi geçen kenarlar arasında ağırlığı en küçük olan kenar e ise, e'yi içeren bir MST vardır. Kesimi geçen en hafif kenar tekse, bu kenar her MST'de bulunur.

İspat. Kanıt yine değiştirme argümanıdır. T, $e = (u, v)$'yi içermeyen bir MST olsun; $u \in S$, $v \in V - S$. T bir ağaç olduğu için u'dan v'ye tek bir yol vardır. Bu yol S'den başlayıp $V - S$'de bittiğine göre kesimi en az bir kez geçer; geçtiği kenar e' olsun. T'ye e eklenir ve e' çıkarılır. e' çıkarılınca ağaç iki parçaya ayrılır, e bu iki parçayı yeniden bağlar; sonuç yine bir kapsayan ağaçtır. Ağırlık $w(e) - w(e')$ kadar değişir. e kesimi geçen en hafif kenar olduğu için bu fark sıfırdan büyük değildir. Yeni ağaç T'den ağır olmadığı için o da bir MST'dir ve e'yi içerir. e tek en hafif kenarsa fark kesin olarak negatiftir; bu durumda T MST olamaz ve e'yi içermeyen MST yoktur. □



Şekil 7.12: Kesim özelliği. Kesikli eğri düğümleri $S = \{p, q, r\}$ ve $V - S = \{s, t, u\}$ diye ayırır. Kesimi geçen kenarlar (mavi kesikli ve kırmızı) 9, 4, 8 ve 11'dir. En hafifi r–s (4) MST'dedir.

Şekil 7.12'deki çizgede kesimi dört kenar geçer: p–s (9), r–s (4), r–t (8) ve q–t (11). En hafifi r–s'dir ve tek en hafif kenar olduğu için her MST'de bulunur. Çizgenin MST'si gerçekten t–u (2), p–r (3), p–q (4), r–s (4) ve s–u (6) kenarlarından oluşur; toplam ağırlık 19'dur. Kesimi geçen kenarlardan yalnız r–s

ağaca girer. Bu tesadüf değildir: MST'nin S ile $V - S$'yi tam olarak bir kenarla bağlaması yeterlidir ve bu kenar en ucuzu olmalıdır.

Kesim özelliğinin açgözlü algoritmalar için anlamı şudur. Algoritma o ana kadar seçtiği kenarlarla bazı bileşenler kurmuştur. Bu bileşenlerden birini öbürlerinden ayıran bir kesim seçilir. Seçilmiş kenarların hiçbiri bu kesimi geçmez. Kesimi geçen en hafif kenar eklenirse, seçilmiş kenarlarla birlikte yine bir MST'nin parçası olarak kalınır. Kanıttaki değiştirme bu durumda da çalışır, çünkü çıkarılan e' kesimi geçer ve bu yüzden seçilmiş kenarlardan biri değildir. Algoritma bu yüzden hiçbir seçimini geri almak zorunda kalmaz.

Ağırlıkların hepsi farklıysa her kesimin en hafif kenarı tektir ve MST de tektir. Şekildeki çizgede p–q ve r–s'nin ikisinin de ağırlığı 4'tür, yani ağırlıklar tümüyle farklı değildir; yine de bu çizgenin MST'si tektir. Ağırlıkların hepsinin farklı olması teklik için yeterlidir, ama gerekli değildir. Eşit ağırlıklar birden çok MST'ye yol açabilir; ama bütün MST'lerin toplam ağırlığı aynıdır.

Örnek 7.6

Dört düğümlü bir çizgede düğümler *{A, B}* ve *{C, D}* olarak ayrılıyor; kesimi geçen kenarlar A–C 5, B–C 3 ve B–D 4'tür. Kesimi geçen en hafif kenar B–C'dir (3) ve tek en hafif kenardır; kesim özelliğine göre her MST'de bulunur. A–C ve B–D için kesim özelliği tek başına bir şey söylemez. Bunların MST'de olup olmadığı çizgenin geri kalan kenarlarına bağlıdır. Örneğin A–B ve C–D kenarları da varsa ve hafifse, MST kesimi yalnız B–C ile geçer.

Sık yapılan hata

Kesim özelliğini "en hafif kenarlar her zaman güvenlidir" diye okumak. Güvenli olan, bir *kesimi geçen* en hafif kenardır. Bütün çizgenin en hafif kenarları sırayla alınırsa bir döngü oluşabilir: döngünün üçüncü kenarı, ilk ikisinin zaten bağladığı iki düğümü yeniden bağlar ve hiçbir kesimi geçmez. Kruskal algoritması bu yüzden döngü kuran kenarı atar; atılan kenarın uçlarını ayıran her kesimi, seçilmiş kenarlardan biri zaten geçer.

7.6.2 Kruskal, Prim ve Borůvka: üç kesim

Üç MST algoritması aynı özelliği farklı kesimlerle kullanır. Fark, her adımda hangi kesimin seçildiğidir. Tablo 7.13 üç algoritmayı yan yana verir.

	Kruskal	Prim	Borůvka
Açgözlü seçim	tüm çizgedeki, döngü kurmayan en hafif kenar	ağaçtan çıkan en hafif kenar	her bileşenden çıkan en hafif kenar
Büyüme	orman → ağaç	tek ağaç, düğüm düğüm	orman, turlarla birleşme
Kullandığı kesim	kenarın uçlarını ayıran iki bileşen	ağaç ile geri kalan düğümler	her bileşen ile geri kalanı

	Kruskal	Prim	Borůvka
Veri yapısı	sıralama ve ayrık kümeler	öncelikli kuyruk	ayrık kümeler
Zaman	$O(E \log V)$	$O(E \log V)$ (ikili heap)	$O(E \log V)$
Uygun olduğu	seyrek çizge, kenar listesi	yoğun çizge, komşuluk yapısı	paralel ya da dağıtık hesaplama

Tablo 7.13: Kruskal, Prim ve Borůvka'nın açgözlü seçimi.

Kruskal kenarları artan ağırlık sırasıyla inceler. Sıradaki kenar iki farklı bileşeni bağlıyorsa eklenir, aynı bileşenin iki düğümünü bağlıyorsa döngü kuracağı için atlanır. Eklenen kenarın kullandığı kesim, uçlarından birinin bileşeni ile geri kalan düğümler arasındaki kesimdir. Bu kesimi geçen daha hafif bir kenar olsaydı daha önce incelenir ve eklenirdi; demek ki eklenen kenar kesimin en hafifidir. Bileşen denetimi ayrık kümeler yapısıyla yapılır. Süreyi sıralama belirler: $O(E \log E)$. $E \le V^2$ olduğu için $\log E \le 2 \log V$'dir ve bu süre $O(E \log V)$ diye de yazılır.

Prim tek bir ağacı bir düğümden başlayarak büyütür. Her adımda ağaçtaki bir düğümü ağaç dışındaki bir düğüme bağlayan en hafif kenar eklenir. Kullanılan kesim, ağacın düğümleri ile geri kalan düğümler arasındaki kesimdir. Ağaç dışındaki her düğümün ağaca en ucuz bağlantısı bir öncelikli kuyrukta tutulur. İkili heap'le süre $O((V + E) \log V)$'dir; bağlı çizgede bu $O(E \log V)$'ye eşittir. Yoğun çizgelerde dizi taramalı bir sürüm $O(V^2)$ ile daha hızlı olabilir.

Borůvka turlar hâlinde çalışır. Her turda her bileşen kendinden çıkan en hafif kenarı bulur ve bu kenarların hepsi birlikte eklenir. Her bileşen kendi kesimini, yani "bu bileşen ile geri kalan her şey" kesimini kullanır. Her tur bileşen sayısını en az yarıya indirdiği için en fazla $\lceil \log_2 V \rceil$ tur vardır ve her tur $O(E)$ sürer; toplam $O(E \log V)$'dir. Bileşenlerin seçimleri birbirinden bağımsız olduğu için algoritma paralel çalışmaya uygundur. Ağırlıklar eşit olabiliyorsa tutarlı bir eşitlik bozucu gerekir; yoksa iki bileşen eşit ağırlıklı farklı kenarlar seçip bir döngü kurabilir.

Üç algoritma aynı çizgede aynı MST'yi bulur, yeter ki ağırlıklar farklı olsun. Ağırlıklar farklıysa MST tektir ve hepsinin bulduğu ağaç odur. Eşit ağırlıklar varsa farklı ağaçlar bulabilirler, ama toplam ağırlıkları aynıdır. Farkları kenarları ele alma sırasında ve gereken veri yapısındadır. Tüm kenarların ağırlığı aynı olduğunda örneğin her kapsayan ağaç bir MST'dir; Kruskal bu durumda kenar listesinin sırasına göre herhangi bir kapsayan ağacı üretir.

Bir kenarın ağırlığı tek başına onun MST'de olup olmayacağını söylemez. Her MST'de bulunan kenarlar, bir kesimin tek en hafif kenarı olanlardır. Hiçbir MST'de bulunmayan kenarlar ise bir döngünün tek en ağır kenarı olanlardır; bu ikinci kurala **döngü özelliği** denir ve ters silme algoritması ona dayanır. Kenarlara aynı sabitin eklenmesi MST'yi değiştirmez, çünkü kenarların sırası korunur ve bütün kapsayan ağaçlarda $V - 1$ kenar olduğundan bütün toplamlar aynı miktarda artar. Kenarlar azalan sırada işlenirse Kruskal aynı biçimde maksimum kapsayan ağacı bulur.

7.7 Huffman kodlama

Bir metin bilgisayarda bitlerle saklanır. Her karaktere bir bit dizisi, yani bir **kod sözcüğü** atanır ve metin bu kodların art arda yazılmasıyla oluşur. ASCII gibi kodlamalarda her karakterin kodu aynı uzunluktadır. Ancak gerçek metinlerde karakterler eşit sıklıkta geçmez. Türkçe bir metinde “a” ve “e” harfleri çok sık, “ğ” ve “j” çok seyrek görülür. Sık karakterlere kısa, seyrek karakterlere uzun kodlar verilirse metnin toplam uzunluğu kısalır. Huffman kodlama bu fikri en iyi biçimde uygulayan açgözlü bir algoritmadır.

7.7.1 Sabit ve değişken uzunluklu kodlar

Huffman kodlama **kayıpsız** bir veri sıkıştırma yöntemidir: sıkıştırılmış veri çözüldüğünde özgün veri bit bit aynen geri gelir. Yöntemi David Huffman 1952'de, öğrenciyken bir dönem ödevi olarak geliştirmiştir. Sunumdaki örnek metin 25 karakter uzunluğundadır ve yalnız altı farklı karakter içerir: F, O, R, G, E ve T. Altı karakteri birbirinden ayırmak için sabit uzunluklu bir kodda $\lceil \log_2 6 \rceil = 3$ bit gerekir; 2 bit yalnız dört karakteri ayırabilir. Bu durumda metin $25 \times 3 = 75$ bit tutar. Her karakter 8 bitlik ASCII ile saklansaydı 200 bit tutardı.

Değişken uzunluklu bir kodda kod sözcüklerinin uzunlukları farklıdır. Metnin toplam uzunluğu her karakterin frekansı ile kod uzunluğunun çarpımlarının toplamıdır:

> Toplam bit = Σ frekans(c) · uzunluk(c)

Bu toplamı küçültmek için en sık karakterlerin kodları kısa tutulmalıdır. Ancak kodlar keyfî biçimde kısaltılamaz; kısaltılmış kodların hâlâ tek anlamlı çözülebilmesi gerekir.

7.7.2 Önek-özgür kodlar

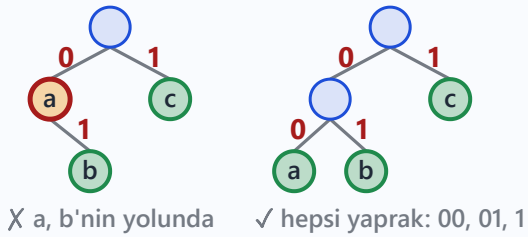
Kodlar a = 0, b = 01, c = 1 olsun. “01” bit dizisi alındığında bunun “ab” mi yoksa yalnız “b” mi olduğu anlaşılamaz. Sorunun kaynağı, a'nın kodu 0'ın, b'nin kodu 01'in başında yer almasıdır. Kod sözcükleri arasına bir ayırıcı konarak bu sorun çözülebilir, ama ayırıcı da bit harcar. Daha iyi bir çözüm, hiçbir kodun bir başka kodun başlangıcı olmamasıdır.

Tanım 7.8

Hiçbir kod sözcüğünün bir başka kod sözcüğünün öneki olmadığı koda **önek-özgür kod** (*prefix-free code*) denir. Önek-özgür bir kodla yazılmış bit dizisi, ayırıcı kullanmadan, soldan sağa okunarak tek anlamlı çözülür.

Önek-özgür kodlar ikili ağaçlarla gösterilir. Kökten başlanır; sol kenar 0, sağ kenar 1 anlamına gelir. Her karakter bir yapraktadır ve kökten o yaprağa giden yoldaki bitler karakterin kodudur. Bir karakter iç düğümde dursaydı, onun kodu altındaki yaprakların kodlarının öneki olurdu. Karakterler yalnız

yapraklarda durduğu için hiçbir kod bir başkasının öneki olamaz. Bir kodun uzunluğu, karakterin ağaçtaki derinliğidir.



Şekil 7.13: Solda a bir iç düğümdedir; kodu 0, b'nin kodu 01'in önekidir ve çözme belirsizdir. Sağda bütün karakterler yapraktadır; kodlar 00, 01 ve 1 önek-özgürdür.

Çözme işlemi ağaç üzerinde yapılır. Kökten başlanır; her bit okunduğunda 0 ise sol, 1 ise sağ çocuğa inilir. Bir yaprağa varılınca o yapraktaki karakter yazılır ve köke dönülür. Bir kod sözcüğünün bittiği an, yaprağa varıldığı andır ve bu an kesin olarak bilinir. Örneğin Şekil 7.13'ün sağındaki ağaçla "00110" dizisi 00 | 1 | 1 | 0... diye okunur: ilk iki bit a'yı, sonraki bit c'yi, bir sonraki yine c'yi verir. Son 0 tek başına bir yaprağa ulaşmaz; dizi eksik gelmiştir.

Optimal bir önek-özgür kodun ağacı **tam** bir ikili ağaçtır: her iç düğümün iki çocuğu vardır. Tek çocuklu bir iç düğüm olsaydı, o düğüm atlanıp çocuğu yukarı taşınabilirdi; altındaki bütün kodlar bir bit kısalırdı ve kodlar yine önek-özgür kalırdı. Şekildeki sol ağaçta kökün sol çocuğu olan a düğümünün tek çocuğu vardır; sağ ağaç bu yüzden daha iyidir.

Örnek 7.7

Dört kod kümesi ele alınsın. *{0, 10, 110, 111}* önek-özgürdür: 0 ile başlayan tek kod 0'dır, 10 ile başlayan tek kod 10'dur, 110 ve 111 birbirinin öneki değildir. Bu küme dört yapraklı, sağa doğru uzayan bir ağaca karşılık gelir. *{0, 01, 110, 111}* önek-özgür değildir; 0, 01'in önekidir. *{1, 10, 00, 01}* önek-özgür değildir; 1, 10'un önekidir. *{00, 01, 0, 11}* önek-özgür değildir; 0 hem 00'ın hem 01'in önekidir.

7.7.3 Algoritmanın adımları

Huffman algoritması ağacı aşağıdan yukarıya kurar. Her karakter başlangıçta kendi frekansıyla tek düğümlü bir ağaçtır. Her adımda frekansı en küçük iki ağaç birleştirilir: ikisi yeni bir iç düğümün çocukları olur ve yeni düğümün frekansı çocuklarının frekanslarının toplamıdır. Tek ağaç kalınca o ağaç Huffman ağacıdır. Adımlar şöyle sıralanır:

1. Her karakterin frekansı hesaplanır.
2. Her karakter için bir yaprak oluşturulur. En küçük frekanslı iki düğüm birleştirilerek yeni bir iç düğüm yapılır; bu adım tek düğüm kalana kadar tekrarlanır.

3. Ağaçta her karakterin kodu, kökten yaprağa giden yoldan okunur: sol kenar 0, sağ kenar 1.
4. Metin bu kodlarla kodlanır.

Açgözlü seçim ikinci adımdadır: her birleştirmede o an en küçük frekanslı iki ağaç seçilir. Bu iki ağacın yaprakları birleştirmeden sonra bir düzey derine iner, yani kodları bir bit uzar. En seyrek karakterlerin kodlarını uzatmak, toplam bit sayısını en az artıran seçimdir. Sunumdaki örneğin frekansları Tablo 7.14'te verilmiştir.

Karakter	F	O	R	G	E	T
Frekans	2	3	4	4	5	7

Tablo 7.14: Örnek metindeki karakter frekansları (toplam 25 karakter).

n karakter için tam olarak *n − 1* birleştirme yapılır, çünkü her birleştirme ağaç sayısını bir azaltır ve *n* ağaçtan bir ağaca inilir. Bu yüzden Huffman ağacında *n* yaprak ve *n − 1* iç düğüm vardır. Örnekte altı karakter, beş birleştirme vardır.

7.7.4 Ağacı kurmak: kod ve izleme

Sunumdaki `huffman` metodu ağacı bir öncelikli kuyrukla kurar:

```
static Dugum huffman(char[] h, int[] f) {
  PriorityQueue<Dugum> kuyruk =
    new PriorityQueue<>((a, b) -> a.f - b.f);
  for (int k = 0; k < h.length; k++)
    kuyruk.add(new Dugum(h[k], f[k]));
  while (kuyruk.size() > 1) {
    Dugum sol = kuyruk.poll();
    Dugum sag = kuyruk.poll();
    Dugum ata = new Dugum('\0', sol.f + sag.f);
    ata.sol = sol;
    ata.sag = sag;
    kuyruk.add(ata);
  }
  return kuyruk.poll();
}
```

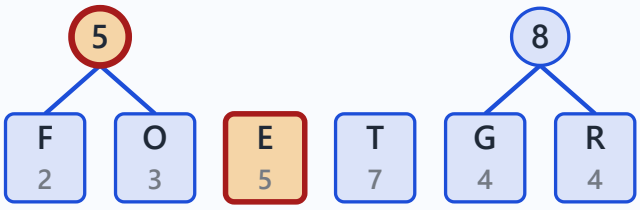
`Dugum` sınıfının dört alanı vardır: karakter `h`, frekans `f`, sol ve sağ çocuk. Yapraklarda çocuklar `null`'dır. Kuyruk, `(a, b) -> a.f - b.f` karşılaştırıcısıyla kurulan bir `PriorityQueue`'dur. Java'nın bu sınıfı bir en küçük heap'tir (*min-heap*); `poll()` her çağrıldığında frekansı en küçük düğümü çıkarır. İlk döngü her karakter için bir yaprak oluşturup kuyruğa ekler. `while` döngüsü kuyrukta birden çok düğüm olduğu sürece döner. Her turda iki kez `poll()` çağrılır; ilk çıkan sol, ikinci çıkan sağ

çocuk olur. Yeni iç düğümün karakteri `'\0'`'dır, çünkü iç düğümler bir karakter taşımaz; frekansı iki çocuğun toplamıdır. Yeni düğüm kuyruğa geri eklenir. Döngü bitince kuyrukta tek düğüm kalır ve bu düğüm ağacın köküdür.

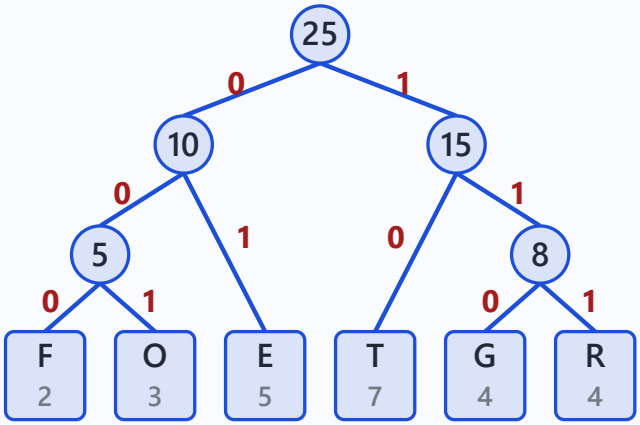
Tur	Sol (1. `poll`)	Sağ (2. `poll`)	Yeni düğüm	Tur sonunda kuyruk
başlangıç	—	—	—	F2, O3, R4, G4, E5, T7
1	F2	O3	(5) = 2 + 3	R4, G4, E5, (5), T7
2	G4	R4	(8) = 4 + 4	E5, (5), T7, (8)
3	(5)	E5	(10) = 5 + 5	T7, (8), (10)
4	T7	(8)	(15) = 7 + 8	(10), (15)
5	(10)	(15)	(25) = 10 + 15	(25)

Tablo 7.15: `huffman(h, f)` izlemesi. Parantez içindeki sayılar iç düğümlerdir.

Tablodaki kuyruk içerikleri frekans sırasıyla yazılmıştır; heap'in dizideki gerçek düzeni farklıdır. İkinci ve üçüncü turlarda eşitlik vardır. İkinci turda G4 ile R4'ün, üçüncü turda (5) ile E5'in frekansı aynıdır. Hangisinin önce çıkacağı heap'in iç düzenine, dolayısıyla karakterlerin kuyruğa eklenme sırasına bağlıdır. Tablo sunumdaki izlemeye uyar. Karakterler F, O, R, G, E, T sırasıyla eklendiğinde Java'nın `PriorityQueue`'su üçüncü turda önce E5'i çıkarır ve E sol çocuk olur. Bu durumda kodlar E = 00, F = 010, O = 011 olur; geri kalan kodlar değişmez. Kod uzunlukları ve toplam bit sayısı iki durumda da aynıdır: E 2 bit, F ve O 3 bit.



Şekil 7.14: Üçüncü turun başı: (5) ve (8) iç düğümleri kurulmuştur. Sıradaki en küçük iki düğüm (5) ve E5'tir (vurgulu).



Şekil 7.15: Tamamlanmış Huffman ağacı. Kök 25'tir; sol kenarlar 0, sağ kenarlar 1 ile etiketlidir.

Ağacın yapısı tablodan okunabilir. Kökün sol çocuğu (10), sağ çocuğu (15)'tir. (10)'un çocukları (5) ve E, (15)'in çocukları T ve (8)'dir. (5)'in altında F ve O, (8)'in altında G ve R vardır. E ve T derinlik 2'de, F, O, G ve R derinlik 3'tedir. En sık iki karakter, en kısa kodları almıştır.

Ağacın toplam bit maliyeti iç düğümlerin frekanslarının toplamına da eşittir: *5 + 8 + 10 + 15 + 25 = 63*. Bunun nedeni şudur: bir karakter, kökten yaprağına giden yoldaki her iç düğümün frekansına bir kez katkı yapar. Yoldaki iç düğüm sayısı karakterin derinliğidir. Böylece her karakterin frekansı derinliği kadar sayılmış olur ve toplam *Σ frekans · derinlik*'e eşit çıkar. Bu gözlem, Huffman'ın her birleştirmede neden en küçük iki frekansı seçtiğini de açıklar: her birleştirmenin maliyeti yeni iç düğümün frekansıdır ve algoritma her adımda bu maliyeti en küçük tutar.

Maliyet şöyledir. *n* yaprağın kuyruğa eklenmesi *O(n log n)* sürer. Döngü *n − 1* kez döner ve her turda iki `poll` ile bir `add` yapar; her biri *O(log n)*'dir. Toplam *O(n log n)*'dir. Burada *n* farklı karakter sayısıdır, metnin uzunluğu değildir. Frekanslar baştan sıralıysa iki sıradan kuyruk ile *O(n)* sürede çalışan bir sürüm de yazılabilir: birinci kuyruk sıralı yaprakları, ikinci kuyruk sırayla oluşan iç düğümleri tutar. Oluşan iç düğümlerin frekansları artan sırada geldiği için ikinci kuyruk kendiliğinden sıralı kalır.

Örnek 7.8

Altı karakterin frekansları a:5, b:9, c:12, d:13, e:16, f:45 olsun (toplam 100). Birleştirmeler sırasıyla şöyledir: a5 + b9 = 14; c12 + d13 = 25; (14) + e16 = 30; (25) + (30) = 55; f45 + (55) = 100. Kök 100'ün sol çocuğu f, sağ çocuğu (55)'tir. Kodlar f = 0, c = 100, d = 101, a = 1100, b = 1101, e = 111 olur. Toplam bit *45 · 1 + 12 · 3 + 13 · 3 + 16 · 3 + 5 · 4 + 9 · 4 = 224*'tür. İç düğümlerin toplamı da aynı sonucu verir: *14 + 25 + 30 + 55 + 100 = 224*. Sabit 3 bitlik kodla metin 300 bit tutardı; kazanç yaklaşık %25'tir. Bu örnekte f çok baskın olduğu için tek bitlik bir kod alır.

7.7.5 Kodları üretmek: kod ve izleme

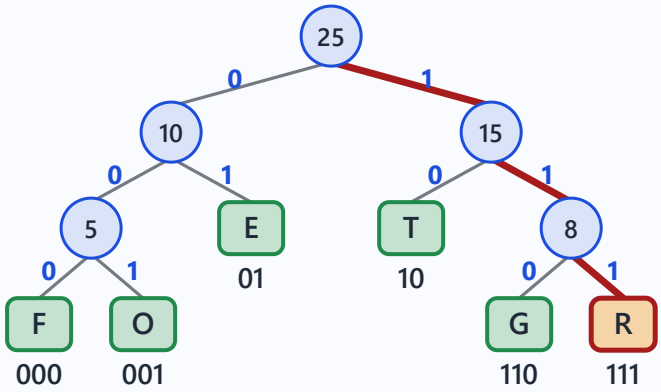
Ağaç kurulduktan sonra her karakterin kodu ağacın bir kez gezilmesiyle bulunur. Sunumdaki özyinelemeli `kodUret` metodu bunu yapar:

```
static void kodUret(Dugum d, String kod) {
  if (d.sol == null) {      // yaprak
    System.out.println(d.h + " = " + kod);
    return;
  }
  kodUret(d.sol, kod + "0"); // sol: 0
  kodUret(d.sag, kod + "1"); // sağ: 1
}
```

Metot bir düğüm ve o düğüme gelene kadar biriken bit dizisini alır. Düğüm bir yapraksa, yani sol çocuğu `null`'sa, karakter ve kodu yazdırılır. Huffman ağacı tam bir ikili ağaç olduğu için yalnız sol çocuğa bakmak yeterlidir: iç düğümlerin iki çocuğu da vardır, yaprakların hiç çocuğu yoktur. Düğüm iç düğümse metot önce sol çocuğa koda “0” ekleyerek, sonra sağ çocuğa “1” ekleyerek iner. Bu bir önce-sıra (*preorder*) gezintisidir; yapraklar soldan sağa sırayla yazdırılır. İlk çağrı `kodUret(kok, "")` biçimindedir.

Çağrı	Düğüm	`kod`	Ne olur?
1	(25), kök	""	iç düğüm; sola "0" ile inilir
2	(10)	"0"	iç düğüm; sola "00" ile inilir
3	(5)	"00"	iç düğüm; sola "000" ile inilir
4	F	"000"	yaprak: `F = 000` yazılır
5	O	"001"	yaprak: `O = 001` yazılır
6	E	"01"	yaprak: `E = 01` yazılır
7	(15)	"1"	iç düğüm; sola "10" ile inilir
8	T	"10"	yaprak: `T = 10` yazılır
9	(8)	"11"	iç düğüm; sola "110" ile inilir
10	G	"110"	yaprak: `G = 110` yazılır
11	R	"111"	yaprak: `R = 111` yazılır

Tablo 7.16: `kodUret(kok, "")` izlemesi (çağrı sırasıyla).



Şekil 7.16: Kod üretiminin sonu. Her yaprağın altında kodu yazılıdır; vurgulu yol son yazılan R = 111'in yoludur.

Metot toplam 11 kez çağrılır: 6 yaprak ve 5 iç düğüm için birer kez. Her düğüm bir kez ziyaret edildiği için gezinti $O(n)$ düğüm ziyareti yapar. Özyineleme derinliği ağacın yüksekliği kadardır; örnekte kökten en derin yaprağa 3 kenar vardır. Dizgi birleştirme `kod + "0"` her çağrıda yeni bir dizgi oluşturur ve uzunluğu kadar zaman alır. Bu maliyet de sayılırsa toplam iş, bütün kodların uzunlukları toplamıyla orantılıdır. Kodlar bir `StringBuilder` ile ya da bir karakter dizisinde tutulursa bu ek maliyet önlenir.

Sık yapılan hata

Metinde tek bir farklı karakter varsa (örneğin “AAAA”) Huffman ağacı tek düğümden oluşur: kök aynı zamanda yapraktır. `kodUret(kok, "")` bu durumda karakterin kodu olarak boş dizgiyi yazar ve metin 0 bitle kodlanmış gibi görünür. Çözücü kaç karakter olduğunu bilemez. Bu özel durumda karaktere en az 1 bitlik bir kod (örneğin “0”) verilmelidir.

7.7.6 Kodlama, çözme ve kazanç

Tablo 7.17 her karakterin kodunu ve metindeki toplam katkısını verir.

Karakter	Frekans	Kod	Uzunluk	Bit (frekans · uzunluk)
F	2	000	3	6
O	3	001	3	9
E	5	01	2	10
T	7	10	2	14
G	4	110	3	12
R	4	111	3	12
Toplam	25			**63**

Tablo 7.17: Huffman kodları ve bit hesabı.

Huffman kodu metni 63 bitle yazar; sabit 3 bitlik kod 75 bit kullanır. Kazanç *(75 − 63) / 75 = %16*'dır. Karakter başına ortalama *63 / 25 = 2,52* bit düşer. Kazanç, frekansların ne kadar dengesiz olduğuna bağlıdır. Bütün karakterler eşit sıklıkta geçseydi kazanç çok az olurdu; a–f örneğindeki gibi bir karakter çok baskınsa kazanç büyür.

Kodlama için her karakterin kodu sırayla yazılır. “FORTE” sözcüğü *000 | 001 | 111 | 10 | 01* olur, yani 13 bit. Sabit kodla aynı sözcük *5 × 3 = 15* bit tutardı. Kodlama sırasında bitler arasına ayırıcı konmaz; buradaki dikey çizgiler yalnız okumayı kolaylaştırmak içindir.

Çözme için ağaç kullanılır. “1100110” bit dizisi alınsın. Kökten başlanır. 1 ile sağa, (15)'e; 1 ile sağa, (8)'e; 0 ile sola, G'ye inilir. G yazılır ve köke dönülür. 0 ile sola, (10)'a; 1 ile sağa, E'ye inilir. E yazılır, köke dönülür. 1 ile sağa, (15)'e; 0 ile sola, T'ye inilir. T yazılır. Sonuç *110 | 01 | 10*, yani “GET”'tir. Çözme her bit için bir kenar iner; *m* bitlik bir dizi *O(m)* sürede çözülür.

Sol ve sağ çocukların yeri keyfîdir. Örneğin T, (15)'in sağ çocuğu yapılabilir. Bu durumda kodlar değişir, ama kod uzunlukları ve toplam 63 bit değişmez. Aynı nedenle eşit frekanslı düğümlerin hangisinin önce seçildiği de kod uzunluklarını değiştirebilir, ama toplam bit sayısını değiştirmez. Bir metnin birden çok en iyi önek-özgür kodu olabilir.

Sıkıştırılmış veri tek başına çözülemez; aynı ağaç ya da kod tablosu çözücünün elinde de bulunmalıdır. Bu yüzden ağaç (ya da her karakterin kod uzunluğu) sıkıştırılmış veriyle birlikte gönderilir. Kısa metinlerde bu ek bilgi kazancın tamamını yiyebilir. 25 karakterlik örnekte altı karakterin kod tablosu, kazanılan 12 bitten çok daha fazla yer tutar. Huffman kodlama bu yüzden uzun metinlerde ve büyük dosyalarda anlamlıdır.

Bir kodun ne kadar iyi olabileceğinin teorik bir alt sınırı da vardır. Karakter *c*'nin metindeki oranı p_c ise karakter başına ortalama bit sayısı $H = -\Sigma\ p_c\ log_2\ p_c$ değerinin altına inemez. Bu değere metnin **entropisi** denir. Örnek metnin entropisi yaklaşık 2,48 bittir; Huffman kodu 2,52 bit kullanır. Huffman kodunun ortalaması her zaman entropi ile entropi artı bir bit arasındadır.

> **Sık yapılan hata**
>
> Huffman ile sıkıştırılmış veriyi ağaç ya da kod tablosu olmadan çözmeye çalışmak. Bit dizisi hangi karakterin hangi kodla yazıldığını içermez. Kodlayıcı ile çözücü aynı ağacı kullanmalıdır. Sol ve sağ çocukların yeri ya da eşitliklerin bozulma biçimi farklı olan iki ağaç, aynı frekanslardan farklı kodlar üretir.

7.7.7 Huffman kodu neden en iyidir?

Bir önek-özgür kod ağacı *T*'nin maliyeti $B(T) = \Sigma\ f(c) \cdot d_T(c)$'dir; *f(c)* karakterin frekansı, $d_T(c)$ ağaçtaki derinliğidir. Huffman'ın kurduğu ağacın bu maliyeti en küçük yaptığı iki adımda gösterilir: açgözlü seçim ve optimal alt yapı.

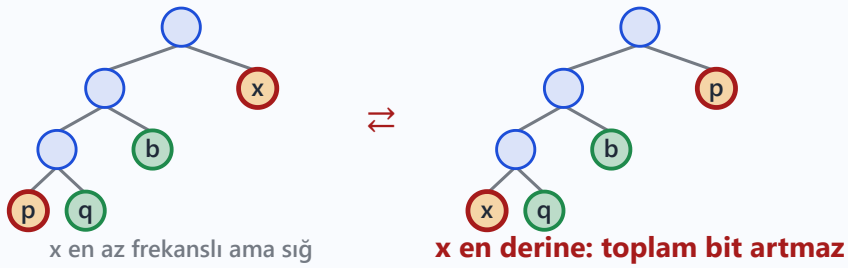
Teorem 7.5 (Huffman'da açgözlü seçim)

Frekansı en küçük iki karakter x ve y olsun. x ile y'nin en derin düzeyde kardeş yapraklar olduğu bir en iyi kod ağacı vardır.

İspat. T bir en iyi ağaç olsun. En iyi ağaç tam bir ikili ağaçtır; bu yüzden en derin düzeyde birbirinin kardeşi olan iki yaprak bulunur. Bunlar a ve b olsun. x ile a'nın yerleri değiştirilsin. Maliyet şu kadar değişir:

$$B(T') - B(T) = f(x) \cdot d(a) + f(a) \cdot d(x) - f(x) \cdot d(x) - f(a) \cdot d(a) = -(f(a) - f(x)) \cdot (d(a) - d(x))$$

x en küçük frekanslı karakter olduğu için $f(a) - f(x) \geq 0$'dır. a en derinde olduğu için $d(a) - d(x) \geq 0$'dır. Çarpım negatif değildir, önündeki eksi işaretiyle fark sıfırdan büyük olamaz: $B(T') \leq B(T)$. Aynı değiştirme y ile b için yapılır. Sonuçta x ve y en derin düzeyde kardeştir ve maliyet artmamıştır; ağaç yine en iyidir. □



Şekil 7.17: Değiştirme argümanı. Solda en düşük frekanslı x sığ bir yerdedir, p en derindedir. Sağda ikisinin yeri değiştirilmiştir; x en derine iner ve toplam bit artmaz.

Teorem 7.6 (Huffman'da optimal alt yapı)

x ve y, frekansı $f(z) = f(x) + f(y)$ olan yeni bir z karakteriyle değiştirilsin ve küçülen alfabe için bir en iyi ağaç T' olsun. T''de z yaprağının yerine, çocukları x ve y olan bir iç düğüm konursa asıl alfabe için bir en iyi ağaç T elde edilir.

İspat. x ve y, z'den bir düzey derindedir. Bu yüzden $B(T) = B(T') - f(z) \cdot d(z) + (f(x) + f(y)) \cdot (d(z) + 1) = B(T') + f(x) + f(y)$ olur. T'nin en iyi olmadığı varsayılsın: maliyeti daha küçük bir T'' vardır. Önceki teoreme göre T'''de x ve y'nin kardeş olduğu varsayılabilir. T'''de bu iki yaprak silinip ebeveynleri z yaprağı yapılırsa küçülen alfabe için maliyeti $B(T'') - f(x) - f(y) < B(T')$ olan bir ağaç çıkar. Bu, T''nin en iyi olmasıyla çelişir. □

İki teorem birlikte tümevarımla Huffman algoritmasının doğruluğunu verir. Algoritma ilk adımda en küçük iki frekansı birleştirir; açgözlü seçim teoremi bu seçimi içeren bir en iyi ağaç olduğunu söyler. Birleştirilen iki karakter tek bir z karakteri gibi davranır ve problem bir karakter küçülür. Optimal alt

yapı teoremi, küçük problemin en iyi ağacından asıl problemin en iyi ağacının elde edildiğini söyler. Böylece Huffman kodu, önek-özgür kodlar arasında *Σ frekans · derinlik* toplamını en küçük yapan koddur.

7.7.8 Maliyet ve kullanım alanları

Huffman kodlamanın bütün aşamalarının maliyeti şöyledir. m karakterlik bir metinde n farklı karakter olsun. Frekans sayımı metni bir kez okur: $O(m)$. Ağacın kurulması $O(n \log n)$, kodların üretilmesi $O(n)$ düğüm ziyareti sürer. Kodlama metni bir kez daha okur ve her karakterin kodunu tablodan alır: $O(m)$ karakter ve çıktının bit sayısı kadar iş. Genellikle n, m'den çok küçüktür (örneğin bir baytlık karakterlerde $n \le 256$); bu yüzden toplam süre pratikte metnin uzunluğuyla doğru orantılıdır.

Huffman kodlama tek başına değil, çoğu zaman başka yöntemlerle birlikte kullanılır. ZIP ve gzip dosyalarının ve PNG resimlerinin kullandığı DEFLATE biçimi, önce tekrar eden dizgileri geriye dönük işaretçilerle kısaltır, sonra çıkan sembolleri Huffman ile kodlar. JPEG resim ve MP3 ses biçimlerinde de veri önce dönüştürülür ve nicelenir; son aşamadaki entropi kodlaması Huffman kodlarıyla yapılır. Dizgi Sıkıştırma simülatörü RLE ve LZW yöntemlerini adım adım gösterir. LZW sözlük tabanlıdır: tekrar eden dizgilere kod verir. Huffman ise frekans tabanlıdır: tek tek karakterlere frekanslarına göre kod verir. İki yaklaşımın farkı simülatördeki örneklerle Huffman'ın sonuçları karşılaştırılarak görülebilir.

7.8 Diğer açgözlü algoritmalar

Açgözlü fikir birçok başka algoritmanın da temelindedir. Bunların bir kısmı kanıtlanmış olarak en iyi sonucu verir. Bir kısmı ise en iyi sonucu garanti etmez, ama sonucun en iyiden ne kadar uzak olabileceği bilinir.

Dijkstra ve Prim. İki algoritma da her adımda bir öncelikli kuyruktan en küçük anahtarlı düğümü alır ve onu kalıcı olarak seçer. Dijkstra'da anahtar, kaynaktan o düğüme bilinen en kısa uzaklıktır; Prim'de, düğümü ağaca bağlayan en ucuz kenarın ağırlığıdır. Seçilen düğüm bir daha değiştirilmez. Dijkstra negatif ağırlıklı kenar bulunmayan çizgelerde en kısa yolları doğru bulur; negatif kenar varsa kalıcı sayılan bir uzaklık sonradan kısalabilir ve açgözlü seçim yanılır. Prim ise kesim özelliğine dayandığı için ağırlıkların işaretinden bağımsız olarak doğrudur.

Çizge boyama. Komşu düğümlerin farklı renk alması koşuluyla bir çizgeyi en az renkle boyamak NP-zor bir problemdir. Açgözlü boyama düğümleri belli bir sırayla ele alır ve her düğüme komşularının kullanmadığı en küçük rengi verir. Hızlıdır, ama kullanılan renk sayısı sıraya bağlıdır. Altı düğümlü bir döngü düşünülsün: a1 – b2 – a3 – b1 – a2 – b3 – a1. Bu döngü iki renkle boyanabilir: a'lar bir renk, b'ler öbür renk. Düğümler a1, a2, a3, b1, b2, b3 sırasıyla boyanırsa gerçekten iki renk yeter. Ancak sıra a1, b1, a2, b2, a3, b3 olursa a1 ve b1 komşu olmadıkları için ikisi de 1 rengini alır. a2, b1'e komşu olduğu için 2'yi, b2, a1'e komşu olduğu için yine 2'yi alır. a3 hem b1'e (1) hem b2'ye (2) komşudur ve 3'ü alır; b3 de aynı nedenle 3'ü alır. Kötü bir sıra bir renk fazla kullandırmıştır. Açgözlü boyama en fazla “en büyük derece artı bir” renk kullanır; bu sınır her zaman sağlanır.

Kutulara yerleştirme (*bin packing*). Boyutları verilen nesneler, kapasitesi sabit olan en az sayıda kutuya yerleştirilmek istenir. Bu problem de NP-zordur. *İlk sığan* kuralı her nesneyi sığdığı ilk kutuya koyar; hiçbirine sığmıyorsa yeni bir kutu açar. *En iyi sığan* kuralı ise nesneyi, konduğunda en az boşluk bırakan kutuya koyar. Kapasite 10 ve nesneler 2, 5, 4, 7, 1, 3, 8 olsun. İlk sığan bu sırayla dört kutu açar: [2, 5, 1], [4, 3], [7], [8]. Nesneler önce büyükten küçüğe sıralanırsa (*azalan sırayla ilk sığan*) üç kutu yeter: [8, 2], [7, 3], [5, 4, 1]. Toplam boyut 30 olduğu için üçten az kutu mümkün değildir. Azalan sırayla ilk sığanın en fazla yaklaşık $11/9 \cdot OPT + 1$ kutu kullandığı kanıtlanmıştır; burada OPT en az kutu sayısıdır.

En büyük kapsama (*maximum coverage*). Bir öğeler evreni ve bu öğelerin alt kümeleri verilir; en fazla k küme seçilerek olabildiğince çok öğe kapsanmak istenir. Açgözlü algoritma her adımda henüz kapsanmamış öğelerden en çoğunu kapsayan kümeyi seçer. Bu yöntemin en iyinin en az $1 - 1/e \approx 0{,}632$'si kadar öğe kapsadığı, yani en iyinin yaklaşık %63'üne ulaştığı kanıtlanmıştır. Daha iyi bir oranı polinom sürede garanti eden bir algoritma, P = NP olmadıkça yoktur.

Bu son örnekler, açgözlü algoritmaların NP-zor problemlerdeki rolünü gösterir. En iyi çözümü bulmak çok pahalıdır, ama açgözlü bir algoritma hızlıca bir çözüm üretir ve bu çözümün kalitesi için kanıtlanmış bir alt sınır verir. Böyle algoritmalara **yaklaşım algoritması** denir; çözümün en iyiye en kötü durumdaki oranına da **yaklaşım oranı** denir.

7.9 Karşılaştırma ve sık yapılan hatalar

Tablo 7.18 notta incelenen problemleri açgözlü ölçüt, süre ve en iyilik bakımından karşılaştırır.

Problem	Açgözlü ölçüt	Zaman	En iyi mi?
Para üstü	sığan en büyük para	$O(k + adet)$	yalnız kanonik sistemlerde
Etkinlik seçimi	en erken biten	$O(n \log n)$	evet
İş sıralama	en kârlı iş, en geç boş dilim	$O(n^2)$	evet
Kesirli sırt çantası	en büyük kâr/ağırlık oranı	$O(n \log n)$	evet (0/1 sürümde hayır)
MST	kesimi geçen en hafif kenar (Kruskal, Prim, Borůvka)	$O(E \log V)$	evet
Huffman	en düşük frekanslı iki ağaç	$O(n \log n)$	evet

Tablo 7.18: Açgözlü algoritmaların karşılaştırması.

Tablonun son sütunu, açgözlü yaklaşımın en önemli dersini özetler. Beş problemde açgözlü algoritma en iyidir ve bu, birer değiştirme argümanıyla kanıtlanmıştır. Para üstünde sonuç para sistemine bağlıdır. 0/1 sırt çantasında ise oran ölçütü en iyiyi kaçırır. İş sıralamadaki $O(n^2)$, sunumdaki doğrudan dilim aramasının en kötü durumudur; ayrık kümelerle bu da $O(n \log n)$'ye iner.

Notun farklı yerlerinde ayrıntılı ele alınan hatalar şöyle toplanabilir:

- **Açgözlüyü kanıtsız doğru saymak.** *{1, 3, 4}* paralarıyla 6 tutarında açgözlü 3 para, en iyi çözüm 2 para kullanır. Her problemde açgözlü seçim özelliği gösterilmelidir; birkaç başarılı deneme kanıt değildir.
- **Yanlış ölçüt.** Etkinlik seçiminde en erken başlayan, en kısa süren ya da en az çakışan etkinliği seçmek en iyi sonucu garanti etmez. Doğru ölçüt en erken bitendir.
- **0/1 sırt çantasında oran ölçütü.** Kesirli problemde en iyi olan oran ölçütü, nesneler bölünemediğinde en iyiyi garanti etmez.
- **En erken boş dilim.** İş sıralamada işi son tarihinden önceki en erken boş dilime koymak, son tarihi yakın işlere yer bırakmaz. En geç boş dilim seçilmelidir.
- **Kesim özelliğini yanlış okumak.** Güvenli olan, bir kesimi geçen en hafif kenardır. Tüm çizgedeki en hafif kenarlar körü körüne alınırsa döngü oluşabilir; Kruskal bu yüzden döngü kuran kenarı atar.
- **Huffman'da ağaçsız çözme.** Kod tablosu ya da ağaç gönderilmezse veri çözülemez. Tek karakterli metinde kök bir yapraktır; koda en az 1 bit verilmelidir.

7.10 Özet

- **Açgözlü algoritma:** çözümü adım adım kurar, her adımda bir ölçüte göre o an en iyi görünen seçimi yapar ve kararlarını geri almaz. Tipik yapı sıralama ve tek geçiştir; süre çoğunlukla *O(n log n)*'dir.
- **Doğruluk koşulları:** açgözlü seçim özelliğine göre ilk seçimi içeren bir en iyi çözüm vardır. Optimal alt yapıya göre kalan problemin en iyi çözümü, ilk seçimle birlikte bütünün en iyi çözümünü verir. İki özellik birlikte sağlanırsa açgözlü algoritma en iyidir; sağlanmazsa yalnız bir sezgiseldir.
- **Değiştirme argümanı:** keyfî bir en iyi çözümün bir parçası açgözlü seçimle değiştirilir; çözümün uygun kaldığı ve kötüleşmediği gösterilir. Etkinlik seçimi, iş sıralama, kesirli sırt çantası, kesim özelliği ve Huffman kanıtları bu kalıbı kullanır.
- **Para üstü:** sığan en büyük para seçilir; *O(k + adet)*. Kanonik sistemlerde (ör. *{50, 25, 10, 5, 1}*) en iyidir; *{4, 3, 1}* sisteminde 6 ve 10 tutarları, *{10, 7, 1}* sisteminde 14 tutarı karşı örnektir. En iyi sonuç *enAz[t] = 1 + min enAz[t − p]* bağıntısıyla *O(k · T)* sürede bulunur.
- **Etkinlik seçimi:** etkinlikler bitişe göre sıralanır; son seçilenin bitişinden sonra başlayan her etkinlik seçilir. *O(n log n)*; sunumdaki on etkinlikten 6'sı seçilir.
- **İş sıralama:** işler kâra göre azalan sırayla, son tarihinden önceki en geç boş dilime konur. Doğrudan aramayla en kötü $O(n^2)$, ayrık kümelerle neredeyse sıralama süresi. Sunumdaki altı işte toplam kâr 140'tır.
- **Kesirli sırt çantası:** nesneler kâr/ağırlık oranına göre azalan sırayla, sığdığı kadar alınır; son nesneden bir parça alınabilir. *O(n log n)*; örnekte 240. 0/1 sürümde aynı ölçüt 160 verir, en iyisi 220'dir.

- **MST ve kesim özelliği:** bir kesimi geçen en hafif kenar bir MST'ye aittir. Kruskal (iki bileşeni ayıran kesim), Prim (ağaç ile geri kalanı) ve Borůvka (her bileşen ile geri kalanı) bu özelliği farklı kesimlerle kullanır; üçü de *O(E log V)*'dir.
- **Huffman kodlama:** en düşük frekanslı iki ağaç tekrar tekrar birleştirilir; sol kenar 0, sağ kenar 1. Kodlar önek-özgürdür ve *Σ frekans · derinlik* toplamı en küçüktür. Örnekte 63 bit (sabit kodla 75); ağaç *O(n log n)* sürede kurulur ve çözücüye gönderilmelidir.
- **Yaklaşım algoritmaları:** çizge boyama, kutulara yerleştirme ve en büyük kapsama gibi NP-zor problemlerde açgözlü algoritmalar en iyiyi garanti etmez, ama kanıtlanmış oranlarla iyi çözümler üretir.

7.11 Kavram listesi

açgözlü algoritma *greedy algorithm* · açgözlü seçim özelliği *greedy-choice property* · optimal alt yapı *optimal substructure* · değiştirme argümanı *exchange argument* · sezgisel *heuristic* · para üstü problemi *coin change problem* · kanonik para sistemi *canonical coin system* · etkinlik seçimi *activity selection* · son tarihli iş sıralama *job sequencing with deadlines* · matroid *matroid* · kesirli sırt çantası *fractional knapsack* · 0/1 sırt çantası *0/1 knapsack* · minimum kapsayan ağaç *minimum spanning tree (MST)* · kesim *cut* · kesim özelliği *cut property* · Huffman kodlama *Huffman coding* · önek-özgür kod *prefix-free code* · sıkıştırma *compression* · entropi *entropy* · öncelikli kuyruk *priority queue* · çizge boyama *graph coloring* · yaklaşım algoritması *approximation algorithm* · yaklaşım oranı *approximation ratio*

7.12 Kendini Dene

1. Para sistemi *{9, 6, 5, 1}* olsun. 11 ve 12 tutarları için açgözlü para üstü kaç para kullanır? En iyi çözümler nedir? Bu sistem kanonik midir?

   Cevap

   11 için açgözlü *9 + 1 + 1* ile 3 para kullanır; en iyisi *6 + 5* ile 2 paradır. 12 için açgözlü *9 + 1 + 1 + 1* ile 4 para kullanır; en iyisi *6 + 6* ile 2 paradır. Sistem kanonik değildir. 11, bu sistemde açgözlünün yanıldığı en küçük tutardır ve en büyük iki paranın toplamı olan 15'ten küçüktür.

2. Sekiz etkinlik *(s, f)* çiftleriyle verilmiştir ve 1'den 8'e numaralıdır: (0, 3), (1, 2), (2, 5), (3, 4), (4, 7),

(5, 6), (6, 9), (8, 10). En erken biten ölçütü hangi etkinlikleri seçer?

Cevap

Bitişe göre sıra: #2 (1, 2), #1 (0, 3), #4 (3, 4), #3 (2, 5), #6 (5, 6), #5 (4, 7), #7 (6, 9), #8 (8, 10). #2 seçilir (son bitiş 2). #1 0'da başlar, atlanır. #4 3'te başlar, seçilir (4). #3 atlanır. #6 5'te başlar, seçilir (6). #5 atlanır. #7 6'da başlar, seçilir (9). #8 8'de başlar, 8 < 9 olduğu için atlanır. Sonuç 4 etkinliktir: #2, #4, #6, #7. Bütün alt kümeler denenirse 5 etkinlikli çakışmasız bir küme bulunamaz.

3. Etkinlik seçiminde her adımda *en geç başlayan* etkinliği seçen algoritma en iyi sonucu verir mi? Gerekçelendiriniz.

Cevap

Evet. Zaman ekseni ters çevrilirse (her *[s, f)* aralığı *[−f, −s)* olur) en geç başlayan etkinlik, en erken biten etkinliğe dönüşür ve çakışma ilişkisi değişmez. En erken biten için yapılan değiştirme argümanı aynen geçerlidir. En geç başlayan *a*, bir en iyi çözümün en son etkinliğinin yerine konabilir. *a* o etkinlikten daha erken başlamaz ve önceki etkinliklerle çakışmaz.

4. Yedi iş (ad, son tarih, kâr) olarak verilmiştir: (A, 3, 35), (B, 4, 30), (C, 4, 25), (D, 2, 20), (E, 3, 15), (F, 1, 12), (G, 2, 5). Açgözlü iş sıralama dilimleri nasıl doldurur ve toplam kâr kaçtır? İşler en erken boş dilime konsaydı sonuç ne olurdu?

Cevap

Kâr sırası A, B, C, D, E, F, G'dir. A → 3. dilim; B → 4. dilim; C'nin son tarihi 4'tür, 4 ve 3 doludur, 2. dilime konur; D'nin son tarihi 2'dir, 2 doludur, 1. dilime konur. E, F ve G için uygun boş dilim kalmamıştır. Dilimler 1–4: D, C, A, B; toplam kâr *20 + 25 + 35 + 30 = 110*, en iyi sonuçtur. En erken boş dilim kuralıyla A 1. dilime, B 2. dilime, C 3. dilime gider; D, E, F ve G'ye yer kalmaz ve toplam 90'da kalır.

5. Kapasite *W = 20*, nesneler (kâr, ağırlık) olarak (48, 12), (30, 10), (28, 10), (5, 1) olsun. Kesirli sırt çantasının değeri nedir? Nesneler bölünemeseydi oran ölçütü ne bulurdu, en iyi sonuç ne olurdu?

Cevap

Oranlar 4; 3; 2,8 ve 5'tir. Oran sırası (5, 1), (48, 12), (30, 10), (28, 10)'dur. (5, 1) ve (48, 12) tümüyle alınır: 13 kg, 53 kâr. Kalan 7 kg için (30, 10)'un 7/10'u alınır: 21. Kesirli değer 74'tür. 0/1 durumunda oran ölçütü (5, 1) ve (48, 12)'yi alır; ardından gelen iki nesne de kalan 7 kg'a sığmaz ve sonuç 53'tür. En iyi 0/1 seçim (30, 10) ile (28, 10)'dur: tam 20 kg, 58 kâr.

6. Frekanslar A:1, B:1, C:2, D:3, E:5, F:8 olsun. Huffman ağacının birleştirmelerini, kod uzunluklarını

ve toplam bit sayısını bulunuz. Sabit uzunluklu kodla karşılaştırınız.

Cevap

Birleştirmeler: $1 + 1 = 2$; $2 + 2 = 4$; $3 + 4 = 7$; $5 + 7 = 12$; $8 + 12 = 20$. Her birleştirme bir önceki birleştirmenin sonucunu içerdiği için ağaç tek yöne doğru uzar. Kod uzunlukları F 1, E 2, D 3, C 4, A ve B 5 bittir. Toplam bit $8 \cdot 1 + 5 \cdot 2 + 3 \cdot 3 + 2 \cdot 4 + 1 \cdot 5 + 1 \cdot 5 = 45$'tir; iç düğümlerin toplamı da $2 + 4 + 7 + 12 + 20 = 45$ verir. Altı karakter için sabit kod 3 bit ister: $20 \times 3 = 60$ bit.

7. Kodlar F = 000, O = 001, E = 01, T = 10, G = 110, R = 111 olsun. 11001101001111 bit dizisi hangi metne çözülür? Aynı metin 3 bitlik sabit kodla kaç bit tutardı?

Cevap

Soldan okunur: 110 → G, 01 → E, 10 → T, 10 → T, 01 → E, 111 → R. Metin “GETTER”dır. Huffman ile 14 bit, sabit kodla $6 \times 3 = 18$ bit tutar. Hiçbir kod bir başkasının öneki olmadığı için her kodun bittiği yer kesin olarak bellidir.

8. Aşağıdaki iki ifade doğru mudur? (a) Ağırlıkları farklı bağlı bir çizgede en hafif kenar her MST'dedir. (b) En ağır kenar hiçbir MST'de bulunmaz.

Cevap

(a) Doğrudur. En hafif kenarın iki ucundan biri S'ye, öteki $V - S$'ye konarak bir kesim kurulur. Bu kenar kesimi geçen kenarların en hafifidir ve tek en hafif olduğu için kesim özelliğine göre her MST'dedir. (b) Yanlıştır. En ağır kenar bir köprü olabilir: silindiğinde çizge ikiye ayrılıyorsa her kapsayan ağaçta bulunmak zorundadır. Bir kenarın MST'den dışlanması için bir döngünün tek en ağır kenarı olması gerekir.

9. Kesirli sırt çantası için verilen değiştirme argümanı, 0/1 sırt çantasında hangi adımda bozulur?

Cevap

Kanıt, en iyi çözümden düşük oranlı bir nesnenin δ kilogramını çıkarıp yerine yüksek oranlı nesnenin δ kilogramını koyar. Bu değiştirme ağırlığı korur ve kârı azaltmaz. 0/1 durumunda nesneler parça parça alınamaz; ancak bütün nesneler değiştirilebilir. Ağırlıkları farklı iki nesne değiştirildiğinde ya kapasite aşılır ya da çantada değersiz bir boşluk kalır. Uygunluk koşulu ya da “daha kötü olmaz” koşulu bozulur.

10. $n = 10^6$ işin hepsinin son tarihi n olsun. Doğrudan dilim aramasıyla iş sıralama kaç dilim bakışı

yapar? Ayrık kümelerle bu sayı yaklaşık ne olur?

Cevap

i. iş en geç dilimden başlayıp geriye doğru ilk boş dilimi arar; önceki $i - 1$ iş en sondaki dilimleri doldurduğu için *i* dilime bakar. Toplam $1 + 2 + \dots + n = n(n+1)/2 \approx 5 \cdot 10^{11}$ bakıştır. Ayrık kümelerle her iş birkaç `bul` ve `birlestir` çağrısıyla yerleşir ve toplam iş neredeyse doğrusal, yani birkaç milyon adım düzeyindedir. Bu durumda süreyi $n \log_2 n \approx 2 \cdot 10^7$ karşılaştırma yapan sıralama belirler.

7.13 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 15: etkinlik seçimi, açgözlü stratejinin öğeleri ve Huffman kodlarının en iyiliği, burada özetlenen kanıtların tam biçimiyle verilir.
- J. Kleinberg ve É. Tardos, *Algorithm Design* (2006), Bölüm 4: aralık çizelgeleme, "açgözlü önde kalır" ve değiştirme argümanı kanıt teknikleri ile en az çakışan ölçütüne karşı örnekler ayrıntılı tartışılır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 5.5: Huffman sıkıştırmasının Java ile eksiksiz bir uygulaması, ağacın dosyaya yazılması ve okunması dahil.
- D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes" (1952): yöntemin ilk yayımlandığı kısa ve okunaklı özgün makale.

7.14 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 15: Greedy Algorithms; Bölüm 21: Minimum Spanning Trees).
2. D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE* 40(9), 1098–1101, 1952.
3. J. Kleinberg, É. Tardos, *Algorithm Design*, Addison-Wesley, 2006 (Bölüm 4: Greedy Algorithms).
4. D. Kozen, S. Zaks, "Optimal Bounds for the Change-Making Problem", *Theoretical Computer Science* 123(2), 377–388, 1994.
5. J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proceedings of the American Mathematical Society* 7(1), 48–50, 1956.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Kesim 4.3: Minimum Spanning Trees; Kesim 5.5: Data Compression).

KISIM III · BÖLÜM 8

Dinamik Programlama

Bazı özyinelemeli çözümler doğrudur ama dayanılmaz derecede yavaştır. Fibonacci sayılarını tanımına bakarak hesaplayan metot, fib(40) için üç yüz milyonu aşkın çağrı yapar; oysa yalnız 41 farklı soruyu cevaplaması gerekir. Aynı sorular tekrar tekrar sorulur, cevaplar ise her seferinde baştan hesaplanır. Dinamik programlama bu israfı ortadan kaldırır: her alt problem bir kez çözülür, cevabı bir tabloya yazılır, gerektiğinde tablodan okunur. Not önce yöntemin iki koşulunu, tasarım adımlarını ve iki uygulama biçimini (bellekleme ve tablolama) tanıtır. Ardından klasik problemler tek tek ele alınır: 0/1 ve sınırsız sırt çantası, alt küme toplamı, en uzun artan alt dizi, en uzun ortak alt dizgi ve en uzun ortak alt dizi. Her problem için durum ve bağıntı yazılır, tablo elle doldurulur, Java kodu satır satır izlenir ve çözüm tablodan geri okunur. Not; düzenleme uzaklığı, para üstü, en büyük toplamlı alt dizi ve matris zinciri çarpımı gibi öteki klasik örneklerle biter.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum08/src`

ÖĞRENME ÇIKTILARI

Tanıma
Bir problemde örtüşen alt problemleri ve optimal alt yapıyı fark etmek; dinamik programlamanın ne zaman uygun olduğuna, ne zaman böl ve fethetin ya da açgözlü yaklaşımın yeteceğine karar vermek.

Formülleme
Alt problemi (durumu) tanımlamak, yineleme bağıntısını ve taban durumlarını yazmak, tablonun boyutunu ve doldurma sırasını belirlemek.

Uygulama
Fibonacci, 0/1 ve sınırsız sırt çantası, alt küme toplamı, en uzun artan alt dizi, en uzun ortak alt dizgi ve en uzun ortak alt dizi tablolarını elle doldurmak ve Java ile yazmak.

Analiz
Bellekleme ile tablolamayı karşılaştırmak; $O(n \cdot W)$, $O(m \cdot n)$, $O(n^2)$ ve $O(n \log n)$ sürelerini gerekçelendirmek; sözde-polinom süreyi açıklamak; çözümün kendisini tablodan geri okumak.

8.1 Temel kavramlar

8.1.1 Dinamik programlama nedir?

Algoritma tasarımında sık kullanılan bir fikir, büyük bir problemi aynı türden daha küçük problemlere bölmektir. Böl ve fethet bu fikri parçalar birbirinden bağımsızken kullanır: birleştirme sıralaması diziyi ikiye böler, iki yarıyı ayrı ayrı sıralar ve birleştirir. İki yarının ortak bir işi yoktur. Bazı problemlerde ise parçalar birbirine karışır. Büyük problemin alt problemleri, kendi alt problemlerini paylaşır. Böyle bir problem düz özyinelemeyle çözülürse aynı küçük soru defalarca sorulur ve her seferinde baştan cevaplanır.

Dinamik programlama (*dynamic programming*, kısaca DP) bu tekrarı ortadan kaldırır. Her alt problemin çözümü bir tabloya yazılır. Aynı alt problemle yeniden karşılaşıldığında hesap yapılmaz, cevap tablodan okunur. Yöntem özellikle bir şeyin en büyüğünü ya da en küçüğünü arayan **optimizasyon problemlerinde** kullanılır: en büyük değer, en kısa yol, en az işlem. Ama aynı fikir sayma ve karar problemlerinde de işe yarar: “kaç farklı yol var?”, “böyle bir alt küme var mı?”

Tanım 8.1

Dinamik programlama, bir problemi daha küçük **alt problemlere** (*subproblem*) ayıran, her alt problemi yalnız bir kez çözen ve çözümleri bir tabloda saklayarak yeniden kullanan algoritma tasarım yöntemidir. Kısaca: DP = özyineleme + hatırlama.

Bu tanımın önemli bir sonucu vardır. Her dinamik programlama çözümünün arkasında bir özyinelemeli çözüm durur. DP problemi farklı bir yolla çözmez; aynı özyinelemeyi gereksiz tekrar olmadan çalıştırır. Bu yüzden DP tasarımının ilk ve en zor adımı doğru özyinelemeyi bulmaktır. Tablo, bu özyinelemenin değerlerini saklayan bir araçtır.

Yöntemin adı 1950'lerde Richard Bellman'dan gelir. Adın içindeki “programlama” sözcüğü kod yazmak anlamına gelmez. O yıllarda bu sözcük planlama ve çizelgeleme anlamında kullanılırdı; doğrusal programlama adı da aynı kökten gelir. Dinamik programlamadaki programlama, kararları bir tablo ile adım adım planlamaktır.

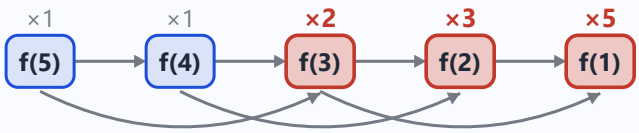
8.1.2 Örtüşen alt problemler

Bir problemin dinamik programlamadan yarar görmesi için iki özellik gerekir. Birincisi alt problemlerin örtüşmesidir.

Tanım 8.2

Bir problemin özyinelemeli çözümü aynı alt problemi birden çok kez çağırıyorsa problemin **örtüşen alt problemleri** (*overlapping subproblems*) vardır. Başka bir deyişle, farklı alt problemlerin sayısı küçüktür, ama özyinelemenin bu alt problemleri çağırma sayısı büyüktür.

En bilinen örnek Fibonacci sayılarıdır. *F(n) = F(n − 1) + F(n − 2)* tanımı doğrudan koda çevrilirse fib(5) hesaplanırken fib(4) ve fib(3) çağrılır. fib(4) de kendi içinde fib(3)'ü çağırır. Böylece fib(3) iki kez hesaplanır. Aynı şey daha küçük argümanlarda katlanarak büyür: fib(2) üç kez, fib(1) beş kez hesaplanır. Şekil 8.1 bu tekrarları gösterir.



Şekil 8.1: fib(5) hesaplanırken alt problemlerin kaç kez çağrıldığı. Oklar "şunu çağırır" anlamındadır; kırmızı kutular birden çok kez çözülen alt problemlerdir.

Burada iki sayı arasındaki fark DP'nin kazancını belirler. fib(5) için yalnız altı farklı alt problem vardır: fib(0)'dan fib(5)'e kadar. Ama düz özyineleme toplam 15 çağrı yapar. n büyüdükçe farklı alt problem sayısı *n + 1* olarak kalır, çağrı sayısı ise üstel büyür. Saklanacak bir tekrar olduğu sürece tablo bu farkı kapatır.

Alt problemler örtüşmüyorsa tablo işe yaramaz. Birleştirme sıralamasında dizinin iki yarısı ayrık parçalardır. Sol yarıyı sıralamak için yapılan hiçbir iş sağ yarıda yeniden yapılmaz. Bir alt dizi ancak bir kez sıralanır; dolayısıyla tabloya yazılacak bir cevabın ikinci kez okunması da hiç gerçekleşmez. Böyle problemler böl ve fethetin alanıdır.

Sık yapılan hata

Her özyinelemeli çözüme tablo eklemenin hızlandırdığı sanılır. Tablo ancak aynı alt problem tekrar çağrılıyorsa kazanç sağlar. İkili aramada ya da birleştirme sıralamasında her alt problem bir kez çözülür; tablo yalnız bellek harcar.

8.1.3 Optimal alt yapı

İkinci özellik, büyük problemin en iyi çözümünün küçük problemlerin en iyi çözümlerinden kurulabilmesidir. Bu özellik olmadan tabloya yazılan alt problem cevapları büyük problemin cevabını kurmaya yetmez.

Tanım 8.3

Bir problemin optimal çözümü, alt problemlerinin optimal çözümlerinden kurulabiliyorsa problemin **optimal alt yapısı** (*optimal substructure*) vardır. Bellman bu özelliği **optimallik ilkesi** olarak dile getirmiştir: optimal bir çözümün içerdiği her alt çözüm, kendi alt problemi için de optimaldir.

Bu özellik bir örnekle açıklık kazanır. Bir dizide **en büyük toplamlı alt dizi** aranıyor olsun. Burada alt dizi, bitişik öğelerden oluşan bir parçadır. $V(i)$, tam olarak i. öğede biten en büyük toplam olsun. i'de biten en iyi parça ya yalnız $a[i]$'den oluşur ya da $i-1$'de biten bir parçanın sonuna $a[i]$ eklenerek elde edilir. İkinci durumda $i-1$'de biten parçanın da en iyisi olması gerekir. Daha iyisi varsa onu kullanmak toplamı büyütürdü. Böylece bağıntı kendiliğinden çıkar:

$$V(i) = \max(\, a[i],\ \ V(i-1) + a[i] \,)$$

Bu tür bağıntılara **Bellman denklemi** de denir: bir durumun değeri, bir önceki alt problemin optimal değerinden hesaplanır. Cevap, bütün $V(i)$ değerlerinin en büyüğüdür. Bu yöntemin adı Kadane algoritmasıdır; $O(n)$ sürede çalışır.

Optimal alt yapı her problemde yoktur. Bir çizgede iki düğüm arasındaki *en uzun basit yol* (düğüm tekrarı olmayan yol) aransın. u'dan w'ye en uzun basit yol bir v düğümünden geçiyorsa, yolun u–v parçasının u'dan v'ye en uzun basit yol olması gerekmez. Daha uzun bir u–v yolu, yolun ikinci yarısındaki düğümleri kullanabilir. İki “en iyi” parça birleştirildiğinde ortaya basit olmayan bir yol çıkar. Alt problemler birbirinden bağımsız çözülemez; bu yüzden bu problem için bilinen verimli bir DP yoktur. En kısa yol probleminde ise durum farklıdır: en kısa yolun her parçası da en kısa yoldur. Bellman–Ford ve Floyd–Warshall bu özelliğe dayanır.

Sık yapılan hata

Örtüşen alt problemleri görüp hemen tablo kurmak yetmez. Önce alt problemlerin optimal çözümlerinin birleştirilmesiyle büyük problemin optimal çözümünün elde edildiği gösterilmelidir. Optimal alt yapı yoksa tablo hızlı ama yanlış bir cevap üretir.

8.1.4 Tasarım adımları

Dinamik programlama çözümleri ilk bakışta birbirinden çok farklı görünür: biri iki boyutlu bir tablo doldurur, öteki tek bir diziyle yetinir. Ama hepsi aynı adımlarla tasarlanır. Bu adımlar sırayla izlenirse yeni bir problemin çözümü de çoğu zaman kendiliğinden ortaya çıkar.

1. **Optimal çözümün yapısı incelenir.** Optimal çözümde son karar nedir? Bu karar verildikten sonra geriye hangi küçük problem kalır?

2. **Durum ve bağıntı yazılır.** Alt problem, birkaç tam sayıyla tanımlanan bir **durum** olarak ifade edilir; ör. "ilk *i* eşya ve *c* kapasite". Durumun değeri daha küçük durumlar cinsinden yazılır (yineleme bağıntısı). Bağıntının dayandığı en küçük durumların değerleri ayrıca verilir (taban durumları).
3. **Hesaplama sırası belirlenir.** Bir hücre hesaplanırken ihtiyaç duyduğu hücreler hazır olmalıdır.
4. **Tablonun boyutu belirlenir.** Kaç boyut gerekir, her boyut kaç değer alır? Örneğin sırt çantasında tablo $(n + 1) \times (W + 1)$ olur.
5. **Tablo doldurulur.** Aşağıdan yukarı döngülerle (tablolama) ya da yukarıdan aşağı özyinelemeyle (bellekleme).
6. **Çözüm geri okunur.** Çoğu problemde yalnız optimal değer değil, o değeri veren seçimler de istenir. Seçimler tablodan geriye doğru okunarak bulunur.

Bu adımların süreyle ilgili basit bir sonucu vardır. Her durum bir kez hesaplanır. Bir durumu hesaplamak için birkaç seçenek denenir. O hâlde toplam süre yaklaşık olarak durum sayısıyla bir durumun hesaplanma süresinin çarpımıdır:

> *toplam süre ≈ (durum sayısı) × (bir durumun hesaplanma süresi)*

Notta geçen bütün karmaşıklıklar bu kuraldan çıkar. Sırt çantasında $(n + 1)(W + 1)$ durum vardır ve her biri iki seçenekle $O(1)$'de hesaplanır: $O(n \cdot W)$. En uzun artan alt dizide n durum vardır ama her biri önceki bütün öğelere bakar: $O(n^2)$. Matris zinciri çarpımında $O(n^2)$ durumun her biri $O(n)$ bölme noktası dener: $O(n^3)$.

8.1.5 Bellekleme ve tablolama

Bir DP bağıntısı iki biçimde çalıştırılabilir. İkisi de her alt problemi bir kez çözer; fark, alt problemlerin hangi sırayla ve hangi araçla çözüldüğündedir.

Bellekleme (*memoization*) **yukarıdan aşağı** (*top-down*) çalışır. Özyinelemeli çözüm olduğu gibi yazılır; yalnız sonuçlar bir önbellekte saklanır. Önbellek bir dizi ya da `HashMap` olabilir. Her çağrının başında tabloya bakılır: cevap oradaysa hemen döndürülür, yoksa hesaplanır, tabloya yazılır ve öyle döndürülür. Hesap büyük problemden başlar ve gerekli olan küçük problemlere iner. Bu yüzden yalnız gerçekten ihtiyaç duyulan alt problemler çözülür. Bedeli özyinelemedir: her çağrının bir maliyeti vardır ve derin özyineleme yığın taşmasına (`StackOverflowError`) yol açabilir.

Tablolama (*tabulation*) **aşağıdan yukarı** (*bottom-up*) çalışır. En küçük alt problemden başlanır ve tablo döngülerle sırayla doldurulur. Özyineleme ve çağrı maliyeti yoktur; bu yüzden tablolama pratikte genellikle daha hızlıdır. Bunun karşılığında doldurma sırası önceden doğru belirlenmelidir: bir hücreye gelindiğinde bağlı olduğu hücreler hazır olmalıdır. Tablolamanın bir üstünlüğü daha vardır. Bağıntı yalnız önceki satırı kullanıyorsa bütün tablo yerine tek satır saklanabilir. Sırt çantasında bellek bu yolla $O(n \cdot W)$'den $O(W)$'ye iner.



Şekil 8.2: Solda bellekleme: fib(5) çağrısı yukarıdan aşağı iner, değerler dönüşte yazılır. Sağda tablolama: f0, f1, … soldan sağa sırayla dolar.

	Bellekleme (yukarıdan aşağı)	**Tablolama (aşağıdan yukarı)**
Yazım	özyinelemeli çözüm + önbellek	iç içe döngüler
Çözülen alt problemler	yalnız gerekenler	tablodaki bütün hücreler
Sıra	özyineleme kendiliğinden belirler	önceden doğru seçilmelidir
Ek maliyet	çağrı maliyeti, yığın taşması riski	yok
Bellek azaltma	zor	önceki satır yetiyorsa kolay
Asimptotik süre	aynı: her alt problem bir kez çözülür	

Tablo 8.1: Bellekleme ve tablolamanın karşılaştırması.

Hangisinin seçileceği probleme bağlıdır. Durum uzayı büyük ama gerçekte kullanılan durumlar azsa bellekleme daha az iş yapar. 0/1 sırt çantası bunun bir örneğidir: bellekleme sürümü, dört eşyalı ve 6 kapasiteli örnekte 28 durumdan yalnız 12'sini hesaplar. Bütün tablo zaten gerekiyorsa ve doldurma sırası belliyse tablolama hem daha hızlı hem daha güvenlidir.

Sık yapılan hata

"Bellekleme her zaman tablolamadan hızlıdır" ya da tersi söylenir. İkisinin asimptotik süresi aynıdır. Tablolama çağrı yapmadığı için pratikte çoğu zaman daha hızlıdır; bellekleme ise gereksiz durumları atladığında öne geçebilir.

8.1.6 DP, böl ve fethet ve açgözlü yaklaşım

Dinamik programlama, iki tanıdık tasarım yöntemiyle birlikte düşünüldüğünde daha iyi anlaşılır. Üçü de problemi küçük parçalara ayırır; farkları, parçalar arasındaki ilişkide ve seçimin nasıl yapıldığındadır.

	Böl ve fethet	**Dinamik programlama**	**Açgözlü (*greedy*)**
Alt problemler	bağımsız	örtüşen	her adımda tek seçim
Seçim	yok; böl ve birleştir	bütün seçenekler denenir, en iyisi tabloya	o anki en iyi seçilir, geri dönülmez
Optimallik	—	optimal alt yapı varsa garanti	yalnız açgözlü seçim özelliği varsa
Örnek	birleştirme sıralaması, ikili arama	0/1 sırt çantası, LCS, Bellman–Ford	kesirli sırt çantası, Dijkstra, Kruskal
Maliyet	yineleme bağıntısı, ana teorem	durum sayısı × durum başına iş	genelde $O(n \log n)$: sıralama + tek geçiş
Bellek	özyineleme yığını	tablo (çoğu zaman $O(n \cdot W)$ ya da $O(m \cdot n)$)	genellikle $O(1)$ ek bellek

Tablo 8.2: Böl ve fethet, dinamik programlama ve açgözlü yaklaşım.

Açgözlü yaklaşım, her adımda yalnız tek bir seçeneğe bakan bir dinamik programlama gibi düşünülebilir. DP bir durumda bütün seçenekleri dener ve en iyisini tabloya yazar. Açgözlü algoritma ise o anda en iyi görünen seçeneği alır ve öteki seçenekleri hiç incelemez. Bu seçimin her zaman bir optimal çözüme götürdüğü kanıtlanabiliyorsa (açgözlü seçim özelliği) tabloya gerek kalmaz ve algoritma çok daha hızlı çalışır. Kanıtlanamıyorsa açgözlü algoritma yanlış cevap verebilir. 0/1 sırt çantası bunun tipik örneğidir.

8.1.7 Uygulama alanları

Dinamik programlama kuramsal bir araç olmanın ötesinde, her gün kullanılan birçok yazılımın içinde çalışır.

- **En kısa yol.** Haritalar ve yönlendirme protokolleri: Bellman–Ford, "en çok k kenarlı en kısa yol" durumuyla; Floyd–Warshall, "yalnız ilk k düğümü ara düğüm olarak kullanan en kısa yol" durumuyla çalışır.

- **Dizgi karşılaştırma.** `diff` ve `git` iki dosyanın farkını en uzun ortak alt diziyle bulur; yazım denetleyicileri düzenleme uzaklığıyla öneri üretir.
- **Biyoinformatik.** DNA ve protein dizilerinin hizalanması (Needleman–Wunsch, Smith–Waterman) ve RNA'nın katlanma yapısının belirlenmesi.
- **Kaynak dağıtımı.** Sırt çantası, bütçe ve yatırım planlama, üretim hattı çizelgeleme.
- **Derleyiciler ve sayısal hesap.** Matris zinciri çarpımının sırası, optimal ikili arama ağacı, bağlamdan bağımsız dilbilgileri için CYK ayrıştırması.
- **Yapay zekâ.** Oyun oynayan programlar, konuşma tanıma ve makine çevirisindeki Viterbi algoritması, pekiştirmeli öğrenmedeki Bellman denklemi.

8.2 Fibonacci: bellekleme ve tablolama

Fibonacci dizisinin özyinelemeli tanımı ve bu tanımı doğrudan koda çeviren metot iyi bilinir. Burada amaç özyinelemeyi yeniden anlatmak değildir. Fibonacci, dinamik programlamanın iki biçimini en küçük ölçekte göstermek için kullanılır: aynı bağıntı önce bellekleme, sonra tablolama ile çalıştırılır ve maliyetleri karşılaştırılır.

8.2.1 Düz özyinelemenin bedeli

Fibonacci dizisi 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … biçiminde ilerler. Taban durumları $F(0) = 0$ ve $F(1) = 1$'dir; $n \geq 2$ için $F(n) = F(n-1) + F(n-2)$'dir. Tanımın birebir kopyası olan metot doğrudur ama çok yavaştır.

```
static long fib(int n) {
  if (n <= 1)
    return n;          // taban durumları
  return fib(n - 1) + fib(n - 2);
}
```

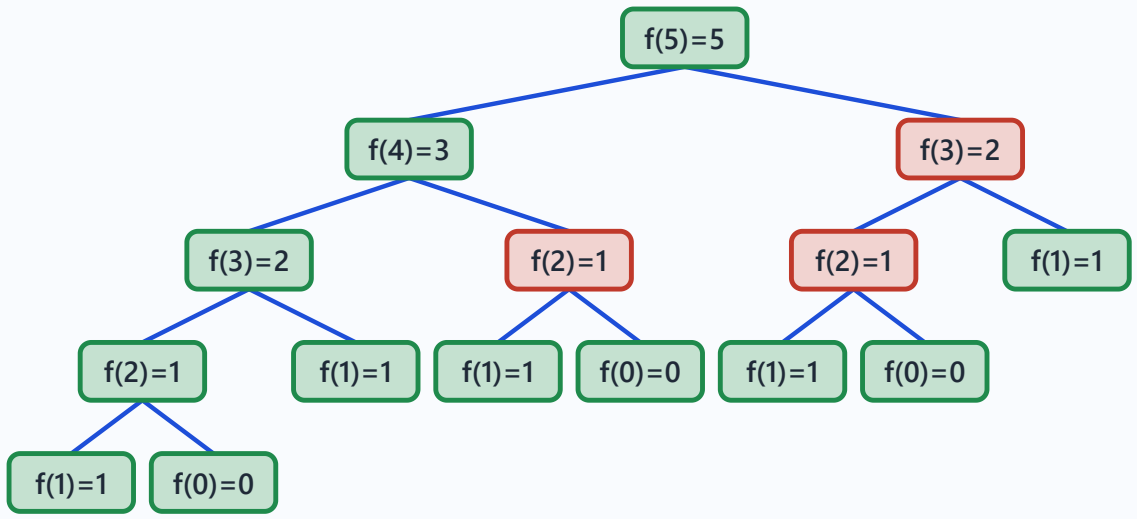
Metodun yaptığı çağrı sayısı $T(n)$ olsun. Her çağrı kendisi için 1 sayılır ve iki alt çağrı yapar: $T(n) = T(n-1) + T(n-2) + 1$, $T(0) = T(1) = 1$. Bu bağıntının çözümü $T(n) = 2 \cdot F(n+1) - 1$'dir; tümevarımla kolayca doğrulanır. Fibonacci sayıları altın oranın kuvvetleri gibi büyüdüğünden çağrı sayısı $\Theta(\varphi^n)$'dir; burada $\varphi \approx 1{,}618$'dir. Tablo 8.3 farkın ne kadar hızlı açıldığını gösterir.

n	F(n)	Düz özyineleme: 2 · F(n + 1) − 1	Bellekleme: 2n − 1
5	5	15	9
10	55	177	19
20	6.765	21.891	39
30	832.040	2.692.537	59

n	F(n)	Düz özyineleme: 2 · F(n + 1) − 1	Bellekleme: 2n − 1
40	102.334.155	331.160.281	79

Tablo 8.3: Düz özyineleme ve bellekleme ile fib(n) hesaplanırken yapılan çağrı sayıları (Java ile sayıldı).

fib(40) üç yüz otuz milyonu aşkın çağrı yapar. fib(50) için bu sayı yaklaşık $4 \cdot 10^{10}$'a çıkar; sıradan bir bilgisayarda dakikalar sürer. Oysa fib(n) hesaplanırken yalnız $n + 1$ farklı alt problem vardır: fib(0), fib(1), …, fib(n). Çağrıların ezici çoğunluğu daha önce cevaplanmış bir soruyu yeniden sorar.



Şekil 8.3: fib(5) özyineleme ağacı, bütün çağrılar bittikten sonra. 15 çağrı yapılır; kırmızı kutular daha önce çözülmüş bir alt problemin yeniden hesaplandığı yerlerdir.

Şekil 8.3'teki ağaçta fib(3) iki kez, fib(2) üç kez, fib(1) beş kez, fib(0) üç kez görünür. Bu sayıların kendileri de Fibonacci sayılarıdır: fib(n) ağacında fib(k) ($k \geq 1$) tam $F(n - k + 1)$ kez çağrılır. Tekrarın büyüklüğü, hesaplanan sayının büyüklüğüyle aynı hızda artar. Dinamik programlamanın özü bu gözlemdir: farklı alt problem sayısı küçük, ama özyinelemenin onları çağırma sayısı büyükse sonuçları saklamak büyük kazanç sağlar.

8.2.2 Bellekleme

Bellekleme, metodun gövdesine iki satır ekler. Hesaplanan her değer `bellek` dizisine yazılır. Bir çağrı, hesap yapmadan önce diziye bakar; değer oradaysa hemen döner.

```
static long[] bellek = new long[50];

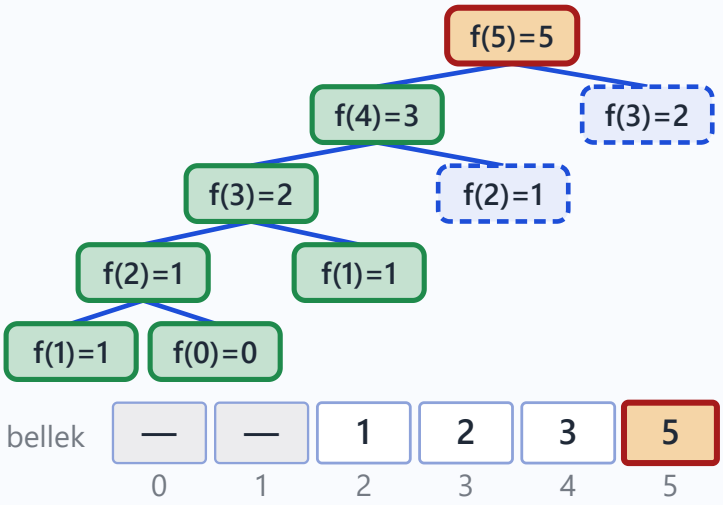
static long fib(int n) {
  if (n <= 1) return n;          // taban
  if (bellek[n] != 0) return bellek[n];
  long a = fib(n - 1);
  long b = fib(n - 2);
  bellek[n] = a + b;
  return bellek[n];
}
```

`bellek` dizisi `static` tanımlandığı için bütün çağrılar aynı diziyi paylaşır. Java'da yeni bir `long` dizisinin bütün öğeleri 0'dır. Kod 0'ı "henüz hesaplanmadı" anlamında kullanır. Bu seçim güvenlidir, çünkü $n \geq 2$ için $F(n)$ hiçbir zaman 0 değildir; taban durumları ise diziye hiç yazılmaz. Dizinin uzunluğu 50 olduğu için kod en çok fib(49)'u hesaplayabilir. Dönüş türünün `long` seçilmesi de bilinçlidir: $F(47) = 2.971.215.073$, `int`'in en büyük değeri olan $2^{31} - 1 = 2.147.483.647$'yi aşar.

Satırlar şöyle çalışır. İlk `if` taban durumlarını karşılar. İkinci `if` bellekleme denetimidir: `bellek[n]` sıfırdan farklıysa değer daha önce hesaplanmıştır ve olduğu gibi döner. Değer yoksa iki alt çağrı yapılır, toplam `bellek[n]`'e yazılır ve döndürülür. Tablo 8.4, fib(5) çağrısının izlemesini verir.

Adım	Çağrı	Olay	bellek[2..5]
1	fib(5)	tabloda yok; önce fib(4) çağrılır	— — — —
2	fib(4)	tabloda yok; önce fib(3) çağrılır	— — — —
3	fib(3)	tabloda yok; önce fib(2) çağrılır	— — — —
4	fib(2)	tabloda yok; önce fib(1) çağrılır	— — — —
5	fib(1)	taban durumu, 1 döner	— — — —
6	fib(0)	taban durumu, 0 döner; `bellek[2] = 1 + 0 = 1`	1 — — —
7	fib(1)	fib(3) içinden; taban durumu, 1 döner; `bellek[3] = 1 + 1 = 2`	1 2 — —
8	fib(2)	fib(4) içinden; `bellek[2] = 1` hazır, yeniden hesaplanmaz; `bellek[4] = 2 + 1 = 3`	1 2 3 —
9	fib(3)	fib(5) içinden; `bellek[3] = 2` hazır; `bellek[5] = 3 + 2 = 5`	1 2 3 5

Tablo 8.4: Bellekleme ile fib(5) izlemesi. "Bellek" sütunu adımdan sonraki `bellek[2..5]` değerlerini verir (— hesaplanmadı).



Şekil 8.4: Bellekleme ile fib(5): gerçekte açılan çağrılar ve dolmuş bellek dizisi. Kesikli kutular, tablodan hazır okunan iki çağrıdır.

Şekil 8.4'teki ağaç artık bir zincirdir. Sol kol fib(5)'ten fib(1)'e iner. Her düzeyde sağ çocuk ya bir taban durumudur ya da tablodan hazır okunur. Toplam 9 çağrı yapılır. Genel durumda fib(2)'den fib(n)'ye kadar her değer bir kez hesaplanır. Çağrılar şöyle sayılır: fib(n) ve fib(n − 1) birer kez çağrılır. $1 \le k \le n-2$ için fib(k) iki kez çağrılır; bir kez açılır, bir kez de tablodan ya da taban durumu olarak hemen cevaplanır. fib(0) ise bir kez çağrılır. Toplam $2 + 2(n-2) + 1 = 2n-1$'dir. Üstel süre doğrusala inmiştir. Ek bellek $O(n)$'dir: `bellek` dizisi ve en derin yerinde n çerçeveli çağrı yığını.

Örnek 8.1

fib(6) düz özyinelemeyle ve bellekleme ile kaç çağrı yapar?

Düz özyineleme: $2 \cdot F(7) - 1 = 2 \cdot 13 - 1 = 25$ çağrı. Ağaçta fib(2) beş, fib(1) sekiz kez görünür. Bellekleme: fib(6) ve fib(5) birer kez çağrılır. fib(4), fib(3), fib(2) ve fib(1) ikişer kez çağrılır; ikinci çağrılar tablodan ya da taban durumundan hemen döner. fib(0) bir kez çağrılır. Toplam $2 + 8 + 1 = 11 = 2 \cdot 6 - 1$ çağrı.

Sık yapılan hata

Önbellek metodun içinde yerel değişken olarak oluşturulur. `long[] bellek = new long[50];` satırı her çağrıda yeni ve boş bir dizi açar. Hiçbir değer paylaşılmaz ve süre yine üstel kalır. Önbellek bütün çağrıların gördüğü bir yerde olmalıdır: `static` alan, nesne alanı ya da parametre olarak geçirilen dizi.

8.2.3 Tablolama

Aynı bağıntı döngüyle de çalıştırılabilir. fib(i), yalnız fib(i − 1) ve fib(i − 2)'ye bağlıdır. Dizi soldan sağa doldurulursa her hücreye gelindiğinde ihtiyaç duyduğu iki hücre hazırdır.

```
static int fibonacci(int n) {
  if (n <= 1) return n;   // n = 0 ve n = 1
  int[] fib = new int[n + 1];
  fib[0] = 0;   // taban durumları
  fib[1] = 1;
  for (int i = 2; i <= n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
  return fib[n];
}
```

Metot önce *n + 1* öğeli bir dizi oluşturur; Java'da bütün öğeler 0 ile başlar. Taban durumları yazıldıktan sonra döngü *i = 2*'den *n*'ye kadar her hücreyi bir toplama ile doldurur. fibonacci(9) için hücreler sırasıyla 1, 2, 3, 5, 8, 13, 21 ve 34 olur. Son hücre `fib[9] = 34` döndürülür. Her hücre bir kez hesaplanır: süre *O(n)*, bellek *O(n)*'dir.



Şekil 8.5: fibonacci(9) bitince `fib` dizisi. Hücreler soldan sağa birer toplama ile doldu.

İlk satır küçük ama gerekli bir ayrıntıdır. Bu satır olmasaydı *n = 0* için dizi tek öğeli olurdu ve `fib[1] = 1` satırı `ArrayIndexOutOfBoundsException` fırlatırdı. `if (n <= 1) return n;` iki taban durumunu dizi kurulmadan karşılar; böylece kod bütün *n ≥ 0* için çalışır. Ayrıca dizi `int` olduğu için fibonacci(47) taşar ve negatif bir sayı döner; Java'da bu sonuç −1.323.752.223'tür.

8.2.4 O(1) bellek ve karşılaştırma

Tablolama sürümünde `fib[i]` hesaplanırken dizinin yalnız son iki hücresi okunur. Daha eski hücrelere bir daha hiç bakılmaz. O hâlde bütün diziyi saklamaya gerek yoktur; son iki değeri tutan iki değişken yeter.

```
static long fib(int n) {
  if (n <= 1) return n;
  long once = 0, simdi = 1;
  for (int i = 2; i <= n; i++) {
    long yeni = once + simdi;
    once = simdi;
    simdi = yeni;
  }
  return simdi;
}
```

Döngünün her turunda `once` ve `simdi` değişkenleri *F(i − 2)* ve *F(i − 1)*'dir. Yeni değer hesaplanır, değişkenler bir adım kaydırılır. Döngü bitince `simdi`, *F(n)*'dir. Süre yine *O(n)*, ek bellek ise *O(1)*'dir. `long` türü $F(92) \approx 7{,}5 \cdot 10^{18}$'e kadar yeter; *F(93)* taşar.

Yöntem	Zaman	Ek bellek
Düz özyineleme	$\Theta(\varphi^n)$	$O(n)$ yığın
Bellekleme	$O(n)$	$O(n)$ tablo + yığın
Tablolama	$O(n)$	$O(n)$ tablo
İki değişken	$O(n)$	$O(1)$

Tablo 8.5: Fibonacci hesaplamanın dört yolu.

Bellek azaltma fikri genel bir kurala dayanır. Bir tablo hücresi yalnız sınırlı sayıda önceki hücreye bağlıysa, artık kullanılmayacak hücreler atılabilir. Sırt çantası tablosunda her satır yalnız bir önceki satıra bağlıdır; bu yüzden iki boyutlu tablo tek bir satıra indirilebilir. Bu kazancın bir bedeli vardır: atılan hücreler, çözümün geri okunması için gerekebilir. Fibonacci'de geri okunacak bir seçim olmadığı için kayıp yoktur.

8.3 0/1 sırt çantası

8.3.1 Problem

Bir yolcu, taşıyabileceği ağırlık sınırlı bir çantaya eşya yerleştirecektir. Her eşyanın bir ağırlığı ve bir değeri vardır. Amaç, ağırlık sınırını aşmadan çantadaki toplam değeri en büyük yapmaktır. Eşyalar bölünemez: bir eşya ya bütünüyle çantaya girer ya da hiç girmez. Problemin adındaki 0/1 bu iki seçeneği anlatır.

Tanım 8.4

0/1 sırt çantası problemi (*0/1 knapsack*): n eşya ve bir W kapasitesi verilir. i. eşyanın değeri v_i, ağırlığı w_i'dir. Her eşya için $x_i \in \{0, 1\}$ seçilerek $\Sigma\, w_i\, x_i \le W$ koşulu altında $\Sigma\, v_i\, x_i$ en büyük yapılır.



Şekil 8.6: 15 kg kapasiteli bir çanta ve beş eşya: 4 ₺ ve 12 kg, 2 ₺ ve 2 kg, 2 ₺ ve 1 kg, 1 ₺ ve 1 kg, 10 ₺ ve 4 kg.

Örnek 8.2

Şekil 8.6'daki çantaya hangi eşyalar konmalıdır?

12 kg'lık eşya dışındaki dört eşyanın toplam ağırlığı $2 + 1 + 1 + 4 = 8$ kg'dır; hepsi sığar ve $2 + 2 + 1 + 10 = 15$ ₺ eder. 12 kg'lık eşya seçilirse geriye 3 kg kalır. Bu 3 kg'a en iyi 2 kg'lık ve 1 kg'lık 2 ₺'lik eşyalar girer; toplam $4 + 2 + 2 = 8$ ₺ olur. 12 kg'lık eşyayı alan hiçbir seçim 15 ₺'ye ulaşamaz. Cevap 15 ₺'dir. Dikkat çekici olan, en pahalı eşyanın optimal çözümde yer almamasıdır.

8.3.2 Kaba kuvvet ve açgözlü seçim neden yetmez?

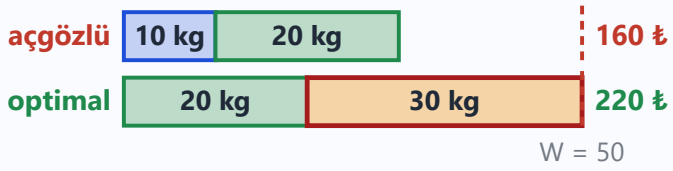
Problemin en doğrudan çözümü bütün seçimleri denemektir. Her eşya için iki seçenek vardır; n eşya için 2^n alt küme oluşur. $n = 50$ için bu sayı $2^{50} \approx 1{,}13 \cdot 10^{15}$'tir. Saniyede bir milyar alt küme denense bile on üç günden uzun sürer. Kaba kuvvet ancak çok küçük girdilerde kullanılabilir.

İkinci akla gelen yol açgözlü seçimdir. Eşyalar değer/ağırlık oranına göre sıralanır ve oranı en yüksek olandan başlayarak sığan her eşya alınır. Bu yöntem bazen doğru cevabı verir, ama her zaman vermez.

Örnek 8.3

$W = 50$ kg; eşyalar (60 ₺, 10 kg), (100 ₺, 20 kg) ve (120 ₺, 30 kg). Açgözlü seçim ne verir, optimal çözüm nedir?

Oranlar sırasıyla 6, 5 ve 4 ₺/kg'dır. Açgözlü algoritma önce birinci eşyayı alır (10 kg dolu), sonra ikinciyi alır (30 kg dolu). Kalan 20 kg'a 30 kg'lık üçüncü eşya sığmaz. Sonuç 160 ₺'dir. Oysa ikinci ve üçüncü eşya birlikte tam 50 kg eder ve 220 ₺ getirir. Açgözlü seçim en verimli eşyayı almakla doğru bir yerel karar verir, ama bu karar çantada kullanılamayan 20 kg'lık bir boşluk bırakır.



Şekil 8.7: Açgözlü seçim 160 ₺'de kalır; optimal seçim 220 ₺'dir.

Açgözlü seçim, eşyalar bölünebiliyorsa optimaldir. Bu probleme **kesirli sırt çantası** denir. Örnekte üçüncü eşyanın üçte ikisi alınabilseydi açgözlü algoritma $60 + 100 + 80 = 240$ ₺ ile optimal sonuca ulaşırdı. 0/1 durumunda ise bir eşyayı almak, sonraki seçenekleri değiştirir. Hangi seçimin daha iyi olduğunu bilmenin tek yolu iki seçeneği de denemektir. Bu deneme her eşyada iki kola ayrılan bir özyineleme demektir; kollar ise aynı alt problemlerde buluşur. Örtüşen alt problemler ve optimal alt yapı birlikte dinamik programlamanın kapısını açar.

8.3.3 Durum ve bağıntı

Tasarım adımlarının ilki son karara bakmaktır. Eşyalar 1'den n'ye numaralansın. Optimal çözümde son eşya için yalnız iki seçenek vardır: ya alınmıştır ya alınmamıştır. Alınmamışsa geriye kalan problem, ilk $n - 1$ eşyayı aynı kapasiteyle en iyi biçimde yerleştirmektir. Alınmışsa değeri v_n kazanılmıştır ve geriye kalan problem, ilk $n - 1$ eşyayı $W - w_n$ kapasiteyle en iyi biçimde yerleştirmektir. İki durumda da geriye kalan problem aynı türdendir. Değişen yalnız iki sayıdır: eşya sayısı ve kapasite. Durum bu iki sayıyla tanımlanır.

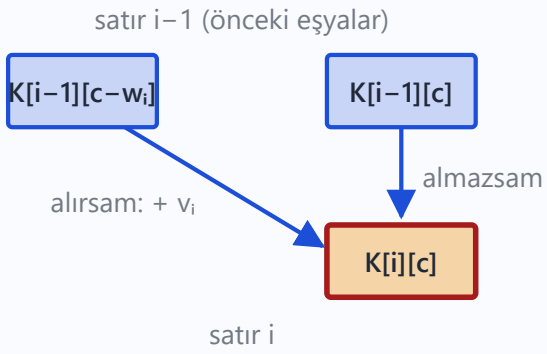
Tanım 8.5

$K[i][c]$, ilk i eşya ve c kapasite ile elde edilebilecek en büyük değerdir ($0 \le i \le n$, $0 \le c \le W$). Taban durumları ve bağıntı:

> $K[0][c] = 0, \quad K[i][0] = 0$
> $w_i > c$ ise $\quad K[i][c] = K[i-1][c]$
> aksi hâlde $\quad K[i][c] = \max(\, K[i-1][c], \;\; v_i + K[i-1][c - w_i] \,)$

Cevap $K[n][W]$'dir.

Bağıntının iki kolu iki kararın değeridir. *Almama* kolunda kapasite aynı kalır ve önceki eşyalarla elde edilen en iyi değer kullanılır: $K[i-1][c]$. *Alma* kolunda v_i kazanılır, kalan $c - w_i$ kapasite önceki eşyalarla en iyi biçimde doldurulur: $v_i + K[i-1][c-w_i]$. Eşya sığmıyorsa alma kolu yoktur. Taban durumları da sezgiyle uyumludur: hiç eşya yoksa ya da kapasite sıfırsa değer 0'dır.



Şekil 8.8: $K[i][c]$ hücresi bir üst satırdaki iki hücreden hesaplanır: tam üstündeki $K[i-1][c]$ (alma) ve w_i sütun soldaki $K[i-1][c-w_i]$ (al).

Şekil 8.8 hesaplama sırasını da belirler. Bir hücre yalnız bir üst satıra bakar. O hâlde tablo satır satır, yukarıdan aşağı doldurulursa her hücreye gelindiğinde ihtiyaç duyduğu hücreler hazır olur. Bir satırın içinde sütunların sırası önemli değildir.

8.3.4 Tablonun doldurulması

Bağıntı doğrudan iki iç içe döngüye çevrilir. Kod eşyaları 0'dan başlayan dizilerde tutar; tablonun *i*. satırı ise *i*. eşyayı, yani dizideki `v[i - 1]` ve `w[i - 1]` öğelerini kullanır.

```
int sirtCantasi(int[] v, int[] w, int W) {
  int n = v.length;
  int[][] K = new int[n + 1][W + 1];
  for (int i = 1; i <= n; i++)
    for (int c = 1; c <= W; c++) {
      int vi = v[i - 1], wi = w[i - 1];
      if (wi > c)            // sığmaz
        K[i][c] = K[i - 1][c];
      else                   // alma / al
        K[i][c] = Math.max(K[i - 1][c],
                    vi + K[i - 1][c - wi]);
    }
  return K[n][W];
}
```

`new int[n + 1][W + 1]` bütün hücreleri 0 ile başlatır; böylece 0. satır ve 0. sütun taban durumlarını kendiliğinden sağlar. Döngüler bu yüzden 1'den başlar. İçteki `if` eşyanın sığıp sığmadığını sınar. Sığmazsa üstteki değer kopyalanır; sığarsa iki kolun büyüğü yazılır.

Örnek olarak dört eşya ve $W = 6$ alınsın. Eşyalar (değer, ağırlık) biçiminde (5, 4), (4, 3), (3, 2) ve (2, 1)'dir. Tablo $(n + 1) \times (W + 1) = 5 \times 7$ boyutludur. Tablo 8.6 doldurulmuş tabloyu verir.

i	v_i	w_i	c = 0	1	2	3	4	5	6
0	—	—	0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	**9**

Tablo 8.6: *K[i][c]* tablosu; eşyalar (5, 4), (4, 3), (3, 2), (2, 1), $W = 6$.

Satırlar tek tek izlensin.

- **Satır 1** (5, 4). $c = 1, 2, 3$ için eşya sığmaz; üstteki 0 kopyalanır. $c = 4$'te $\max(0, 5 + K[0][0]) = 5$. $c = 5$ ve 6'da da aynı hesap 5 verir; tek eşya bir kez alınabilir.
- **Satır 2** (4, 3). $c = 3$'te ikinci eşya ilk kez sığar: $\max(0, 4 + K[1][0]) = 4$. $c = 4$'te almama kolu 5, alma kolu $4 + K[1][1] = 4$'tür; 5 kalır. $c = 5$ ve 6'da da 5 kalır: iki eşya birlikte 7 kg eder, sığmaz.
- **Satır 3** (3, 2). $c = 2$'de 3 yazılır. $c = 3$'te almama 4, alma $3 + K[2][1] = 3$'tür; 4 kalır. $c = 5$'te alma kolu $3 + K[2][3] = 3 + 4 = 7$ olur ve almama kolundaki 5'i geçer. $c = 6$'da $3 + K[2][4] = 3 + 5 = 8$ yazılır.
- **Satır 4** (2, 1). En hafif eşya her sütunda sığar. $c = 1$: $\max(0, 2) = 2$. $c = 3$: $\max(4, 2 + K[3][2]) = \max(4, 5) = 5$. $c = 4$: $\max(5, 2 + 4) = 6$. $c = 5$: $\max(7, 2 + 5) = 7$; iki kol eşittir ve almama korunur. $c = 6$: $\max(8, 2 + K[3][5]) = \max(8, 9) = 9$.



Şekil 8.9: Tablo doldu. Cevap $K[4][6] = 9$; 24 hücre hesaplandı, her biri $O(1)$.

Cevap sağ alt hücrededir: *K[4][6] = 9*. Tablonun her satırı bir soruyu cevaplar: “yalnız ilk *i* eşya varken her kapasite için en iyi değer nedir?” Örneğin 3. satır, (5, 4), (4, 3) ve (3, 2) eşyalarıyla 5 kg'lık bir çantanın en çok 7 ₺ değer taşıyabileceğini söyler.

> **Sık yapılan hata**
>
> Satır *i*'de eşya bilgisi `v[i]` ve `w[i]` ile okunur. Tablo *n + 1* satırlıdır ama diziler *n* öğelidir. `v[i]` yazılırsa her satır bir sonraki eşyayı kullanır ve son satırda `ArrayIndexOutOfBoundsException` oluşur. Doğrusu `v[i - 1]` 'dir.

8.3.5 Geri okuma

Tablo yalnız en büyük değeri verir. Hangi eşyaların seçildiği tablonun içinde saklıdır ve sondan başa doğru okunarak bulunur. Bu işleme **geri okuma** (*traceback*) denir.

Kural basittir. *K[i][c] ≠ K[i − 1][c]* ise değer ancak *i*. eşyayı alarak artmıştır; eşya alınmıştır ve kapasite $c \leftarrow c - w_i$ olur. Eşitse eşya alınmadan da aynı değer elde edilebilir; eşya alınmamış sayılır ve *c* değişmez. Her iki durumda bir üst satıra çıkılır.

Hücre	Karşılaştırma	Karar	Yeni c
K[4][6] = 9	≠ K[3][6] = 8	eşya 4 alındı (v = 2, w = 1)	6 − 1 = 5
K[3][5] = 7	≠ K[2][5] = 5	eşya 3 alındı (v = 3, w = 2)	5 − 2 = 3
K[2][3] = 4	≠ K[1][3] = 0	eşya 2 alındı (v = 4, w = 3)	3 − 3 = 0
K[1][0] = 0	= K[0][0] = 0	eşya 1 alınmadı	0

Tablo 8.7: *K[4][6] = 9* hücresinden geri okuma.



Şekil 8.10: Geri okumanın izlediği yol. Seçilen eşyalar 2, 3 ve 4: değer *4 + 3 + 2 = 9*, ağırlık *3 + 2 + 1 = 6 = W*.

Geri okuma her satırda bir karşılaştırma yapar; süresi *O(n)*'dir. Aynı kural koda kolayca çevrilir. Aşağıdaki metot doldurulmuş tabloyu ve ağırlıkları alır, seçilen eşyaların numaralarını (1'den başlayarak) döndürür.

```
List<Integer> geriOku(int[][] K, int[] w, int W) {
  List<Integer> secilen = new ArrayList<>();
  int c = W;
  for (int i = K.length - 1; i >= 1; i--)
    if (K[i][c] != K[i - 1][c]) {   // değer arttı: eşya i alındı
      secilen.add(0, i);
      c -= w[i - 1];
    }
  return secilen;
}
```

Metot örnek tablo için `[2, 3, 4]` döndürür. Eşitlik durumunda eşyanın alınmamış sayılması, birden çok optimal çözüm varsa bunlardan birinin seçilmesi demektir. Örneğin *K[4][5] = 7* hücresinde iki kol eşitti; o hücreden başlayan bir geri okuma 4. eşyayı almaz ve (4, 3) ile (3, 2) eşyalarını bulur.

8.3.6 Karmaşıklık ve sözde-polinom süre

Tablo *(n + 1)(W + 1)* hücrelidir ve her hücre sabit sürede hesaplanır. Süre ve bellek *O(n · W)*'dir. İlk bakışta bu bir polinom süre gibi görünür. Ama burada önemli bir incelik vardır.

Bir algoritmanın süresi, girdinin *boyutuna*, yani girdiyi yazmak için gereken bit sayısına göre ölçülür. *n* eşyayı yazmak için *n* ile orantılı yer gerekir. Oysa *W* sayısını yazmak için yalnız $\log_2 W$ bit gerekir. $W = 10^9$ otuz bitle yazılır, ama tablo 10^9 sütun ister. *W*'nin bit sayısı bir artınca *W*'nin değeri ve tablonun genişliği iki katına çıkar. Süre, girdi uzunluğuna göre üsteldir.

> **Tanım 8.6**
>
> Süresi girdinin uzunluğuna göre değil, girdideki bir sayının *değerine* göre polinom olan algoritmaya **sözde-polinom** (*pseudo-polynomial*) süreli algoritma denir. 0/1 sırt çantasının *O(n · W)* DP'si bu türdendir.

Bu ayrımın pratik bir sonucu vardır. 0/1 sırt çantası NP-zor bir problemdir; bilinen hiçbir algoritma onu her girdide polinom sürede çözemez. DP bu gerçeği değiştirmez. Yalnız *W* küçükken, örneğin birkaç bin ya da birkaç milyon olduğunda, problemi pratikte çözülebilir kılar. Ağırlıklar gram cinsinden ve kapasite birkaç ton ise tablo milyarlarca sütuna ulaşır ve kurulamaz.

> **Sık yapılan hata**
>
> *O(n · W)* polinom sanılır ve "0/1 sırt çantası DP ile polinom sürede çözülür" denir. *W* bir sayıdır; girdide *log W* bit kaplar. Süre bu bit sayısına göre üsteldir.

8.3.7 Tek boyutlu tablo

i. satır yalnız *i − 1*. satırı kullanır. Bu gözlem belleği *O(W)*'ye indirir: tek boyutlu bir `dp` dizisi tutulur. Eşyalar sırayla işlenir; her eşyada dizi, eski satırdan yeni satıra yerinde güncellenir.

```
int[] dp = new int[W + 1];
for (int i = 0; i < n; i++)
  for (int c = W; c >= w[i]; c--)
    dp[c] = Math.max(dp[c],
                     v[i] + dp[c - w[i]]);
return dp[W];
```

Kodun en önemli ayrıntısı iç döngünün yönüdür: kapasite **büyükten küçüğe** gezilir. Nedeni şudur. `dp[c]` güncellenirken okunan `dp[c - w[i]]`, eski satırın değeri olmalıdır; yani *i*. eşya henüz hesaba katılmamış olmalıdır. Büyükten küçüğe gezildiğinde $c - w_i < c$ olan hücreler bu turda henüz güncellenmemiştir ve eski değeri taşır. Küçükten büyüğe gezilseydi `dp[c - w[i]]` bu turda zaten güncellenmiş olurdu; aynı eşya iki kez alınabilirdi. Döngünün alt sınırı `c >= w[i]`, sığmayan kapasiteleri hiç ziyaret etmez. Bu hücrelerde değer değişmez; eski satırdan kopyalanmış sayılır.

Eşya (v, w)	Değişen hücreler (c = 6'dan aşağı)	dp[0..6]
başlangıç	—	0 0 0 0 0 0 0
0: (5, 4)	dp[6]: 0 → 5, dp[5]: 0 → 5, dp[4]: 0 → 5	0 0 0 0 5 5 5
1: (4, 3)	dp[3]: 0 → 4	0 0 0 4 5 5 5
2: (3, 2)	dp[6]: 5 → 8, dp[5]: 5 → 7, dp[2]: 0 → 3	0 0 3 4 5 7 8
3: (2, 1)	dp[6]: 8 → 9, dp[4]: 5 → 6, dp[3]: 4 → 5, dp[1]: 0 → 2	0 2 3 5 6 7 9

Tablo 8.8: Tek boyutlu 0/1 sırt çantası; eşyalar (5, 4), (4, 3), (3, 2), (2, 1), *W = 6*. Her satır, o eşya işlendikten sonraki `dp[0..6]`'dır.

Her satır, iki boyutlu tablonun aynı numaralı satırıyla birebir aynıdır. Örneğin eşya 0 işlenirken `dp[6]`, `dp[5]` ve `dp[4]` sırayla 5 olur. Bu sırada okunan `dp[2]`, `dp[1]` ve `dp[0]` henüz değişmemiştir ve 0'dır; eşya bir kez alınır. Sonuç `dp[6] = 9`'dur. Kodda eşyalar 0'dan numaralandığı için seçim "eşya 1, 2, 3" diye okunur: *4 + 3 + 2*. İki boyutlu tablonun numaralandırmasıyla bu, eşya 2, 3 ve 4'tür.



Şekil 8.11: Dört eşya işlendikten sonra tek satırlık tablo: `dp[6] = 9`, iki boyutlu tablonun sağ alt hücresiyle aynı.

Örnek 8.4

Aynı dört eşya için iç döngü yanlışlıkla küçükten büyüğe yazılırsa (`for (int c = w[i]; c <= W; c++)`) sonuç ne olur?

Eşya 3 (2, 1) işlenirken `dp[1]` 2 olur; ardından `dp[2] = max(3, 2 + dp[1]) = 4` olur. Bu hesap az önce güncellenen `dp[1]`'i kullanır, yani aynı eşyayı ikinci kez alır. Zincir sürer: `dp[3] = 6`, `dp[4] = 8`, `dp[5] = 10`, `dp[6] = 12`. Sonuç 12'dir: en hafif eşyadan altı tane alınmış gibi. Bu bir 0/1 çözümü değildir; sınırsız sırt çantasının cevabıdır.

Tek boyutlu sürümün bir bedeli vardır. Ara satırlar silindiği için geri okuma artık yapılamaz. Seçilen eşyalar gerekiyorsa her eşya ve kapasite için kararı saklayan ayrı bir `boolean[][] dahil` tablosu tutulur. Bu tablo da $n \cdot W$ boyutludur, ama `int` yerine `boolean` sakladığı için daha az yer kaplar.

8.3.8 Yukarıdan aşağı çözüm

Aynı bağıntı bellekleme ile de yazılabilir. Burada alt problem "*i*. eşyadan itibaren, *c* kapasiteyle elde edilebilecek en büyük değer" olarak tanımlanır; eşyalar baştan sona taranır. Tanım tablolamadakinin aynadaki görüntüsüdür ve aynı cevabı verir.

```
Integer[][] bellek;   // bellek[i][c]: i. eşyadan itibaren, c kapasiteyle en iyi

int f(int[] v, int[] w, int i, int c) {
  if (i == v.length) return 0;              // eşya kalmadı
  if (bellek[i][c] != null) return bellek[i][c];
  int sonuc = f(v, w, i + 1, c);            // alma
  if (w[i] <= c)                            // al
    sonuc = Math.max(sonuc, v[i] + f(v, w, i + 1, c - w[i]));
  bellek[i][c] = sonuc;
  return sonuc;
}

int sirtCantasiBellek(int[] v, int[] w, int W) {
  bellek = new Integer[v.length][W + 1];
  return f(v, w, 0, W);
}
```

Önbellek `Integer[][]` türündedir. `Integer` nesneleri `null` ile başlar ve `null` "henüz hesaplanmadı" anlamına gelir. `int[][]` kullanılsaydı 0 değeri hem "hesaplanmadı" hem de "gerçek cevap 0" anlamına gelirdi; cevabı 0 olan durumlar tekrar tekrar hesaplanırdı. Metot önce taban durumuna bakar: eşya kalmadıysa değer 0'dır. Sonra önbelleğe bakar. Değer yoksa almama kolunu hesaplar, eşya sığıyorsa alma kolunu da hesaplar ve büyüğünü saklar.

Örnekte `sirtCantasiBellek(v, w, 6)` 9 döndürür. Önbellek $4 \times 7 = 28$ hücrelidir, ama bunlardan yalnız 12'si doldurulur. Kökten başlayan özyineleme yalnız gerçekten ulaşılabilen kapasitelere iner: 0. eşyada yalnız $c = 6$; 1. eşyada $c = 6$ ve 2; 2. eşyada 6, 3 ve 2; 3. eşyada 6, 4, 3, 1, 2 ve 0. Ağırlıklar büyük ve az sayıdaysa bu fark daha da büyür. Örneğin bütün ağırlıklar 1000'in katıysa yalnız 1000'in katı olan kapasitelere inilir.

8.4 Sınırsız sırt çantası

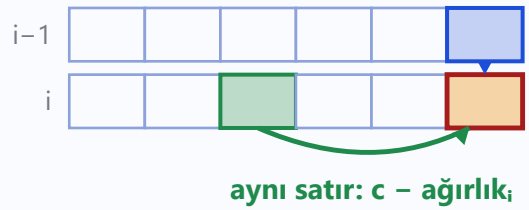
8.4.1 Problem ve bağıntı

Bazı problemlerde eşyalar tek tek değil, türler hâlinde verilir ve her türden istenildiği kadar alınabilir. Bir kargo firması aynı boyutta istediği kadar koli yükleyebilir; bir kesme makinesi aynı uzunlukta istediği kadar parça kesebilir. Bu probleme **sınırsız sırt çantası** (*unbounded knapsack*) denir. Geri kalan her şey 0/1 sırt çantasıyla aynıdır: kapasite aşılmadan toplam kâr en büyük yapılır.

Durum yine iki sayıyla tanımlanır, ama bu kez eşya türleri 0'dan numaralanır: *dp[i][c]*, 0'dan *i*'ye kadar olan eşya türleri ve *c* kapasite ile elde edilebilecek en büyük kârdır. Son kararın yapısı da benzer. *i*. türden hiç alınmazsa cevap *dp[i − 1][c]*'dir. En az bir tane alınırsa $kâr_i$ kazanılır ve geriye $c - ağırlık_i$ kapasite kalır. Bu kalan kapasitede *aynı türden yine alınabilir*. Bu yüzden kalan problem *i − 1*. satırda değil, yine *i*. satırdadır.

> $dp[i][c] = max(\ dp[i-1][c],\ \ kâr_i + dp[i][c - ağırlık_i]\ )\quad (ağırlık_i \le c \text{ ise})$

Eşya sığmıyorsa yalnız ilk kol kalır. Satır indisi 0'dan başladığı için $dp[-1][c] = 0$ kabul edilir: hiç eşya türü yokken kâr 0'dır. 0/1 sırt çantasıyla tek fark, "alırsam" kolunun baktığı satırdır.



Şekil 8.12: Sınırsız sırt çantasında alma kolu aynı satırdaki $c - ağırlık_i$ hücresine bakar: eşya alındıktan sonra yine alınabilir.

Bu fark, hesaplama sırasını da etkiler. Hücre artık aynı satırda soldaki bir hücreye bağlıdır. Satır soldan sağa doldurulursa o hücre hazırdır. Satırlar yine yukarıdan aşağı gezilir.

8.4.2 Tablonun doldurulması

Örnek olarak dört eşya türü alınsın: kârlar *[15, 50, 60, 90]*, ağırlıklar *[1, 3, 4, 5]*, kapasite 8. Tablo *4 × 9* boyutludur. İlk satırda yalnız 0. tür (15 kâr, 1 kg) kullanılabilir.

i	kâr	ağırlık	c = 0	1	2	3	4	5	6	7	8
0	15	1	0	15	30	45	60	75	90	105	120
1	50	3	0	15	30	50	65	80	100	115	130
2	60	4	0	15	30	50	65	80	100	115	130
3	90	5	0	15	30	50	65	90	105	120	**140**

Tablo 8.9: Sınırsız sırt çantası tablosu; kâr *[15, 50, 60, 90]*, ağırlık *[1, 3, 4, 5]*, kapasite 8.

Satırların hesabı şöyle ilerler.

- **Satır 0.** Kapasite 0 sütunu 0'dır. *c = 1*'de tek seçenek 0. türü almaktır: *15 + dp[0][0] = 15*. *c = 2*'de *15 + dp[0][1] = 30*: aynı satır tekrar kullanıldı, yani 0. türden iki tane alındı. Satır böylece 15'in katlarıyla dolar: *dp[0][c] = 15c*.
- **Satır 1** (50, 3 kg). *c = 1, 2*'de eşya sığmaz, üst satır kopyalanır. *c = 3*: *max(45, 50 + dp[1][0]) = 50*. *c = 4*: *max(60, 50 + dp[1][1]) = 65*. *c = 6*: *max(90, 50 + dp[1][3]) = max(90, 100) = 100*; burada *dp[1][3]*, kendisi de 1. türü içeren bir hücredir. 100, iki tane 50'lik eşyadır. *c = 8*: *max(120, 50 + dp[1][5]) = 130*.
- **Satır 2** (60, 4 kg). 60'lık eşya hiçbir sütunda üst satırı geçemez. Örneğin *c = 8*'de *60 + dp[2][4] = 60 + 65 = 125 < 130*. Satır, üst satırın kopyasıdır.

- **Satır 3** (90, 5 kg). *c = 5*: *max(80, 90 + 0) = 90*. *c = 6*: *max(100, 90 + dp[3][1]) = 105*. *c = 7*: *max(115, 90 + 30) = 120*. *c = 8*: *max(130, 90 + dp[3][3]) = max(130, 140) = 140*.

			kapasite →								
kâr	ağır.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90	105	120	140

Şekil 8.13: Doldurulmuş tablo. Cevap *dp[3][8] = 140 = 90 + dp[3][3]*; *dp[3][3] = 50*, bir tane 1. tür eşyadır.

Cevap *dp[3][8] = 140*'tır. Geri okuma 0/1'deki gibi yapılır, tek farkla: bir tür alındığında aynı satırda kalınır. *dp[3][8] = 140 ≠ dp[2][8] = 130* olduğu için 3. türden bir tane alınmıştır ve *c = 3* olur. *dp[3][3] = dp[2][3] = 50* olduğundan yukarı çıkılır. *dp[2][3] = dp[1][3]* yine yukarı. *dp[1][3] = 50 ≠ dp[0][3] = 45*: 1. türden bir tane alınmıştır ve *c = 0* olur. Çözüm, bir tane 90'lık (5 kg) ve bir tane 50'lik (3 kg) eşyadır.

8.4.3 Tek boyutlu çözüm ve döngü yönü

Sınırsız sırt çantasında satırlar tamamen birleştirilebilir. Her türden istenildiği kadar alınabildiğine göre, c kapasitenin en iyi değeri son alınan eşyaya bakılarak yazılabilir. Son alınan eşya j ise kalan kapasite $c - w_j$'dir ve orada da bütün türler serbesttir:

> $$dp[c] = \max \{\, dp[c - w_j] + v_j : w_j \le c \,\}, \quad dp[0] = 0$$

```
static int sinirsizSirtCantasi(int[] w, int[] v, int W) {
  int[] dp = new int[W + 1];       // dp[c]: c kapasiteyle en iyi
  for (int c = 1; c <= W; c++)
    for (int j = 0; j < w.length; j++)
      if (w[j] <= c)
        dp[c] = Math.max(dp[c], dp[c - w[j]] + v[j]);
  return dp[W];
}
```

Dış döngü kapasiteyi 1'den *W*'ye kadar artırır, iç döngü her türü dener. *dp[c]* hesaplanırken daha küçük bütün kapasiteler hazırdır. Örnek için sonuç dizisi *dp = [0, 15, 30, 50, 65, 90, 105, 120, 140]*'tır; iki boyutlu tablonun son satırının aynısıdır. Süre $O(n \cdot W)$, bellek $O(W)$'dir.

Tek boyutlu 0/1 sırt çantasıyla bu kod arasındaki ilişki öğreticidir. İkisi de tek bir dizide şu güncellemeyi yapar: `dp[c] = max(dp[c], v + dp[c - w])`. Fark, döngü yönündedir. Kapasite **küçükten büyüğe** gezilirse `dp[c - w]` bu turda zaten güncellenmiştir ve eşyayı içerebilir; eşya tekrar alınabilir, sonuç sınırsız sırt çantasıdır. Kapasite **büyükten küçüğe** gezilirse `dp[c - w]` eski değeri taşır; eşya en çok bir kez alınır, sonuç 0/1'dir. Aynı tek satırlık güncelleme, yalnız döngünün yönüyle iki farklı problemi çözer.

Örnek 8.5

Tek bir eşya var: değer 10, ağırlık 3; kapasite 9. Tek boyutlu tablo iki yönde doldurulursa `dp[9]` kaç olur?

Büyükten küçüğe: *c* = 9'da `dp[6]` henüz 0'dır, `dp[9] = 10` olur; *c* = 6 ve *c* = 3'te de 10 yazılır. Sonuç 10'dur; eşya bir kez alınır. Küçükten büyüğe: `dp[3] = 10`, `dp[6] = 10 + dp[3] = 20`, `dp[9] = 10 + dp[6] = 30`. Eşya üç kez alınmıştır. 10, 0/1'in; 30, sınırsız sırt çantasının cevabıdır.

Sık yapılan hata

Tek boyutlu 0/1 sırt çantasında kapasite küçükten büyüğe gezilir. Kod derlenir, çalışır ve makul görünen bir sayı döndürür; ama bu sayı sınırsız sırt çantasının cevabıdır. Hatayı yakalamanın en kolay yolu, tek eşyalı küçük bir örnekte sonucun eşya değerini aşıp aşmadığına bakmaktır.

8.4.4 Kod izleme

Tek boyutlu kodun izlemesi her kapasitede hangi seçeneğin kazandığını gösterir. Aşağıdaki sürüm sunumdaki izleme kodudur ve yukarıdakiyle aynı işi yapar.

```
static int sinirsizSirtCantasi(
    int[] w, int[] v, int W) {
  int[] dp = new int[W + 1];  // dp[c]
  for (int c = 1; c <= W; c++)
    for (int j = 0; j < w.length; j++)
      if (w[j] <= c)
        dp[c] = Math.max(dp[c],
                  dp[c - w[j]] + v[j]);
  return dp[W];
}
```

c	Adaylar $dp[c - w_j] + v_j$	dp[c]
1	dp[0] + 15 = 15	15

c	Adaylar `dp[c − w_j] + v_j`	dp[c]
2	dp[1] + 15 = 30	30
3	dp[2] + 15 = 45, dp[0] + 50 = 50	50
4	dp[3] + 15 = 65, dp[1] + 50 = 65, dp[0] + 60 = 60	65
5	dp[4] + 15 = 80, dp[2] + 50 = 80, dp[1] + 60 = 75, dp[0] + 90 = 90	90
6	dp[5] + 15 = 105, dp[3] + 50 = 100, dp[2] + 60 = 90, dp[1] + 90 = 105	105
7	dp[6] + 15 = 120, dp[4] + 50 = 115, dp[3] + 60 = 110, dp[2] + 90 = 120	120
8	dp[7] + 15 = 135, dp[5] + 50 = 140, dp[4] + 60 = 125, dp[3] + 90 = 140	140

Tablo 8.10: `sinirsizSirtCantasi(w, v, 8)` izlemesi; kâr *[15, 50, 60, 90]*, ağırlık *[1, 3, 4, 5]*. Her satırda sığan türlerin adayları ve kazanan değer.

c = 2 satırı sınırsızlığın kaynağını gösterir: `dp[1]` bu turda güncellenmişti ve 0. türü bir kez içeriyordu; ona yine 15 eklenince aynı türden iki tane alınmış olur. Son satırda iki aday 140'a ulaşır: *dp[5] + 50* ve *dp[3] + 90*. İkisi de aynı seçimi anlatır: bir tane 50'lik (3 kg) ve bir tane 90'lık (5 kg). Aynı 140'ı veren 15 + 15 + 50 + 60 seçimi ise 9 kg'dır ve çantaya sığmaz. Tablo bütün birleşimleri tek tek saymadan, *O(n · W)* sürede denemiş olur.

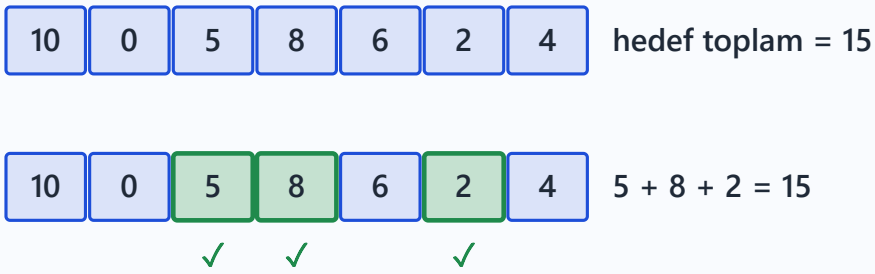


Şekil 8.14: İzlemenin sonunda `dp` dizisi; `dp[8] = 140`.

8.5 Alt küme toplamı

8.5.1 Problem ve bağıntı

Bir dizi tam sayı ve bir hedef değer verilsin. Toplamı tam olarak hedefe eşit olan bir alt küme var mıdır? Bu soru bir optimizasyon değil, **karar problemidir**: cevap evet ya da hayırdır. Problemin adı **alt küme toplamıdır** (*subset sum*).



Şekil 8.15: 10, 0, 5, 8, 6, 2, 4 dizisinde hedef 15: örneğin 5, 8 ve 2 seçilerek toplam 15 elde edilir.

Son karar yine bir öğenin seçilip seçilmediğidir. Son öğe seçilmezse soru, ilk $n - 1$ öğeyle aynı hedefin elde edilip edilemeyeceğidir. Seçilirse hedef öğe kadar azalır. Durum iki sayıyla tanımlanır ve değeri bir mantıksal değerdir.

Tanım 8.7

dp[i][j], ilk *i* öğeyle toplamı tam olarak *j* olan bir alt küme kurulabiliyorsa doğru, kurulamıyorsa yanlıştır.

> $dp[i][0] = \text{doğru}, \quad dp[0][j] = \text{yanlış} \ \ (j > 0)$
> $dp[i][j] = dp[i-1][j] \ \text{ veya } \ (a_i \le j \ \text{ ve } \ dp[i-1][j-a_i])$

Sütun 0 her satırda doğrudur, çünkü boş küme toplam 0'ı verir. Satır 0'da sıfırdan büyük hiçbir toplam elde edilemez. Bağıntı, 0/1 sırt çantasının bağıntısıyla aynı biçimdedir: sırt çantasındaki `max` yerine mantıksal veya, değerin toplanması yerine de yalnız ulaşılabilirlik vardır. Alt küme toplamı, sırt çantasının "değer = ağırlık" özel durumu olarak da görülebilir. Her öğenin değeri ağırlığına eşit olsun ve kapasite hedef olsun. Sırt çantasının cevabı hedefe eşitse istenen alt küme vardır.

8.5.2 Tablo ve geri okuma

Dizi $a = 10, 0, 5, 8, 6, 2, 4$ ve hedef 15 olsun. Tablo 8×16 boyutludur. Her satırı bütün hücreleriyle yazmak yerine, o satırda ulaşılabilen toplamları listelemek daha okunaklıdır. Bir satır, bir önceki satırın bütün toplamlarını korur; ayrıca o toplamların her birine a_i eklenerek yeni toplamlar ekler (hedefi aşanlar dışarıda kalır).

i	a_i	Ulaşılabilen toplamlar (dp[i][j] doğru)	Yeni
0	—	0	—
1	10	0, 10	10
2	0	0, 10	yok
3	5	0, 5, 10, 15	5, 15

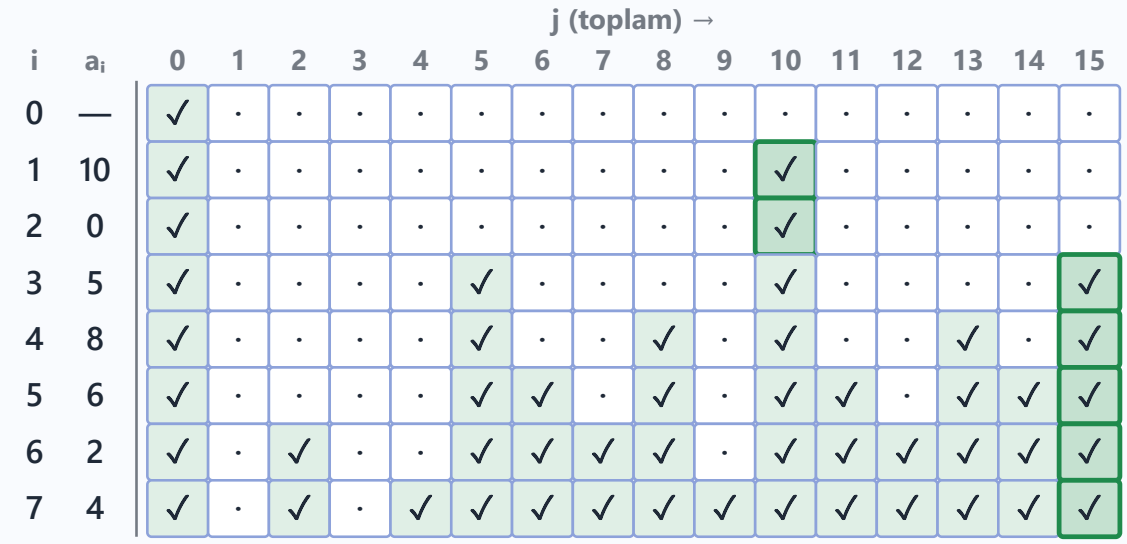
i	a_i	Ulaşılabilen toplamlar (dp[i][j] doğru)	Yeni
4	8	0, 5, 8, 10, 13, 15	8, 13
5	6	0, 5, 6, 8, 10, 11, 13, 14, 15	6, 11, 14
6	2	0, 2, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15	2, 7, 12
7	4	0, 2, 4, 5, 6, …, 15 (1 ve 3 dışında hepsi)	4, 9

Tablo 8.11: Alt küme toplamı tablosu satır satır; *a = 10, 0, 5, 8, 6, 2, 4*, hedef 15. Satırda doğru olan sütunlar listelenmiştir.

2. satır ilginçtir: 0 eklemek hiçbir toplamı değiştirmez, satır üst satırın kopyasıdır. Hedef 15 ilk kez 3. satırda ulaşılabilir olur: 10 ve 5 birlikte. Son hücre *dp[7][15]* doğrudur; cevap evettir.

Geri okuma son hücreden başlar. *dp[i − 1][j]* doğruysa a_i olmadan da toplam j kurulabilir; öğe seçilmez ve yukarı çıkılır. Yanlışsa toplam ancak a_i ile kurulabilir; öğe seçilir ve $j \leftarrow j - a_i$ olur.

- *dp[7][15] = dp[6][15] =* doğru: 4 seçilmez. Aynı gerekçeyle 2, 6 ve 8 de seçilmez; 4. satıra kadar çıkılır.
- *dp[3][15]* doğru ama *dp[2][15]* yanlış: 5 seçilir; *j = 15 − 5 = 10*.
- *dp[2][10] = dp[1][10] =* doğru: 0 seçilmez.
- *dp[1][10]* doğru ama *dp[0][10]* yanlış: 10 seçilir; *j = 0* ve okuma biter.



Şekil 8.16: Doldurulmuş tablo ve geri okuma yolu. Geri okuma {5, 10} alt kümesini bulur: *10 + 5 = 15*.

Bu dizide toplamı 15 olan altı alt küme vardır: {10, 5}, {5, 8, 2}, {5, 6, 4} ve bunların her birine 0 eklenmiş hâlleri. Geri okuma yalnız birini verir. Hangisini vereceği, eşitlik kuralına bağlıdır. Buradaki kural "öğe olmadan da olabiliyorsa seçme" olduğu için okuma, dizinin başındaki öğeleri kullanan çözüme yönelir.

8.5.3 Kod ve analiz

```
boolean altKume(int[] a, int hedef) {
  int n = a.length;
  boolean[][] dp = new boolean[n + 1][hedef + 1];
  for (int i = 0; i <= n; i++)
    dp[i][0] = true;           // boş küme
  for (int i = 1; i <= n; i++) {
    int x = a[i - 1];
    for (int j = 1; j <= hedef; j++)
      dp[i][j] = dp[i - 1][j]  // x alınmaz
        || (x <= j && dp[i - 1][j - x]);
  }
  return dp[n][hedef];
}
```

Java'da yeni bir `boolean` dizisi `false` ile başlar. İlk döngü sütun 0'ı doğru yapar. İç döngüdeki ifade bağıntının birebir karşılığıdır. `||` işlecinin kısa devre davranışı burada işe yarar: `dp[i - 1][j]` doğruysa sağ taraf hiç hesaplanmaz. Sağ taraftaki `x <= j` sınaması da `&&`'nin kısa devresi sayesinde `j - x` negatifken diziye erişilmesini önler.

Süre ve bellek *O(n · hedef)*'tir. Bu da sözde-polinom bir süredir: hedef büyükse tablo kurulamaz. Alt küme toplamı NP-tam bir problemdir; DP onu yalnız hedef küçükken pratik kılar. Bellek, 0/1 sırt çantasındaki gibi tek boyuta indirilebilir. Bunun için *j* büyükten küçüğe gezilir ki her öğe en çok bir kez kullanılsın.

Örnek 8.6

Eşit bölme problemi. *{1, 5, 11, 5}* dizisi toplamları eşit iki parçaya bölünebilir mi?

Toplam *S = 22*'dir. Bir parçanın toplamı *S/2 = 11* ise öteki parçanın toplamı da 11'dir. Soru, hedef 11 ile alt küme toplamına dönüşür. {11} ya da {1, 5, 5} bu toplamı verir; cevap evettir. *S* tek olsaydı cevap hesap yapmadan hayır olurdu.

Sık yapılan hata

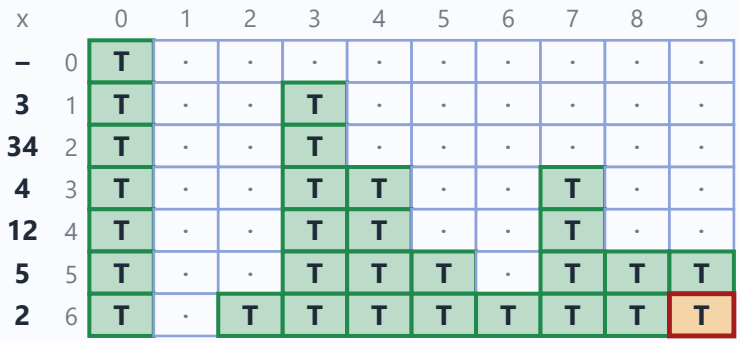
Sütun 0 doğru olarak başlatılmaz. Tablo baştan sona `false` kalır ve hiçbir toplam ulaşılabilir görünmez; her girdide cevap "hayır" olur. Boş kümenin toplamı 0'dır ve her satırda geçerlidir.

8.5.4 Kod izleme

Kodun satır satır davranışı daha küçük bir örnekte izlensin: `altKume({3, 34, 4, 12, 5, 2}, 9)`. Tablo *(n + 1) × (hedef + 1) = 7 × 10 = 70* hücrelidir.

i	x	dp[i][j] doğru olan j'ler	Açıklama
0	—	0	yalnız boş küme
1	3	0, 3	yeni toplam: 3
2	34	0, 3	*34 > 9*: hiçbir j için sığmaz, satır kopyalanır
3	4	0, 3, 4, 7	yeni toplamlar: 4, 7
4	12	0, 3, 4, 7	*12 > 9*: satır kopyalanır
5	5	0, 3, 4, 5, 7, 8, 9	yeni: 5, 8, 9; *dp[5][9] = dp[4][4] → 4 + 5 = 9*
6	2	0, 2, 3, 4, 5, 6, 7, 8, 9	yeni: 2, 6

Tablo 8.12: `altKume({3, 34, 4, 12, 5, 2}, 9)` izlemesi; her satırda doğru olan sütunlar.



Şekil 8.17: İzlemenin sonu: `dp[6][9] = true`. T doğru, nokta yanlış hücrelerdir.

Metot `true` döndürür. Toplamı 9 olan iki alt küme vardır: {4, 5} ve {3, 4, 2}. Geri okuma *dp[6][9]*'dan başlar; *dp[5][9]* doğru olduğu için 2 seçilmez. *dp[4][9]* yanlış olduğu için 5 seçilir ve *j = 4* olur. 12 seçilmez; *dp[2][4]* yanlış olduğu için 4 seçilir ve *j = 0* olur. Bulunan alt küme {4, 5}'tir. Aynı dizide hedef 30 sorulsaydı cevap hayır olurdu: 34 hiçbir zaman kullanılamaz, öteki öğelerin toplamı ise *3 + 4 + 12 + 5 + 2 = 26 < 30*'dur.

8.6 En uzun artan alt dizi

8.6.1 Alt dizi ve alt dizgi

Bu ve sonraki iki kesimdeki üç problem (LIS, ortak alt dizgi, LCS), bir dizinin parçalarıyla ilgilidir. Parça sözcüğü iki farklı anlamda kullanılır ve bu iki anlamın karıştırılması bu konudaki hataların başında gelir.

Tanım 8.8

Bir dizinin **alt dizisi** (*subsequence*), dizinin bazı öğeleri atlanarak ve kalan öğelerin sırası korunarak elde edilen dizidir. Seçilen öğelerin bitişik olması gerekmez.

Bir dizinin **alt dizgisi** ya da bitişik parçası (*substring, subarray*), dizinin ardışık öğelerinden oluşan parçadır.

"GXTXAYB" dizgisinde G, T, A, B öğeleri sırasıyla 0, 2, 4 ve 6. konumlarda geçer. Göreli sıraları korunduğu için "GTAB" bir alt dizidir. Öğeler bitişik olmadığı için alt dizgi değildir. Buna karşılık "XAY" 3, 4 ve 5. konumlarda bitişik geçer; hem alt dizgi hem alt dizidir. Her alt dizgi aynı zamanda bir alt dizidir; tersi doğru değildir. n uzunluklu bir dizinin 2^n alt dizisi, ama yalnız $n(n + 1)/2$ boş olmayan bitişik parçası vardır.

Tanım 8.9

En uzun artan alt dizi (*longest increasing subsequence*, LIS), öğeleri kesin olarak artan alt diziler arasında en uzun olanıdır.

Örnek olarak 0'dan 15'e kadar sayıların şu dizilişi alınsın: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15. Bu dizinin bir LIS'i 0, 2, 6, 9, 11, 15'tir ve uzunluğu 6'dır.



Şekil 8.18: 0, 2, 6, 9, 11, 15 öğeleri atlanarak seçilir; göreli sıraları korunur ve değerleri artar.

Aynı dizinin başka LIS'leri de vardır. Örneğin 0, 4, 6, 9, 13, 15 ve 0, 2, 6, 9, 13, 15 de 6 uzunlukta artan alt dizilerdir; bu dizide toplam dört farklı LIS vardır. LIS tek olmak zorunda değildir, ama uzunluğu tektir. DP önce bu uzunluğu bulur; dizinin kendisi gerekiyorsa ek bilgiyle geri okunur.

8.6.2 $O(n^2)$ dinamik programlama

Doğru durumu seçmek bu problemin anahtarıdır. "İlk i öğenin LIS uzunluğu" iyi bir durum değildir: bu değeri bilmek, $i + 1$. öğenin o diziyi uzatıp uzatamayacağını söylemez, çünkü dizinin hangi sayıyla bittiği bilinmez. Durum, bitiş noktasını sabitlemelidir.

Tanım 8.10

dp[i], tam olarak a[i]'de biten en uzun artan alt dizinin uzunluğudur.

> $dp[i] = 1 + max\ \{\ dp[j] : j < i,\ \ a[j] < a[i]\ \}$

Böyle bir *j* yoksa *dp[i] = 1*'dir. Cevap *max dp[i]*'dir.

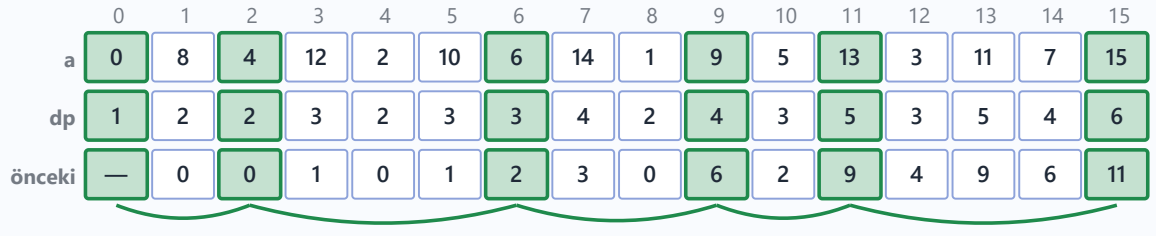
Bağıntının anlamı şudur. *a[i]*'de biten artan bir alt dizide *a[i]*'den önceki öğe, solda yer alan ve *a[i]*'den küçük bir *a[j]*'dir. O öğede biten en uzun dizinin sonuna *a[i]* eklenir. Bütün adaylar denenir ve en uzunu seçilir. Önünde daha küçük öğe yoksa *a[i]* tek başına 1 uzunluklu bir dizidir. Cevap son hücre değildir: LIS dizinin herhangi bir yerinde bitebilir.

Tablo 8.13 16 öğeli örneğin bütün *dp* değerlerini verir. "Önceki" satırı, en büyük *dp[j]*'yi veren *j*'dir; eşitlikte soldaki ilk *j* seçilmiştir.

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
a[i]	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp[i]	1	2	2	3	2	3	3	4	2	4	3	5	3	5	4	6
önceki	—	0	0	1	0	1	2	3	0	6	2	9	4	9	6	11

Tablo 8.13: 16 öğeli dizide *dp* ve önceki indisler.

Birkaç hücrenin hesabı ayrıntılı yazılsın. *i = 3*'te *a[3] = 12*'dir; soldaki adaylar 0, 8 ve 4'tür ve dp değerleri 1, 2, 2'dir. En büyüğü *j = 1*'deki 2'dir: *dp[3] = 3* (0, 8, 12). *i = 9*'da *a[9] = 9*'dur; adaylar 0, 8, 4, 2, 6 ve 1'dir. Bunlardan en uzun diziyle biteni 6'dır (*dp[6] = 3*): *dp[9] = 4* (0, 4, 6, 9). *i = 15*'te *a[15] = 15* bütün öğelerden büyüktür; adayların en büyük dp değeri 5'tir ve ilk kez *j = 11*'de görülür: *dp[15] = 6*.



Şekil 8.19: En büyük dp 6'dır (*i = 15*). Önceki indisler izlenir: 15 → 11 → 9 → 6 → 2 → 0; LIS 0, 4, 6, 9, 13, 15.

Dizinin kendisi önceki indislerden geri okunur. En büyük dp değerinin bulunduğu indisten başlanır ve önceki indislere atlanır: 15 → 11 → 9 → 6 → 2 → 0. Bu indislerdeki öğeler sondan başa 15, 13, 9, 6, 4, 0'dır; ters çevrilince 0, 4, 6, 9, 13, 15 elde edilir. Bu, Şekil 8.18'deki LIS'ten farklıdır ama aynı uzunluktadır. Farkın nedeni eşitlik kuralıdır: *i = 15*'te hem *j = 11* hem *j = 13* dp değeri 5 olan adaylardır ve ilk görülen seçilir.

8.6.3 Kod ve izleme

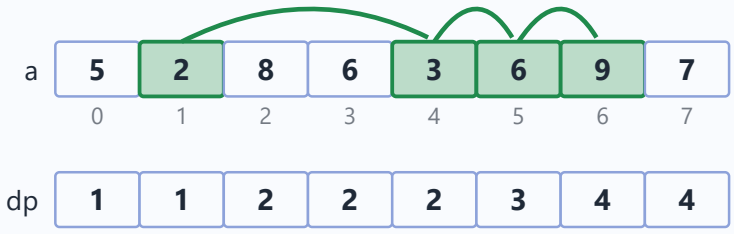
```
static int lis(int[] a) {
  int n = a.length, enUzun = 0;
  int[] dp = new int[n];
  for (int i = 0; i < n; i++) {
    dp[i] = 1;                // tek başına
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;    // uzat
    enUzun = Math.max(enUzun, dp[i]);
  }
  return enUzun;
}
```

Dış döngü her *i* için önce `dp[i] = 1` yazar: öğe tek başına bir dizidir. İç döngü soldaki her *j*'yi dener. `a[j] < a[i]` koşulu dizinin artan olmasını, `dp[j] + 1 > dp[i]` koşulu ise yalnız daha uzun bir dizinin kabul edilmesini sağlar. Kesin büyüktür işareti eşitlikte ilk *j*'yi korur. Her turun sonunda `enUzun` güncellenir. İç döngü toplam $0 + 1 + \dots + (n-1) = n(n-1)/2$ kez döner; süre $O(n^2)$, bellek $O(n)$'dir.

Kodu daha kısa bir dizide izlemek, satırların etkisini açıkça gösterir: `lis({5, 2, 8, 6, 3, 6, 9, 7})`.

i	a[i]	Soldaki küçükler	Seçilen j (dp[j])	dp[i]	enUzun
0	5	yok	—	1	1
1	2	yok	—	1	1
2	8	5, 2	0 (1)	2	2
3	6	5, 2	0 (1)	2	2
4	3	2	1 (1)	2	2
5	6	5, 2, 3	4 (2)	3	3
6	9	5, 2, 8, 6, 3, 6	5 (3)	4	4
7	7	5, 2, 6, 3, 6	5 (3)	4	4

Tablo 8.14: `lis({5, 2, 8, 6, 3, 6, 9, 7})` izlemesi.



Şekil 8.20: İzlemenin sonu: `enUzun = 4`. Bir LIS 2, 3, 6, 9'dur (dp değerleri 1, 2, 3, 4 olan zincir).

i = 5 satırı durumun neden "*a[i]*'de biten" diye tanımlandığını gösterir. *a[5] = 6*'dan küçük üç öğe vardır: 5, 2 ve 3. 5 ve 2 tek başına birer dizinin sonudur (dp = 1). 3 ise 2, 3 dizisinin sonudur (dp = 2). 6, en uzun zinciri uzatır: 2, 3, 6. Son öğe 7'nin dp değeri 4'tür ve `enUzun` ile eşittir; bu bir rastlantıdır. Dizinin sonuna 1 eklenseydi son hücre 1, cevap yine 4 olurdu.

Diziyi kurmak için her öğenin önceki indisi saklanır. Aşağıdaki metot `lis`'e bir `onceki` dizisi ekler ve en uzun dizinin bittiği yerden geriye doğru zinciri izler.

```
List<Integer> lisDizi(int[] a) {
  int n = a.length, enSon = 0;
  int[] dp = new int[n], onceki = new int[n];
  for (int i = 0; i < n; i++) {
    dp[i] = 1; onceki[i] = -1;
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i]) {
        dp[i] = dp[j] + 1; onceki[i] = j;
      }
    if (dp[i] > dp[enSon]) enSon = i;
  }
  LinkedList<Integer> lis = new LinkedList<>();
  for (int i = enSon; i != -1; i = onceki[i])
    lis.addFirst(a[i]);
  return lis;
}
```

Metot örnek dizi için `[2, 3, 6, 9]`, 16 öğeli dizi için `[0, 4, 6, 9, 13, 15]` döndürür. `LinkedList.addFirst`, geri okumanın sondan başa ilerlemesini dengeler: öğeler başa eklendiği için liste doğru sırada oluşur.

Sık yapılan hata

Metot `enUzun` yerine `dp[n - 1]` döndürür. {5, 1, 6, 2, 7, 0} dizisinde *dp = [1, 1, 2, 2, 3, 1]*'dir. Son öğe 0 hiçbir diziyi uzatamaz; `dp[5] = 1` döner. Doğru cevap 3'tür (5, 6, 7 ya da 1, 2, 7).

8.6.4 Aktif listeler

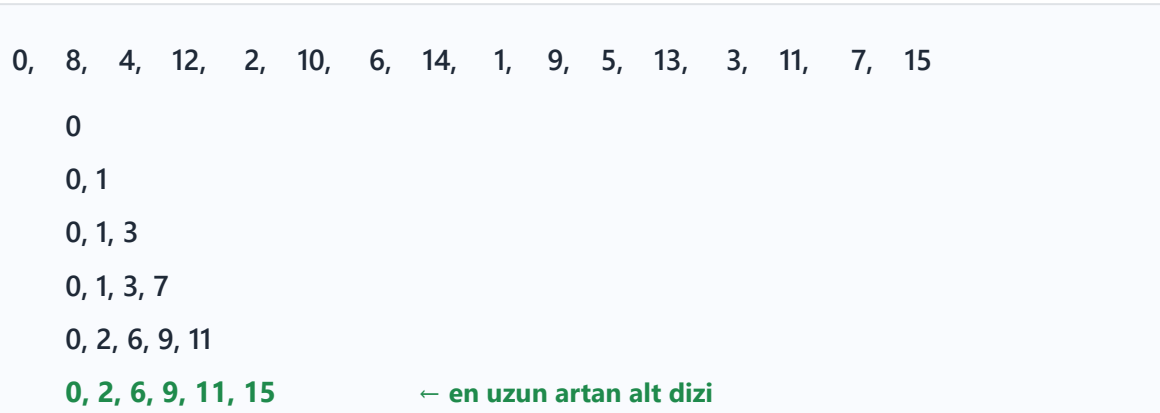
$O(n^2)$ algoritma her öğede soldaki bütün öğelere bakar. Daha hızlı bir yöntem için şu gözlem kullanılır: aynı uzunlukta iki artan dizi varsa, sonu küçük olan her zaman daha iyidir. Sonradan gelecek bir sayı, sonu büyük olanı uzatabiliyorsa sonu küçük olanı da uzatabilir; tersi doğru değildir. O hâlde her uzunluk için yalnız bir liste saklamak yeter: o uzunluktaki artan listeler arasında **sonu en küçük olan**. Bu listelere **aktif listeler** denir.

Yeni bir x öğesi geldiğinde üç durum vardır:

1. x bütün aktif listelerin sonlarından küçükse 1 uzunluklu yeni bir liste $[x]$ başlar ve eski 1 uzunluklu listenin yerini alır.
2. x bütün sonlardan büyükse en uzun liste kopyalanır, sonuna x eklenir; liste sayısı bir artar.
3. Aksi hâlde sonu x'ten küçük olan en uzun liste bulunur, kopyalanıp sonuna x eklenir. Yeni liste, aynı uzunluktaki eski listenin yerini alır; eski listenin sonu daha büyük olduğu için artık gereksizdir.

x	Olay	Liste sayısı	En uzun aktif liste
0	ilk liste [0]	1	0
8	hepsinden büyük: uzat	2	0, 8
4	[0]'ı uzatır → [0, 4]; [0, 8] gereksiz	2	0, 4
12	hepsinden büyük: uzat	3	0, 4, 12
2	[0]'ı uzatır → [0, 2]; [0, 4] gereksiz	3	0, 4, 12
10	[0, 2]'yi uzatır → [0, 2, 10]; [0, 4, 12] gereksiz	3	0, 2, 10
6	[0, 2]'yi uzatır → [0, 2, 6]; [0, 2, 10] gereksiz	3	0, 2, 6
14	hepsinden büyük: uzat	4	0, 2, 6, 14
1	[0]'ı uzatır → [0, 1]; [0, 2] gereksiz	4	0, 2, 6, 14
9	[0, 2, 6]'yı uzatır → [0, 2, 6, 9]; [0, 2, 6, 14] gereksiz	4	0, 2, 6, 9
5	[0, 1]'i uzatır → [0, 1, 5]; [0, 2, 6] gereksiz	4	0, 2, 6, 9
13	hepsinden büyük: uzat	5	0, 2, 6, 9, 13
3	[0, 1]'i uzatır → [0, 1, 3]; [0, 1, 5] gereksiz	5	0, 2, 6, 9, 13
11	[0, 2, 6, 9]'u uzatır → [0, 2, 6, 9, 11]; [0, 2, 6, 9, 13] gereksiz	5	0, 2, 6, 9, 11
7	[0, 1, 3]'ü uzatır → [0, 1, 3, 7]; [0, 2, 6, 9] gereksiz	5	0, 2, 6, 9, 11
15	hepsinden büyük: uzat	6	0, 2, 6, 9, 11, 15

Tablo 8.15: 16 öğeli dizide aktif listeler. Her satır, öğe işlendikten sonraki en uzun liste ve liste sayısıdır.



Şekil 8.21: Dizi bittiğinde altı aktif liste. En uzunu 0, 2, 6, 9, 11, 15'tir; LIS uzunluğu 6.

Dizi bittiğinde liste sayısı LIS uzunluğudur ve en uzun aktif liste bir LIS'tir. Yöntemin doğruluğu aşağıdaki değişmeze dayanır.

Teorem 8.1 (Sonlar artandır)

Her adımdan sonra, uzunluğu k olan aktif listenin sonu, uzunluğu $k + 1$ olan aktif listenin sonundan kesin olarak küçüktür.

İspat. Uzunluğu $k + 1$ olan aktif listenin ilk k öğesi, uzunluğu k olan artan bir listedir ve son öğesi, $k + 1$'lik listenin sonundan küçüktür. k uzunluklu aktif liste, o uzunluktaki listeler arasında sonu en küçük olandır; sonu bu öğeden büyük olamaz. Dolayısıyla k'lık listenin sonu, $k + 1$'lik listenin sondan bir önceki öğesine eşit ya da ondan küçüktür; bu da $k + 1$'lik listenin sonundan kesin olarak küçüktür. □

Teoremin iki önemli sonucu vardır. Birincisi, yeni öğenin hangi listeyi uzatacağı sıralı bir dizide arama yapılarak bulunabilir: sonlar dizisi artandır ve aranan yer, sonu x'ten büyük ya da eşit olan ilk listedir. Bu ikili aramayla $O(\log n)$ sürede bulunur. İkincisi, listelerin kendilerini saklamaya gerek yoktur; yalnız sonlarını saklamak yeter.

8.6.5 O(n log n) kod

Aşağıdaki kod aktif listelerin yalnız sonlarını saklar. `son[k]`, uzunluğu $k + 1$ olan artan alt diziler arasında en küçük son öğedir. `son[0..uz − 1]` her zaman artandır; başta `uz = 0`'dır.

```
static int lisHizli(int[] a) {
  int[] son = new int[a.length];
  int uz = 0;                 // aktif liste sayısı
  for (int x : a) {
    int alt = 0, ust = uz;    // son[alt] >= x ilk yer
    while (alt < ust) {
      int orta = (alt + ust) / 2;
      if (son[orta] < x) alt = orta + 1;
      else ust = orta;
    }
    son[alt] = x;             // yerine koy / uzat
    if (alt == uz) uz++;
  }
  return uz;
}
```

Her *x* için `while` döngüsü, `son[alt] >= x` koşulunu sağlayan ilk konumu ikili aramayla bulur. Arama aralığı `[alt, ust)` yarı açıktır; `ust` başlangıçta `uz`'dur. `son[orta] < x` ise aranan yer ortanın sağındadır; değilse orta da aday olabilir ve aralık sola daraltılır. Döngü bittiğinde `alt`, *x*'in yazılacağı yerdir. `alt == uz` ise *x* bütün sonlardan büyüktür ve liste sayısı artar; aksi hâlde *x* daha büyük bir sonun yerini alır. Bu son durum, aktif listeler yönteminin üçüncü kuralıdır: aynı uzunluktaki liste artık daha küçük bir sayıyla biter.

x	alt	karş.	İşlem	uz	son[0..uz – 1]
5	0	0	liste boş; uzar	1	5
2	0	1	5 yerine 2	1	2
8	1	1	hepsinden büyük; uzar	2	2, 8
6	1	2	8 yerine 6	2	2, 6
3	1	2	6 yerine 3	2	2, 3
6	2	1	hepsinden büyük; uzar	3	2, 3, 6
9	3	2	hepsinden büyük; uzar	4	2, 3, 6, 9
7	3	2	9 yerine 7	4	2, 3, 6, 7

Tablo 8.16: `lisHizli({5, 2, 8, 6, 3, 6, 9, 7})` izlemesi; "karş." ikili aramadaki karşılaştırma sayısıdır.



Şekil 8.22: İzlemenin sonu: `uz = 4`, `son = [2, 3, 6, 7]`.

Metot 4 döndürür; $O(n^2)$ algoritmayla aynı sonuçtur. Her öğe için ikili arama en çok $\lceil \log_2(uz + 1) \rceil$ karşılaştırma yapar. Toplam süre $O(n \log n)$, ek bellek $O(n)$'dir. Bir milyon öğelik bir dizide $O(n^2)$ algoritma yaklaşık $5 \cdot 10^{11}$ karşılaştırma yaparken bu yöntem yirmi milyon civarında karşılaştırmayla biter.

`son` dizisinin içeriği konusunda dikkatli olmak gerekir. Bu dizi bir LIS olmak zorunda değildir. 16 öğeli örnekte işlem bitince `son = [0, 1, 3, 7, 11, 15]` olur. Uzunluk doğrudur (6), ama bu sayılar dizide bu sırayla geçmez: 7, 14. konumdadır, 11 ise 13. konumda, yani 7'den öncedir. Dizinin kendisi gerekiyorsa her öğe `son`'a yazıldığında, solundaki konumda (`alt - 1`) bulunan öğenin indisi önceki indis olarak saklanır; geri okuma $O(n^2)$ sürümdeki gibi yapılır.

Sık yapılan hata

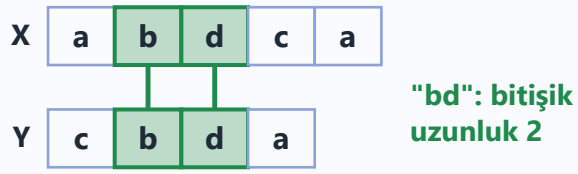
İkili aramada `son[orta] < x` yerine `son[orta] <= x` yazılır. Bu durumda eşit bir öğe, eşit sonun yerine değil, sağına yazılır ve liste uzar. Algoritma artık kesin artan değil, azalmayan (eşitliğe izin veren) en uzun alt diziyi bulur. {2, 2, 2} için doğru cevap 1, hatalı kodun cevabı 3'tür.

8.7 En uzun ortak alt dizgi

8.7.1 Problem ve bağıntı

İki dizgi verilsin. İkisinde de bitişik olarak geçen en uzun parça hangisidir? Bu probleme **en uzun ortak alt dizgi** (*longest common substring*) denir. İntihal denetiminde iki metnin birebir aynı olan en uzun bölümü, sürüm karşılaştırmada iki dosyanın ortak bloğu bu soruyla bulunur.

Örnek olarak “abdca” ve “cbda” dizgileri alınsın. Ortak alt dizgiler “a”, “b”, “d”, “c” ve “bd”'dir. En uzunu “bd”'dir ve uzunluğu 2'dir.



Şekil 8.23: “abdca” ile “cbda”nın en uzun ortak alt dizgisi “bd”'dir.

Kaba kuvvet yöntemi, birinci dizginin bütün alt dizgilerini ikinci dizgide arar. m uzunluklu bir dizginin $O(m^2)$ alt dizgisi vardır; her biri n uzunluklu dizgide aranırsa süre $O(m^2 \cdot n)$ ya da daha kötü olur. DP aynı soruyu $O(m \cdot n)$ sürede cevaplar.

Durumun seçimi LIS'tekine benzer. Ortak parçanın nerede bittiği sabitlenmelidir. Dizgiler X (uzunluk m, tablonun sütunları) ve Y (uzunluk n, tablonun satırları) olsun.

Tanım 8.11

L[i][j], *Y*'nin ilk *i* karakterinin ve *X*'in ilk *j* karakterinin *tam sonunda* biten en uzun ortak parçanın uzunluğudur.

> *Y[i − 1] = X[j − 1] ise L[i][j] = L[i − 1][j − 1] + 1*
> *aksi hâlde L[i][j] = 0*

0. satır ve 0. sütun sıfırdır. Cevap, tablodaki en büyük değerdir.

İki karakter eşitse, bir önceki karakterlerde biten ortak parça bir karakter uzar: bu çapraz hücreye 1 eklemektir. Karakterler farklıysa bu iki konumda biten hiçbir ortak parça yoktur; bitişiklik bozulur ve sayaç sıfırlanır. Cevap son hücre değildir, çünkü ortak parça dizgilerin herhangi bir yerinde bitebilir.

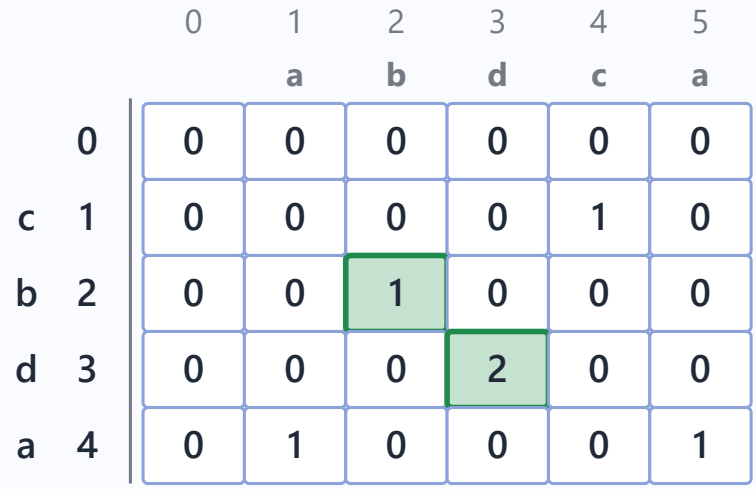
8.7.2 Tablonun doldurulması

Sütunlar *X* = “abdca”, satırlar *Y* = “cbda” olsun. Tablo *5 × 6*'dır ve satır satır doldurulur.

	j = 0	**1 (a)**	**2 (b)**	**3 (d)**	**4 (c)**	**5 (a)**
i = 0	0	0	0	0	0	0
1 (c)	0	0	0	0	1	0
2 (b)	0	0	1	0	0	0
3 (d)	0	0	0	**2**	0	0
4 (a)	0	1	0	0	0	1

Tablo 8.17: En uzun ortak alt dizgi tablosu; sütunlar X = “abdca”, satırlar Y = “cbda”.

Tablonun çoğu sıfırdır; yalnız eşleşen karakterlerin kesiştiği hücreler doludur. 1. satırda 'c' yalnız *X[3]* = 'c' ile eşleşir: *L[1][4] = L[0][3] + 1 = 1*. 2. satırda 'b', *X[1]* ile eşleşir: *L[2][2] = 1*. 3. satırda 'd', *X[2]* ile eşleşir ve çaprazındaki *L[2][2] = 1*'i uzatır: *L[3][3] = 2*. Bu, “b” ile başlayıp “d” ile biten ortak parçadır. 4. satırda 'a' iki yerde eşleşir. *L[4][1] = L[3][0] + 1 = 1*'dir. *L[4][5] = L[3][4] + 1 = 0 + 1 = 1*'dir; çaprazdaki 'd' ile 'c' eşleşmediği için parça uzamaz.



Şekil 8.24: En büyük değer *L[3][3] = 2*; çapraz geriye 2 adım gidilince *X[1..2]* = "bd" bulunur. Son hücre *L[4][5] = 1* cevap değildir.

Parçanın kendisi en büyük hücreden okunur. En büyük değer *L[3][3] = 2*'dir; parça *X*'te 3. karakterde biter ve 2 uzunluktadır. O hâlde parça *X[1..2]* = "bd"'dir. Burada ayrıca geri yürümeye gerek yoktur: bitiş konumu ve uzunluk parçayı tek başına belirler.

8.7.3 Kod

Sunumda bu problemin yalnız bağıntısı ve tablosu vardır. Aşağıdaki kod aynı bağıntıyı Java'ya çevirir. Kod, en büyük değeri ve onun *X*'teki bitiş konumunu tablo doldurulurken izler; sonunda parçayı `substring` ile döndürür.

```
String ortakAltDizgi(String x, String y) {
  int m = x.length(), n = y.length();
  int[][] L = new int[n + 1][m + 1];
  int enIyi = 0, bitis = 0;           // bitis: X'te parçanın bittiği yer
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      if (y.charAt(i - 1) == x.charAt(j - 1)) {
        L[i][j] = L[i - 1][j - 1] + 1;
        if (L[i][j] > enIyi) { enIyi = L[i][j]; bitis = j; }
      }                               // eşleşmezse L[i][j] = 0 kalır
  return x.substring(bitis - enIyi, bitis);
}
```

Eşleşmeyen hücrelere bir şey yazılmaz; Java'nın başlangıç değeri olan 0 orada kalır. `L[i][j] > enIyi` koşulu kesin büyüktür; eşit uzunlukta bir parça daha bulunursa ilk bulunan korunur. Metot "abdca" ve "cbda" için "bd", "ababc" ve "babca" için "babc" döndürür. Süre ve bellek *O(m · n)*'dir. Her hücre yalnız çaprazındaki hücreye bağlı olduğu için bellek iki satıra, dolayısıyla *O(min(m, n))*'ye indirilebilir.

Sık yapılan hata

Cevap son hücreden okunur. Son hücre, iki dizginin *ikisinin de sonunda* biten ortak parçanın uzunluğudur. Örnekte *L[4][5] = 1*'dir, ama cevap 2'dir. Ortak alt dizgide cevap tablonun en büyük değeridir.

8.8 En uzun ortak alt dizi

8.8.1 Problem ve bağıntı

Bitişiklik koşulu kaldırılınca problem değişir. İki dizide de aynı göreli sırayla geçen en uzun dizi aranır; aradaki karakterler atlanabilir. Bu probleme **en uzun ortak alt dizi** (*longest common subsequence*, LCS) denir. `diff` ve `git` iki dosyanın değişmeyen satırlarını LCS ile bulur; değişen satırlar LCS'nin dışında kalanlardır. DNA dizilerinin benzerliği ve intihal tespiti de LCS'nin uygulamalarıdır.

“abdca” ve “cbda” için LCS “bda”'dır ve uzunluğu 3'tür; ortak alt dizgi yalnız 2'ydi. Bir başka örnek: “AGGTAB” ve “GXTXAYB” dizilerinin LCS'si “GTAB”'dir ve uzunluğu 4'tür. “GTAB” iki dizide de alt dizidir, ama hiçbirinde alt dizgi değildir. LCS tek olmak zorunda değildir; uzunluğu ise tektir. “ABCBDAB” ve “BDCABA” dizilerinin üç farklı LCS'si vardır: “BCBA”, “BDAB” ve “BCAB”.

Bağıntı, iki dizinin son karakterlerine bakılarak bulunur. Durum ortak alt dizgidekiyle aynı biçimde tanımlanır, ama anlamı farklıdır: *L[i][j]*, *Y*'nin ilk *i* ve *X*'in ilk *j* karakterinin LCS uzunluğudur. Bu kez parçanın sonda bitmesi gerekmez.

Teorem 8.2 (LCS'nin optimal alt yapısı)

> *Y[i − 1] = X[j − 1] ise L[i][j] = L[i − 1][j − 1] + 1*
> *aksi hâlde L[i][j] = max(L[i − 1][j], L[i][j − 1])*

0. satır ve 0. sütun sıfırdır; cevap son hücre *L[n][m]*'dir.

İspat. Son karakterler eşit olsun; bu karakter *c* olsun. Herhangi bir ortak alt dizi *c* ile bitmiyorsa sonuna *c* eklenerek uzatılabilir. O hâlde bir LCS *c* ile biter ve *c*'den önceki kısmı, iki dizinin son karakterleri atılmış hâllerinin ortak alt dizisidir. Bu kısım daha uzun olabilseydi LCS de uzardı; demek ki o da bir LCS'dir: *L[i − 1][j − 1] + 1*. Son karakterler farklıysa bir ortak alt dizi bu iki karakterden en az birini kullanmaz; o karakter atılabilir. Atılan karakter *Y*'ninki ise cevap *L[i − 1][j]*, *X*'inki ise *L[i][j − 1]* olur. Hangisinin atılacağı bilinmediği için büyüğü alınır. □

İki problemin bağıntıları yalnız eşleşmeme durumunda ayrılır. Tablo 8.18 farkları özetler.

	Eşleşme	Eşleşmeme	Cevap
Ortak alt dizgi	çapraz + 1	0	tablonun en büyük değeri
Ortak alt dizi (LCS)	çapraz + 1	max(üst, sol)	son hücre

Tablo 8.18: Ortak alt dizgi ve ortak alt dizi (LCS) bağıntılarının karşılaştırması.

Eşleşmeme durumundaki fark, iki problemin özünü yansıtır. Ortak alt dizgide eşleşmeyen bir karakter zinciri koparır; sayaç sıfırlanır. LCS'de eşleşmeyen karakter yalnız atlanır; o ana kadar bulunan ortak alt dizi kaybolmaz ve üstten ya da soldan taşınır. Bu yüzden LCS tablosunun değerleri sağa ve aşağı doğru hiç azalmaz; son hücre en büyük değerdir.

8.8.2 Kod ve izleme

```
static int lcs(String x, String y) {
  int m = x.length(), n = y.length();
  int[][] L = new int[n + 1][m + 1];
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      if (y.charAt(i - 1) == x.charAt(j - 1))
        L[i][j] = L[i - 1][j - 1] + 1;
      else
        L[i][j] = Math.max(L[i - 1][j],
                           L[i][j - 1]);
  return L[n][m];
}
```

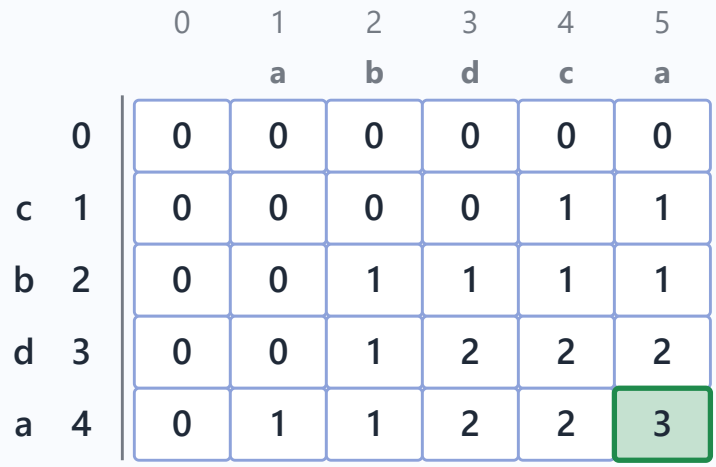
Kod, bağıntıyı iç içe iki döngüyle uygular. Dizgilerin karakterlerine `charAt(i - 1)` ve `charAt(j - 1)` ile erişilir, çünkü tablonun *i*. satırı *Y*'nin *i*. karakterini, yani 0 tabanlı *i − 1* indisli karakterini temsil eder. Aynı “abdca” ve “cbda” örneği için tablo şöyle dolar:

	j = 0	1 (a)	2 (b)	3 (d)	4 (c)	5 (a)
i = 0	0	0	0	0	0	0
1 (c)	0	0	0	0	**1**	1
2 (b)	0	0	**1**	1	1	1
3 (d)	0	0	1	**2**	2	2
4 (a)	0	**1**	1	2	2	**3**

Tablo 8.19: LCS tablosu; sütunlar X = “abdca”, satırlar Y = “cbda”. Kalın hücreler eşleşmelerdir.

Bazı hücrelerin hesabı şöyledir. *L[1][5]*: 'c' ≠ 'a'; üst 0, sol 1; değer 1'dir. 'c' karakterinin bulduğu ortak alt dizi sağa taşınmıştır. *L[3][3]*: 'd' = 'd'; çapraz *L[2][2] = 1*'e 1 eklenir ve 2 olur (“bd”). *L[4][3]*: 'a' ≠ 'd'; üst 2, sol 1; değer 2'dir. *L[4][5]*: 'a' = 'a'; çapraz *L[3][4] = 2*'ye 1 eklenir: 3. Ortak alt dizgi tablosun-

da bu çapraz hücre 0'dı; LCS'de ise “bd” sağa taşınarak 2 olmuştu. Bitişiklik koşulu kalkınca 'a', “bd”nin sonuna eklenebilmiştir.



Şekil 8.25: Tablo doldu: *L[4][5] = 3*. 20 hücre hesaplandı, her biri *O(1)*.

Süre ve bellek *O(m · n)*'dir. Yalnız uzunluk gerekiyorsa iki satır yeter ve bellek *O(min(m, n))*'ye iner. Dizinin kendisi de isteniyorsa bütün tablo saklanmalıdır.

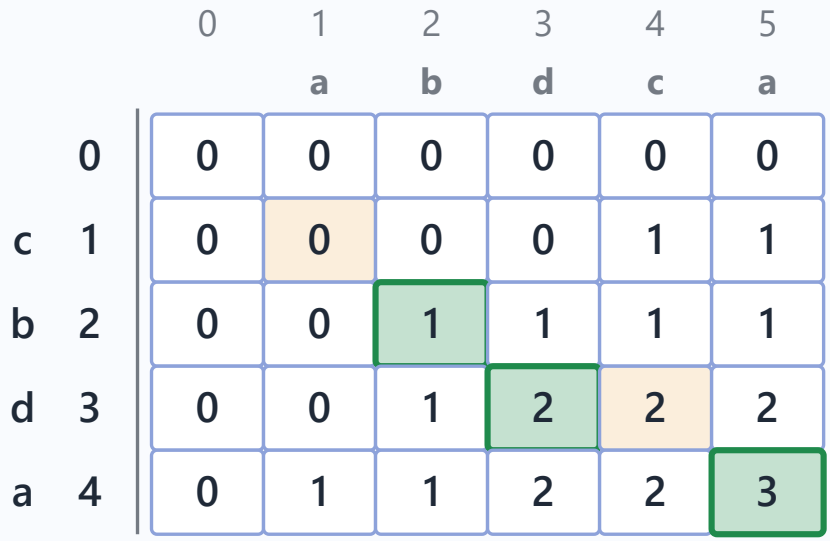
8.8.3 Geri okuma

LCS'nin kendisi son hücreden başlayarak geri okunur. Her hücrede bağıntının hangi kolunun kullanıldığına bakılır:

- Karakterler eşitse karakter LCS'ye eklenir ve çapraz hücreye geçilir.
- Eşit değilse değerin geldiği yöne gidilir: sol hücre üstten büyükse sola, değilse yukarı.
- 0. satıra ya da 0. sütuna ulaşınca durulur. Karakterler sondan başa bulunduğu için sonuç ters çevrilir.

Hücre	Karakterler	Karar	Bulunan (sondan)
L[4][5] = 3	'a' = 'a'	ekle, çapraz	a
L[3][4] = 2	'd' ≠ 'c'	sol (2) > üst (1): sola	a
L[3][3] = 2	'd' = 'd'	ekle, çapraz	da
L[2][2] = 1	'b' = 'b'	ekle, çapraz	bda
L[1][1] = 0	'c' ≠ 'a'	üst (0) ≥ sol (0): yukarı	bda
L[0][1]	—	sınır; dur	bda

Tablo 8.20: “abdca” ve “cbda” için LCS geri okuması.



Şekil 8.26: Geri okuma yolu. Yeşil hücreler eşleşmelerdir: LCS = “bda”, uzunluk 3. Geri okuma *O(m + n)* adım sürer.

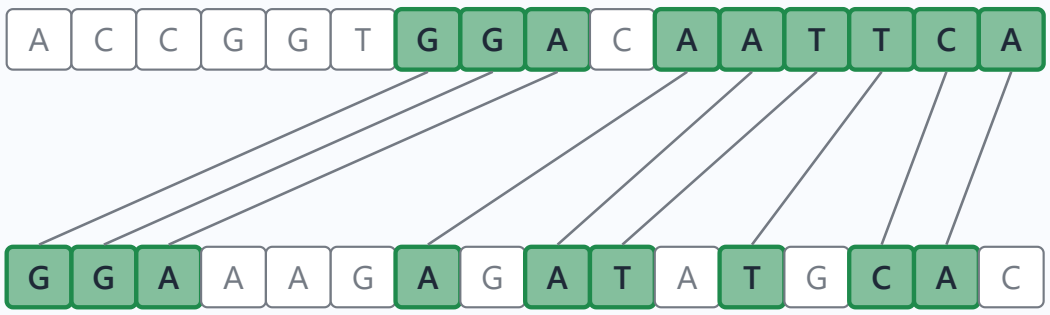
Her adımda ya satır ya sütun ya da ikisi birden bir azalır; geri okuma en çok *m + n* adım sürer. Aşağıdaki metot tabloyu doldurur ve LCS'nin kendisini döndürür.

```
String lcsDizgi(String x, String y) {
  int m = x.length(), n = y.length();
  int[][] L = new int[n + 1][m + 1];
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      L[i][j] = y.charAt(i - 1) == x.charAt(j - 1) ? L[i - 1][j - 1] + 1
              : Math.max(L[i - 1][j], L[i][j - 1]);
  StringBuilder sb = new StringBuilder();
  int i = n, j = m;
  while (i > 0 && j > 0)
    if (y.charAt(i - 1) == x.charAt(j - 1)) {   // eşleşme: çapraz
      sb.append(x.charAt(j - 1)); i--; j--;
    } else if (L[i][j - 1] > L[i - 1][j]) j--;  // sol büyükse sola
    else i--;                                   // değilse yukarı
  return sb.reverse().toString();
}
```

Metot “abdca” ve “cbda” için “bda”, “AGGTAB” ve “GXTXAYB” için “GTAB” döndürür. Eşitlikte yukarı gitme kuralı, birden çok LCS varsa hangisinin bulunacağını belirler. “ABCBDAB” (sütunlar) ve “BDCABA” (satırlar) için metot “BDAB”'yi bulur; kural değiştirilip eşitlikte sola gidilirse başka bir LCS bulunabilir.

8.8.4 DNA örneği

Biyoinformatikte iki DNA dizisinin ne kadar benzediği LCS'nin uzunluğuyla ölçülebilir. Şekil 8.27'deki diziler ACCGGTGGACAATTCA ve GGAAAGAGATATGCAC'tir. Her ikisi de 16 bazdan oluşur.



Şekil 8.27: İki DNA dizisi ve bir LCS: GGAAATTCA, uzunluk 9. Eşleşen bazları birleştiren çizgiler kesişmez.

Şekildeki çizgiler bir ortak alt dizinin görsel karşılığıdır. Eşleşen karakterleri birleştiren çizgiler kesişmez, çünkü göreli sıra iki dizide de korunur. Kesişmeyen çizgilerle kurulan her eşleştirme bir ortak alt dizidir. LCS, bu biçimde çizilebilecek en çok çizgili eşleştirmedir. DP sonucu GGAAATTCA'dır ve uzunluğu 9'dur. Diziler yer değiştirince geri okuma GGGGAATCA gibi başka bir LCS bulabilir; uzunluk yine 9'dur.

Göz kararı eşleştirme yanıltıcıdır. Örneğin AAAATTCA da geçerli bir ortak alt dizidir: ilk dizide A'lar 0, 8, 10, 11. konumlarda, ardından T, T, C, A gelir; ikinci dizide de aynı sıra bulunur. Ama uzunluğu 8'dir ve en uzun değildir. Doğru cevabı ancak bütün seçenekleri dolaylı olarak deneyen DP tablosu verir. *16 × 16* bir tablo 256 hücreyle bu işi bitirir; olası bütün alt dizileri denemek ise $2^{16} = 65.536$ alt diziyi öteki dizide aramak demektir.

8.9 Diğer klasik DP problemleri

Notta incelenen problemler bir kalıp izler: durum birkaç tam sayıyla tanımlanır, bağıntı son kararın seçeneklerini dener, tablo uygun sırayla doldurulur. Aynı kalıp başka birçok klasik problemi çözer. Tablo 8.21 bunlardan yedisini özetler; ardından dördü küçük örneklerle ele alınır.

Problem	Durum ve bağıntı	Süre
Düzenleme uzaklığı (*edit distance*)	$D[i][j] = \min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + [a_i \neq b_j])$; $D[i][0] = i$, $D[0][j] = j$	$O(m \cdot n)$
Para üstü (*coin change*)	en az para: $P[x] = 1 + \min P[x - c]$ $(c \le x)$; $P[0] = 0$	$O(k \cdot X)$
En büyük toplamlı alt dizi	$V(i) = \max(a[i], V(i-1) + a[i])$ (Kadane); cevap $\max V(i)$	$O(n)$
Matris zinciri çarpımı	$M[i][j] = \min \{ M[i][k] + M[k+1][j] + p_{i-1} \cdot p_k \cdot p_j : i \le k < j \}$	$O(n^3)$
Gezgin satıcı (Held–Karp)	$dp[S][i]$: S kümesini gezip i'de biten en kısa yol; $dp[S][i] = \min dp[S - i][j] + d(j, i)$	$O(n^2 \cdot 2^n)$

Problem	Durum ve bağıntı	Süre
Bellman–Ford	$d_k(v)$: en çok k kenarlı en kısa yol; $d_k(v) = min(d_{k-1}(v), d_{k-1}(u) + w(u, v))$	$O(V \cdot E)$
Optimal ikili arama ağacı	$e[i][j] = min\ \{\ e[i][r-1] + e[r+1][j] : i \le r \le j\ \} + w(i, j)$	$O(n^3)$

Tablo 8.21: Diğer klasik dinamik programlama problemleri.

8.9.1 Düzenleme uzaklığı

İki dizgi arasındaki **düzenleme uzaklığı** (Levenshtein uzaklığı), birini ötekine çevirmek için gereken en az tek karakterlik işlem sayısıdır. İşlemler üç türlüdür: bir karakter silmek, bir karakter eklemek ve bir karakteri başka bir karakterle değiştirmek. Yazım denetleyicisi yanlış yazılmış bir kelimeye uzaklığı küçük olan sözlük kelimelerini önerir.

D[i][j], *a*'nın ilk *i* karakterini *b*'nin ilk *j* karakterine çevirmenin en az maliyetidir. Son karakter için üç seçenek vardır. *a*'nın son karakteri silinir: *D[i − 1][j] + 1*. *b*'nin son karakteri eklenir: *D[i][j − 1] + 1*. Son karakterler eşleştirilir: eşitlerse ücretsiz, değilse bir değiştirme ile, $D[i-1][j-1] + [a_i \neq b_j]$. Köşeli ayraç, koşul doğruysa 1, yanlışsa 0 demektir. Taban durumları açıktır: boş dizgiden *j* karakterlik dizgiye *j* eklemeyle, *i* karakterlik dizgiden boş dizgiye *i* silmeyle varılır.

```
int duzenlemeUzakligi(String a, String b) {
  int m = a.length(), n = b.length();
  int[][] D = new int[m + 1][n + 1];
  for (int i = 0; i <= m; i++) D[i][0] = i;     // i silme
  for (int j = 0; j <= n; j++) D[0][j] = j;     // j ekleme
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++) {
      int fark = a.charAt(i - 1) == b.charAt(j - 1) ? 0 : 1;
      D[i][j] = Math.min(Math.min(D[i - 1][j] + 1,   // sil
                                  D[i][j - 1] + 1),  // ekle
                         D[i - 1][j - 1] + fark);    // değiştir / aynı
    }
  return D[m][n];
}
```

Örnek 8.7

"kitap" ile "kayak" arasındaki düzenleme uzaklığı kaçtır?

	""	k	a	y	a	k
""	0	1	2	3	4	5
k	1	0	1	2	3	4
i	2	1	1	2	3	4
t	3	2	2	2	3	4
a	4	3	2	3	2	3

	""	k	a	y	a	k
p	5	4	3	3	3	**3**

Sonuç *D[5][5] = 3*'tür. Köşegen boyunca okunursa k ve a karakterleri eşleşir; i → a, t → y ve p → k değiştirilir. Örneğin *D[2][2]*: 'i' ≠ 'a'; seçenekler *D[1][2] + 1 = 2*, *D[2][1] + 1 = 2* ve *D[1][1] + 1 = 1*'dir; en küçüğü 1'dir. *D[4][4]*: 'a' = 'a'; çapraz *D[3][3] = 2* ücretsiz taşınır.

Düzenleme uzaklığı ile LCS yakından ilişkilidir. Yalnız ekleme ve silmeye izin verilirse en az işlem sayısı $m + n - 2 \cdot LCS$ olur: LCS'de olmayan karakterler bir dizgiden silinir, ötekinden eklenir.

8.9.2 Para üstü

Para birimleri $c_1, \ldots, c_k$ ve bir X tutarı verilsin. Bu problemin iki ünlü sorusu vardır: tutar en az kaç parayla ödenir ve kaç farklı biçimde ödenebilir? Her paradan sınırsız sayıda bulunduğu için ikisi de sınırsız sırt çantasının akrabasıdır.

En az para. *P[x]*, x tutarını ödemek için en az para sayısıdır. Son verilen para c ise geriye $x - c$ kalır: $P[x] = 1 + \min P[x - c]$ ($c \le x$), $P[0] = 0$. Küçük tutarlar önce hazır olmalıdır; x 1'den X'e doldurulur. Tablo $X + 1$ hücrelidir, her hücrede k para denenir: $O(k \cdot X)$ süre, $O(X)$ bellek. Bu da sözde-polinom bir süredir.

```
int enAzPara(int[] para, int X) {
  int[] P = new int[X + 1];            // P[0] = 0
  for (int x = 1; x <= X; x++) {
    P[x] = Integer.MAX_VALUE;
    for (int c : para)
      if (c <= x && P[x - c] != Integer.MAX_VALUE)
        P[x] = Math.min(P[x], P[x - c] + 1);
  }
  return P[X];                         // MAX_VALUE: ödenemez
}

long yolSayisi(int[] para, int X) {
  long[] d = new long[X + 1];
  d[0] = 1;                            // 0 tutarının tek yolu: hiç para
  for (int c : para)                   // dış döngü paralar üzerinde
    for (int x = c; x <= X; x++)
      d[x] += d[x - c];
  return d[X];
}
```

Örnek 8.8

Paralar {1, 3, 4}, tutar 6. En az kaç para gerekir? Açgözlü yöntem kaç para kullanır?

Tablo: *P = [0, 1, 2, 1, 1, 2, 2]*. Örneğin *P[5] = 1 + min(P[4], P[2], P[1]) = 1 + 1 = 2* (4 + 1). Son hücre: *P[6] = 1 + min(P[5], P[3], P[2]) = 1 + min(2, 1, 2) = 2*; ödeme 3 + 3'tür. Açgözlü yöntem önce en büyük parayı alır: 4, sonra 1, sonra 1. Üç para kullanır ve optimal değildir. Günlük kullanılan para sistemlerinin çoğunda açgözlü yöntem doğru sonuç verir, ama bu her para sistemi için geçerli değildir.

Yol sayısı. *d[x]*, *x* tutarını ödemenin farklı biçimlerinin sayısıdır; *d[0] = 1*'dir (hiç para vermemek). Burada döngülerin sırası belirleyicidir. Dış döngü paralar, iç döngü tutarlar üzerinde olursa her biçim bir kez sayılır: paralar sabit bir sırayla eklenir ve "2 + 1" ile "1 + 2" aynı kabul edilir. Döngüler yer değiştirirse sıralı diziler sayılır. {1, 2, 5} ile 5 tutarı için doğru cevap 4'tür: 5, 2 + 2 + 1, 2 + 1 + 1 + 1 ve 1 + 1 + 1 + 1 + 1. Döngüler ters kurulsa sonuç 9 olurdu.

Sık yapılan hata

Para üstünde açgözlü seçimin her zaman doğru olduğu sanılır. {1, 3, 4} ile 6 tutarı bunun karşı örneğidir: açgözlü 3 para, DP 2 para verir. Sırt çantasında olduğu gibi, açgözlü seçimin doğruluğu ayrıca kanıtlanmadıkça DP gerekir.

8.9.3 En büyük toplamlı alt dizi

Bir dizide toplamı en büyük olan bitişik parça aranır. Bağıntı optimal alt yapı tartışılırken yazılmıştı: *V(i) = max(a[i], V(i − 1) + a[i])*. Her adımda iki seçenek vardır: önceki parçayı uzatmak ya da *a[i]* ile yeniden başlamak. Önceki parçanın toplamı negatifse onu taşımak zarardır; yeniden başlamak daha iyidir. *V(i)* yalnız *V(i − 1)*'e bağlı olduğu için tek değişken yeter.

```
int kadane(int[] a) {
  int V = a[0], enIyi = a[0];
  for (int i = 1; i < a.length; i++) {
    V = Math.max(a[i], V + a[i]);   // yeniden başla ya da uzat
    enIyi = Math.max(enIyi, V);
  }
  return enIyi;
}
```

Örnek 8.9

[−2, 1, −3, 4, −1, 2, 1, −5, 4] dizisinin en büyük toplamlı alt dizisi nedir?

V değerleri sırasıyla −2, 1, −2, 4, 3, 5, 6, 1, 5'tir. 1 indisinde önceki toplam −2 olduğu için yeniden başlanır: *V = max(1, −1) = 1*. 3 indisinde önceki toplam −2'dir; 4 ile yeniden başlanır. Sonra parça uzar: *4 − 1 = 3*, *3 + 2 = 5*, *5 + 1 = 6*. −5 parçayı 1'e düşürür ama sıfırın altına indirmez. En büyük değer 6'dır: *[4, −1, 2, 1]*. Süre *O(n)*, bellek *O(1)*'dir.

8.9.4 Matris zinciri çarpımı

$p \times q$ boyutlu bir matrisle $q \times r$ boyutlu bir matrisin olağan çarpımı $p \cdot q \cdot r$ skaler çarpma ister. Matris çarpımı birleşmelidir: $(AB)C = A(BC)$. Sonuç aynıdır, ama maliyet parantezlemeye göre çok farklı olabilir. Problem, bir matris zincirini en az skaler çarpmayla hesaplayan parantezlemeyi bulmaktır.

A_i matrisi $p_{i-1} \times p_i$ boyutlu olsun. $M[i][j]$, $A_i \dots A_j$ zincirinin en az maliyetidir. Bu zincirin son çarpımı bir k noktasında ikiye ayrılır: önce $A_i \dots A_k$ ve $A_{k+1} \dots A_j$ hesaplanır, sonra $p_{i-1} \times p_k$ ve $p_k \times p_j$ boyutlu iki sonuç çarpılır. Bütün k'ler denenir. Burada durumlar zincir uzunluğuna göre doldurulur: kısa zincirler önce hazır olmalıdır.

```
long matrisZinciri(int[] p) {          // A_i matrisi p[i-1] × p[i]
  int n = p.length - 1;
  long[][] M = new long[n + 1][n + 1];
  for (int uz = 2; uz <= n; uz++)      // zincir uzunluğu
    for (int i = 1; i + uz - 1 <= n; i++) {
      int j = i + uz - 1;
      M[i][j] = Long.MAX_VALUE;
      for (int k = i; k < j; k++)
        M[i][j] = Math.min(M[i][j],
            M[i][k] + M[k + 1][j] + (long) p[i - 1] * p[k] * p[j]);
    }
  return M[1][n];
}
```

Örnek 8.10

A_1 10 × 30, A_2 30 × 5, A_3 5 × 60 boyutludur ($p = 10, 30, 5, 60$). En iyi parantezleme hangisidir?

$(A_1A_2)A_3$: $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 1500 + 3000 = 4500$ çarpma. $A_1(A_2A_3)$: $30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 9000 + 18.000 = 27.000$ çarpma. İlki altı kat daha ucuzdur; metot 4500 döndürür.

Durum sayısı $O(n^2)$'dir ($i \le j$ çiftleri); her durumda $O(n)$ bölme noktası denenir. Süre $O(n^3)$'tür. Bütün parantezlemeleri tek tek denemek ise Catalan sayıları kadar seçenek demektir: 10 matris için 4862, 20 matris için 1,7 milyardan fazla.

8.9.5 Gezgin satıcı, Bellman–Ford ve optimal ikili arama ağacı

Tablodaki son üç problem, durumun farklı biçimler alabileceğini gösterir: bir küme ile bir düğüm, bir düğüm ile bir kenar sayısı, bir anahtar aralığı.

- **Gezgin satıcı (Held–Karp).** Durum bir küme ve bir düğümdür: $dp[S][i]$, başlangıçtan çıkıp S kümesindeki düğümlerin hepsini gezen ve i'de biten en kısa yoldur. Küme bir bit maskesiyle saklanır. $2^n \cdot n$ durum vardır, her biri $O(n)$ önceki düğüm dener: $O(n^2 \cdot 2^n)$. Bu süre hâlâ üsteldir, ama bütün

turları denemenin $(n-1)!$'inden çok daha küçüktür. 20 düğüm için $20^2 \cdot 2^{20} \approx 4{,}2 \cdot 10^8$'e karşılık $19! \approx 1{,}2 \cdot 10^{17}$.

- **Bellman–Ford.** Durum bir düğüm ve bir kenar sayısıdır: $d_k(v)$, en çok k kenar kullanan en kısa yoldur. Her turda bütün kenarlar gevşetilir; $V-1$ tur yeter. Tablo tek boyuta indirilir ve süre $O(V \cdot E)$ olur.
- **Optimal ikili arama ağacı.** Anahtarların erişim olasılıkları biliniyorsa beklenen arama maliyeti en küçük olan ağaç aranır. Durum bir anahtar aralığıdır; bağıntı aralık için bütün kök seçeneklerini dener. Yapı matris zinciriyle aynıdır ve süre $O(n^3)$'tür.

8.10 Sık yapılan hatalar

Dinamik programlama hataları çoğunlukla derleme hatası vermez. Kod çalışır, bir sayı döndürür ve sayı yanlıştır. Aşağıdaki hatalar, sunumdaki listeyle birlikte not boyunca karşılaşılanları toplar.

Sık yapılan hata

Taban durumlarını unutmak. $K[0][c]$, $dp[i][0]$ ya da fib(0) ve fib(1) yanlış başlatılırsa bütün tablo kayar. Alt küme toplamında sütun 0 doğru yapılmazsa her cevap “hayır” olur; para üstünde $P[0] = 0$ yerine sonsuz yazılırsa hiçbir tutar ödenemez.

Sık yapılan hata

İndis kayması. Tablo $n+1$ boyutludur ama dizi n öğelidir: i. satır `v[i - 1]`'i, LCS tablosunun i. satırı `charAt(i - 1)`'i kullanır. `v[i]` yazmak her satırı bir sonraki öğeye kaydırır ve son satırda dizinin dışına taşar.

Sık yapılan hata

Yanlış döngü yönü. Tek boyutlu 0/1 sırt çantasında kapasiteyi küçükten büyüğe gezmek eşyayı birden çok kez alır; sonuç sınırsız sırt çantasının cevabı olur. Para üstünde yol sayısı için dış döngünün tutar üzerinde kurulması, aynı ödemenin farklı sıralarını ayrı ayrı sayar.

Sık yapılan hata

Alt dizgi ile alt diziyi karıştırmak. Ortak alt dizgide eşleşmeme 0 yazar ve cevap tablonun en büyük değeridir. LCS'de eşleşmeme $max(üst, sol)$ yazar ve cevap son hücredir. LIS'te de cevap son hücre değil, en büyük $dp[i]$'dir.

Sık yapılan hata

Açgözlü seçimi yeterli sanmak. 0/1 sırt çantasında (60, 100, 120 ₺ örneği) ve para üstünde ({1, 3, 4} ile 6) açgözlü seçim optimal değildir. Açgözlü yöntem ancak doğruluğu kanıtlanmışsa kullanılır.

Sık yapılan hata

***O(n · W)*'yi polinom sanmak.** *W* sayısal bir değerdir; girdi boyutu *log W* bittir. Büyük *W*'de tablo kurulamaz.

Sık yapılan hata

Önbellekte "hesaplanmadı" işaretini yanlış seçmek. `int` tablosunda 0 hem "boş" hem "cevap 0" anlamına gelir; cevabı 0 olan durumlar her seferinde yeniden hesaplanır. `Integer` ve `null`, ya da −1 ile doldurulmuş bir tablo bu karışıklığı önler.

Sık yapılan hata

Bellek azaltırken geri okumayı kaybetmek. Tek boyutlu tablo optimal değeri verir, ama seçimleri geri okumak için gereken satırlar silinmiştir. Seçimler de isteniyorsa ya bütün tablo ya da ayrı bir karar tablosu saklanır.

8.11 Özet

- Dinamik programlama = örtüşen alt problemler + optimal alt yapı. Her alt problem bir kez çözülür, sonucu tabloda saklanır.
- Tasarım adımları: optimal çözümün yapısı, durum ve bağıntı (taban durumlarıyla), hesaplama sırası, tablo boyutu, tablonun doldurulması, çözümün geri okunması.
- Toplam süre ≈ durum sayısı × durum başına iş. Bu kural notta geçen bütün karmaşıklıkları açıklar.
- Bellekleme yukarıdan aşağı çalışır ve yalnız gereken durumları çözer; tablolama aşağıdan yukarı çalışır, çağrı maliyeti yoktur ve bellek azaltmaya uygundur. Asimptotik süreleri aynıdır.
- Açgözlü yaklaşım her adımda tek seçeneğe bakar; ancak açgözlü seçim özelliği kanıtlanmışsa doğrudur. Böl ve fethette alt problemler bağımsızdır ve tablo gerekmez.

Problem	Durum / bağıntı	Zaman · bellek
Fibonacci	$F(n) = F(n-1) + F(n-2)$; bellekleme ya da tablo	$O(n) \cdot O(1)$
0/1 sırt çantası	$K[i][c] = \max(K[i-1][c], v_i + K[i-1][c-w_i])$	$O(nW) \cdot O(W)$

Problem	Durum / bağıntı	Zaman · bellek
Sınırsız sırt çantası	$dp[i][c] = max(dp[i-1][c], p_i + dp[i][c-w_i])$	$O(nW) \cdot O(W)$
Alt küme toplamı	$dp[i][j] = dp[i-1][j] \vee dp[i-1][j-a_i]$	$O(n \cdot hedef)$
LIS	$dp[i] = 1 + max\ dp[j]$ ($j < i, a[j] < a[i]$); ya da aktif listeler	$O(n^2)$ / $O(n \log n)$
Ortak alt dizgi	eşitse çapraz + 1, değilse 0; cevap tablonun en büyüğü	$O(mn)$
LCS	eşitse çapraz + 1, değilse max(üst, sol); cevap son hücre	$O(mn)$

Tablo 8.22: Notta incelenen problemlerin özeti.

- 0/1 sırt çantası, alt küme toplamı ve para üstü sözde-polinomdur: süre girdideki sayının değerine bağlıdır, bit uzunluğuna değil.
- Tek boyutlu tabloda döngü yönü problemi belirler: büyükten küçüğe 0/1, küçükten büyüğe sınırsız.
- Çözümün kendisi tablodan geri okunur: sırt çantasında değer değişen satırlar, LCS'de çapraz adımlar, LIS'te önceki indis zinciri.

8.12 Kavram listesi

dinamik programlama *dynamic programming (DP)* · alt problem *subproblem* · örtüşen alt problemler *overlapping subproblems* · optimal alt yapı *optimal substructure* · bellekleme *memoization* · tablolama *tabulation* · yukarıdan aşağı / aşağıdan yukarı *top-down / bottom-up* · geri okuma *traceback* · açgözlü seçim özelliği *greedy-choice property* · 0/1 sırt çantası *0/1 knapsack* · kesirli sırt çantası *fractional knapsack* · sınırsız sırt çantası *unbounded knapsack* · sözde-polinom *pseudo-polynomial* · alt küme toplamı *subset sum* · en uzun artan alt dizi *longest increasing subsequence (LIS)* · en uzun ortak alt dizgi *longest common substring* · en uzun ortak alt dizi *longest common subsequence (LCS)* · düzenleme uzaklığı *edit distance* · para üstü *coin change* · matris zinciri çarpımı *matrix chain multiplication*

8.13 Kendini Dene

Sorular 2 ve 3 aynı eşyaları kullanır: (değer, ağırlık) biçiminde (12, 2), (17, 3), (30, 5); kapasite 6.

1. Düz özyinelemeli fib(6) çağrısında fib(2) kaç kez çağrılır? Bellekleme ile fib(2) kaç kez hesaplanır,

toplam çağrı sayısı kaçtır?

Cevap

fib(n) ağacında fib(k), $F(n - k + 1)$ kez çağrılır: fib(2) için $F(5) = 5$ kez. Düz özyineleme toplam $2 \cdot F(7) - 1 = 25$ çağrı yapar. Bellekleme ile fib(2) bir kez hesaplanır, ikinci çağrısı tablodan döner; toplam çağrı $2 \cdot 6 - 1 = 11$'dir.

2. 0/1 sırt çantası tablosunun son satırı nedir? En büyük değer kaçtır, hangi eşyalar seçilir?

Cevap

Satırlar: *[0, 0, 12, 12, 12, 12, 12]*, *[0, 0, 12, 17, 17, 29, 29]*, *[0, 0, 12, 17, 17, 30, 30]*. Cevap $K[3][6] = 30$'dur. Geri okuma: $30 \neq K[2][6] = 29$, 3. eşya alınır ve $c = 1$ olur; 1 kg'a başka eşya sığmaz. Yalnız 30'luk eşya seçilir. 12 + 17 = 29 bir eksik kalır.

3. Aynı eşyalardan sınırsız sayıda alınabilseydi cevap ne olurdu? Tek boyutlu 0/1 kodunda iç döngü küçükten büyüğe yazılsaydı hangi sayı dönerdi?

Cevap

Sınırsız: *dp = [0, 0, 12, 17, 24, 30, 36]*; cevap 36'dır, üç tane 12'lik (2 kg) eşya. Küçükten büyüğe gezilen tek boyutlu 0/1 kodu da 36 döndürür, çünkü o kod artık sınırsız sırt çantasını çözer.

4. 0/1 sırt çantası DP'sinde W iki katına çıkarsa süre nasıl değişir? W'nin ikili gösterimine bir bit eklenirse ne olur?

Cevap

İki durumda da tablo genişliği yaklaşık iki katına çıkar, süre de iki katına çıkar. Bir bitlik artış girdinin boyutunu yalnız bir bit büyütür; süre ise iki katına çıkar. b bitlik bir W için süre $O(n \cdot 2^b)$'dir; sözde-polinom bu anlamdadır.

5. {2, 3, 7, 8, 10} kümesinde toplamı 11 olan bir alt küme var mıdır? Tablonun son satırında hangi toplamlar ulaşılabilirdir?

Cevap

Evet: {3, 8}; bu tek çözümdür. Son satırda ulaşılabilir toplamlar 0, 2, 3, 5, 7, 8, 9, 10 ve 11'dir (1, 4 ve 6 elde edilemez). Geri okuma *dp[5][11]*'den başlar: 10 olmadan da 11 kurulabildiği için 10 seçilmez; 8 seçilir ($j = 3$); 7 seçilmez; 3 seçilir ($j = 0$).

6. [3, 10, 2, 1, 20, 4, 6, 7] dizisinde $O(n^2)$ algoritmanın dp dizisi nedir? `lisHizli` bitince `son` dizisi

nedir?

Cevap

dp = [1, 2, 1, 1, 3, 2, 3, 4]; LIS uzunluğu 4'tür, örneğin 3, 4, 6, 7. `son` sırasıyla [3], [3, 10], [2, 10], [1, 10], [1, 10, 20], [1, 4, 20], [1, 4, 6], [1, 4, 6, 7] olur. Bu örnekte son dizi bir LIS'tir (1, 4, 6, 7), ama bu bir rastlantıdır; dizinin LIS olması gerekmez.

7. “kalem” ve “kelam” için en uzun ortak alt dizgi ve LCS uzunlukları kaçtır?

Cevap

Ortak alt dizgi 1'dir: iki harfli parçaların (ka, al, le, em ile ke, el, la, am) hiçbiri ortak değildir. LCS 3'tür; “kam”, “kem” ve “klm” üç farklı LCS'dir. Bitişiklik koşulunun kalkması uzunluğu üç katına çıkarır.

8. “kedi” ile “kadın” arasındaki düzenleme uzaklığı kaçtır?

Cevap

3'tür: e → a ve i → ı değiştirilir, sona n eklenir. Tablonun son satırı *[4, 3, 3, 2, 2, 3]*'tür; *D[4][5] = 3*.

9. Paralar {1, 5, 6, 9}, tutar 11. En az kaç para gerekir? Açgözlü yöntem kaç para kullanır?

Cevap

DP: 2 para (5 + 6). Açgözlü: 9 + 1 + 1, 3 para. Bu para sisteminde de açgözlü seçim optimal değildir.

10. A_1 5 × 10, A_2 10 × 3, A_3 3 × 12 boyutludur. En az kaç skaler çarpma gerekir?

Cevap

$(A_1A_2)A_3 = 5 \cdot 10 \cdot 3 + 5 \cdot 3 \cdot 12 = 150 + 180 = 330$. $A_1(A_2A_3) = 10 \cdot 3 \cdot 12 + 5 \cdot 10 \cdot 12 = 360 + 600 = 960$. En az 330'dur.

8.14 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 14: çubuk kesme, matris zinciri, LCS ve optimal ikili arama ağacı üzerinden DP'nin ilkelerini kanıtlarıyla verir.
- J. Kleinberg ve É. Tardos, *Algorithm Design* (2005), Bölüm 6: ağırlıklı aralık çizelgeleme, sırt çantası, dizi hizalama ve Bellman–Ford örnekleriyle durum seçimini adım adım tartışır.

- S. Dasgupta, C. Papadimitriou ve U. Vazirani, *Algorithms* (2006), Bölüm 6: DP'yi yönlü çevrimsiz çizgede en kısa yol olarak yorumlar; LIS, düzenleme uzaklığı, sırt çantası ve gezgin satıcıyı kısa ve açık biçimde işler.
- R. Bellman, *Dynamic Programming* (1957): yöntemin ve optimallik ilkesinin ilk kapsamlı anlatımıdır.

8.15 Kaynaklar

1. R. Bellman, *Dynamic Programming*, Princeton University Press, 1957.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 14: Dynamic Programming).
3. S. Dasgupta, C. Papadimitriou, U. Vazirani, *Algorithms*, McGraw-Hill, 2006 (Bölüm 6).
4. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005 (Bölüm 6).

KISIM IV

Zor Problemler

Bölüm 9–11

KISIM IV · BÖLÜM 9

Geri İzleme Algoritmaları

Bazı problemler için hızlı ve kesin bir algoritma bilinmez. Sekiz veziri birbirini tehdit etmeyecek biçimde tahtaya dizmek, bir Sudoku'yu doldurmak ya da bir haritayı üç renkle boyamak bu türdendir. Bu problemlerde çözüm, seçimler denenerek aranır. Geri izleme bu aramayı düzenli yürüten bir yöntemdir. Çözüm adım adım kurulur. Bir adım kuralları bozduğu anda son karar geri alınır ve sıradaki seçenek denenir. Not önce zor problemlerin hangi sınıflara ayrıldığını açıklar. Ardından geri izlemenin genel şablonu, durum uzayı ağacı ve budama kavramları labirent ve permütasyon örnekleriyle kurulur. n-vezir problemi Java koduyla ve satır satır izlemesiyle ayrıntılı işlenir; en az çakışma sezgiseli aynı probleme bambaşka bir yoldan yaklaşır. Sudoku, çizge boyama, Hamilton yolu, kelime arama ve alt küme toplamı şablonun başka uygulamalarıdır. Not, karmaşıklık analizi, budamayı güçlendiren sezgiseller ve geri izlemenin öteki tasarım yöntemleriyle karşılaştırılmasıyla biter.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum09/src`

ÖĞRENME ÇIKTILARI

Kavram
Geri izlemeyi kısmi çözüm, aday, kısıt ve budama kavramlarıyla açıklamak; küçük bir problemin durum uzayı ağacını çizmek; P, NP, NP-zor ve NP-tam sınıflarını ayırt etmek.

Şablon
"Seç → keşfet → geri al" özyinelemeli şablonunu Java ile yazmak ve yeni bir probleme uyarlamak. Tek çözümde duran sürümü bütün çözümleri sayan sürümden ayırt etmek.

Uygulamalar
n-vezir, Sudoku, labirent, çizge boyama, Hamilton yolu, kelime arama ve alt küme toplamı problemlerini adım adım çözmek ve kodlarını izlemek.

Analiz
En kötü durumun neden üstel olduğunu göstermek; budamanın, O(1) kısıt testinin ve MRV ile en az çakışma gibi sezgisellerin etkisini sayılarla değerlendirmek.

9.1 Zor problemler: P, NP ve NP-zor

Dersin bu noktasına kadar görülen algoritmaların çoğu hızlıdır. Bir diziyi birleştirmeli sıralama $O(n \log n)$ sürede sıralar. Dijkstra algoritması en kısa yolları, Kruskal algoritması minimum kapsayan ağacı polinom sürede bulur. Ama her problem bu kadar uysal değildir. Bir çizgenin her düğümünden tam bir kez geçen bir yol var mı? Bir haritanın ülkeleri, komşu ülkeler farklı renkte olacak biçimde üç renkle boyanabilir mi? Bu soruların ikisi için de bugüne kadar polinom süreli bir algoritma bulunamamıştır. Bulunamayacağı da kanıtlanamamıştır. Geri izleme bu türden problemler için geliştirilmiş bir arama yöntemidir. Yöntemin neden gerektiğini görmek için önce problemlerin zorluk sınıflarına kısaca bakmak yararlıdır.

9.1.1 Karar problemleri ve polinom zaman

Zorluk sınıfları çoğunlukla **karar problemleri** (*decision problems*) üzerinden tanımlanır. Karar probleminin cevabı yalnız "evet" ya da "hayır"dır. "Bu çizgede Hamilton yolu var mı?" bir karar problemidir. "Bu çizgedeki en kısa Hamilton turunun uzunluğu nedir?" ise bir **en iyileme problemidir** (*optimization problem*). En iyileme problemi çoğu zaman bir karar problemine çevrilebilir: "Uzunluğu en çok 100 olan bir tur var mı?" Karar sürümü hızlı çözülebilseydi, eşik değeri ikili aramayla daraltılarak en iyi değer de bulunabilirdi. Bu yüzden karar sürümünün zorluğu, en iyileme sürümünün zorluğu hakkında da bilgi verir.

Bir algoritmanın süresi bir k sabiti için $O(n^k)$ ise algoritma **polinom zamanlıdır** (*polynomial time*). n burada girdinin boyudur. $O(n)$, $O(n \log n)$ ve $O(n^3)$ polinom sürelerdir. $O(2^n)$ ve $O(n!)$ ise değildir. Fark küçük girdilerde göze çarpmaz, ama girdi büyüdükçe uçurum açılır. $n = 60$ için $n^3 = 216.000$ işlem bir bilgisayar için bir an sürer. $2^{60} \approx 1{,}15 \cdot 10^{18}$ işlem ise saniyede bir milyar işlem yapan bir makinede otuz yılı aşar. Polinom süre bu yüzden "pratikte çözülebilir" anlamında kullanılır.

9.1.2 P ve NP

> **Tanım 9.1**
>
> **P**, polinom zamanda *çözülebilen* karar problemlerinin sınıfıdır. **NP** ise önerilen bir çözümü polinom zamanda *doğrulanabilen* karar problemlerinin sınıfıdır. Cevap "evet" olduğunda bunu gösteren bir tanık (*certificate*) vardır. Tanığın doğruluğu polinom sürede denetlenebilir.

P sınıfına iki örnek iki ağırlıklı çizge sorusudur: "s'den t'ye uzunluğu en çok 50 olan bir yol var mı?" ve "ağırlığı en çok 30 olan bir kapsayan ağaç var mı?" Birinci soruyu Dijkstra, ikincisini Kruskal algoritması polinom sürede cevaplar.

NP'nin tanımı çözmekle değil, denetlemekle ilgilidir. Hamilton yolu problemi buna iyi bir örnektir. Bir çizgede Hamilton yolu bulmak zordur. Ama biri bir düğüm sırası önerirse, sıranın her düğümü bir kez içerip içermediği ve ardışık düğümlerin komşu olup olmadığı $O(n)$ adımda denetlenir. Sudoku da aynı

türdendir. Doldurulmuş bir tabloyu denetlemek kolaydır: her satır, sütun ve kutu bir kez taranır. Boş bir tabloyu doldurmak ise arama gerektirir.

Polinom sürede çözülebilen bir problemin çözümü polinom sürede doğrulanabilir de. Doğrulayıcı, önerilen tanığı yok sayıp problemi baştan çözebilir. Bu yüzden P, NP'nin içindedir. Asıl soru ters yöndedir: doğrulaması kolay her problemin çözümü de kolay mıdır? Bu soru $P = NP$ sorusu olarak anılır ve bilgisayar biliminin en önemli açık problemidir. Araştırmacıların çoğu $P \neq NP$ olduğuna inanır, ama bu henüz kanıtlanmamıştır.

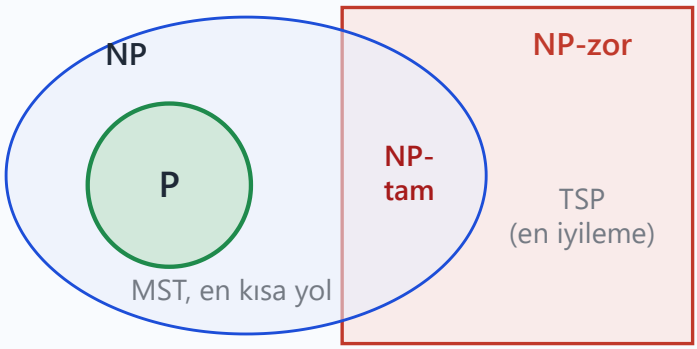
9.1.3 İndirgeme, NP-zor ve NP-tam

İki problemin zorluğunu karşılaştırmanın aracı **indirgemedir** (*reduction*). A probleminin her örneği polinom sürede B probleminin bir örneğine dönüştürülebiliyorsa ve iki örneğin cevabı aynıysa, A, B'ye indirgenir. Bu durumda B'yi çözen hızlı bir algoritma A'yı da hızlıca çözer: önce dönüştürülür, sonra B'nin algoritması çalıştırılır. Yani B en az A kadar zordur.

> **Tanım 9.2**
>
> NP'deki her problem bir B problemine polinom zamanda indirgenebiliyorsa B, **NP-zor**dur (*NP-hard*). B hem NP'de hem NP-zor ise **NP-tam**dır (*NP-complete*). NP-zor bir problem en az NP'deki en zor problem kadar zordur; kendisinin NP'de olması gerekmez.

NP-tam problemler NP'nin "en zor" üyeleridir. İçlerinden biri polinom sürede çözülebilseydi, indirgemeler sayesinde NP'deki her problem polinom sürede çözülürdü ve $P = NP$ olurdu. İlk NP-tam problem mantıksal doyurulabilirliktir (*satisfiability*, SAT): mantıksal değişkenlerden kurulmuş bir formülü doğru yapan bir değer ataması var mı? SAT'ın kısıtlı bir biçimi olan 3-SAT da NP-tamdır. Hamilton yolu ve Hamilton döngüsü, çizge 3-boyama ve alt küme toplamının karar sürümü de NP-tam problemler arasındadır.

Gezgin satıcı problemi (*traveling salesman problem*, TSP) ve sırt çantası probleminin **en iyileme** sürümleri ise NP-zordur. Bu sürümler karar problemi değildir; bir sayı ya da bir tur döndürürler. Önerilen bir turun en kısa tur olduğunu denetlemek de kolay değildir. Bu yüzden NP içinde sayılmazlar, ama karar sürümleri NP-tam olduğu için en az o kadar zordurlar.



Şekil 9.1: *P ≠ NP* varsayımıyla sınıfların ilişkisi. P, NP'nin içindedir. NP-zor bölgesinin NP ile kesişimi NP-tam problemlerdir. TSP'nin en iyileme sürümü NP-zordur, ama NP'de değildir.

Şekil 9.1'deki çizim bir varsayıma dayanır. *P = NP* olsaydı P, NP ve NP-tam bölgeleri (önemsiz birkaç istisnayla) çakışırdı. Çizimin gösterdiği ayrım, bugünkü bilgimizin değil, yaygın kanaatin resmidir.

Problem	Sürüm	Sınıf
En kısa yol, minimum kapsayan ağaç	karar	P
SAT, 3-SAT	karar	NP-tam
Hamilton yolu, Hamilton döngüsü	karar	NP-tam
Çizge 3-boyama	karar	NP-tam
Alt küme toplamı	karar	NP-tam
Gezgin satıcı, 0/1 sırt çantası	en iyileme	NP-zor

Tablo 9.1: Notta geçen problemlerin sınıfları.

9.1.4 Kesin çözüm için ne yapılabilir?

P ≠ NP ise NP-tam ve NP-zor problemler için polinom zamanlı **kesin** algoritma yoktur. Bu durumda üç yol kalır.

- **Üstel ama akıllı arama.** Kesin çözüm şartsa olası çözümler aranır. Arama uzayı üsteldir, ama çoğu zaman uzayın küçük bir kısmını gezmek yeter. Amaç, işe yaramayacağı anlaşılan bölümleri erkenden atmaktır. Bu işleme **budama** (*pruning*) denir. Geri izleme ve dal sınır yöntemleri bu yolu izler.
- **Yaklaşık çözüm.** Polinom süre şartsa en iyi çözümden vazgeçilir. Yaklaşım algoritmaları en iyiye kanıtlı bir oranla yakın bir çözüm üretir.
- **Sezgisel arama.** Hiçbir garanti vermeyen ama pratikte çoğu zaman iyi çalışan yöntemler kullanılır. En az çakışma sezgiseli bu türdendir.

Bu notun konusu birinci yoldur. Geri izleme en kötü durumda üstel süre harcar. Ama iyi bir budamayla, pratikte karşılaşılan örneklerin çoğunu makul sürede çözer. n-vezir problemi bu farkı sayılarla gösterir: 8 vezir için 16,7 milyon tam yerleşimi tek tek denemek yerine yalnız 2056 kısmi yerleşim üretmek yeter.

9.2 Geri izlemenin temel fikri

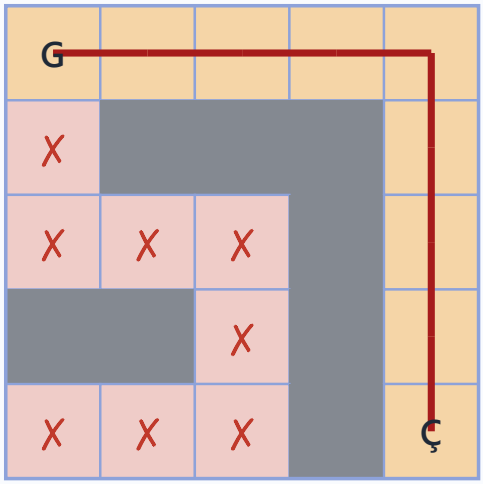
9.2.1 Geri dönüp denemek

Bir problemi çözmek için bazen farklı yollar denemek ve yanlış yollardan geri dönmek gerekir. Kilitli bir çantanın üç haneli şifresini unutan biri bütün kombinasyonları sırayla dener. Bir bulmaca çözen kişi bir hücreye bir rakam yazar, birkaç adım sonra çelişkiye düşünce silip başka rakam dener. **Geri izleme** (*backtracking*) bu sezgisel davranışı sistemli bir algoritmaya dönüştürür.

> **Tanım 9.3**
>
> **Geri izleme**, çözümü adım adım bir **kısmi çözüm** olarak kuran bir arama yöntemidir. Her adımda kısmi çözüme bir **aday** eklenir. Eklenen aday problemin **kısıtlarını** bozuyorsa o yol terk edilir. Kısmi çözüm tamamlanmışsa bir çözüm bulunmuştur. Bir adımda denenecek aday kalmamışsa son karar geri alınır ve bir önceki adımda sıradaki aday denenir.

Tanımdaki dört terim notun her yerinde geçer. **Kısmi çözüm** o ana kadar yapılan seçimlerdir: n-vezir probleminde ilk birkaç satıra konmuş vezirler, Sudoku'da doldurulmuş hücreler. **Aday** bir sonraki adımda yapılabilecek seçimdir: sıradaki satırın sütunlarından biri, boş hücreye yazılabilecek 1–9 rakamlarından biri. **Kısıt**, geçerli bir çözümün uyması gereken kuraldır: iki vezir birbirini tehdit etmez, bir satırda aynı rakam iki kez geçmez. **Geri alma** ise son eklenen adayı çıkarıp durumu bir önceki hâline getirmektir.

Geri izlemeyi **kaba kuvvetten** (*brute force*) ayıran budur. Kaba kuvvet bütün tam aday çözümleri üretir ve her birini sonradan test eder. Geri izleme ise kısmi çözüm geçersiz olur olmaz durur. Geçersiz bir kısmi çözümün devamı hiç üretilmez. İlk iki vezir birbirini tehdit ediyorsa, kalan altı vezirin hiçbir yerleşimi bu hatayı düzeltemez. Kaba kuvvet bu altı vezirin $8^6 = 262.144$ yerleşimini tek tek dener; geri izleme hiçbirini denemez.

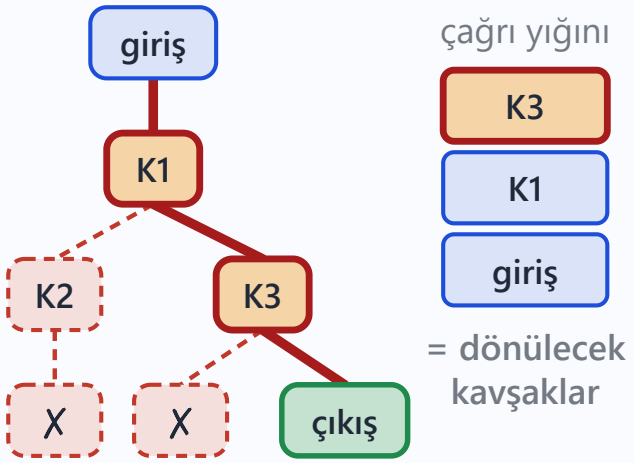


Şekil 9.2: Bir labirentte geri izleme. G giriş, Ç çıkıştır. Turuncu hücreler bulunan yolu, X işaretli hücreler denenip bırakılan çıkmazları gösterir.

9.2.2 Labirentteki yolcu

Geri izleme algoritması bir labirentte yol arayan bir yolcuya benzetilebilir. Yolcu her kavşakta bir karar verir ve bir koridora girer. Koridor çıkışa götürüyorsa iş biter. Çıkmaza girerse son kavşağa geri döner ve henüz denemediği bir koridoru dener. O kavşaktaki bütün koridorlar çıkmazsa bir önceki kavşağa döner. Bu deneme-yanılma süreci çıkış bulunana ya da bütün seçenekler tükenene kadar sürer.

Benzetme algoritmanın en önemli gereksinimini de gösterir. Geri dönebilmek için nereden gelindiğini hatırlamak gerekir. Yolcu her kavşakta yere bir işaret bırakır ya da girdiği kavşakları bir listeye yazar. Geri izleme algoritmalarında bu listeyi özyineleme kendiliğinden sağlar. Her özyinelemeli çağrı bir kavşaktır. Çağrı yığınında üst üste duran çerçeveler, girilmiş ama henüz bitirilmemiş kavşakların listesidir. Bir çağrı `return` ile bittiğinde denetim bir önceki çerçeveye döner. Bu da yolcunun bir önceki kavşağa geri dönmesidir.

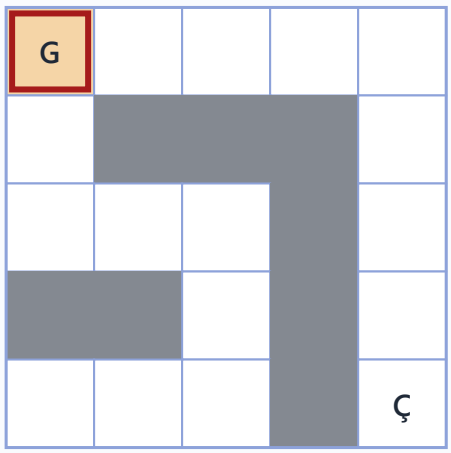


Şekil 9.3: Kavşaklar ve çağrı yığını. Girişten K1'e, oradan K3'e geçilmiştir. K2 dalı ve K3'ün ilk dalı çıkmazdır. Yığında giriş, K1 ve K3 durur: bunlar geri dönülecek kavşaklardır.

Labirent örneği yapay zekâ ve bulmaca problemlerinin ortak yapısını taşır. Sudoku'da her boş hücre bir kavşaktır, yazılabilecek rakamlar koridorlardır. Satrançta her hamle bir kavşaktır. Hepsinde aynı soru sorulur: bu kararla devam etmek mümkün mü, değilse hangi karara geri dönülmeli?

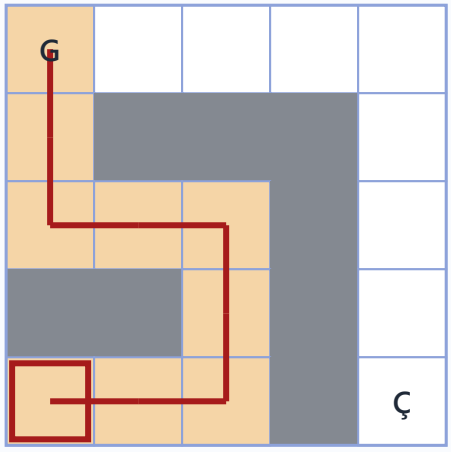
9.2.3 Labirentte geri dönüş adım adım

Labirent *5 × 5*'lik bir ızgaradır. Hücreler (satır, sütun) çiftiyle gösterilir; giriş (0, 0), çıkış (4, 4) hücresidir. Gri hücreler duvardır: (1, 1), (1, 2), (1, 3), (2, 3), (3, 0), (3, 1), (3, 3) ve (4, 3). Algoritma her hücrede dört yönü sabit bir sırayla dener: aşağı, sağ, yukarı, sol. Ziyaret edilen her hücre işaretlenir, böylece aynı hücreye ikinci kez girilmez. İşaret olmasaydı yolcu iki hücre arasında sonsuza kadar gidip gelebilirdi.



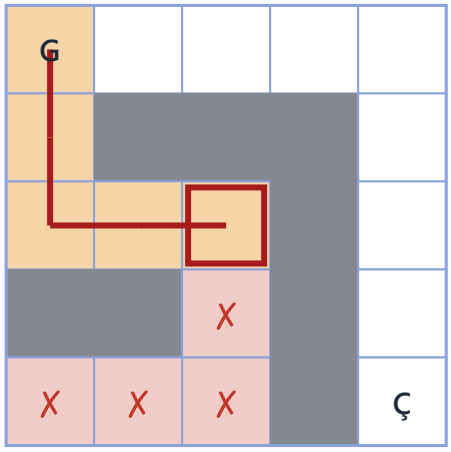
Şekil 9.4: Başlangıç: giriş (G) hücresindeyiz. Yön sırası aşağı, sağ, yukarı, soldur.

Arama şöyle ilerler. Girişten aşağı iki adım atılır: (1, 0), (2, 0). (3, 0) duvar olduğu için aşağı yön kapalıdır; sıradaki yön olan sağ denenir. Sağa iki adımla (2, 1) ve (2, 2)'ye varılır. (2, 2) bir kavşaktır: aşağısı açık, sağı (2, 3) ve yukarısı (1, 2) duvardır. Önce aşağı denenir: (3, 2), (4, 2). (4, 2)'nin aşağısı ızgaranın dışı, sağı (4, 3) duvardır; sola dönülür: (4, 1), (4, 0). (4, 0) hücresinde dört yön de kapalıdır: aşağısı ve solu sınır, yukarısı duvar, sağı ziyaret edilmiş. Bu bir çıkmazdır.



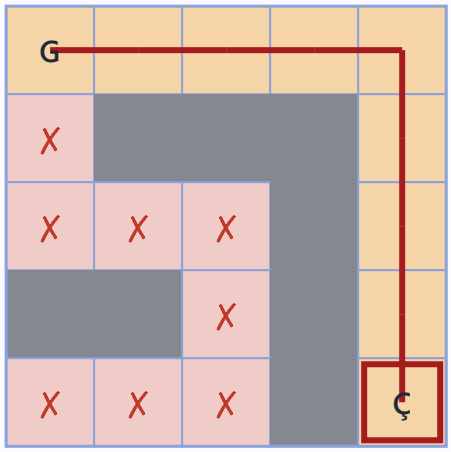
Şekil 9.5: Aşağı ve sola ilerleyen koridor (4, 0) hücresinde çıkmaza girer.

Şimdi geri dönüş başlar. (4, 0), (4, 1), (4, 2) ve (3, 2) yoldan çıkarılır, çünkü bu hücrelerin denenmemiş açık yönü yoktur. (2, 2) kavşağına geri gelinir. Bu kavşağın kalan yönleri de kapalıdır: sağı ve yukarısı duvar, solu ziyaret edilmiş. Geri dönüş sürer: (2, 2), (2, 1), (2, 0) ve (1, 0) da çıkmaz olarak bırakılır. Sonunda girişe dönülür. Girişte aşağı yön tükenmiştir; sıradaki yön olan sağ denenir.



Şekil 9.6: İlk geri dönüş: (4, 0), (4, 1), (4, 2) ve (3, 2) yoldan çıkarılır (X). Kavşak (2, 2) yeniden denenir, ama kalan yönleri de kapalıdır.

Üst koridor boyunca sağa gidilir: (0, 1), (0, 2), (0, 3), (0, 4). (0, 1), (0, 2) ve (0, 3)'ün aşağısı duvardır; (0, 4)'ün aşağısı ise açıktır. (1, 4), (2, 4), (3, 4) üzerinden çıkışa, (4, 4)'e varılır.



Şekil 9.7: Çıkışa ulaşıldı. Yol 9 hücredir; 8 hücre çıkmaz sokak olarak denenip bırakılmıştır.

Örnek 9.1

Soru. Labirentte girişten çıkışa kaç farklı basit yol vardır? (Basit yolda bir hücre en çok bir kez kullanılır.) Yön sırası "sağ, aşağı, yukarı, sol" olsaydı arama kaç hücreyi çıkmaz olarak işaretlerdi?

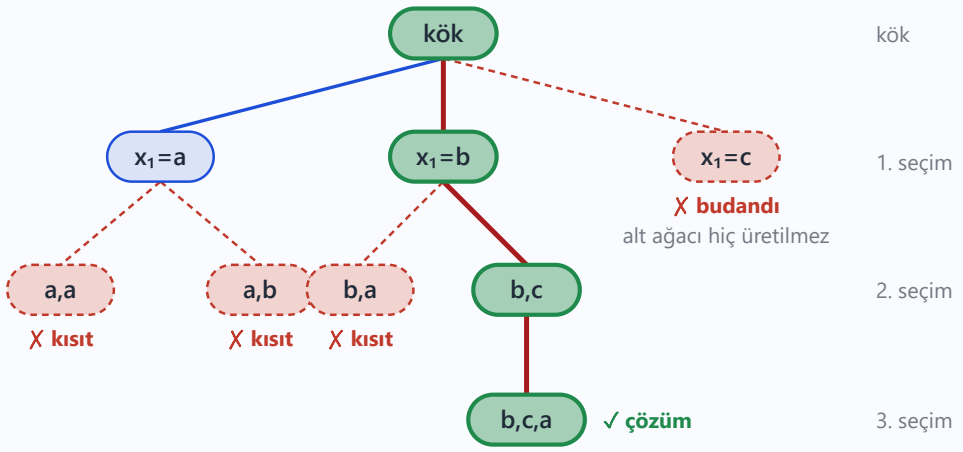
Çözüm. Sol alt bölge, yani (1, 0), (2, 0), (2, 1), (2, 2), (3, 2) ve (4, 0)–(4, 2) hücreleri, labirentin geri kalanına yalnız girişten bağlıdır. (2, 3), (3, 3), (4, 3) ve (1, 2) duvar olduğu için bu bölgeden sağ sütuna geçilemez. Çıkışa varan tek bağlantı üst koridor ve sağ sütundur. Bu yüzden yalnız 1 basit yol vardır: 9 hücrelik turuncu yol. Yön sırası sağla başlasaydı girişten ilk denenen yön üst koridor olurdu. Arama her hücrede önce sağa gider ve (0, 4)'e varır. Orada sağ yön ızgaranın dışıdır; aşağı inilir ve sağ sütun boyunca çıkışa varılır. Arama hiç çıkmaza girmez: 0 hücre çıkmaz olarak işaretlenir. Aynı labirentte yön sırası aramanın maliyetini 8 çıkmaz hücreden 0'a indirir. Hangi seçeneğin önce deneneceği, geri izlemenin hızını belirleyen önemli bir karardır.

9.2.4 Durum uzayı ağacı

Geri izlemenin yaptığı aramayı bir ağaç üzerinde düşünmek yararlıdır. Kökte henüz hiçbir seçim yapılmamıştır. Kökün çocukları birinci seçimin seçenekleridir. Her çocuğun çocukları ikinci seçimin seçenekleridir ve bu böyle sürer. Bu ağaca **durum uzayı ağacı** (*state-space tree*) denir.

Tanım 9.4

Durum uzayı ağacı, kısmi çözümlerin düğüm, seçimlerin kenar olduğu köklü ağaçtır. Kök boş kısmi çözümdür. Kökten bir düğüme giden yol, o ana kadar yapılan seçimlerdir. Bir düğümün çocukları, bir sonraki düzeyde yapılabilecek seçimlerdir. Yapraklar ya tam çözümlerdir ya da devam ettirilemeyen kısmi çözümlerdir.



Şekil 9.8: Üç seçimli bir problemin durum uzayı ağacı. x_1 = c düğümü budanmıştır; alt ağacı hiç üretilmez. Kısıtı bozan (a, a), (a, b) ve (b, a) dalları da kesilir. Kalın yol çözüme, (b, c, a)'ya götürür.

Şekil 9.8 tanımdaki terimleri bir arada gösterir. **Kısmi çözüm** kökten bir düğüme giden yoldur: x_1 = b, x_2 = c düğümü (b, c) kısmi çözümüdür. **Adaylar** düğümün çocuklarıdır. **Kısıt** bozulursa düğümün alt ağacı budanır: (a, a) kısmi çözümü geçersizdir ve altındaki hiçbir düğüm üretilmez. **Hedef testi** tam ve geçerli bir çözüme ulaşılıp ulaşılmadığını sorar; bu ağaçta (b, c, a) bir yapraktır ve çözümdür.

Ağacın iki özelliği önemlidir. Birincisi, ağaç önceden kurulmaz. Algoritma yalnız o an bulunduğu düğümü ve köke giden yolu bellekte tutar. Ağaç gezilirken üretilir ve gezilen dallar unutulur. İkincisi, gezinme sırası **derinlik önceliklidir** (*depth-first search*, DFS). Algoritma bir dalda dibe kadar iner. Dal biterse ya da budanırsa bir üst düğüme çıkar ve sıradaki çocuğa geçer. Geri izleme, durum uzayı ağacında yapılan bir derinlik öncelikli aramadır. Fark yalnız şudur: geçersiz düğümlerin alt ağaçları hiç üretilmez.

Sık yapılan hata

Durum uzayı ağacını bellekte kurulan bir veri yapısı sanmak. Ağaç yalnız aramayı düşünmek için kullanılan bir modeldir. Bellekte aynı anda yalnız kökten geçerli düğüme uzanan tek yol bulunur. n-vezirde bu yol en çok *n* düğümdür, oysa ağacın kendisi binlerce düğüm içerebilir.

9.2.5 Genel şablon: seç, keşfet, geri al

Geri izleme algoritmalarının hemen hepsi aynı iskeletle yazılır. İskelet bir özyinelemeli metottur. Metot bir kısmi çözüm alır, tam bir çözüm olup olmadığına bakar, değilse her adayı sırayla dener.

```
void geriDon(Durum kismi) {
  if (cozumMu(kismi)) {
    kaydet(kismi);
    return;
  }
  for (Aday a : adaylar(kismi)) {
    if (gecerliMi(kismi, a)) {
      ekle(kismi, a);    // seç
      geriDon(kismi);    // keşfet
      cikar(kismi, a);   // geri al
    }
  }
}
```

Kodun dört parçası vardır.

1. **Hedef testi.** `cozumMu(kismi)` kısmi çözümün tamamlanıp tamamlanmadığına bakar. Tamamlanmışsa çözüm kaydedilir ve metot döner. Tek bir çözüm yetiyorsa burada arama tamamen durdurulabilir.
2. **Seç.** `adaylar(kismi)` bir sonraki adımın seçeneklerini verir. Geçerli bir aday `ekle(kismi, a)` ile kısmi çözüme eklenir.
3. **Keşfet.** `geriDon(kismi)` özyinelemeyle bir düzey derine iner. Bu çağrı, eklenen adayın altındaki bütün alt ağacı gezer.
4. **Geri al.** Çağrı döndüğünde `cikar(kismi, a)` adayı kısmi çözümden çıkarır. Durum çağrıdan önceki hâline döner ve döngü sıradaki adayla sürer.

Budama `gecerliMi` içinde olur. Geçersiz bir aday için özyinelemeli çağrı hiç yapılmaz. Böylece o adayın altındaki bütün alt ağaç tek bir `if` ile atlanır. Budamanın gücü buradan gelir: bir denetim, belki binlerce düğümlük bir alt ağacı ortadan kaldırır.

Geri alma adımının neden şart olduğunu görmek için kısmi çözümün nasıl saklandığına bakmak gerekir. `kismi` bütün çağrıların paylaştığı tek bir nesnedir: bir tahta, bir dizi ya da bir liste. Her çağrı bu nesneyi değiştirir. Bir çağrı döndüğünde nesne, çağrının bıraktığı hâlde kalır. Sıradaki aday denenmeden önce bu değişiklik silinmezse yeni aday eskisinin kalıntılarıyla birlikte sınanır. Yalnız değerle geçirilen ilkel parametreler, örneğin bir `int` toplam ya da bir indis, kendiliğinden eski hâline döner. Her çağrının kendi kopyası vardır.

Tek bir çözüm yetiyorsa şablon küçük bir değişiklikle `boolean` döndürür. Hedef testinde `true` döndürülür. Özyinelemeli çağrı `true` döndürürse arayan da hemen `true` döndürür. Böylece ilk çözüm bulunduğunda bütün çağrılar sırayla sona erer. n-vezir, Sudoku, çizge boyama ve Hamilton yolu kodları bu biçimdedir.

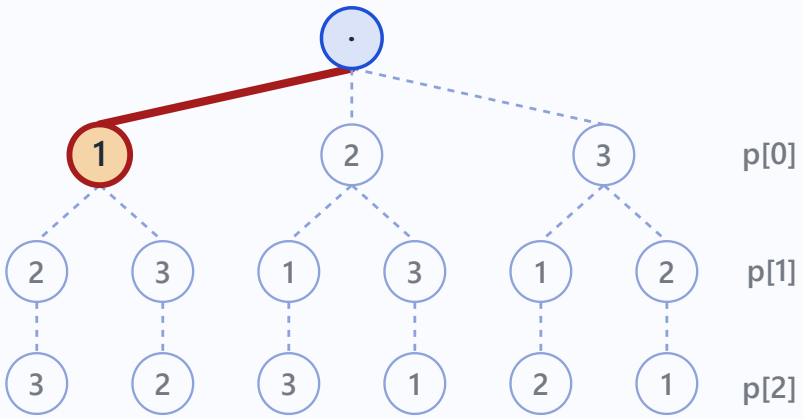
9.2.6 Şablonun uygulaması: permütasyonlar

Şablonun en temiz uygulaması bir kümenin bütün permütasyonlarını üretmektir. 1, 2 ve 3 sayılarının her sıralaması bir permütasyondur. Kısmi çözüm, ilk *k* konuma yazılmış sayılardır. Aday, henüz kullanılmamış bir sayıdır. Kısıt tektir: bir sayı iki kez kullanılamaz.

```
void permutasyon(int[] p, boolean[] kullan, int k) {
  if (k == p.length) {              // çözüm
    System.out.println(Arrays.toString(p));
    return;
  }
  for (int a = 1; a <= p.length; a++) {
    if (!kullan[a]) {               // geçerli
      p[k] = a; kullan[a] = true;   // seç
      permutasyon(p, kullan, k + 1); // keşfet
      kullan[a] = false;            // geri al
    }
  }
}
```

`p` dizisi kısmi çözümü tutar; `p[0..k-1]` doludur. `kullan` dizisi hangi sayıların kullanıldığını gösterir. Dizinin boyu *n + 1*'dir, çünkü sayılar 1'den başlar ve `kullan[0]` boş kalır. `k == p.length` hedef testidir: bütün konumlar dolduysa permütasyon yazdırılır. Döngü 1'den *n*'ye kadar her sayıyı dener. `!kullan[a]` geçerlilik testidir. Sayı kullanılmamışsa `p[k]`'ya yazılır ve işaretlenir (seç). Özyinelemeli çağrı bir sonraki konumu doldurur (keşfet). Çağrı dönünce işaret kaldırılır (geri al).

Geri alma satırında yalnız `kullan[a] = false` vardır; `p[k]` sıfırlanmaz. Bunun nedeni, `p[k]`'nın bir sonraki adayda zaten üzerine yazılacak olmasıdır. Geçerlilik testi `p`'ye değil, `kullan`'a bakar. Bu yüzden `p[k]`'daki eski değer hiçbir kararı etkilemez. Kısıtı denetleyen yapı ise mutlaka geri alınmalıdır.



Şekil 9.9: İlk adım: `k = 0`'da aday 1 seçildi. Kesikli düğümler henüz üretilmemiştir.

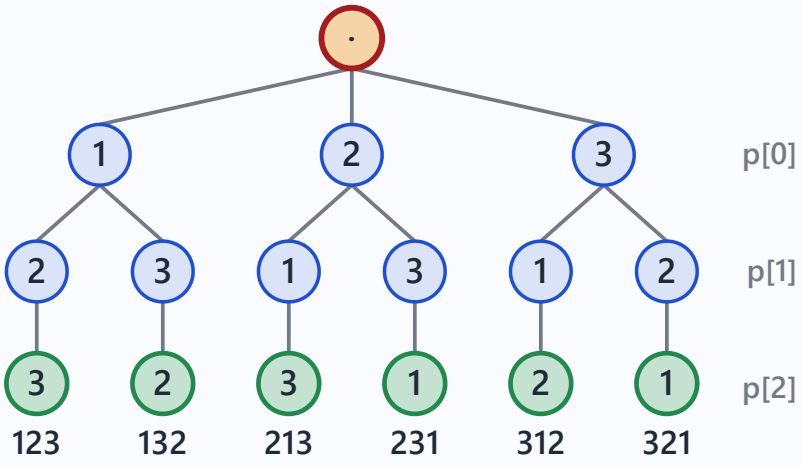
Adım	k	İşlem	p	Kullanılan	Çıktı
1	0	a = 1 seçilir	[1, –, –]	{1}	
2	1	a = 1 kullanılmış; a = 2 seçilir	[1, 2, –]	{1, 2}	
3	2	yalnız 3 kullanılmamış; seçilir	[1, 2, 3]	{1, 2, 3}	
4	3	`k == p.length` : çözüm	[1, 2, 3]	{1, 2, 3}	[1, 2, 3]
5	1	3 ve 2 geri alınır; a = 3 seçilir	[1, 3, 3]	{1, 3}	
6	2	yalnız 2 kullanılmamış; seçilir	[1, 3, 2]	{1, 2, 3}	
7	3	çözüm	[1, 3, 2]	{1, 2, 3}	[1, 3, 2]
8	0	2, 3, 1 geri alınır; a = 2 seçilir	[2, 3, 2]	{2}	
9	1–3	a = 1, sonra 3	[2, 1, 3]	{1, 2, 3}	[2, 1, 3]
10	1–3	geri al; a = 3, sonra 1	[2, 3, 1]	{1, 2, 3}	[2, 3, 1]
11	0–3	3 ile başlayan dal aynı biçimde	[3, 1, 2], [3, 2, 1]	—	[3, 1, 2], [3, 2, 1]

Tablo 9.2: `permutasyon(p, kullan, 0)` çağrısının izlemesi (*n = 3*). "Kullanılan" sütunu adımdan sonra `true` olan sayıları gösterir.

5. adımdaki `p` değeri dikkat çekicidir: [1, 3, 3]. `p[2]` 'de bir önceki permütasyondan kalan 3 durur. Bu değer hiçbir şeyi bozmaz, çünkü `k = 1` 'deyiz ve `p[2]` henüz kısmi çözümün parçası değildir. 6. adımda üzerine 2 yazılır. Program çıktısı şudur:

```
[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]
```

Adaylar küçükten büyüğe denendiği için permütasyonlar **sözlüksel sırada** (*lexicographic order*) çıkar. Bu, bütün geri izleme algoritmalarında geçerli bir gözlemdir: adaylar hangi sırayla denenirse çözümler de o sırayla bulunur.



Şekil 9.10: Arama bitince durum uzayı ağacının tamamı. Altı yaprak altı permütasyondur; her yaprağın altında kökten o yaprağa giden yolun okunuşu yazılıdır.

Örnek 9.2

Soru. $n = 3$ için durum uzayı ağacında kök dahil kaç düğüm vardır? Kaç kez "seç" ve kaç kez "geri al" yapılır? Aynı sorular $n = 4$ için nasıl cevaplanır?

Çözüm. Düzeyleri ayrı ayrı saymak yeter. Kökte 1 düğüm vardır. Birinci düzeyde 3 seçenek, ikinci düzeyde her birinin altında kalan 2 seçenek, üçüncü düzeyde 1 seçenek bulunur. Düğüm sayısı $1 + 3 + 3 \cdot 2 + 3 \cdot 2 \cdot 1 = 1 + 3 + 6 + 6 = 16$'dır. Kök dışındaki her düğüm bir "seç" işlemidir ve her seçin bir "geri al"ı vardır: 15 seç, 15 geri al. Yaprak sayısı $3! = 6$'dır. $n = 4$ için düğümler $1 + 4 + 12 + 24 + 24 = 65$'tir; 64 seç, 64 geri al ve 24 yaprak vardır.

Genel olarak n öğenin permütasyon ağacında k. düzeyde $n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = n!/(n-k)!$ düğüm vardır. Son iki düzeyin düğüm sayısı eşittir ($n!$) ve toplam $n!$'in küçük bir sabit katıdır ($e \cdot n!$'e yakındır). Her yaprakta permütasyon yazdırmak $O(n)$ sürer. Bu yüzden bütün permütasyonları üretmenin süresi $O(n \cdot n!)$'dir. Burada budama işi azaltmaz, çünkü bütün yapraklar zaten çözümdür. Budamanın kazandırdığı şey, kısıtı bozan dalların hiç açılmamasıdır: aynı sayının tekrar seçilmesine izin veren budamasız ağaçta $n = 4$ için $4^4 = 256$ yaprak olurdu.

Sık yapılan hata

Bütün permütasyonları bir listede toplarken kısmi çözümün kendisini listeye eklemek. Aşağıdaki kodda `sonuclar.add(yol)` yazılsaydı listeye altı kez *aynı* nesnenin referansı eklenirdi. Arama bitince `yol` boşalmış olacağı için liste altı boş listeden oluşurdu. Doğrusu, çözüm anında bir kopya saklamaktır.

```
List<List<Integer>> sonuclar = new ArrayList<>();

void topla(List<Integer> yol, boolean[] kul, int n) {
  if (yol.size() == n) {
    sonuclar.add(new ArrayList<>(yol));   // kopyası saklanır
    return;
  }
  for (int v = 1; v <= n; v++) {
    if (!kul[v]) {
      kul[v] = true; yol.add(v);          // seç
      topla(yol, kul, n);                 // keşfet
      yol.remove(yol.size() - 1);         // geri al
      kul[v] = false;
    }
  }
}
```

Bu sürüm kısmi çözümü bir `List` içinde tutar. Seçme adımı listeye ekler, geri alma adımı son öğeyi siler. `new ArrayList<>(yol)` o anki içeriğin bağımsız bir kopyasıdır; sonraki değişiklikler kopyayı etkilemez. `topla(new ArrayList<>(), new boolean[4], 3)` çağrısının ardından `sonuclar` altı permütasyonu sözlüksel sırada içerir.

9.2.7 Kaba kuvvet ve budama

Budamanın ne kadar iş kazandırdığı, 8 vezir problemiyle sayısal olarak gösterilebilir. Problem şudur: *8 × 8*'lik satranç tahtasına birbirini tehdit etmeyen 8 vezir yerleştirilecek. Problemi çözmenin birkaç yolu vardır ve her biri farklı sayıda aday inceler.

Yaklaşım	İncelenen aday	Açıklama
64 kareden 8'ini seç	4.426.165.368	*C(64, 8)*: her yerleşimi üret, sonra test et
Her satıra bir vezir	16.777.216	8^8 tam yerleşim
Her satıra bir vezir, sütunlar farklı	40.320	*8!* permütasyon
Geri izleme, bütün çözümler	2056	güvenli yerleştirme sayısı (92 çözüm)
Geri izleme, ilk çözüm	113	güvenli yerleştirme; 105 geri alma

Tablo 9.3: 8 vezir probleminde farklı yaklaşımların incelediği aday sayısı.

Tablonun ilk üç satırı kaba kuvvetin giderek akıllanan sürümleridir. İlk satır en saf yaklaşımdır: 64 kareden 8'ini seçmenin *C(64, 8) = 4.426.165.368* yolu vardır. İkinci satır bir gözlem kullanır: her satırda tam olarak bir vezir bulunmalıdır, çünkü aynı satırdaki iki vezir birbirini tehdit eder. Her satırdaki vezirin 8 sütun seçeneği vardır ve $8^8 = 16.777.216$ yerleşim kalır. Üçüncü satır aynı mantığı sütunlara da uygular: sütunlar da farklı olmalıdır. Bu durumda yerleşim, sütunların bir permütasyonudur ve *8! = 40.320* aday kalır. Bu üç yaklaşımın ortak yanı, her adayın *tam* bir yerleşim olması ve ancak sonunda test edilmesidir.

Geri izleme ise tam yerleşimleri değil, *kısmi* yerleşimleri üretir. Vezirler satır satır konur. Bir vezir öncekilerden biriyle çakışıyorsa o kısmi yerleşimin devamı hiç denenmez. Sonuç çarpıcıdır: bütün 92 çözümü bulmak için yalnız 2056 güvenli yerleştirme yapılır. İlk çözüm 113 yerleştirmede bulunur. 8^8 tam yerleşimle karşılaştırıldığında bütün çözümleri bulmak bile yaklaşık *16.777.216 / 2056 ≈ 8160* kat daha az iş gerektirir.

Budama en kötü durumu değiştirmez; geri izleme yine üstel bir algoritmadır. Ama pratikte durum uzayı ağacının çok küçük bir kısmı gezilir. Budamanın gücü kısıtın ne kadar erken denetlendiğine bağlıdır. Kısıt kısmi çözüm üzerinde, her yeni adayda denetlenmelidir. Sona bırakılan bir denetim kaba kuvvetten farksızdır.

9.2.8 Güçlü ve zayıf yanlar

Geri izlemenin güçlü yanları genelliğinden gelir.

- **Geniş uygulama alanı.** Çözümü bir dizi seçim olarak ifade edilebilen her probleme uygulanır. Aynı şablon permütasyon üretiminden Sudoku'ya kadar değişmeden kullanılır.
- **Çoğu zaman verimli.** Kısıtları erkenden denetlenebilen problemlerde ağacın büyük bölümü budanır.
- **Kısa ve anlaşılır kod.** Algoritma birkaç satırlık bir özyinelemeli metottur.
- **Tam arama.** Geri izleme bütün çözümleri bulur ya da hiç çözüm olmadığını kesin olarak gösterir. Arama ağacın tamamını gezip bir şey bulamazsa çözüm yoktur. Bu özelliğe **tamlık** denir.

Zayıf yanları da aynı ölçüde belirgindir.

- **Üstel en kötü durum.** Zaman karmaşıklığı 2^n, $n!$ ya da 9^m gibi üstel fonksiyonlarla sınırlanır.
- **Büyük arama uzaylarında yavaşlık.** Olası çözüm sayısı çok büyük olduğunda iyi bir budama ve sezgisel gerekir.
- **Yığın belleği.** Derin özyineleme her düzey için bir çerçeve kullanır: $O(derinlik)$ bellek. Çok derin aramalarda Java yığını taşabilir.
- **Tekrar eden iş.** Aynı alt problem farklı yollardan tekrar tekrar çözülebilir. Dinamik programlama bu sonuçları tabloda saklar; geri izleme saklamaz.

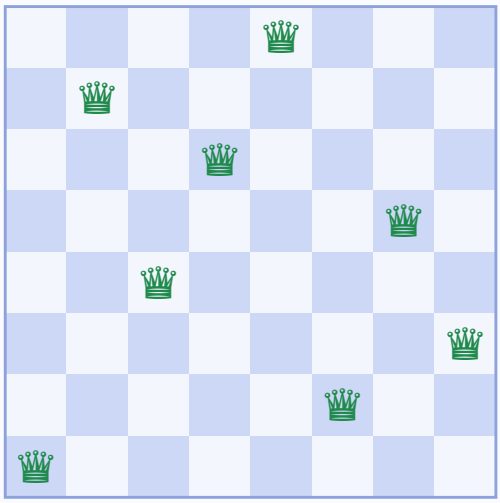
9.3 n-vezir problemi

9.3.1 Problem

Satrançta vezir en güçlü taştır. Bulunduğu satır, sütun ve iki çapraz boyunca istediği kadar ilerleyebilir. Bu dört doğru üzerindeki her taşı tehdit eder. **8 vezir problemi** (*eight queens*) 1848'de bir satranç dergisinde sorulmuş eski bir bulmacadır: 8 vezir, hiçbiri bir diğerini tehdit etmeyecek biçimde 8×8'lik tahtaya yerleştirilebilir mi?

> **Tanım 9.5**
>
> **n-vezir problemi** (*N-queens problem*): $n \times n$'lik bir tahtaya n vezir yerleştirilir. Herhangi iki vezir aynı satırda, aynı sütunda ya da aynı çaprazda olmamalıdır. Çözüm böyle bir yerleşimdir.

Problemin genel biçiminde bazı küçük durumlar ilginçtir. $n = 1$ için tek vezir tek kareye konur; bu bir çözümdür. $n = 2$ ve $n = 3$ için çözüm yoktur. 2×2'lik tahtada her iki kare çifti ya aynı satırda, ya aynı sütunda ya da aynı çaprazdadır. $n \geq 4$ için her zaman en az bir çözüm vardır. 8 vezir probleminin 92 çözümü bulunur. Tahtanın döndürülmesi ve yansıtılmasıyla birbirine dönüşen çözümler aynı sayılırsa 12 **temel çözüm** kalır.



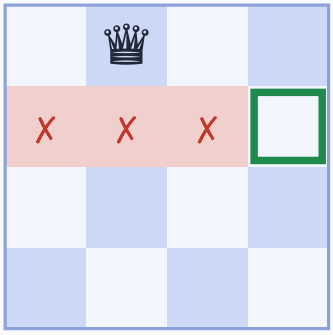
Şekil 9.11: 8 vezir probleminin bir çözümü: a1, b7, c4, d6, e8, f2, g5, h3. Satranç gösteriminde harf sütunu, sayı satırı belirtir; 1. satır tahtanın en altındadır.

Şekil 9.11'deki çözüm satranç gösterimiyle verilmiştir. a1, a sütununun 1. satırı, yani sol alt köşedir. Çözümün doğruluğu denetlenebilir: sekiz harf farklıdır (sütunlar farklı), sekiz sayı farklıdır (satırlar farklı). Çapraz denetiminde her çift için satır farkı ile sütun farkının mutlak değerleri karşılaştırılır; hiçbir çiftte eşit değildir. Bu denetim, problemin NP'de olmasının küçük bir örneğidir: çözümü bulmak arama ister, denetlemek ise *C(8, 2) = 28* çift karşılaştırmasıyla biter.

9.3.2 Satır satır yerleştirme

Geri izleme çözümü şu adımlarla ilerler.

1. Tahtanın her hücresi başlangıçta boştur.
2. İlk vezir ilk satırda bir sütuna yerleştirilir.
3. Bir sonraki satırda her sütun sırayla denenir. Vezir o sütuna konduğunda önceki vezirlerle çakışıp çakışmadığı denetlenir: aynı sütunda ya da iki çaprazdan birinde bir vezir var mı?
4. Çakışma yoksa konum geçerli kabul edilir ve bir sonraki satıra geçilir.
5. Bir satırda bütün sütunlar denenip güvenli yer bulunamazsa bir önceki satırdaki vezir kaldırılır ve o satırda sıradaki sütun denenir. Bu geri dönüştür.
6. Bütün vezirler yerleştirildiğinde çözüm bulunmuştur.



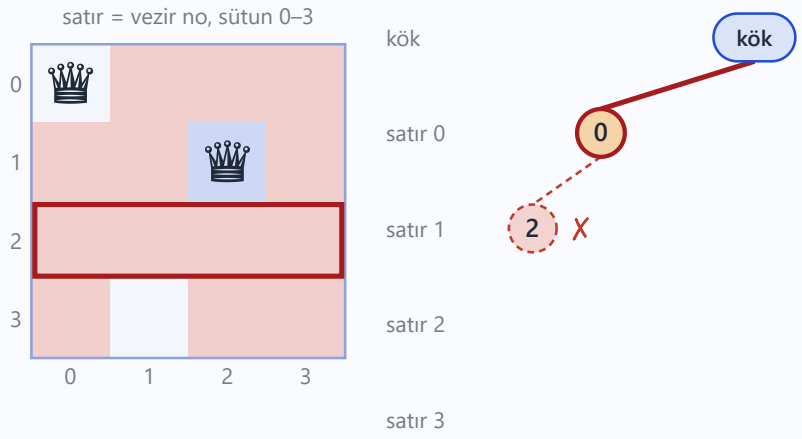
Şekil 9.12: 4 × 4 tahtada 0. satırın 1. sütununda bir vezir varken 1. satır: sütun 0, 1 ve 2 tehdit altındadır, sütun 3 güvenlidir.

Vezirlerin satır satır konması bilinçli bir tasarım kararıdır. Her satırda tam olarak bir vezir olmak zorundadır. Bu gözlem satır kısıtını yapının içine gömer. Vezir 0 satır 0'a, vezir 1 satır 1'e konur; iki vezirin aynı satırda olması imkânsız hâle gelir. Böylece her yerleştirmede yalnız sütun ve iki çapraz denetlenir. Arama uzayı da 64^8'den (her vezir için 64 kare) 8^8'e iner. Kısıtları denetlemek yerine yapıyla sağlamak, geri izlemede sık kullanılan bir tekniktir.

9.3.3 4 vezir: tahta ve durum uzayı ağacı

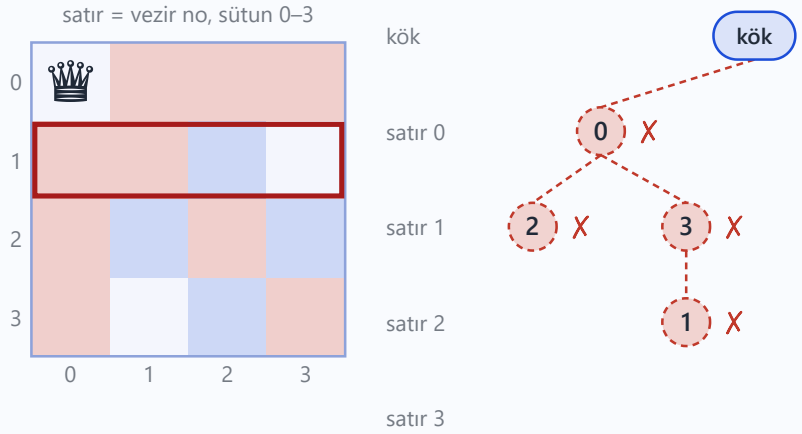
Algoritmanın işleyişi en iyi küçük bir örnekte görülür. $n = 4$ için vezirler satır satır yerleştirilir; her satırda sütun 0, 1, 2, 3 sırayla denenir. Durum uzayı ağacında kökün altındaki her düzey bir satırdır; bir düğüm, o satırda seçilen sütunu gösterir. Tahtadaki kırmızı hücreler, yerleşmiş vezirlerin tehdit ettiği karelerdir.

1. Satır 0: sütun 0 güvenlidir; vezir yerleştirilir ve satır 1'e inilir.
2. Satır 1: sütun 0 ve 1 tehdit altındadır; sütun 2 güvenlidir. Vezir konur, satır 2'ye inilir.
3. Satır 2: dört sütun da tehdit altındadır. Bu bir çıkmazdır. Satır 1'deki vezir (sütun 2) kaldırılır; satır 1'de sıradaki sütun denenir.



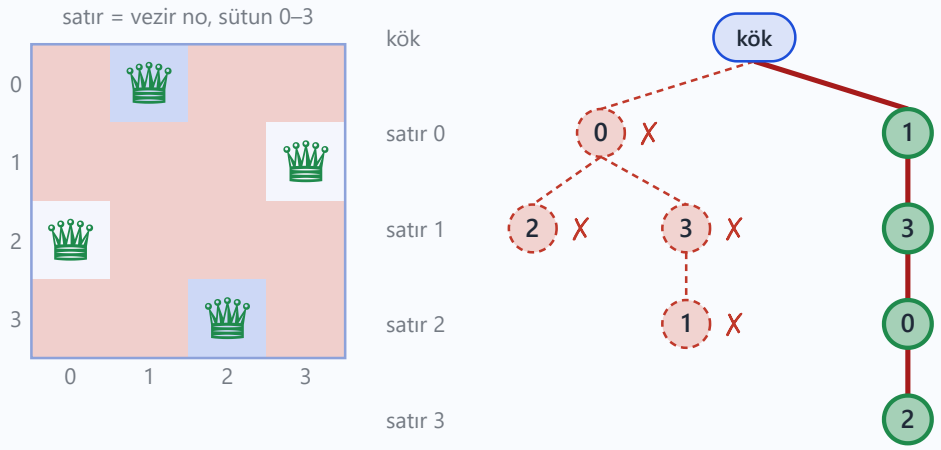
Şekil 9.13: İlk çıkmaz. Vezirler (0, 0) ve (1, 2)'deyken satır 2'nin bütün kareleri tehdit altındadır. Ağaçta 2 düğümü kesikli çizilmiştir: bu dal geri alınacaktır.

4. Satır 1: sütun 3 güvenlidir; vezir konur, satır 2'ye inilir.
5. Satır 2: sütun 0 tehdit altında, sütun 1 güvenlidir. Vezir konur, satır 3'e inilir.
6. Satır 3: dört sütun da tehdit altındadır; çıkmaz. Satır 2'deki vezir (sütun 1) kaldırılır.
7. Satır 2: kalan sütunlar 2 ve 3 de tehdit altındadır; çıkmaz. Satır 1'deki vezir (sütun 3) kaldırılır.
8. Satır 1'in denenecek sütunu kalmamıştır. Satır 0'daki vezir (sütun 0) da kaldırılır.



Şekil 9.14: Sütun 0 ile başlayan bütün dallar çıkmaza girmiştir. Geri dönüş köke kadar uzanır; satır 0'da sıradaki sütun denenecektir.

9. Satır 0: sütun 1 güvenlidir; vezir konur.
10. Satır 1: sütun 0, 1, 2 tehdit altında, sütun 3 güvenlidir.
11. Satır 2: sütun 0 güvenlidir.
12. Satır 3: sütun 0 ve 1 tehdit altında, sütun 2 güvenlidir. Dördüncü vezir yerleşir ve `satir == n` olur. Çözüm bulunmuştur: sütunlar [1, 3, 0, 2].



Şekil 9.15: Çözüm [1, 3, 0, 2]. Ağaçta yeşil yol çözümü, kesikli düğümler geri dönülen dalları gösterir. Toplam 8 yerleştirme ve 4 geri dönüş yapılmıştır.

Ağaçtaki düğümler sayılırsa işin dökümü çıkar. Kök dışında 8 düğüm vardır: çözüm yolundaki 4 düğüm ve geri alınan 4 düğüm (0, 2, 3 ve 1). Her düğüm bir yerleştirmedir; geri alınan her düğüm bir geri dönüştür. Tehdit altındaki karelere hiç vezir konmadığı için bu kareler ağaçta düğüm olarak bile görünmez. Budamanın kazancı tam da budur.

Çözümün gösterimine dikkat edilmelidir. [1, 3, 0, 2] dizisinin *i*. öğesi *i*. satırdaki vezirin sütunudur. Satır 0'daki vezir sütun 1'de, satır 1'deki sütun 3'te, satır 2'deki sütun 0'da, satır 3'teki sütun 2'dedir. Bu gösterim, satır kısıtının yapıya gömülmesinin doğal sonucudur: her satır için tek bir sayı yeter.

Örnek 9.3

Soru. 4 vezir probleminin kaç çözümü vardır? İkinci çözüm birincisinden nasıl elde edilir?

Çözüm. Arama ilk çözümde durdurulmazsa satır 0'da sütun 2 ve 3 de denenir. Sütun 2 dalı ikinci çözümü verir: [2, 0, 3, 1]. Sütun 3 dalı çıkmaza girer. 4 vezir probleminin yalnız 2 çözümü vardır. İkinci çözüm birincinin aynadaki görüntüsüdür: her sütun *c*, *3 − c* ile değiştirilirse [1, 3, 0, 2] dizisi [2, 0, 3, 1] olur. Tahta dikey eksene göre yansıtıldığında tehdit ilişkileri korunur. Bu yüzden bir çözümün yansıması da çözümdür.

9.3.4 Java kodu

Algoritma iki metotla yazılır. `guvenliMi`, *(s, k)* karesine konacak bir vezirin daha önce konmuş vezirlerle çakışıp çakışmadığını denetler. `coz` satır satır ilerleyen geri izleme metodudur. Tahta `int[][] t` dizisidir; `t[i][j] == 1`, *(i, j)* karesinde vezir olduğunu gösterir. `n` sınıfın bir alanıdır.

```
boolean guvenliMi(int[][] t,
        int s, int k) {
  // aynı sütun
  for (int i = 0; i < s; i++)
    if (t[i][k] == 1) return false;
  // sol üst çapraz
  for (int i = s-1, j = k-1;
       i >= 0 && j >= 0; i--, j--)
    if (t[i][j] == 1) return false;
  // sağ üst çapraz
  for (int i = s-1, j = k+1;
       i >= 0 && j < n; i--, j++)
    if (t[i][j] == 1) return false;
  return true;
}
```

`guvenliMi` yalnız üç yöne bakar: yukarı, sol üst çapraz ve sağ üst çapraz. Aşağıdaki satırlara bakmaya gerek yoktur, çünkü vezirler satır satır konur ve *s*. satıra vezir konurken alt satırlar boştur. Aynı satıra da bakılmaz, çünkü her satırda en çok bir vezir vardır. İlk döngü sütundaki *s* hücreyi tarar. İkinci döngü *(s − 1, k − 1)*'den başlayıp satırı ve sütunu birlikte azaltarak sol üst köşeye doğru ilerler. Üçüncü döngü satırı azaltırken sütunu artırır. Döngü koşulları tahtanın dışına çıkılmasını engeller: `j >= 0` sol kenarı, `j < n` sağ kenarı korur. Herhangi bir döngü bir vezir bulursa metot hemen `false` döndürür.

```
boolean coz(int[][] t, int satir) {
  if (satir == n)
    return true;
  for (int sutun = 0; sutun < n;
       sutun++) {
    if (guvenliMi(t, satir, sutun)) {
      t[satir][sutun] = 1; // yerleştir
      if (coz(t, satir + 1))
        return true;
      t[satir][sutun] = 0; // geri al
    }
  }
  return false;
}
```

`coz` şablonun doğrudan uygulamasıdır. `satir == n` hedef testidir: *n* vezir yerleşmişse çözüm bulunmuştur. Döngü satırdaki her sütunu dener. `guvenliMi` geçerlilik testidir ve budamayı yapar. Güvenli bir kareye vezir konur (seç), bir sonraki satır için `coz` çağrılır (keşfet). Çağrı `true` döndürürse çözüm bulunmuştur. Bu `true` değeri hemen yukarı iletilir; tahta olduğu gibi kalır ve çözümü içerir. Çağrı `false` döndürürse bu sütunla devam etmenin yolu yoktur; vezir kaldırılır (geri al) ve döngü sıradaki sütunla sürer. Döngü biterse bu satırda hiçbir sütun işe yaramamıştır ve `false` döndürülür. Bu `false`, bir üst düzeydeki `coz`'u kendi vezirini kaldırmaya zorlar.

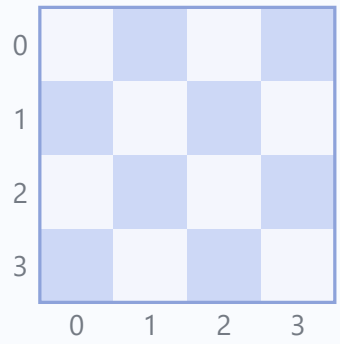
Maliyet iki parçadan oluşur. `guvenliMi` en çok üç döngü çalıştırır; her döngü en çok n adımdır. Bu yüzden bir çağrı $O(n)$ sürer. `coz` için en kötü durum, durum uzayı ağacının tamamının gezilmesidir. İlk satırda n, ikincide en çok $n - 1$ güvenli sütun vardır, çünkü en azından bir sütun tutulmuştur. Bu sınırla düğüm sayısı $O(n!)$'dir. Her düğümde n sütun için `guvenliMi` çağrılır. Gerçek düğüm sayısı $n!$'den çok daha küçüktür, ama kaba üst sınır $O(n!)$'dir. Bellek kullanımı tahta için $O(n^2)$, çağrı yığını için $O(n)$'dir.

Sık yapılan hata

Özyinelemeli çağrının sonucunu yaymamak. `if (coz(t, satir + 1)) return true;` yerine yalnız `coz(t, satir + 1);` yazılırsa çözüm bulunduğunda bile döngü sürer. Bir sonraki kod satırında `t[satir][sutun] = 0` çalışır ve çözüm tahtadan silinir. Arama ağacın tamamını gezer ve sonunda `false` döner; tahta boş kalır.

9.3.5 Kod izleme: coz(t, 0), n = 4

Kodu satır satır izlemek, ağaç izlemesinde görünmeyen bir ayrıntıyı ortaya çıkarır: her güvenlik testi bir `guvenliMi` çağrısıdır ve metot *ilk* bulduğu tehditte döner. Metot önce sütunu, sonra sol üst çaprazı, en son sağ üst çaprazı denetler. Bir kare birden çok vezirce tehdit ediliyorsa yalnız ilk denetimin bulduğu tehdit görünür. Örneğin (2, 2) karesi hem (1, 2)'deki vezirin sütunundadır hem de (0, 0)'daki vezirin çaprazındadır. Metot önce sütunu taradığı için tehdidi (1, 2)'de bulur.



Şekil 9.16: `coz(t, 0)` çağrıldı; tahta boş, `satir = 0`.

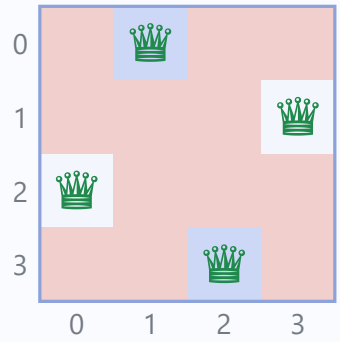
#	Kare	`guvenliMi`	Neden / işlem	Yer	Yığın derinliği
1	(0, 0)	true	`t[0][0] = 1`, `coz(t, 1)`	[0, –, –, –]	2
2	(1, 0)	false	aynı sütun: (0, 0)	[0, –, –, –]	2
3	(1, 1)	false	sol üst çapraz: (0, 0)	[0, –, –, –]	2

#	Kare	`guvenliMi`	Neden / işlem	Yer	Yığın derinliği
4	(1, 2)	true	`t[1][2] = 1`, `coz(t, 2)`	[0, 2, –, –]	3
5	(2, 0)	false	aynı sütun: (0, 0)	[0, 2, –, –]	3
6	(2, 1)	false	sağ üst çapraz: (1, 2)	[0, 2, –, –]	3
7	(2, 2)	false	aynı sütun: (1, 2)	[0, 2, –, –]	3
8	(2, 3)	false	sol üst çapraz: (1, 2)	[0, 2, –, –]	3
—	—	—	satır 2'de döngü bitti, `return false`; geri al: `t[1][2] = 0`	[0, –, –, –]	2
9	(1, 3)	true	`t[1][3] = 1`, `coz(t, 2)`	[0, 3, –, –]	3
10	(2, 0)	false	aynı sütun: (0, 0)	[0, 3, –, –]	3
11	(2, 1)	true	`t[2][1] = 1`, `coz(t, 3)`	[0, 3, 1, –]	4
12	(3, 0)	false	aynı sütun: (0, 0)	[0, 3, 1, –]	4
13	(3, 1)	false	aynı sütun: (2, 1)	[0, 3, 1, –]	4
14	(3, 2)	false	sol üst çapraz: (2, 1)	[0, 3, 1, –]	4
15	(3, 3)	false	aynı sütun: (1, 3)	[0, 3, 1, –]	4
—	—	—	satır 3 çıkmaz; geri al: `t[2][1] = 0`	[0, 3, –, –]	3
16	(2, 2)	false	sol üst çapraz: (0, 0)	[0, 3, –, –]	3
17	(2, 3)	false	aynı sütun: (1, 3)	[0, 3, –, –]	3
—	—	—	satır 2 çıkmaz; geri al: `t[1][3] = 0`; sütun 4 = n, satır 1'de de döngü biter	[0, –, –, –]	2
—	—	—	satır 1 çıkmaz; geri al: `t[0][0] = 0`	[–, –, –, –]	1
18	(0, 1)	true	`t[0][1] = 1`, `coz(t, 1)`	[1, –, –, –]	2
19	(1, 0)	false	sağ üst çapraz: (0, 1)	[1, –, –, –]	2
20	(1, 1)	false	aynı sütun: (0, 1)	[1, –, –, –]	2

#	Kare	`guvenliMi`	Neden / işlem	Yer	Yığın derinliği
21	(1, 2)	false	sol üst çapraz: (0, 1)	[1, –, –, –]	2
22	(1, 3)	true	`t[1][3] = 1`, `coz(t, 2)`	[1, 3, –, –]	3
23	(2, 0)	true	`t[2][0] = 1`, `coz(t, 3)`	[1, 3, 0, –]	4
24	(3, 0)	false	aynı sütun: (2, 0)	[1, 3, 0, –]	4
25	(3, 1)	false	aynı sütun: (0, 1)	[1, 3, 0, –]	4
26	(3, 2)	true	`t[3][2] = 1`, `coz(t, 4)` → `satir == n`, `true`	[1, 3, 0, 2]	5

Tablo 9.4: `coz(t, 0)` çağrısının izlemesi ($n = 4$). "Yer" sütunu adımdan sonraki yerleşimi satır sırasıyla verir; – boş satırdır.

Tablo kodun bütün davranışını gösterir. `guvenliMi` 26 kez çağrılır. Bunların 8'i `true` döner; bu 8 çağrı yerleştirmedir. 4 kez geri alma yapılır. Yığın derinliği, o an açık olan `coz` çerçevelerinin sayısıdır. Derinlik 5'e ulaştığında `coz(t, 4)` çerçevesi açıktır ve hedef testi `true` döner. Bu değer dört çerçeve boyunca yukarı taşınır. Hiçbir çerçeve kendi vezirini kaldırmaz; tahta çözümü içerir.



Şekil 9.17: `satir == n` olunca `true` değeri yığın boyunca yukarı taşınır; tahtada çözüm [1, 3, 0, 2] kalır.

Örnek 9.4

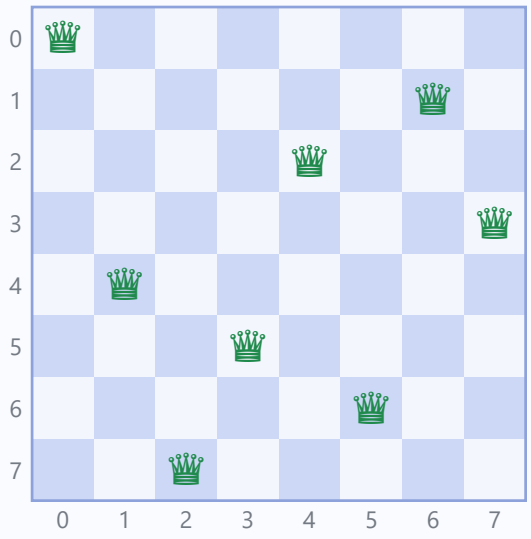
Soru. $n = 5$ için `coz(t, 0)` ilk çözümü nasıl bulur? Kaç yerleştirme ve kaç geri dönüş yapılır?

Çözüm. Satır 0'da sütun 0 güvenlidir. Satır 1'de sütun 0 aynı sütunda, sütun 1 çaprazdadır; sütun 2 güvenlidir. Satır 2'de sütun 0, 1, 2 ve 3 tehdit altındadır; sütun 4 güvenlidir. Satır 3'te sütun 0 tehdit altında, sütun 1 güvenlidir. Satır 4'te sütun 0, 1 ve 2 tehdit altında, sütun 3 gü-

venlidir. Çözüm [0, 2, 4, 1, 3]'tür. 5 yerleştirme yapılır ve hiç geri dönülmez: her satırda ilk güvenli sütun doğru seçimdir. 4 vezirde 8 yerleştirme ve 4 geri dönüş gerekiyordu. Daha büyük bir problem daha az iş gerektirebilir; ilk çözümü bulmanın maliyeti *n* ile düzgün artmaz.

9.3.6 8 vezir: algoritmanın ilk çözümü

`coz(t, 0)` *n = 8* için çalıştırıldığında sütunlar küçükten büyüğe denendiği için her zaman sözlüksel sıradaki ilk çözüm bulunur. Bu çözüm [0, 4, 7, 5, 2, 6, 1, 3]'tür: satır 0'daki vezir sütun 0'da, satır 1'deki sütun 4'te ve böyle sürer.



Şekil 9.18: Algoritmanın 8 vezir için bulduğu ilk çözüm. Satır 0–7 için sütunlar: [0, 4, 7, 5, 2, 6, 1, 3].

Bu çözüme ulaşana kadar `guvenliMi` 876 kez çağrılır, 113 vezir yerleştirilir ve 105 kez geri dönülür. Sayılar arasındaki ilişki dikkat çekicidir: *113 − 105 = 8*. Her yerleştirme ya geri alınır ya da çözümde kalır; çözümde 8 vezir kaldığına göre geri kalan 105 yerleştirme geri alınmıştır. Bu eşitlik ilk çözümde duran her n-vezir aramasında geçerlidir: *yerleştirme − geri alma = n*.

Şekil 9.11'deki a1, b7, c4, d6, e8, f2, g5, h3 yerleşimi de bir çözümdür, ama algoritmanın ilk bulduğu çözüm değildir. 92 çözümün hepsi geçerlidir; algoritma yalnız aday sırasının belirlediği ilkinde durur. Bütün çözümleri bulmak için hedef testinde çözüm kaydedilir ve `false` döndürülür. `false`, aramanın sürmesi anlamına gelir: üstteki çağrı vezirini kaldırır ve sıradaki sütunu dener.

9.3.7 Çözüm ve yerleştirme sayıları

Algoritmanın *n* büyüdükçe nasıl davrandığını görmek için iki sayı ölçülür: bütün çözümleri bulmak için yapılan güvenli yerleştirme sayısı ve ilk çözüme kadar yapılan yerleştirme sayısı. “Yerleştirme”, güvenli bulunup tahtaya konan vezir sayısıdır; durum uzayı ağacındaki kök dışı düğüm sayısına eşittir.

n	Çözüm	n^n (kaba)	Yerleştirme (bütün çözümler)	İlk çözüme kadar
1	1	1	1	1
2	0	4	2	—
3	0	27	5	—
4	2	256	16	8
5	10	3.125	53	5
6	4	46.656	152	31
7	40	823.543	551	9
8	92	16.777.216	2056	113
9	352	387.420.489	8393	41
10	724	10.000.000.000	35.538	102

Tablo 9.5: n-vezir için çözüm sayısı, kaba kuvvetin tam yerleşim sayısı ve geri izlemenin yaptığı yerleştirmeler.

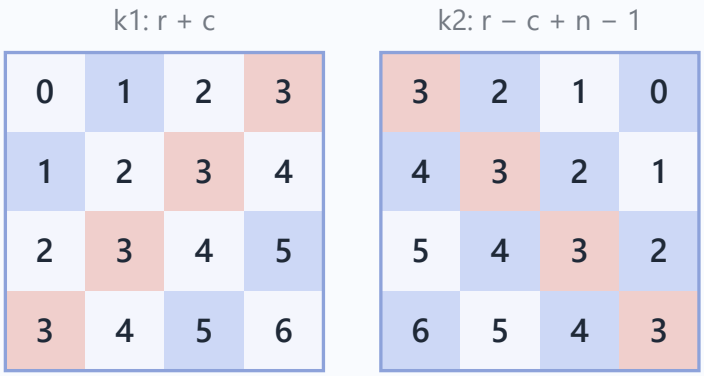
Tablodan üç sonuç çıkar.

- *n = 2* ve *n = 3* için çözüm yoktur. Algoritma bütün ağacı gezer ve `false` döndürür. *n = 3*'te ağacın kök dışında yalnız 5 düğümü vardır; bu 5 yerleştirmenin hepsi geri alınır. Bu, "çözüm yok" cevabının kanıtıdır: arama tamdır ve hiçbir olasılığı atlamamıştır.
- Bütün çözümleri bulmanın maliyeti n^n'den çok küçüktür, ama yine de hızla büyür. *n = 9*'dan 10'a geçişte yerleştirme sayısı yaklaşık 4,2 kat artar. Büyüme kabaca *n!* gibidir: *10! = 3.628.800*'ün yaklaşık yüzde biri.
- İlk çözüme kadar geçen iş *n* ile düzgün artmaz. *n = 6*'da 31 yerleştirme gerekirken *n = 7*'de yalnız 9 yeter. *n = 6*'nın sözlüksel ilk çözümü [1, 3, 5, 0, 2, 4] sütun 1 ile başlar; sütun 0 ile başlayan bütün dallar boşa gezilir. *n = 7*'nin ilk çözümü [0, 2, 4, 6, 1, 3, 5] ise neredeyse hiç geri dönüş gerektirmez: yalnız 2 geri alma yapılır.

9.3.8 O(1) güvenlik testi

`guvenliMi` her çağrıda sütunu ve iki çaprazı tarar ve *O(n)* sürer. 8 vezirde ilk çözüme kadar 876 çağrı yapıldığı düşünülürse bu maliyet önemsiz değildir. Tarama yerine her sütun ve her çapraz için bir bayrak tutulursa test *O(1)*'e iner. Bunun için çaprazların nasıl numaralandırılacağı bilinmelidir.

Bir kare *(r, c)* ile gösterilsin. Sağ üstten sol alta inen **ters köşegen** üzerinde bir kareden ötekine geçerken satır bir artar, sütun bir azalır. Bu yüzden *r + c* toplamı sabit kalır. Sol üstten sağ alta inen **köşegende** ise satır ve sütun birlikte artar; *r − c* farkı sabittir. *n × n* tahtada *r + c*, 0 ile *2n − 2* arasında değer alır; *2n − 1* ters köşegen vardır. *r − c* ise *−(n − 1)* ile *n − 1* arasındadır. Bu değere *n − 1* eklenirse indis 0'dan başlar ve yine *2n − 1* değer çıkar.



Şekil 9.19: 4 × 4 tahtada her karenin `k1` ($r + c$) ve `k2` ($r - c + n - 1$) indisi. Vurgulu kareler aynı indisi paylaşır: soldakiler 3 numaralı ters köşegeni, sağdakiler 3 numaralı köşegeni oluşturur.

```
boolean[] sut = new boolean[n];
boolean[] k1 = new boolean[2*n-1]; // r+c
boolean[] k2 = new boolean[2*n-1]; // r-c+n-1

void coz(int r) {
  if (r == n) { sayac++; return; }
  for (int c = 0; c < n; c++) {
    int a = r + c, b = r - c + n - 1;
    if (sut[c] || k1[a] || k2[b])
      continue;
    sut[c] = k1[a] = k2[b] = true;
    coz(r + 1);
    sut[c] = k1[a] = k2[b] = false;
  }
}
```

Kodda üç `boolean` dizisi vardır. `sut[c]`, c. sütunda vezir olup olmadığını; `k1[a]`, $a = r + c$ numaralı ters köşegende; `k2[b]`, $b = r - c + n - 1$ numaralı köşegende vezir olup olmadığını gösterir. Bir karenin güvenli olup olmadığı üç dizi erişimiyle anlaşılır. Bayraklardan biri `true` ise kare tehdit altındadır ve `continue` ile sıradaki sütuna geçilir. Değilse üç bayrak birden `true` yapılır (seç), bir alt satır çözülür (keşfet), sonra üç bayrak `false` yapılır (geri al).

Bu sürüm iki açıdan da farklıdır. Birincisi, tahta dizisine gerek kalmaz. Bayraklar çakışmayı tek başına denetler. Çözümün kendisi gerekiyorsa bir `yer[satir] = sutun` dizisi tutmak yeter. İkincisi, metot `void`'dir ve ilk çözümde durmaz. Hedef testinde `sayac` artırılır ve metot döner; arama sürer. Bu yüzden kod bütün çözümleri sayar. $n = 8$ için `sayac` 92 olur ve 2056 yerleştirme yapılır; bu sayılar tahta tabanlı sürümle aynıdır, çünkü iki sürüm aynı ağacı aynı sırayla gezer. Değişen yalnız her düğümdeki iştir.

Örnek 9.5

Soru. *n = 8* için bayraklı sürümde yalnız (3, 1) karesinde bir vezir vardır. Hangi bayraklar `true` 'dur? (5, 3), (1, 3) ve (4, 4) kareleri güvenli midir?

Çözüm. `sut[1]`, `k1[3 + 1] = k1[4]` ve `k2[3 − 1 + 7] = k2[9]` `true` 'dur. (5, 3) için *a = 8*, *b = 5 − 3 + 7 = 9*; `k2[9]` doludur. Kare güvenli değildir: (3, 1), (4, 2), (5, 3) aynı köşegendedir. (1, 3) için *a = 4*; `k1[4]` doludur. (3, 1), (2, 2), (1, 3) aynı ters köşegendedir; kare güvenli değildir. (4, 4) için `sut[4]`, `k1[8]` ve `k2[7]` boştur; kare güvenlidir.

Sık yapılan hata

Köşegen indisinde *n − 1*'i eklemeyi unutmak. `k2[r - c]` yazılırsa (0, 3) karesi için indis *−3* olur ve `ArrayIndexOutOfBoundsException` fırlar. Dizinin boyunu *n* almak da aynı hatayı verir: köşegen sayısı *n* değil *2n − 1*'dir.

9.3.9 Bütün çözümleri saymak

Tahta tabanlı `coz` da bütün çözümleri sayacak biçimde değiştirilebilir. Değişiklik yalnız hedef testindedir.

```
int sayac = 0;

boolean coz(int[][] t, int satir) {
  if (satir == n) {
    sayac++;          // çözümü say
    return false;     // aramayı sürdür
  }
  for (int sutun = 0; sutun < n; sutun++) {
    if (guvenliMi(t, satir, sutun)) {
      t[satir][sutun] = 1;
      coz(t, satir + 1);
      t[satir][sutun] = 0;   // geri al
    }
  }
  return false;
}
```

Hedef testinde `sayac` artırılır ve `false` döndürülür. `false` değeri üst çağrıya "bu yol bitti, aramayı sürdür" der. Üst çağrı vezirini kaldırır ve sıradaki sütunu dener. Metot hiçbir zaman `true` döndürmediği için özyinelemeli çağrının sonucuna bakmaya da gerek kalmaz. Kod *n = 4, 5, 6, 7, 8* için sırasıyla 2, 10, 4, 40 ve 92 çözüm sayar.

Hedef testinde `true` döndürmek ise yanlış olurdu. Bu durumda ilk çözümde bütün çağrılar sona erer ve `sayac` 1'de kalır. Aynı şekilde geri alma satırını silmek de yanlış olurdu. Tahta önceki denemelerin vezirleriyle dolar ve sonraki denemeler hatalı biçimde budanır.

Bütün çözümler aranırken tahtanın simetrisinden yararlanılabilir. Bir çözüm dikey eksene göre yansıtılırsa yine bir çözüm elde edilir. İlk satırdaki vezir sütun c'deyse yansıması $n - 1 - c$'dedir. Bu yüzden ilk vezir yalnız sol yarıda denenir ve bulunan her çözüm iki kez sayılır. n tekse orta sütunda başlayan çözümlerin yansıması da orta sütunda başlar; onlar bir kez sayılır. $n = 8$ için bu yöntem 2056 yerine 1028 yerleştirmeyle aynı 92 sonucuna ulaşır: iş tam yarıya iner. $n = 9$'da yerleştirme 8393'ten 4703'e düşer; orta sütun tek başına sayıldığı için kazanç biraz azdır.

9.4 En az çakışma sezgiseli

9.4.1 Tam yerleşimden onarıma

Geri izleme çözümü boş bir tahtadan başlayarak kurar ve her adımda geçerliliği korur. n-vezir problemine bambaşka bir yoldan da yaklaşılabilir. Önce bütün vezirler tahtaya konur; her sütuna bir vezir. Bu yerleşim büyük olasılıkla geçersizdir: bazı vezirler birbirini tehdit eder. Sonra çakışmalar küçük, yerel değişikliklerle tek tek onarılır. Bu yaklaşımın adı **en az çakışma sezgiselidir** (*min-conflicts heuristic*).

> **Tanım 9.6**
>
> **En az çakışma sezgiseli**, tam ama geçersiz olabilen bir atamadan başlayan bir yerel arama yöntemidir. Her adımda kısıtları bozan değişkenlerden biri seçilir. Bu değişkene, öteki değişkenlerle en az çakışma yaratan değer verilir. Çakışma kalmayınca ya da adım sınırına ulaşılınca arama durur.

n-vezir için değişkenler sütunlardır ve her değişkenin değeri o sütundaki vezirin satırıdır. `yer[k] = r`, k. sütundaki vezirin r. satırda olduğunu gösterir. Sütun kısıtı yapıya gömülüdür; her sütunda tam olarak bir vezir vardır. Çakışmalar satırlarda ve çaprazlarda çıkar.

```
enAzCakisma(n, maksAdim):
  // başlangıç: açgözlü yerleşim
  for k = 0 .. n-1:
    yer[k] = en az çakışmalı satır
  for adim = 1 .. maksAdim:
    if çakışma yok: return yer
    k = rastgele çakışan sütun
    yer[k] = argmin_r cakisma(k, r)
             // eşitlikte rastgele
  return başarısız
```

Sözde kodun iki evresi vardır. **Başlangıç** evresi vezirleri sütun sütun yerleştirir. Her sütunda, o ana kadar konmuş vezirlerle en az çakışan satır seçilir; eşitlik varsa satırlardan biri rastgele alınır. Bu, açgözlü bir başlangıçtır ve çoğu zaman yalnız birkaç çakışma bırakır. **Onarım** evresi her adımda çakışan vezirlerden birini rastgele seçer. Seçilen vezir kendi sütunu içinde, en az çakışma yaratan satıra taşınır. Burada da eşitlik rastgele bozulur. Çakışma kalmadığında yerleşim bir çözümdür. `maksAdim` adım içinde çözüm bulunamazsa arama başarısız sayılır.

9.4.2 Çakışma sayısı

Algoritmanın temel hesabı şudur: *k*. sütundaki vezir *r*. satıra taşınsa kaç vezirle çakışır? Bu sayı, aynı satırdaki ve aynı çaprazlardaki öteki vezirlerin sayısıdır.

```
int cakisma(int[] yer, int k, int r) {
  int c = 0;
  for (int j = 0; j < yer.length; j++) {
    if (j == k) continue;                      // vezirin kendisi
    if (yer[j] == r                            // aynı satır
        || Math.abs(yer[j] - r) == Math.abs(j - k))   // çapraz
      c++;
  }
  return c;
}
```

Metot bütün sütunları tarar. Vezirin kendi sütunu atlanır. *j*. sütundaki vezir aynı satırdaysa sayaç artar. Satır farkının mutlak değeri sütun farkının mutlak değerine eşitse iki kare aynı çaprazdadır; sayaç yine artar. Hesap $O(n)$ sürer. Bir sütunun bütün satırları için hesaplanırsa $O(n^2)$ olur. Büyük n için her satırda, her köşegende ve her ters köşegende kaç vezir olduğu sayaçlarla tutulur. Bu durumda tek bir çakışma sayısı $O(1)$'de okunur.

Örnek 9.6

Soru. 4×4 tahtada `yer = [1, 3, 0, 3]` 'tür. Hangi vezirler çakışır? Onarım için sütun 3 seçilirse bu vezirin her satır için çakışma sayısı nedir? Vezir nereye taşınır?

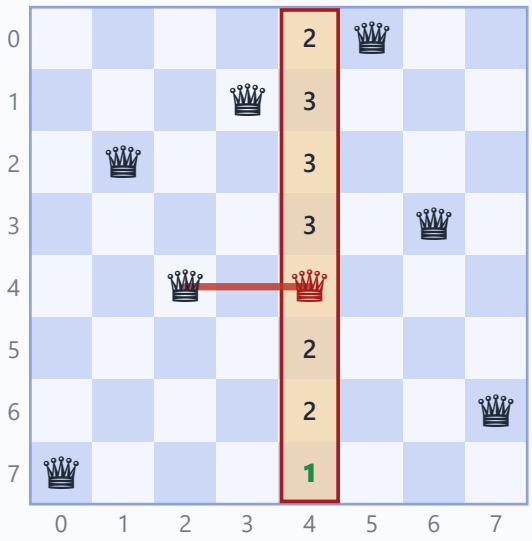
Çözüm. Vezirler (satır, sütun) olarak (1, 0), (3, 1), (0, 2) ve (3, 3)'tedir. Sütun 1 ile sütun 3'teki vezirler aynı satırdadır (3); başka çakışma yoktur. Sütun 3 için hesap:

- Satır 0: (0, 2) aynı satırda: 1.
- Satır 1: (1, 0) aynı satırda; (0, 2) çaprazda (satır farkı 1, sütun farkı 1); (3, 1) çaprazda (2 ve 2): 3.
- Satır 2: hiçbir vezirle çakışmaz: 0.
- Satır 3: (3, 1) aynı satırda: 1.

Sayılar 1, 3, 0, 1'dir. En küçük değer 0, yalnız satır 2'dedir. Vezir oraya taşınır ve yerleşim [1, 3, 0, 2] olur. Hiçbir çakışma kalmaz; tek onarım adımıyla 4 vezir çözümü bulunmuştur.

9.4.3 Onarım adımları

8 vezirde algoritmanın bir çalışması şöyle ilerler. Açgözlü başlangıç `yer = [7, 2, 4, 1, 4, 0, 3, 6]` yerleşimini üretir. Bu yerleşimde yalnız bir çift çakışır: sütun 2 ve sütun 4'teki vezirler aynı satırdadır (4). Şekillerde vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir; vezirin kendi satırı boş bırakılmıştır.



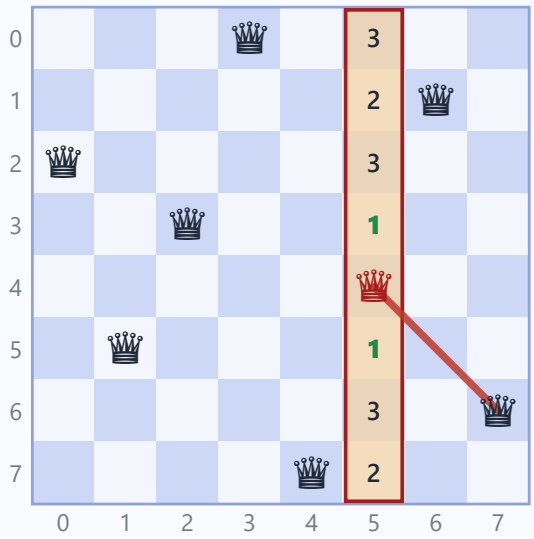
Şekil 9.20: Açgözlü başlangıç. Sütun 4'teki vezir seçildi. Sütunundaki çakışma sayıları satır 0'dan 7'ye 2, 3, 3, 3, (kendi satırı), 2, 2, 1'dir; en küçüğü satır 7'dedir.

Adım	`yer`	Çakışan	Seçilen	En küçük değer	Karar
1	[7, 2, 4, 1, 4, 0, 3, 6]	2 ve 4 (satır 4)	sütun 4	1 (satır 7)	satır 7'ye
2	[7, 2, 4, 1, 7, 0, 3, 6]	0 ve 4 (satır 7)	sütun 0	1 (satır 0, 5)	rastgele biri
…	birkaç onarım adımı				
3	[2, 5, 3, 0, 7, 3, 1, 6]	2 ve 5 (satır 3)	sütun 5	1 (satır 4, 5)	satır 4'e
4	[2, 5, 3, 0, 7, 4, 1, 6]	5 ve 7 (çapraz)	sütun 5	1 (satır 3, 5)	plato; satır 5'e
5	[2, 5, 3, 0, 7, 5, 1, 6]	1 ve 5 (satır 5)	sütun 1	1 (satır 7)	satır 7'ye
6	[2, 7, 3, 0, 7, 5, 1, 6]	1 ve 4 (satır 7)	sütun 4	1 (satır 0, 2)	satır 2'ye
7	[2, 7, 3, 0, 2, 5, 1, 6]	0 ve 4 (satır 2)	sütun 0	0 (satır 4)	satır 4'e
8	[4, 7, 3, 0, 2, 5, 1, 6]	yok	—	—	çözüm

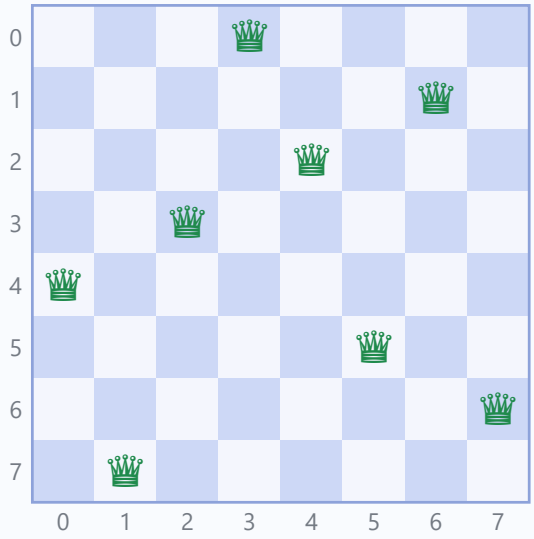
Tablo 9.6: En az çakışma sezgiselinin bir çalışması ($n = 8$). `yer[k]`, k. sütundaki vezirin satırıdır.

Tablo yerel aramanın iki önemli davranışını gösterir. Birincisi, her onarım çakışmayı gidermez; bazen yerini değiştirir. 1. adımda sütun 4'teki vezir satır 7'ye taşınır. Sütun 2 ile çakışma biter, ama sütun 0'daki vezir de satır 7'dedir ve yeni bir çakışma doğar. Çakışan çift sayısı 1'de kalır. Algoritma çakışma sayısını hiç artırmaz, ama azaltmayı da garanti etmez.

İkincisi, algoritma **platolara** takılabilir. 4. adımda sütun 5'teki vezir satır 4'tedir ve (6, 7) karesindeki vezirle aynı çaprazdadır. Satır 3, 4 ve 5'in hepsinde çakışma sayısı 1'dir. Hiçbir taşıma durumu iyileştirmez; bu bir platodur. Sezgisel eşitliği rastgele bozar ve vezir satır 5'e geçer. Orada bu kez sütun 1 ile aynı satırdadır. Sütun 5 platoda 3, 4 ve 5 arasında birkaç kez gidip gelebilir. Rastgele seçim sayesinde sonunda başka bir çakışan vezir, sütun 1, seçilir ve arama platodan çıkar. 7. adımda sütun 0 için satır 4'te 0 çakışma vardır; vezir oraya taşınır ve çözüm bulunur.



Şekil 9.21: Plato. Sütun 5'teki vezir için satır 3, 4 ve 5'in hepsinde çakışma 1'dir; taşıma durumu iyileştirmez.



Şekil 9.22: Çakışan vezir kalmadı: çözüm [4, 7, 3, 0, 2, 5, 1, 6].

Bulunan çözüm, satır sırasıyla değil sütun sırasıyla yazılmıştır. Geri izleme algoritmasının [0, 4, 7, 5, 2, 6, 1, 3] çözümüyle karşılaştırılırken bu fark akılda tutulmalıdır. İki gösterim de bir permütasyondur; biri satırdan sütuna, öteki sütundan satıra eşlemedir.

9.4.4 Yerel arama ve tamlık

En az çakışma bir **yerel arama** (*local search*) yöntemidir. Geri izlemeden üç temel farkı vardır.

- **Durum.** Geri izlemede durum kısmi ama her zaman geçerlidir. Yerel aramada durum tamdır, ama geçersiz olabilir.
- **Bellek.** Yerel aramada geri dönüş yığını yoktur. Algoritma yalnız şu anki yerleşimi tutar; geçmiş kararları hatırlamaz.
- **Tamlık.** Yerel arama **tam değildir**. Platolarda dolaşabilir ve döngüye girebilir; bu yüzden bir adım sınırı gerekir. Yalnız bir çözüm bulur ve çözüm olmadığını kanıtlayamaz. Adım sınırına ulaşmak "çözüm yok" anlamına gelmez; yalnız "bu çalışmada bulunamadı" anlamına gelir.

Buna karşılık en az çakışma büyük n için şaşırtıcı biçimde hızlıdır. Minton ve arkadaşlarının 1992'deki çalışmasına göre iyi bir açgözlü başlangıçla bir milyon vezirlik problem bile ortalama birkaç on onarım adımında çözülür. Geri izleme ise n büyüdükçe hızla yavaşlar. İki yöntemin güçlü yanları birbirini tamamlar. Bütün çözümleri saymak ya da çözüm olmadığını göstermek için geri izleme gerekir. Çok büyük bir örnekte tek bir çözüm hızla isteniyorsa yerel arama daha uygundur.

> **Sık yapılan hata**
>
> Yerel aramayı tam sanmak. $n = 3$ için en az çakışma her zaman adım sınırına ulaşır, çünkü çözüm yoktur. Ama $n = 100$ için adım sınırına ulaşmak da olasıdır: sınır küçük seçilmiş ya da arama kötü bir platoya düşmüş olabilir. İki durum algoritmanın çıktısına bakarak ayırt edilemez. Yerel arama "çözüm yok" cevabını kanıtlayamaz.

9.5 Sudoku

9.5.1 Kısıt sağlama problemi olarak Sudoku

Sudoku'nun amacı 9×9'luk bir tabloyu doldurmaktır. Her satırda, her sütunda ve dokuz 3×3'lük kutunun her birinde 1'den 9'a kadar rakamlar birer kez kullanılır. Bazı hücreler baştan verilir; bunlara **ipucu** denir. Kalan hücreler bu kurallara uyacak biçimde doldurulur. İyi kurulmuş bir bulmacanın tek bir çözümü vardır.

Sudoku, daha genel bir problem türünün örneğidir.

Tanım 9.7

Kısıt sağlama problemi (*constraint satisfaction problem*, CSP) üç parçadan oluşur. Birincisi bir değişkenler kümesidir. İkincisi her değişkenin alabileceği değerlerin kümesidir (**alan**, *domain*). Üçüncüsü değişkenlerin birlikte uyması gereken **kısıtlardır**. Çözüm, her değişkene alanından bir değer veren ve bütün kısıtları sağlayan bir atamadır.

Sudoku'da değişkenler boş hücrelerdir. Her değişkenin alanı *{1, ..., 9}*'dur. Kısıtlar "aynı birimde farklı" biçimindedir: aynı satırdaki, aynı sütundaki ve aynı kutudaki hücreler farklı rakam almalıdır. n-vezir de bir kısıt sağlama problemidir: değişkenler satırlar, alanlar sütunlardır. Çizge boyamada değişkenler düğümler, alanlar renklerdir. Geri izleme kısıt sağlama problemlerinin genel çözüm yöntemidir.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

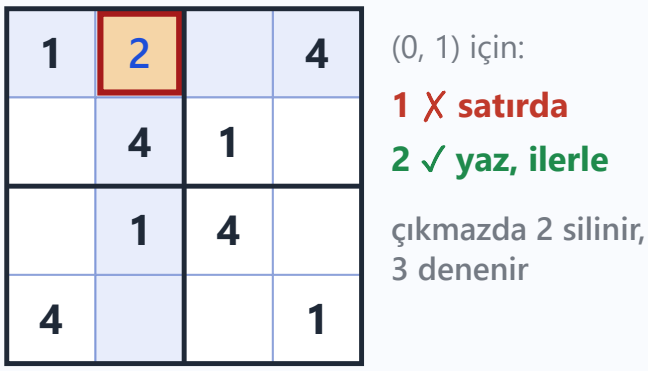
Şekil 9.23: Örnek bulmaca. 30 hücre ipucudur, 51 hücre boştur.

9.5.2 Algoritma

Geri izleme algoritması Sudoku'yu şu adımlarla çözer.

1. Boş bir hücreye bir rakam atanır.
2. Atanan rakam geçerliyse, yani aynı satırda, sütunda ve kutuda yoksa, bir sonraki boş hücreye geçilir.
3. Geçerli değilse sıradaki rakam denenir.
4. Bir hücrede bütün rakamlar denenmiş ve hiçbiri işe yaramamışsa önceki hücreye geri dönülür ve orada farklı bir rakam denenir.
5. Adımlar bütün hücreler dolana kadar tekrarlanır.

Bu, genel şablonun doğrudan uygulamasıdır. Kısmi çözüm doldurulmuş hücrelerdir. Adaylar 1–9 rakamlarıdır. Kısıt testi `gecerliMi`'dir. Hedef, boş hücre kalmamasıdır.



Şekil 9.24: 4 × 4'lük küçük bir Sudoku'da (0, 1) hücresi: 1 satırda var, reddedilir; 2 geçerlidir, yazılır ve sonraki hücreye geçilir. İleride bir çıkmaza girilirse 2 silinir ve 3 denenir.

Şekildeki küçük bulmaca *2 × 2*'lik kutularla oynanır ve rakamlar 1–4'tür. Bu bulmacada hiç çıkmaza girilmez: (0, 1)'e 2, (0, 2)'ye 3 yazılır ve kalan altı boş hücre de ilk geçerli rakamla dolar. Sekiz yerleştirmede çözüm bulunur. Büyük bulmacada durum çok farklıdır.

9.5.3 Java kodu

```
boolean coz(int[][] t) {
  for (int s = 0; s < 9; s++)
    for (int k = 0; k < 9; k++)
      if (t[s][k] == 0) {
        for (int r = 1; r <= 9; r++)
          if (gecerliMi(t, s, k, r)) {
            t[s][k] = r;
            if (coz(t)) return true;
            t[s][k] = 0;   // geri al
          }
        return false;  // rakam yok
      }
  return true;  // boş hücre yok
}
```

`coz` her çağrıda tabloyu baştan tarar ve ilk boş hücreyi arar. Boş hücre *(s, k)* bulunca 1'den 9'a kadar her rakamı dener. Geçerli bir rakam yazılır ve `coz` yeniden çağrılır. Bu çağrı bir sonraki boş hücreyi bulur. Çağrı `true` döndürürse çözüm bulunmuştur. `false` döndürürse rakam silinir ve sıradaki rakam denenir. Dokuz rakamın hiçbiri işe yaramazsa `return false` çalışır. Bu satır geri dönüşü tetikler: üst çağrı kendi hücresindeki rakamı siler.

Kodun yapısında ince bir nokta vardır. `return false` iki döngünün *içinde*, ilk boş hücrenin işlendiği bloğun sonundadır. Metot ilk boş hücreyi işledikten sonra kalan hücrelere bakmaz. İlk boş hücre doldurulamıyorsa geri kalan hücreleri doldurmaya çalışmanın anlamı yoktur. Döngüler hiç boş hücre

bulmadan biterse en alttaki `return true` çalışır: tablo doludur.

```
boolean gecerliMi(int[][] t,
    int s, int k, int r) {
  for (int i = 0; i < 9; i++)
    if (t[s][i] == r || t[i][k] == r)
      return false;
  int bs = s / 3 * 3;
  int bk = k / 3 * 3;
  for (int i = bs; i < bs + 3; i++)
    for (int j = bk; j < bk + 3; j++)
      if (t[i][j] == r)
        return false;
  return true;
}
```

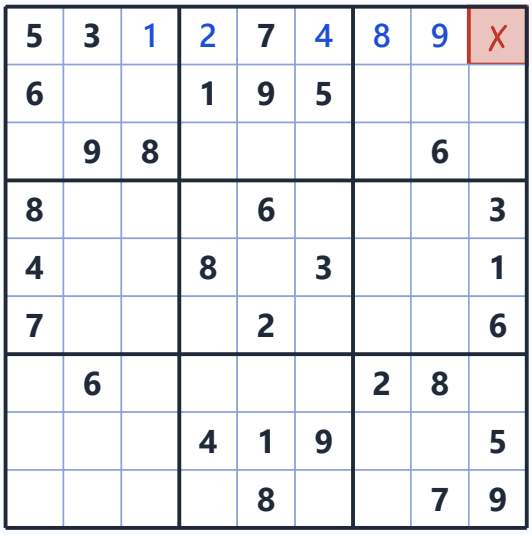
`gecerliMi`, *r* rakamının *(s, k)* hücresine yazılıp yazılamayacağını denetler. İlk döngü tek geçişte hem satırı (`t[s][i]`) hem sütunu (`t[i][k]`) tarar. Ardından hücrenin kutusu taranır. Kutunun sol üst köşesi `bs = s / 3 * 3` ve `bk = k / 3 * 3` ile bulunur. Tam sayı bölmesi aşağı yuvarlar; 3 ile çarpmak da sonucu kutu numarasından indise çevirir. (7, 4) hücresi için *7 / 3 = 2*, *2 · 3 = 6*; *4 / 3 = 1*, *1 · 3 = 3*. Kutu 6–8. satırları ve 3–5. sütunları kapsar. Metot satır ve sütun için 18, kutu için 9 olmak üzere en çok 27 hücreye bakar; bu sabit bir iştir.

Sık yapılan hata

Kutu köşesini `s / 3` ile hesaplamak. Bu ifade kutunun numarasını (0, 1 ya da 2) verir, ilk satırın indisini değil. (7, 4) hücresi için kutu (2, 1) bulunur ve 2–4. satırlar ile 1–3. sütunlar taranır. Bu bölge yanlış kutudur. Kutu denetimini tamamen unutmak da benzer bir hata doğurur: satır ve sütun kurallarına uyan ama kutuları geçersiz "çözümler" üretilir.

9.5.4 Adım adım çözüm

Algoritma örnek bulmacada boş hücreleri satır satır, soldan sağa doldurur; her hücrede 1–9 sırayla denenir. Satır 0'da ipuçları 5, 3 ve 7'dir. İlk beş boş hücreye (0, 2), (0, 3), (0, 5), (0, 6) ve (0, 7) sırasıyla 1, 2, 4, 8 ve 9 girilir. Her biri o hücre için geçerli en küçük rakamdır. (0, 8) hücresinde hiçbir rakam geçerli değildir: satırda eksik kalan tek rakam 6'dır, o da 8. sütunda (5, 8) hücresinde vardır. Bu ilk çıkmazdır; o ana kadar 5 yerleştirme yapılmış, hiç geri alınmamıştır.



Şekil 9.25: İlk çıkmaz. Mavi rakamlar algoritmanın yazdıklarıdır. $(0, 8)$ hücresine hiçbir rakam yazılamaz (X).

Geri dönüş başlar. (0, 7)'deki 9 silinir; o hücrede 9'dan büyük rakam yoktur ve (0, 7) de çıkmaz olur. (0, 6)'daki 8 silinir ve 9 denenir. Bu kez (0, 7)'ye hiçbir rakam uymaz; 9 da silinir. (0, 6) da çıkmazdır. Geri dönüş (0, 5)'e kadar uzanır ve orada 4 yerine 6 denenir. Bu noktada 7 yerleştirme ve 4 geri alma yapılmıştır. Ardından (0, 6)'ya 4, (0, 7)'ye 9, (0, 8)'e 8 yazılır ve satır 0 ilk kez tamamlanır: 5 3 1 2 7 6 4 9 8. Bu satır kendi içinde geçerlidir, ama çözüme götürüp götürmediği henüz bilinmez.

Gerçekten de götürmez. Satır 1'de yapılan bütün denemeler çıkmaza girer. Geri dönüş satır 0'a kadar uzanır ve (0, 8)'deki 8 silinir; o ana kadar 31 yerleştirme ve 26 geri alma yapılmıştır. Erken verilen bir karar ancak çok sonra yanlış çıkabilir. Algoritma bunu ancak o karara dayanan her şeyi denedikten sonra anlar.

Satır	Yerleştirme	Geri alma	Not
0	10	4	5 3 1 2 7 6 4 9 8 (sonradan değişecek)
1	229	218	
2	377	360	ilk üç satır artık son çözümdekiyle aynı
3	395	372	
4	1520	1492	
5	4185	4151	işin büyük kısmı bu satıra kadar
6	4197	4157	
7	4202	4157	geri alma artık yok
8	4208	4157	çözüm

Tablo 9.7: Her satırın ilk kez tamamlandığı anda yapılmış toplam yerleştirme ve geri alma sayıları.

Tablo aramanın düzensiz ilerleyişini gösterir. İlk üç satır 377 yerleştirmede yerine oturur. Satır 4 ile satır 5 arasında ise 2665 yerleştirme yapılır. Bu aralıkta algoritma satır 3–5'teki seçimleri defalarca değiştirir. Satır 6'dan sonra neredeyse hiç geri dönüş kalmaz: son 11 yerleştirmenin hiçbiri geri alınmaz. Sonunda 51 boş hücrenin hepsi dolar ve bütün kısıtlar sağlanır. Toplam 4208 yerleştirme ve 4157 geri alma yapılmıştır. Fark *4208 − 4157 = 51*'dir: çözümde kalan rakamlar.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Şekil 9.26: Çözüm bulundu. Siyah rakamlar ipucu, mavi rakamlar algoritmanın yazdıklarıdır; turuncu hücre son yazılan hücredir.

9.5.5 Maliyet ve MRV

Kaba kuvvet 51 boş hücrenin her birine 9 rakamdan birini yazıp sonra denetleseydi $9^{51} \approx 4{,}6 \cdot 10^{48}$ atama gerekirdi. Budama bu sayıyı 4208'e indirir. En kötü durum için üst sınır yine $O(9^m)$'dir; m boş hücre sayısıdır. Ama pratikte ağacın küçük bir kısmı gezilir.

Aynı bulmaca çok daha az işle de çözülebilir. Satır satır ilk boş hücreyi doldurmak zorunlu değildir. Hangi hücrenin önce doldurulacağı bir seçimdir ve bu seçim aramanın boyunu belirler. **MRV** (*minimum remaining values*, en az kalan değer) sezgiseli her adımda *en az adayı kalan* boş hücreyi doldurur.

```
boolean cozMRV(int[][] t) {
  int es = -1, ek = -1, enAz = 10;
  for (int s = 0; s < 9; s++)
    for (int k = 0; k < 9; k++)
      if (t[s][k] == 0) {
        int aday = 0;
        for (int r = 1; r <= 9; r++)
          if (gecerliMi(t, s, k, r)) aday++;
        if (aday < enAz) { enAz = aday; es = s; ek = k; }
      }
  if (es < 0) return true;              // boş hücre yok
  for (int r = 1; r <= 9; r++)
    if (gecerliMi(t, es, ek, r)) {
      t[es][ek] = r;
      if (cozMRV(t)) return true;
      t[es][ek] = 0;                    // geri al
    }
  return false;
}
```

Metot önce bütün boş hücreleri tarar ve her biri için kaç geçerli rakam kaldığını sayar. Adayı en az olan hücre *(es, ek)* seçilir. Boş hücre yoksa tablo doludur. Seçilen hücrede geçerli rakamlar sırayla denenir; yapı öteki geri izleme kodlarıyla aynıdır. Bir hücrede aday sayısı 0 ise döngü hiç çalışmaz ve metot hemen `false` döner. Bu, çelişkinin erkenden fark edilmesidir.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Şekil 9.27: Bulmacanın çözümü. İki strateji de aynı tabloya ulaşır.

Strateji	Yerleştirme	Geri alma
Satır satır, ilk boş hücre	4208	4157

Strateji	Yerleştirme	Geri alma
MRV: en az adaylı hücre	51	0

Tablo 9.8: Örnek bulmacada iki hücre seçme stratejisinin maliyeti.

Bu bulmacada MRV her adımda tek adaylı bir hücre bulur. Başlangıçta 51 boş hücrenin 4'ünün tek adayı vardır. Bunlardan biri doldurulunca komşu hücrelerin adayları azalır ve yeni tek adaylı hücreler ortaya çıkar. Zincir hiç kopmaz. 51 yerleştirmenin hepsi doğrudur ve hiç geri dönüş gerekmez. Her adımdaki tarama 81 hücre ve 9 rakam için `gecerliMi` çağırdığı için adım başına iş artar. Ama adım sayısı 4208'den 51'e düştüğü için toplam iş yine çok azalır.

MRV her bulmacada geri dönüşü sıfırlamaz. Zor bulmacalarda bir noktada her boş hücrenin en az iki adayı kalır. O zaman MRV iki adaylı bir hücre seçer ve yanlış adayı denerse geri dönmek zorunda kalır. Yine de en az adaylı hücre ile başlamak, dallanmayı en küçük tutar ve çelişkileri olabildiğince erken ortaya çıkarır.

9.6 Çizge boyama

9.6.1 Problem

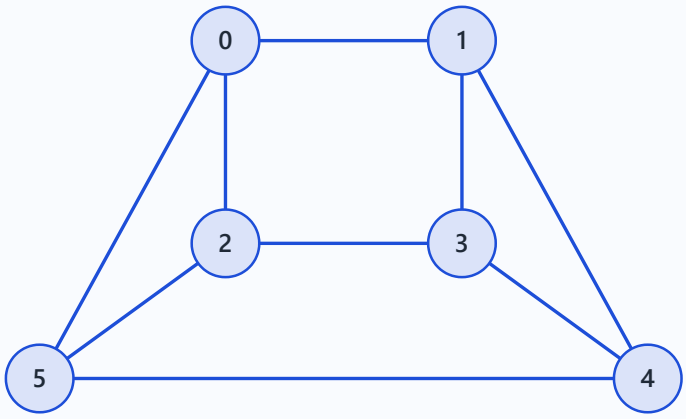
Bir haritada komşu ülkelerin farklı renklerle boyanması istenir. Ülkeler düğüm, ortak sınırlar kenar olarak düşünülürse bu soru bir çizge problemine dönüşür. Aynı problem başka kılıklarda da karşımıza çıkar. Sınav çizelgesinde aynı öğrencinin girdiği iki sınav aynı saate konamaz; sınavlar düğüm, çakışmalar kenar, saatler renktir. Derleyicide aynı anda canlı olan iki değişken aynı yazmacı paylaşamaz; değişkenler düğüm, yazmaçlar renktir.

> **Tanım 9.8**
>
> **m-boyama problemi** (*m-coloring*): bir çizgenin düğümlerine 1'den m'ye kadar renkler, kenarla bağlı iki düğüm aynı rengi almayacak biçimde atanabilir mi? Bir çizgeyi boyamaya yeten en küçük m değerine çizgenin **renk sayısı** (*chromatic number*) denir.

Geri izleme bu probleme doğal biçimde uyar. Düğümler 0, 1, 2, … sırasıyla boyanır. Kısmi çözüm, ilk k düğümün renkleridir. Adaylar m renktir. Kısıt, bir düğümün renginin komşularından hiçbirinin rengiyle aynı olmamasıdır. 3-boyamanın karar sürümü NP-tamdır; $m = 2$ için ise problem polinom sürede çözülür, çünkü 2-boyanabilirlik iki parçalı olmaya denktir ve bir genişlik öncelikli aramayla sınanabilir.

9.6.2 Adım adım: altı düğümlü çizge

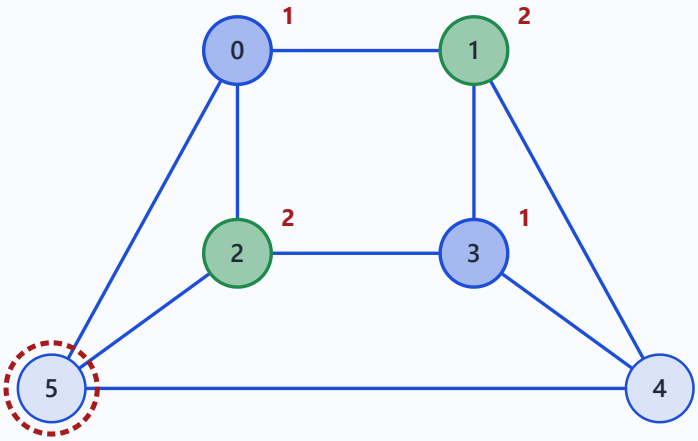
Çizgenin altı düğümü ve dokuz kenarı vardır: 0–1, 0–2, 0–5, 1–3, 1–4, 2–3, 2–5, 3–4 ve 4–5. Üç renk kullanılır: 1 mavi, 2 yeşil, 3 turuncu. Her düğümde renkler 1, 2, 3 sırasıyla denenir.



Şekil 9.28: Boyanacak altı düğümlü çizge, *m = 3*.

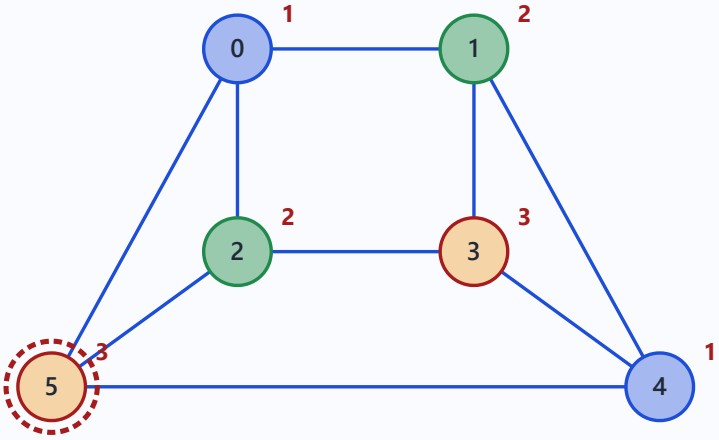
Adım	Düğüm	Komşu renkleri	Karar	`renk[0..5]`
1	0	—	renk 1	[1, 0, 0, 0, 0, 0]
2	1	0: 1	1 dolu; renk 2	[1, 2, 0, 0, 0, 0]
3	2	0: 1	1 dolu; renk 2	[1, 2, 2, 0, 0, 0]
4	3	1: 2, 2: 2	renk 1	[1, 2, 2, 1, 0, 0]
5	4	1: 2, 3: 1	1, 2 dolu; renk 3	[1, 2, 2, 1, 3, 0]
6	5	0: 1, 2: 2, 4: 3	çıkmaz; düğüm 4'ün rengi silinir	[1, 2, 2, 1, 0, 0]
7	4	—	denenecek renk kalmadı; düğüm 3'ün rengi silinir	[1, 2, 2, 0, 0, 0]
8	3	1: 2, 2: 2	2 dolu; renk 3	[1, 2, 2, 3, 0, 0]
9	4	1: 2, 3: 3	renk 1	[1, 2, 2, 3, 1, 0]
10	5	0: 1, 2: 2, 4: 1	1, 2 dolu; renk 3	[1, 2, 2, 3, 1, 3]

Tablo 9.9: Altı düğümlü çizgenin 3-boyaması. “Komşu renkleri” sütunu, o an boyalı komşuların renklerini verir.



Şekil 9.29: 6. adım öncesi: düğüm 5'in komşuları 0, 2 ve 4 üç rengin üçünü de kullanıyor. Hiçbir renk uygun değildir; çıkmaz.

Çıkmazın kaynağı 4. adımdaki karardır. Düğüm 3 mavi boyanınca düğüm 4 turuncu olmak zorunda kalır. Düğüm 5 ise mavi düğüm 0'a, yeşil düğüm 2'ye ve turuncu düğüm 4'e komşudur. Geri dönüş önce düğüm 4'e gider. Düğüm 4'te 3'ten büyük renk yoktur ve o da çıkmaza girer. Geri dönüş düğüm 3'e uzanır. Düğüm 3 turuncu olunca düğüm 4 mavi olabilir; düğüm 5'in komşuları mavi, yeşil ve mavi olur ve düğüm 5 turuncu boyanır. Geçerli 3-boyama [1, 2, 2, 3, 1, 3] bulunur.



Şekil 9.30: Altı düğüm de boyandı. Her düğümün yanındaki sayı rengidir: [1, 2, 2, 3, 1, 3].

9.6.3 Java kodu ve kod izleme

```
boolean boya(int v) {
  if (v == n) return true;          // bitti
  for (int c = 1; c <= m; c++) {
    if (uygunMu(v, c)) {
      renk[v] = c;                  // seç
      if (boya(v + 1)) return true; // keşfet
      renk[v] = 0;                  // geri al
    }
  }
  return false;                     // çıkmaz
}
// uygunMu: v'nin c renkli komşusu yoksa true
```

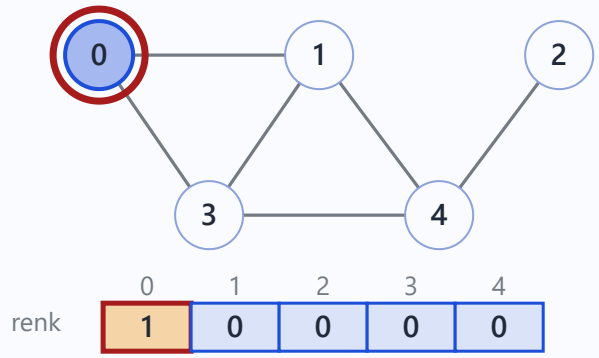
`boya(v)`, düğüm 0'dan *v* – *1*'e kadarının boyandığını varsayar ve *v*. düğümden itibaren boyamayı sürdürür. `v == n` ise bütün düğümler boyanmıştır. Döngü renkleri 1'den *m*'ye kadar dener. Uygun bir renk `renk[v]`'ye yazılır ve bir sonraki düğüme geçilir. Alt çağrı başarısız olursa renk 0 yapılır; 0, "boyanmamış" anlamına gelir. Bütün renkler başarısız olursa `false` döner.

`uygunMu`'nun gövdesi sunumda tek satırlık bir açıklamayla verilmiştir. Komşuluk matrisiyle yazılmış bir sürümü şöyledir:

```
boolean uygunMu(int v, int c) {
  for (int u = 0; u < n; u++)
    if (komsu[v][u] == 1 && renk[u] == c)
      return false;   // c renkli bir komşu var
  return true;
}
```

Metot *v*'nin bütün komşularına bakar. Komşulardan biri *c* rengindeyse `false` döner. Henüz boyanmamış komşuların rengi 0'dır ve hiçbir *c* ile eşleşmez. Bu yüzden yalnız önceden boyanmış komşular kısıtı etkiler. Komşuluk matrisiyle bu denetim *O(n)*, komşuluk listesiyle *O(derece)* sürer.

Kod izlemesi için daha küçük bir çizge kullanılsın: beş düğüm ve altı kenar, 0–1, 0–3, 1–3, 1–4, 2–4 ve 3–4. *m = 3*'tür.

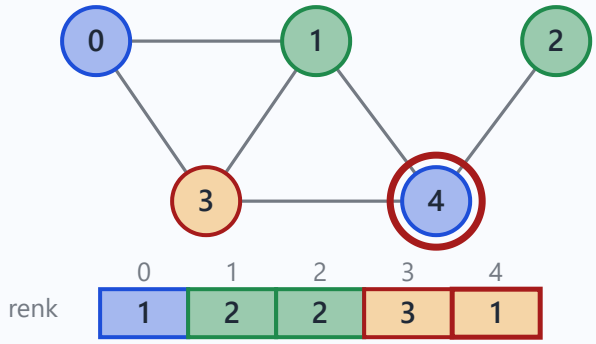


Şekil 9.31: `boya(0)` başlıyor. Düğüm 0'a renk 1 atandı; `renk` dizisinin öteki hücreleri 0'dır.

Çağrı	Denenen renkler	Sonuç	`renk`
`boya(0)`	1 uygun	`renk[0] = 1`	[1, 0, 0, 0, 0]
`boya(1)`	1 (0'da var), 2 uygun	`renk[1] = 2`	[1, 2, 0, 0, 0]
`boya(2)`	1 uygun	`renk[2] = 1`	[1, 2, 1, 0, 0]
`boya(3)`	1 (0'da), 2 (1'de), 3 uygun	`renk[3] = 3`	[1, 2, 1, 3, 0]
`boya(4)`	1 (2'de), 2 (1'de), 3 (3'te)	çıkmaz, `false`; `renk[3] = 0`	[1, 2, 1, 0, 0]
`boya(3)` sürer	renk kalmadı	`false`; `renk[2] = 0`	[1, 2, 0, 0, 0]
`boya(2)` sürer	2 uygun	`renk[2] = 2`	[1, 2, 2, 0, 0]
`boya(3)`	1, 2 dolu; 3 uygun	`renk[3] = 3`	[1, 2, 2, 3, 0]
`boya(4)`	1 uygun	`renk[4] = 1`	[1, 2, 2, 3, 1]
`boya(5)`	—	`v == n`, `true`	[1, 2, 2, 3, 1]

Tablo 9.10: `boya(0)` çağrısının izlemesi (beş düğüm, $m = 3$).

İzlemenin ilginç noktası düğüm 2'dir. Düğüm 2'nin tek komşusu düğüm 4'tür ve 2 boyanırken düğüm 4 henüz boyanmamıştır. Bu yüzden düğüm 2 ilk denemede renk 1'i alır. Bu seçim yerel olarak doğrudur, ama düğüm 4'ün üç komşusunu (1, 2 ve 3) üç farklı renge zorlar. Hata ancak düğüm 4'e gelince anlaşılır. Geri dönüş düğüm 3'ten geçerek düğüm 2'ye kadar uzanır. Düğüm 2 yeşil olunca düğüm 4 mavi boyanabilir. Sonuç [1, 2, 2, 3, 1]'dir.



Şekil 9.32: `v = 5 = n` : bütün düğümler boyandı; geçerli 3-boyama [1, 2, 2, 3, 1].

Maliyet açısından en kötü durumda her düğüm m rengin hepsini dener. Ağacın yaprak sayısı en çok m^n'dir. Her düğümde `uygunMu` $O(n)$ sürer. Toplam süre $O(n \cdot m^n)$'dir. Bellek `renk` dizisi ve çağrı yığını için $O(n)$'dir.

Renklerin simetrisi gereksiz işin bir kaynağıdır. Bir boyamada renklerin adları değiştirilirse yine geçerli bir boyama elde edilir: [1, 2, 2, 3, 1] ile [2, 1, 1, 3, 2] aynı yapıyı verir. Bu yüzden düğüm 0'ın rengini 1 olarak sabitlemek aramayı kabaca m kat küçültür. Bir çıkmaz bulunduğunda düğüm 0'ı başka renkle denemek anlamsızdır; renk adları değişse de aynı çıkmaza girilir.

9.6.4 Tek uzunluklu döngü ve en az renk sayısı

Örnek 9.7

Soru. Beş düğümlü döngü çizgesi C_5 (kenarlar 0–1, 1–2, 2–3, 3–4, 4–0) `boya(0)` ile $m = 2$ renkle boyanmaya çalışılıyor. Arama ne döndürür? En az kaç renk gerekir?

Çözüm. Düğüm 0 renk 1'i alır. Düğüm 1'in komşusu 0 renk 1'dedir; renk 2'yi alır. Düğüm 2 renk 1'i, düğüm 3 renk 2'yi alır. Düğüm 4'ün komşuları 0 (renk 1) ve 3'tür (renk 2). İki renk de doludur; çıkmaz. Geri dönüşler sonucu değiştirmez. İki renkle bir döngüde renkler zorunlu olarak dönüşümlü gider: 1, 2, 1, 2, … Düğüm 0 renk 2 ile denense bile dizi 2, 1, 2, 1 olur ve düğüm 4 yine iki renge de komşudur. Arama ağacın tamamını gezer ve `false` döner. Bu sonuç, çözüm olmadığının kanıtıdır; arama tamdır. $m = 3$ ile `boya(0)` ilk denemede [1, 2, 1, 2, 3] boyamasını bulur. C_5'in renk sayısı 3'tür.

Örnek genel bir kuralın özel hâlidir. Tek uzunluklu bir döngü iki renkle boyanamaz. Çift uzunluklu döngüler ise iki renkle boyanabilir: renkler dönüşümlü dağıtılır ve son düğüm ilk düğümden farklı renkte kalır. Daha genel olarak bir çizge ancak ve ancak tek uzunluklu döngü içermiyorsa, yani iki parçalıysa, 2-boyanabilir. Bir çizgenin renk sayısını bulmak için `boya` artan m değerleriyle çalıştırılabilir: 1, 2, 3, … İlk başarılı m renk sayısıdır. Başarısız her denemenin ağacın tamamını gezdiğine dikkat edilmelidir; en pahalı adım genellikle son başarısız denemedir.

9.7 Hamilton yolu

9.7.1 Problem ve adım adım çözüm

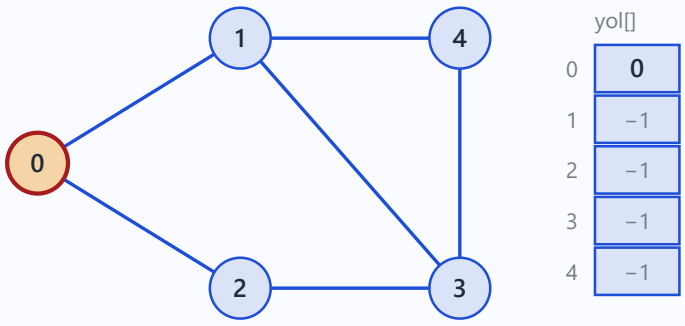
Tanım 9.9

Hamilton yolu (*Hamiltonian path*), bir çizgenin her düğümünü tam bir kez ziyaret eden yoldur. Yolun son düğümü ilk düğüme de komşuysa yol kapanır ve bir **Hamilton döngüsü** (*Hamiltonian cycle*) elde edilir.

Hamilton yolu, her *kenardan* tam bir kez geçen Euler yoluna benzer görünür, ama zorluk açısından çok farklıdır. Euler yolunun varlığı düğüm derecelerine bakılarak doğrusal sürede anlaşılır. Hamilton yolu problemi ise NP-tamdır; bilinen bütün kesin algoritmalar üstel süre harcar.

Geri izlemede kısmi çözüm bir yoldur: `yol[0..konum - 1]`. Aday, yola eklenecek bir sonraki düğümdür. Kısıt iki parçalıdır: yeni düğüm yolun son düğümüne komşu olmalı ve yolda daha önce bulunmamalıdır. Hedef, yolun *n* düğüme ulaşmasıdır.

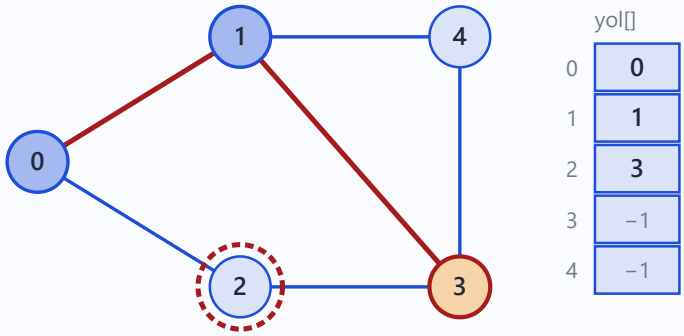
Örnek çizgenin beş düğümü ve altı kenarı vardır: 0–1, 0–2, 1–3, 1–4, 2–3 ve 3–4. Yol düğüm 0'dan başlar; `yol[0] = 0`. Her konumda düğümler 0, 1, 2, … sırasıyla denenir.



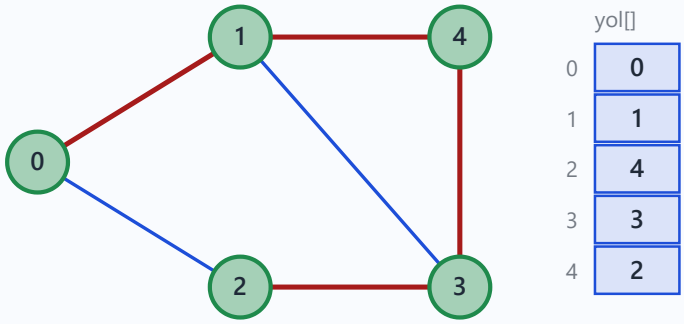
Şekil 9.33: Başlangıç: `yol[0] = 0`, öteki konumlar −1 (boş).

1. Konum 1: düğüm 1, 0'a komşudur; `yol[1] = 1`.
2. Konum 2: 1'in komşuları 0, 3 ve 4'tür. 0 yoldadır; ilk uygun düğüm 3'tür. `yol[2] = 3`.
3. Konum 3: 3'ün komşuları 1, 2 ve 4'tür. İlk uygun düğüm 2'dir. `yol[3] = 2`.
4. Konum 4: son düğüm 2'nin komşuları 0 ve 3'tür; ikisi de yoldadır. Çıkmaz. Düğüm 2 çıkarılır.
5. Konum 3: sıradaki uygun düğüm 4'tür. `yol[3] = 4`.
6. Konum 4: 4'ün komşuları 1 ve 3 yoldadır. Çıkmaz. Düğüm 4 çıkarılır.
7. Konum 3: 3'ün denenmemiş komşusu kalmadı. Düğüm 3 de çıkarılır.

8. Konum 2: 1'in sıradaki uygun komşusu 4'tür. `yol[2] = 4`.
9. Konum 3: 4'ün yolda olmayan komşusu 3'tür. `yol[3] = 3`.
10. Konum 4: 3'ün yolda olmayan komşusu 2'dir. `yol[4] = 2`. Yol beş düğüme ulaştı.



Şekil 9.34: 4. adım: yol 0 → 1 → 3 → 2. Son düğüm 2'nin bütün komşuları yolda; 2 çıkarılacak.



Şekil 9.35: Hamilton yolu bulundu: 0 → 1 → 4 → 3 → 2.

9.7.2 Java kodu

```
boolean bul(int[][] graf, int[] yol, int konum) {
  if (konum == graf.length) return true;
  for (int v = 0; v < graf.length; v++) {
    if (uygunMu(v, graf, yol, konum)) {
      yol[konum] = v;
      if (bul(graf, yol, konum + 1)) return true;
      yol[konum] = -1;     // geri al
    }
  }
  return false;
}

boolean uygunMu(int v, int[][] graf,
                int[] yol, int konum) {
  if (graf[yol[konum - 1]][v] == 0) return false;
  for (int i = 0; i < konum; i++)
    if (yol[i] == v) return false;
  return true;
}
```

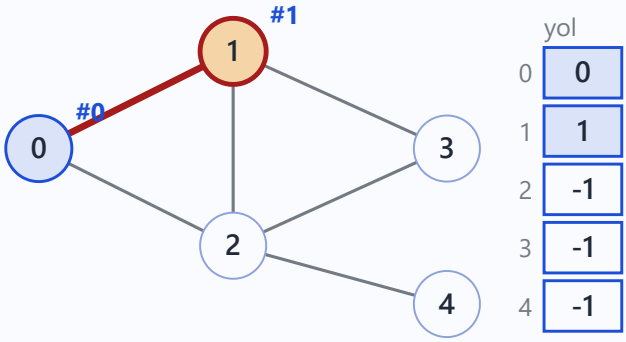
`graf` komşuluk matrisidir; `graf[u][v] == 1`, u ile v arasında kenar olduğunu gösterir. Çağıran kod `yol` dizisini −1 ile doldurur, `yol[0] = 0` yapar ve `bul(graf, yol, 1)`'i çağırır. `konum == graf.length` hedef testidir: yolda n düğüm vardır. Döngü her düğümü aday olarak dener. `uygunMu` iki kısıtı denetler. İlk satır, adayın son düğüme, yani `yol[konum - 1]`'e komşu olup olmadığına bakar. Döngü ise adayın yolda olup olmadığını tarar. Geri alma satırı `yol[konum] = -1`'dir.

Burada da permütasyon kodundaki gözlem geçerlidir: geri alma satırı mantık için şart değildir. `uygunMu` yalnız `yol[0..konum - 1]`'e bakar ve `yol[konum]`'daki eski değer bir sonraki denemede üzerine yazılır. Yine de −1 yazmak durumu temiz tutar ve hata ayıklamayı kolaylaştırır.

Maliyet hesabı permütasyonlara benzer. Yolun her konumu kalan düğümlerden biridir; en kötü durumda $(n - 1)!$ sıra denenir. Her düğümde n aday için `uygunMu` $O(n)$ sürer. Kaba sınır $O(n \cdot n!)$'dir. Yolda olup olmadığı bir `boolean` dizisiyle tutulursa `uygunMu` $O(1)$'e iner.

9.7.3 Kod izleme: ikinci çizge

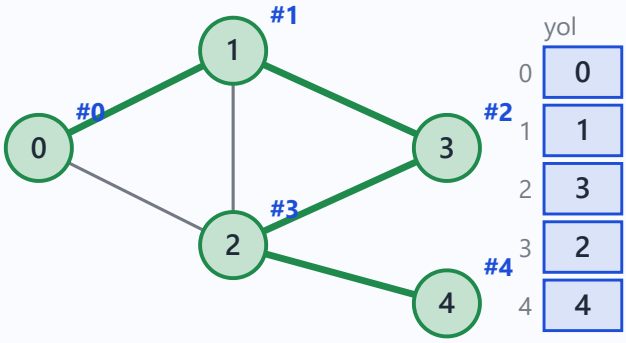
Kod ikinci bir çizgede izlensin. Beş düğüm ve altı kenar vardır: 0–1, 0–2, 1–2, 1–3, 2–3 ve 2–4. Düğüm 4'ün tek komşusu 2'dir. Bu yüzden düğüm 4 bir Hamilton yolunda ancak uçta bulunabilir.



Şekil 9.36: İkinci çizge. `yol[0] = 0` ile `bul(graf, yol, 1)` çağrıldı; konum 1'de 0'a komşu ilk düğüm 1 seçildi.

Konum	Son düğüm	Karar	`yol`
1	0	0 yolda; ilk uygun 1	[0, 1, −1, −1, −1]
2	1	ilk uygun 2	[0, 1, 2, −1, −1]
3	2	ilk uygun 3	[0, 1, 2, 3, −1]
4	3	3'ün komşuları 1, 2 yolda; çıkmaz, 3 çıkarılır	[0, 1, 2, −1, −1]
3	2	sıradaki uygun 4	[0, 1, 2, 4, −1]
4	4	4'ün tek komşusu 2 yolda; çıkmaz, 4 çıkarılır	[0, 1, 2, −1, −1]
3	2	aday kalmadı; çıkmaz, 2 çıkarılır	[0, 1, −1, −1, −1]
2	1	sıradaki uygun 3	[0, 1, 3, −1, −1]
3	3	ilk uygun 2	[0, 1, 3, 2, −1]
4	2	ilk uygun 4	[0, 1, 3, 2, 4]
5	—	`konum == n`, `true`	Hamilton yolu 0 → 1 → 3 → 2 → 4

Tablo 9.11: `bul(graf, yol, 1)` çağrısının izlemesi (ikinci çizge).



Şekil 9.37: Bulunan yol: 0 → 1 → 3 → 2 → 4. Düğümlerin yanındaki #0–#4 etiketleri yoldaki sırayı gösterir.

İzlemede algoritma, düğüm 4'ün yalnız uçta olabileceğini bilmez. Önce 2'den sonra 3'ü dener, sonra 4'ü. İki deneme de çıkmaza girer. Bir insan bu çizgeye bakınca "4 son düğüm olmalı, 2 de sondan bir önceki" der ve aramayı hemen daraltır. Bu tür akıl yürütmeler budama kurallarına çevrilebilir. Örneğin yolda olmayan bir düğümün yolda olmayan komşusu kalmamışsa ve bu düğüm son düğüme de komşu değilse dal hemen kesilebilir.

9.7.4 Hamilton döngüsü

Hamilton döngüsü aramak için koddaki tek değişiklik hedef testindedir. Yol *n* düğüme ulaştığında son düğümün başlangıç düğümüne komşu olup olmadığına da bakılır.

```
boolean dongu(int[][] graf, int[] yol, int konum) {
  if (konum == graf.length)
    return graf[yol[konum - 1]][yol[0]] == 1;   // son düğüm başlangıca komşu mu?
  for (int v = 0; v < graf.length; v++) {
    if (uygunMu(v, graf, yol, konum)) {
      yol[konum] = v;
      if (dongu(graf, yol, konum + 1)) return true;
      yol[konum] = -1;     // geri al
    }
  }
  return false;
}
```

Hedef testi `false` döndürürse arama sürer; bu yol bir döngüye kapanmamıştır ve başka yollar denenir. İlk örnek çizgede bulunan 0 → 1 → 4 → 3 → 2 yolunun son düğümü 2, başlangıç düğümü 0'a komşudur. Bu yüzden `dongu` aynı yolu bulur ve 2 → 0 kenarıyla kapanan bir Hamilton döngüsü verir. İkinci çizgede ise Hamilton döngüsü yoktur. Düğüm 4'ün tek komşusu vardır; bir döngüde her düğümün iki komşusu kullanılır. `dongu` bütün ağacı gezer ve `false` döndürür.

> **Sık yapılan hata**
>
> Hamilton döngüsü ararken yalnız `konum == n` testine bakmak. Bu test bir Hamilton *yolu* bulur. Son düğümün ilk düğüme komşu olduğu ayrıca denetlenmezse ikinci çizgede 0 → 1 → 3 → 2 → 4 yolu yanlışlıkla döngü sanılır.

9.8 Kelime arama

9.8.1 Problem

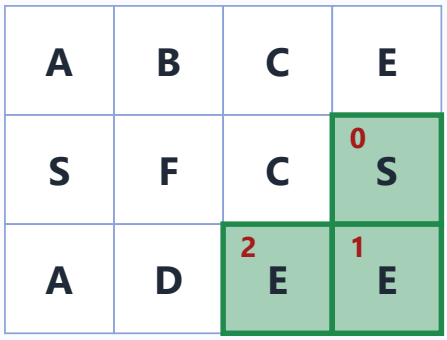
Bir harf ızgarasında bir kelime aranır. Kelime, komşu hücrelerden geçen bir yol boyunca okunmalıdır. Komşuluk dört yönlüdür: yukarı, aşağı, sol ve sağ. Bir hücre aynı yolda en çok bir kez kullanılabilir. Soru şudur: kelimeyi oluşturan böyle bir yol var mı?

Örnek ızgaranın üç satırı vardır: ABCE, SFCS ve ADEE. Aranan kelime “SEE” olsun. Her hücre sırayla başlangıç olarak denenir. Komşu yönler yukarı, aşağı, sol, sağ sırasıyla denenir.

A	B	C	E
S	F	C	S
A	D	E	E

Şekil 9.38: Harf ızgarası: ABCE / SFCS / ADEE.

1. (1, 0) = 'S' kelimenin ilk harfiyle eşleşir ve ziyaret edildi olarak işaretlenir. Komşuları A, A ve F'dir; hiçbiri 'E' değildir. Geri dönülür ve işaret kaldırılır.
2. (1, 3) = 'S' eşleşir ve işaretlenir. Yukarı komşusu (0, 3) = 'E' eşleşir ve işaretlenir.
3. (0, 3)'ün komşuları: yukarısı ızgara dışı, aşağısı (1, 3) ziyaretli, solu (0, 2) = 'C', sağı ızgara dışı. Üçüncü harf bulunamaz; işaret kaldırılır.
4. (1, 3)'ün aşağı komşusu (2, 3) = 'E' eşleşir ve işaretlenir.
5. (2, 3)'ün sol komşusu (2, 2) = 'E' eşleşir. Son harf de eşleşti: “SEE” bulundu.



Şekil 9.39: "SEE" bulundu. Küçük sayılar harfin kelimedeki indisidir: (1, 3) S, (2, 3) E, (2, 2) E.

9.8.2 Java kodu

```
boolean ara(char[][] iz, String kelime, int i,
            int s, int k, boolean[][] ziyaret) {
  if (i == kelime.length()) return true;
  if (s < 0 || s >= iz.length || k < 0
      || k >= iz[0].length || ziyaret[s][k]
      || iz[s][k] != kelime.charAt(i))
    return false;
  ziyaret[s][k] = true;
  for (int d = 0; d < 4; d++)
    if (ara(iz, kelime, i + 1, s + ds[d],
            k + dk[d], ziyaret))
      return true;
  ziyaret[s][k] = false;   // geri al
  return false;
}
```

`ara`, kelimenin *i*. harfinden sonrasının *(s, k)* hücresinden başlayarak bulunup bulunamayacağını söyler. `i == kelime.length()` hedef testidir: bütün harfler eşleşmiştir. İkinci `if` altı durumu birden eler: hücre ızgaranın dışındaysa, bu yolda zaten kullanılmışsa ya da harfi beklenen harf değilse `false` döner. Bu eleme, kısıt testinin özyinelemeli çağrının *başında* yapıldığı bir yazım biçimidir. n-vezir kodunda ise test çağrıdan önce, `guvenliMi` ile yapılıyordu. İki biçim de aynı budamayı yapar.

Testlerden geçen hücre işaretlenir (seç). Dört komşu için `ara` bir sonraki harfle çağrılır (keşfet). `ds` ve `dk` yön dizileridir. Komşulardan biri başarılı olursa `true` hemen yukarı iletilir. Hiçbiri başarılı olmazsa işaret kaldırılır (geri al) ve `false` döner. Dışarıdan bakan kod her hücreyi başlangıç olarak dener:

```
int[] ds = {-1, 1, 0, 0};   // yukarı, aşağı, sol, sağ
int[] dk = {0, 0, -1, 1};

boolean varMi(char[][] iz, String kelime) {
  for (int s = 0; s < iz.length; s++)
    for (int k = 0; k < iz[0].length; k++) {
      boolean[][] ziyaret = new boolean[iz.length][iz[0].length];
      if (ara(iz, kelime, 0, s, k, ziyaret))
        return true;
    }
  return false;
}
```

Hedef testinin sınır denetiminden önce yapılmasının bir nedeni vardır. Kelimenin son harfi ızgaranın kenarındaki bir hücrede eşleşirse dört komşu çağrısının bazıları ızgara dışına düşer. Ama bu çağrılarda *i* zaten kelimenin boyuna eşittir ve ilk satır `true` döndürür. Sıra ters olsaydı iki sorun çıkardı. Izgara dışına düşen çağrılar `false` döndürürdü. Izgara içindeki çağrılar ise `kelime.charAt(i)` ile kelimenin dışını okumaya çalışır ve istisna fırlatırdı. Örneğin tek hücreli bir ızgarada tek harfli bir kelime hiç bulunamazdı.

Kelime	Sonuç	Yol ya da neden
ABCCED	var	(0, 0) A → (0, 1) B → (0, 2) C → (1, 2) C → (2, 2) E → (2, 1) D
SEE	var	(1, 3) S → (2, 3) E → (2, 2) E
ABCB	yok	B iki kez kullanılamaz
SADE	var	(1, 0) S → (2, 0) A → (2, 1) D → (2, 2) E

Tablo 9.12: ABCE / SFCS / ADEE ızgarasında bazı kelimeler.

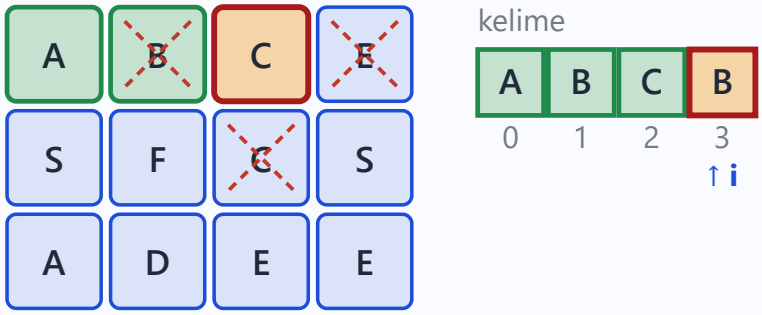
“SADE” araması da bir geri dönüş içerir. (1, 0)'daki S'den önce yukarıdaki (0, 0) = 'A' denenir. Bu A'nın komşuları B, S (ziyaretli) ve ızgara dışıdır; 'D' bulunamaz. İşaret kaldırılır ve aşağıdaki (2, 0) = 'A' denenir. Oradan (2, 1) = 'D' ve (2, 2) = 'E' ile kelime tamamlanır.

9.8.3 Kod izleme: “ABCB”

“ABCB” kelimesi ızgarada yoktur, ama neden olmadığını görmek için arama izlenmelidir. Arama (0, 0)'dan başlar.

1. (0, 0) = 'A' = kelime[0]. Hücre işaretlenir.
2. kelime[1] = 'B' aranır. Yukarı ızgara dışı, aşağı (1, 0) = 'S' ≠ 'B', sol ızgara dışı. Sağdaki (0, 1) = 'B' eşleşir ve işaretlenir.
3. kelime[2] = 'C' aranır. Yukarı ızgara dışı, aşağı (1, 1) = 'F' ≠ 'C', sol (0, 0) ziyaretli. Sağdaki (0, 2) = 'C' eşleşir ve işaretlenir.
4. kelime[3] = 'B' aranır. Yukarı ızgara dışı, aşağı (1, 2) = 'C' ≠ 'B', sol (0, 1) ziyaretli, sağ (0, 3) = 'E' ≠ 'B'. Dört yön de başarısızdır. Izgaradaki tek B (0, 1)'dedir ve bu yolda kullanılmıştır.

5. (0, 2) için geri alma: `ziyaret[0][2] = false`, `false` döner.
6. (0, 1)'in ve (0, 0)'ın kalan yönleri de başarısızdır. İkisinin de işareti kaldırılır. `ara(…, 0, 0, 0, …)` `false` döner.
7. Dış döngü öteki hücreleri dener. A ile başlayan tek öteki hücre (2, 0)'dır; komşuları S, D ve ızgara dışıdır, B yoktur. "ABCB" yoktur.



Şekil 9.40: 4. adım: (0, 2)'den 'B' aranıyor. Çarpı işaretli hücreler reddedilen komşulardır; B'nin bulunduğu (0, 1) zaten bu yolda kullanılmıştır.

Maliyet için kelimenin uzunluğu L, ızgaranın boyutları $R \times C$ olsun. Her hücre bir başlangıçtır. İlk harften sonra her adımda en çok 3 yeni yön vardır, çünkü gelinen hücre ziyaretlidir. Bu yüzden bir başlangıçtan gezilen yol sayısı $O(3^L)$ ile sınırlıdır ve toplam süre $O(R \cdot C \cdot 3^L)$'dir. Bellek `ziyaret` dizisi için $O(R \cdot C)$, yığın için $O(L)$'dir. Pratikte harf eşleşmesi aramayı çok erken keser; "ABCB" araması yalnız 4 hücreyi işaretler.

Sık yapılan hata

Geri dönüşte ziyaret işaretini kaldırmamak. İşaret kalırsa başarısız bir denemede geçilen hücreler sonraki denemelerde "kullanılmış" görünür. "SADE" aramasında (0, 0)'daki A denenip bırakıldıktan sonra işareti kalsa bile sonuç değişmezdi. Ama (1, 0)'daki S işaretli kalsaydı ve kelime başka bir yolda bu S'yi kullanacak olsaydı, var olan kelime bulunamazdı. Doğru sıra şudur: hücre önce işaretlenir, sonra komşular keşfedilir, en sonda işaret kaldırılır.

9.9 Diğer uygulamalar

9.9.1 Alt küme toplamı

Pozitif tam sayılardan oluşan bir küme ve bir hedef değer verilsin. Toplamı tam olarak hedefe eşit olan bir alt küme var mı? Bu soru **alt küme toplamı** (*subset sum*) problemidir ve karar sürümü NP-tamdır. Durum uzayı ağacı bu kez ikili bir ağaçtır. i. düzeyde i. öğe için iki seçenek vardır: öğe alınır ya da alınmaz. n öğe için ağacın 2^n yaprağı vardır; her yaprak bir alt kümedir.

```
int[] a = {3, 5, 6, 7};
int HEDEF = 15;
boolean[] secildi = new boolean[a.length];

boolean ara(int i, int toplam) {
  if (toplam == HEDEF) return true;
  if (i == a.length || toplam > HEDEF) return false;   // budama
  secildi[i] = true;                    // al
  if (ara(i + 1, toplam + a[i])) return true;
  secildi[i] = false;                   // alma
  return ara(i + 1, toplam);
}
```

`ara(i, toplam)`, ilk *i* öğe için karar verilmiş ve seçilenlerin toplamı `toplam` iken çağrılır. Hedefe ulaşılmışsa `true` döner. Öğe kalmamışsa ya da toplam hedefi aşmışsa `false` döner. İkinci koşul budamadır: öğeler pozitif olduğu için toplam bir kez hedefi aştıktan sonra bir daha azalamaz. Sonra iki dal sırayla denenir. Önce *i*. öğe alınır; bu dal başarısız olursa öğe bırakılır ve ikinci dal denenir. Bu kodda kısmi çözümün bir parçası olan `toplam` değerle geçirildiği için geri almaya gerek yoktur. Yalnız `secildi` dizisi geri alınır.

#	Çağrı	Anlamı	Sonuç
1	`ara(0, 0)`	kök	3'ü alma dalına iner
2	`ara(1, 3)`	3 alındı	5'i alma dalına iner
3	`ara(2, 8)`	5 alındı	6'yı alma dalına iner
4	`ara(3, 14)`	6 alındı	7'yi alma dalına iner
5	`ara(4, 21)`	7 alındı	*21 > 15*: budandı, `false`
6	`ara(4, 14)`	7 alınmadı	öğe kalmadı, 14 ≠ 15: `false`
7	`ara(3, 8)`	6 alınmadı	7'yi alma dalına iner
8	`ara(4, 15)`	7 alındı	`toplam == HEDEF`: `true`

Tablo 9.13: `a = {3, 5, 6, 7}`, `HEDEF = 15` için `ara(0, 0)` çağrılarının sırası.

Arama 8 çağrıda biter ve `secildi = [true, true, false, true]` olur: bulunan alt küme {3, 5, 7}'dir. Ağacın tamamında $2^5 - 1 = 31$ düğüm vardır. 5. çağrı budamanın işlediği yerdir: 3, 5, 6 ve 7'nin toplamı 21'dir, hedefi aşar.

Bu budama yalnız "fazla" toplamları keser. Ters yönde de bir budama yapılabilir: kalan öğelerin hepsi alınsa bile hedefe ulaşılamıyorsa dal kesilir. Bunun için `kalan[i]`, *i*. öğeden sonuna kadar olan öğelerin toplamı (sonek toplamı) olarak önceden hesaplanır ve şu satır eklenir:

```
if (toplam + kalan[i] < HEDEF) return false;
```

Örnek 9.8

Soru. Aynı kümede hedef 22 olsun. Kalan-toplam budaması olmadan ve olunca `ara(0, 0)` kaç çağrı yapar?

Çözüm. Öğelerin toplamı *3 + 5 + 6 + 7 = 21*'dir; 22'ye hiçbir alt küme ulaşamaz. Budamasız kodda hiçbir toplam 22'yi aşmaz, bu yüzden "fazla" budaması hiç çalışmaz. Arama ağacın 31 düğümünün hepsini gezer ve `false` döner. Kalan-toplam budamasıyla ilk çağrıda *0 + kalan[0] = 21 < 22*'dir. Arama tek çağrıda biter. Aynı kümede hedef 15 olduğunda ise arama iki sürümde de 8 çağrı yapar; budama bu örnekte işe yaramaz. Budamanın kazancı her zaman girdiye bağlıdır.

9.9.2 Geri izlemeyle çözülen öteki problemler

Şablon çok sayıda probleme uyarlanabilir. Her uyarlamada iki soru cevaplanır: her düzeyde hangi seçim yapılır ve hangi durumda dal kesilir?

Problem	Her düzeydeki seçim	Budama (kısıt)
Alt küme toplamı / sırt çantası	*i*. öğe alınsın mı? (2 dal)	toplam ya da ağırlık sınırı aşıldı
Gezgin satıcı (TSP)	sıradaki şehir	yol, bilinen en iyi turdan uzun
Kutu yerleştirme (*bin packing*)	eşya hangi kutuya?	kutu kapasitesi aşıldı
Maksimum kapsama	küme seçilsin mi?	kalan kümelerle iyileşme mümkün değil
Mantıksal doyurulabilirlik (SAT)	değişken doğru mu, yanlış mı?	bir tümce (*clause*) yanlış oldu
Mayın tarlası	hücre mayın mı?	açılmış sayılarla çelişki
Permütasyon / kombinasyon üretimi	sıradaki öğe	öğe zaten kullanıldı

Tablo 9.14: Geri izlemeyle çözülen bazı problemler.

Tablodaki problemlerin bir kısmı karar ya da sayma problemidir, bir kısmı ise en iyileme problemidir. Gezgin satıcı ve sırt çantasında amaç *bir* çözüm değil, *en iyi* çözümdür. Bu problemlerde kısıt budamasının yanına ikinci bir budama eklenir. Arama sırasında o ana kadar bulunan en iyi çözüm saklanır. Bir dalın varabileceği en iyi değer, bilinen en iyi çözümden kötüyse dal kesilir. Gezgin satıcıda yarım kalmış bir turun uzunluğu zaten bilinen en iyi turdan büyükse turu tamamlamaya gerek yoktur. Bu fikir **dal sınır** (*branch and bound*) yönteminin temelidir.

SAT satırı ayrıca önemlidir. Bugün kullanılan SAT çözücülerinin temeli de geri izlemedir: değişkenlere sırayla değer verilir, bir tümce yanlış olunca geri dönülür. Modern çözücüler bu iskelete birim yayılım, çelişkiden öğrenme ve akıllı değişken sıralaması ekler. Böylece milyonlarca değişkenli endüstriyel formülleri çözebilirler.

9.10 Analiz

9.10.1 Karmaşıklık

Bir geri izleme algoritmasının süresi iki çarpandan oluşur:

> *Zaman = durum uzayı ağacında gezilen düğüm sayısı × düğüm başına iş*

Düğüm başına iş genellikle polinomdur: bir kısıt testi, bir kopyalama, bir yazdırma. Düğüm sayısı ise ağacın şekline bağlıdır. Her düzeyde b dal ve d düzey varsa ağaçta en çok b^d yaprak olur. Seçenekler düzey ilerledikçe azalıyorsa (permütasyonlarda olduğu gibi) yaprak sayısı $n!$ gibi büyür.

Problem	En kötü zaman	Açıklama
n-vezir	$O(n!)$	satır başına bir vezir, sütunlar farklı; dizi taramalı test ile $O(n \cdot n!)$
Sudoku	$O(9^m)$	m: boş hücre sayısı; her hücrede 9 aday
Çizge m-boyama	$O(n \cdot m^n)$	n düğüm, her biri m renk; denetim $O(n)$ ya da $O(derece)$
Hamilton yolu	$O(n \cdot n!)$	yoldaki her konum için kalan düğümler
Kelime arama	$O(R \cdot C \cdot 3^L)$	ilk harften sonra en çok 3 yeni yön
Alt kümeler / permütasyonlar	$O(2^n)$ / $O(n \cdot n!)$	ağaçtaki yaprak sayısı

Tablo 9.15: Notta incelenen problemlerin en kötü durum süreleri.

Bu sınırlar budamayı hesaba katmaz. Budama düğüm sayısını çoğu zaman büyük ölçüde küçültür, ama en kötü durumu genellikle değiştirmez. Her zaman, budamanın hiç işlemediği kötü girdiler kurulabilir. Örneğin hiçbir alt kümenin hedefe ulaşamadığı ve toplamların hedefi hiç aşmadığı bir alt küme toplamı girdisi ağacın tamamını gezdirir.

Bellek açısından geri izleme çok tutumludur. Aynı anda yalnız tek bir kısmi çözüm ve çağrı yığınındaki çerçeveler tutulur. Yığının derinliği ağacın derinliğine eşittir. n-vezirde derinlik n, Sudoku'da m, kelime aramada L'dir. Bu yüzden bellek $O(derinlik)$ artı kısmi çözümün kendi boyudur. Ağacın genişliği ne kadar büyük olursa olsun bellek değişmez. Geniş arama yapan yöntemlerle karşılaştırıldığında bu önemli bir üstünlüktür.

Örnek 9.9

Soru. 8 vezirde kaba kuvvet 8^8 tam yerleşimi test etseydi ve her test 28 vezir çiftini karşılaştırsaydı kaç karşılaştırma yapılırdı? Geri izleme ilk çözüm için kaç `guvenliMi` çağrısı yapar?

Çözüm. *16.777.216 × 28 = 469.762.048* çift karşılaştırması, yani yaklaşık 470 milyon işlem gerekirdi. Geri izleme ilk çözüme kadar 876 kez `guvenliMi` çağırır ve her çağrı en çok $3 \cdot 8 = 24$ hücreye bakar. Üst sınır $876 \times 24 = 21.024$ hücre okumasıdır. Fark dört büyüklük basamağından fazladır.

9.10.2 Budamayı güçlendiren sezgiseller

Geri izlemenin hızı iki şeye bağlıdır: ağacın ne kadarının budandığına ve her düğümün ne kadar sürdüğüne. Notta görülen teknikler bu iki çarpanı küçültür.

- **Değişken sıralaması (MRV).** En az seçeneği kalan değişken önce doldurulur. Çelişkiler erken ortaya çıkar ve dallanma küçük kalır. Örnek Sudoku'da geri dönüş sayısı 4157'den 0'a iner.
- **Değer sıralaması.** Bir değişkenin değerleri arasında önce en az kısıtlayan değer denenir. Bu, en az çakışma fikrinin geri izlemedeki karşılığıdır. Değer sıralaması bütün çözümler arandığında işi azaltmaz, ama ilk çözüm daha erken bulunur. Labirentte yön sırasını değiştirmek çıkmaz hücre sayısını 8'den 0'a indiriyordu.
- **İleri denetim (*forward checking*).** Bir atamadan sonra komşu değişkenlerin alanlarından çelişen değerler silinir. Alanı boşalan bir değişken varsa hemen geri dönülür. Böylece bir çıkmaz, o değişkene sıra gelmeden fark edilir.
- **Hızlı kısıt testi.** n-vezirde sütun ve köşegen bayrakları, Sudoku'da satır, sütun ve kutu için bit maskeleri kullanılır. Test $O(n)$'den $O(1)$'e iner.
- **Simetri kırma.** Simetrik çözümlerden yalnız biri aranır. n-vezirde ilk vezir yalnız sol yarıda denenir ve bulunan çözümler yansıtılır: 8 vezirde yerleştirme sayısı 2056'dan 1028'e iner. Çizge boyamada ilk düğümün rengi sabitlenir.
- **Sınır (*bound*).** En iyilemede kısmi çözümün ulaşabileceği en iyi değer, bilinen en iyi çözümden kötüyse dal kesilir. Bu, dal sınır yöntemidir.

Bu tekniklerin ortak bir dersi vardır: bir aramanın maliyeti, aranan uzayın boyundan çok, aramanın nasıl düzenlendiğine bağlıdır. 9^{51} büyüklüğündeki bir uzay, doğru hücre sırasıyla 51 adımda çözülür.

9.10.3 Geri izlemeyi ayıran ne?

Geri izleme dersin öteki algoritma tasarım yöntemleriyle karşılaştırıldığında yeri daha iyi anlaşılır.

Yöntem	Fikir	Geri izlemeden farkı
Böl ve fethet	Bağımsız alt problemlere böl, birleştir	Seçim ve deneme yok; her alt problem tam bir kez çözülür
Açgözlü	Her adımda yerel en iyi; geri dönmez	Tek dal izlenir; yanlış seçim düzeltilemez
Dinamik programlama	Örtüşen alt problemleri bir kez çöz, sakla	Bütün seçenekler denenir ama sonuçlar tabloda; alt problem sayısı polinom olmalı
Geri izleme	Kısmi çözümü genişlet, geçersizse geri al	Seçimler denenir ve geri alınır; kısıt sağlama, bütün çözümler, karar problemleri

Tablo 9.16: Algoritma tasarım yöntemleri ve geri izlemeden farkları.

Açgözlü yöntemle karşılaştırma öğreticidir. Açgözlü bir algoritma durum uzayı ağacında kökten yaprağa tek bir yol izler. Her düğümde en iyi görünen çocuğa iner ve bir daha geri dönmez. Geri izleme aynı ağaçta derinlik öncelikli bir arama yapar. Açgözlü seçim yanlışsa geri izleme bunu fark eder ve

düzeltir. En az çakışmanın açgözlü başlangıcı bunu gösterir: başlangıç neredeyse doğrudur, ama tek bir çakışmayı gidermek için yine de onarım gerekir.

Dinamik programlamayla karşılaştırma da önemlidir. İki yöntem de bütün seçenekleri dener. Fark, alt problemlerin saklanıp saklanmamasındadır. Alt küme toplamında aynı (*i*, `toplam`) çifti farklı yollardan birden çok kez oluşabilir. Örneğin {1, 2, 3, 4} kümesinde ilk dört karardan sonra {1, 4} ve {2, 3} seçimleri aynı toplamı, 5'i verir. Geri izleme bu çifti her seferinde baştan çözer. Dinamik programlama ise sonucu tabloda saklar. Hedef küçükse dinamik programlama *O(n · HEDEF)* sürede çalışır. Ama n-vezir ya da Hamilton yolu gibi problemlerde alt problemi tanımlayan durum, kullanılmış sütunların ya da ziyaret edilmiş düğümlerin kümesidir. Bu kümelerin sayısı üsteldir ve tablo pratik olmaktan çıkar.

Özetle geri izleme, durum uzayı ağacında derinlik öncelikli aramadır. Ağaç önceden kurulmaz; gezilirken üretilir. Geçersiz dallar hiç üretilmez.

9.11 Sık yapılan hatalar

Geri izleme kodları kısadır, ama küçük hatalar çok farklı sonuçlar doğurur. Aşağıdaki hataların her biri notun bir yerinde ayrıntılı ele alınmıştır.

Sık yapılan hata

Geri almayı unutmak. Özyinelemeden dönünce `t[s][k] = 0` ya da `ziyaret[s][k] = false` yapılmazsa eski seçim sonraki dalları bozar. n-vezirde tahta önceki denemelerin vezirleriyle dolar ve geçerli sütunlar yanlışlıkla reddedilir.

Sık yapılan hata

Sonucu yaymamak. `if (coz(...)) return true;` yerine yalnız `coz(...)` yazılırsa çözüm bulunsa da arama sürer ve tahta geri alınır. Program çözüm bulduğu hâlde boş bir tahta gösterir.

Sık yapılan hata

Erken `return false`**.** İlk geçersiz adayda döngüden çıkmak. `if (!guvenliMi(...)) return false;` yazılırsa sıradaki sütunlar hiç denenmez. 4 vezirde satır 1'de sütun 0 reddedilince metot döner ve sütun 2 hiç denenmez.

Sık yapılan hata

Paylaşılan listeyi kaydetmek. Bütün çözümler toplanırken `sonuclar.add(yol)` yerine bir kopya, `new ArrayList<>(yol)`, eklenmelidir. Aksi hâlde liste aynı nesneye işaret eden referanslarla dolar.

Sık yapılan hata

Eksik kısıt denetimi. n-vezirde çaprazlardan yalnız birini, Sudoku'da 3×3'lük kutuyu denetlemeyi unutmak. Algoritma hızla "çözüm" bulur, ama bulduğu yerleşimler geçersizdir.

Sık yapılan hata

Yerel aramayı tam sanmak. En az çakışma çözüm olmadığını kanıtlayamaz. Adım sınırına ulaşmak "çözüm yok" demek değildir. Kesin bir "yok" cevabı için ağacın tamamını gezen bir geri izleme gerekir.

9.12 Özet

- **Zor problemler:** P polinom sürede çözülen, NP polinom sürede doğrulanan karar problemleridir. NP-tam problemler hem NP'de hem NP-zordur. $P \neq NP$ ise bu problemlerin polinom zamanlı kesin algoritması yoktur; kesin çözüm için budamalı üstel arama gerekir.
- **Fikir:** çözüm adım adım bir kısmi çözüm olarak kurulur. Bir aday kısıtı bozarsa o dal terk edilir; bir adımda aday kalmazsa son karar geri alınır ve sıradaki aday denenir. Geçersiz kısmi çözümlerin devamı hiç üretilmez.
- **Şablon:** hedef testi, ardından her aday için: geçerliyse seç, özyinele, geri al. Tek çözüm için `boolean` döndürülür ve `true` yukarı iletilir; bütün çözümler için hedefte sayılıp `false` döndürülür.
- **Durum uzayı ağacı:** kısmi çözümler düğüm, seçimler kenardır. Geri izleme bu ağaçta derinlik öncelikli aramadır; ağaç gezilirken üretilir ve budama geçersiz alt ağaçları atlar.
- **n-vezir:** vezirler satır satır konur; sütun ve iki çapraz denetlenir. 4 vezirde ilk çözüm [1, 3, 0, 2] (8 yerleştirme, 4 geri dönüş); 8 vezirde 92 çözüm, ilk çözüm [0, 4, 7, 5, 2, 6, 1, 3] (113 yerleştirme, 105 geri dönüş). Bayraklarla ($r + c$, $r - c + n - 1$) güvenlik testi $O(1)$ olur.
- **En az çakışma:** tam bir yerleşimden başlanır, çakışan bir vezir en az çakışmalı satıra taşınır. Hızlıdır ama tam değildir: platolarda dolaşabilir ve çözüm olmadığını kanıtlayamaz.
- **Sudoku:** boş hücreye 1–9 denenir; satır, sütun ve kutu denetlenir. Satır satır sırada örnek bulmaca 4208 yerleştirme ve 4157 geri alma ister; MRV ile 51 yerleştirme, 0 geri alma.

- **Öteki uygulamalar:** çizge boyama (renk kısıtı, $O(n \cdot m^n)$), Hamilton yolu ve döngüsü (komşuluk ve tekrar kısıtı, $O(n \cdot n!)$), kelime arama (ziyaret işaretinin geri alınması, $O(R \cdot C \cdot 3^L)$), alt küme toplamı (al/alma ikili ağacı, fazla ve eksik toplam budaması).
- **Maliyet:** zaman, gezilen düğüm sayısı ile düğüm başına işin çarpımıdır; en kötü durum üsteldir (2^n, $n!$, 9^m). Bellek yalnız *O(derinlik)*'tir. Değişken ve değer sıralaması, ileri denetim, hızlı kısıt testi ve simetri kırma gezilen ağacı küçültür.

9.13 Kavram listesi

geri izleme *backtracking* · kısmi çözüm *partial solution* · budama *pruning* · durum uzayı ağacı *state-space tree* · kaba kuvvet *brute force* · derinlik öncelikli arama *depth-first search (DFS)* · karar problemi *decision problem* · polinom zaman *polynomial time* · indirgeme *reduction* · NP-zor *NP-hard* · NP-tam *NP-complete* · n-vezir problemi *N-queens problem* · en az çakışma sezgiseli *min-conflicts heuristic* · yerel arama *local search* · sezgisel *heuristic* · kısıt sağlama problemi *constraint satisfaction problem* · MRV *minimum remaining values* · ileri denetim *forward checking* · çizge boyama *graph coloring* · Hamilton yolu *Hamiltonian path* · alt küme toplamı *subset sum* · sözlüksel sıra *lexicographic order* · dal sınır *branch and bound*

9.14 Kendini Dene

1. Permütasyon kodu `permutasyon(new int[4], new boolean[5], 0)` ile çağrılıyor. Yazdırılan 7. permütasyon hangisidir? Arama boyunca kaç kez “seç” yapılır?

   Cevap

   Permütasyonlar sözlüksel sırada çıkar: [1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2] ve 7. olarak [2, 1, 3, 4]. Seç sayısı ağacın kök dışı düğüm sayısıdır: $4 + 12 + 24 + 24 = 64$.

2. Dört öğeli bir kümenin alt kümelerini “al / alma” kararlarıyla üreten aramanın durum uzayı ağacında kök dahil kaç düğüm ve kaç yaprak vardır?

   Cevap

   Her düzeyde düğüm sayısı ikiye katlanır: $1 + 2 + 4 + 8 + 16 = 31$ düğüm. Yaprak sayısı $2^4 = 16$'dır; her yaprak bir alt kümedir.

3. *n = 4* için bayraklı sürümde vezirler (0, 1) ve (1, 3) karelerindedir. Hangi `sut`, `k1` ve `k2` hücreleri `true`'dur? (2, 0) ve (2, 2) kareleri güvenli midir?

   Cevap

   `sut[1]`, `sut[3]`; `k1[1]`, `k1[4]`; `k2[0 - 1 + 3] = k2[2]`, `k2[1 - 3 + 3] = k2[1]`. (2, 0) için `sut[0]`, `k1[2]` ve `k2[5]` boştur: güvenli. (2, 2) için `k1[4]` doludur: (1, 3) ile aynı ters köşegendedir, güvenli değildir.

4. *n = 6* için `coz(t, 0)` ilk çözümü bulana kadar 31 vezir yerleştirir. Kaç geri alma yapılır? Neden ilk çözüm sütun 0 ile başlamaz?

   Cevap

   Her yerleştirme ya geri alınır ya da çözümde kalır; çözümde 6 vezir vardır. *31 − 6 = 25* geri alma yapılır. 6 vezir probleminin dört çözümünün hiçbiri 0. satırda sütun 0'ı kullanmaz. Sütun 0 ile başlayan bütün dallar çıkmaza girer ve sözlüksel ilk çözüm [1, 3, 5, 0, 2, 4] olur.

5. 4 × 4 tahtada `yer = [1, 3, 0, 0]`'dır (`yer[k]`, *k*. sütundaki vezirin satırı). Onarım için sütun 2 seçiliyor. Bu vezirin satır 0–3 için çakışma sayıları nedir? En küçük değerli satıra taşınması sorunu çözer mi?

   Cevap

   Sayılar 1, 2, 1, 2'dir. Satır 0 vezirin şimdiki yeridir (sütun 3 ile aynı satır). Satır 2 de 1 çakışmalıdır: (3, 1) ile aynı çaprazdadır. Vezir satır 2'ye taşınırsa `yer = [1, 3, 2, 0]` olur ve bu kez sütun 1 ile sütun 2 çakışır; çakışan çift sayısı yine 1'dir. Bu bir platodur. Sütun 3 seçilseydi sayılar 1, 3, 0, 1 olurdu ve vezir satır 2'ye taşınarak çözüm [1, 3, 0, 2] bulunurdu.

6. Örnek Sudoku bulmacasında (4, 7) hücresinin kutusu hangi satır ve sütunları kapsar? Başlangıçta bu hücrenin adayları nelerdir?

   Cevap

   4 / 3 · 3 = 3, *7 / 3 · 3 = 6*: kutu 3–5. satırlar ve 6–8. sütunlardır. Satır 4'te 4, 8, 3, 1; sütun 7'de 6, 8, 7; kutuda 3, 1, 6 vardır. Kullanılmayan rakamlar 2, 5 ve 9'dur. Çözümde hücreye 9 yazılır.

7. Kenarları 0–1, 1–2, 2–0 ve 2–3 olan çizge `boya(0)` ile *m = 2* ve *m = 3* renkle boyanıyor. Sonuçlar nedir?

   Cevap

   0, 1, 2 düğümleri bir üçgendir; üçü de birbirine komşudur ve iki renk yetmez. *m = 2* için arama bütün ağacı gezer ve `false` döner. *m = 3* için ilk denemede [1, 2, 3, 1] bulunur: düğüm 3'ün tek komşusu 2'dir (renk 3), bu yüzden renk 1 uygundur.

8. Düğüm 0 merkezde, 1, 2 ve 3 yalnız merkeze bağlı olan yıldız çizgede `bul(graf, yol, 1)` ne döndürür? Başka bir başlangıç düğümü sonucu değiştirir mi?

   Cevap

   `false`. 0'dan 1'e gidilince 1'in yolda olmayan komşusu kalmaz; 2 ve 3 için de aynısı olur. Başlangıç değiştirmek de işe yaramaz: bir yolda en çok iki uç düğüm vardır, ama bu çizgede derecesi 1 olan üç düğüm vardır. Hamilton yolu yoktur.

9. ABCE / SFCS / ADEE ızgarasında "ASFB" ve "ECE" kelimeleri var mıdır?

   Cevap

   "ASFB" vardır: (0, 0) A → (1, 0) S → (1, 1) F → (0, 1) B. "ECE" yoktur. C harfi (0, 2) ve (1, 2)'dedir. (0, 2)'nin E komşusu yalnız (0, 3)'tür; o da yolun başında kullanılmıştır. (1, 2)'nin E komşusu yalnız (2, 2)'dir; yol oradan başlarsa ikinci E için başka komşu kalmaz. Her iki durumda da bir E hücresi iki kez kullanılmak zorunda kalır.

10. `a = {2, 4, 5, 9}` ve hedef 13 için alt küme toplamı araması (önce "al" dalı) hangi alt kümeyi bulur? Kalan-toplam budaması eklenirse çağrı sayısı nasıl değişir?

    Cevap

    2 ile başlayan bütün dallar başarısız olur: 2 + 4 + 5 = 11, 9 eklenince 20; 2 + 4 + 9 = 15; 2 + 5 + 9 = 16; 2 + 9 = 11. Ardından 2 bırakılır ve 4 alınır. 4 + 5 = 9'a 9 eklenince 18 olur ve dal budanır. 5 bırakılıp 9 alınınca 4 + 9 = 13 bulunur. Sonuç {4, 9}'dur. Budamasız kod 23, kalan-toplam budamalı kod 21 çağrı yapar; kazanç küçüktür, çünkü toplamların çoğu hedefi aşarak zaten budanır.

9.15 İleri okuma

- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3. baskı, 12. bölüm: geri izleme ile dal sınırı aynı çerçevede anlatır. n-vezir, Hamilton döngüsü ve alt küme toplamı örneklerini çözer.
- S. Russell ve P. Norvig, *Artificial Intelligence: A Modern Approach*, 4. baskı, 6. bölüm: kısıt sağlama problemlerini, MRV, ileri denetim ve en az çakışma sezgiselini ayrıntılı ele alır.
- D. E. Knuth, *The Art of Computer Programming*, Cilt 4B: geri izleme programlamasının tarihini, maliyet tahmini yöntemlerini ve "dancing links" tekniğini verir.
- S. W. Golomb ve L. D. Baumert, "Backtrack Programming" (1965): yöntemi genel bir programlama tekniği olarak tanımlayan ilk makalelerden biridir.

9.16 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3. baskı, Pearson, 2012 (12. bölüm: Backtracking).
3. S. Russell ve P. Norvig, *Artificial Intelligence: A Modern Approach*, 4. baskı, 2020 (6. bölüm: kısıt sağlama, en az çakışma).
4. S. Minton, M. D. Johnston, A. B. Philips ve P. Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems", *Artificial Intelligence* 58, 1992.
5. D. E. Knuth, *The Art of Computer Programming*, Cilt 4B, Addison-Wesley, 2022 (Backtrack programming, Dancing Links).
6. S. W. Golomb ve L. D. Baumert, "Backtrack Programming", *Journal of the ACM* 12(4), 516–524, 1965.
7. S. A. Cook, "The Complexity of Theorem-Proving Procedures", *Proceedings of the 3rd ACM Symposium on Theory of Computing (STOC)*, 151–158, 1971.

KISIM IV · BÖLÜM 10

Dal Sınır Algoritmaları

Birçok optimizasyon probleminde olası çözümlerin sayısı girdiyle birlikte üstel ya da faktöriyel hızla büyür. Bütün adayları tek tek denemek birkaç düzine öğede bile olanaksızlaşır. Dal sınır yöntemi bu aramayı akıllıca kısaltır. Çözümler bir ağaç üzerinde adım adım kurulur. Her dal için o dalın en fazla ne kadar iyi olabileceği önceden tahmin edilir. Tahmin, o ana kadar bulunan en iyi çözümü geçemiyorsa dala hiç girilmez. Bu not önce yöntemin kavramlarını, genel şemasını, sınırın nasıl seçildiğini ve üç arama stratejisini anlatır. Ardından yöntemi beş problemde adım adım uygular: 0/1 sırt çantası, gezgin satıcı, atama, tam sayılı doğrusal programlama ve maksimum klik. Her uygulamada sınırın neden geçerli olduğu gösterilir, kod satır satır açıklanır ve izleme tablosuyla çalıştırılır. Not, tasarım paradigmalarının karşılaştırmasıyla biter.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum10/src`

ÖĞRENME ÇIKTILARI

Paradigma

Dal sınır yönteminin durum uzayı ağacı, dallanma, sınırlama ve budama adımlarını açıklamak; yöntemin geri izlemeden farkını söylemek.

Sınır tasarımı

En büyüklemede üst, en küçüklemede alt sınır kurmak; sınırın geçerli, yani iyimser olmasının neden şart olduğunu gerekçelendirmek; gevşetme yoluyla sınır elde etmek.

Arama stratejileri

FIFO, LIFO ve en düşük maliyet (en iyi önce) stratejilerini aynı örnekte izlemek ve üretilen düğüm sayısı ile bellek açısından karşılaştırmak. Tasarım paradigmalarını birbirinden ayırt etmek.

Uygulamalar

0/1 sırt çantası, gezgin satıcı, atama, tam sayılı doğrusal programlama ve maksimum klik problemlerini dal sınır ile adım adım çözmek.

10.1 Dal sınır fikri

Karmaşık bir problemi çözmek bazen bir yol ayrımında doğru yönü bulmaya benzer. Her ayrımda birkaç seçenek vardır ve her seçenek yeni ayrımlara götürür. Hedef, sonunda en iyi noktaya varan yolu bulmaktır. Bütün yolları sonuna kadar yürümek kesin sonuç verir ama çok uzun sürer. Dal sınır yöntemi bir ara yol önerir: her ayrımda, o yönde en iyi durumda nereye varılabileceği tahmin edilir. Tahmin, elde bulunan en iyi sonuçtan kötüyse o yöne hiç gidilmez.

10.1.1 Kombinatoryal optimizasyon ve kaba kuvvetin sınırı

Dal sınır yönteminin hedefi **kombinatoryal optimizasyon** problemleridir. Bu problemlerde çözüm sonlu sayıda ayrık seçimden kurulur: hangi eşyaların alınacağı, şehirlerin hangi sırayla gezileceği, hangi kişiye hangi işin verileceği. Olası çözümlerin sayısı sonludur ama çok büyüktür. Aranan, bir amaç fonksiyonunu en büyükleyen ya da en küçükleyen çözümdür. Yöntem kesin, yani en iyi (*optimal*) çözümü bulmayı amaçlar; yaklaşık bir çözümle yetinmez.

Bu tür problemlerin en basit çözümü **kaba kuvvettir**: bütün adaylar üretilir, her biri değerlendirilir ve en iyisi seçilir. Sorun, aday sayısının büyüme hızıdır. n eşyalı bir sırt çantası probleminde her eşya ya alınır ya alınmaz; 2^n alt küme vardır. n şehirli gezgin satıcı probleminde başlangıç şehri sabitlenince kalan şehirlerin her sıralaması ayrı bir turdur; $(n-1)!$ tur vardır. Tablo 10.1 bu iki sayının nasıl büyüdüğünü gösterir.

n	2^n alt küme	$(n-1)!$ tur
5	32	24
10	1.024	362.880
20	1.048.576	$\approx 1{,}2 \cdot 10^{17}$
40	$\approx 1{,}1 \cdot 10^{12}$	$\approx 2{,}0 \cdot 10^{46}$

Tablo 10.1: Kaba kuvvetin denemesi gereken aday sayısı.

Saniyede bir milyar tur değerlendiren bir bilgisayar, 20 şehirli bütün turları yaklaşık $1{,}2 \cdot 10^{8}$ saniyede, yani üç yılı aşkın sürede bitirir. 40 şehir için gereken süre evrenin yaşının çok ötesindedir. Sırt çantasında büyüme daha yavaştır ama 40 eşyada da trilyon mertebesinde alt küme denenir. Bu problemlerin çoğu **NP-zor**dur (*NP-hard*): bilinen hiçbir algoritma onları en kötü durumda polinom sürede çözemez. Dal sınır da en kötü durumu değiştirmez. Değiştirdiği şey, pratikte incelenen aday sayısıdır.

Yöntemin temel fikri tek cümleyle şöyle söylenebilir: bir dalın en fazla ne kadar iyi olabileceği önceden tahmin edilir; tahmin yeterince iyi değilse o dala hiç girilmez. Bu tahmin doğru kurulursa en iyi çözüm hiçbir zaman kaybedilmez, ama adayların büyük bölümü hiç üretilmeden elenir.

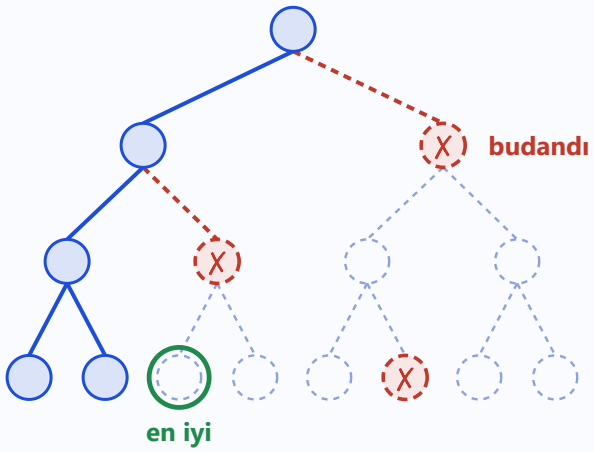
10.1.2 Dallanma, sınırlama, budama

Tanım 10.1

Dal sınır (*branch and bound*), bir optimizasyon probleminin çözüm uzayını ağaç biçiminde alt problemlere ayıran bir algoritma tasarım yöntemidir. Her alt problem için ulaşılabilecek en iyi değer tahmin edilir. Tahmini, o ana kadar bulunan en iyi çözümden daha iyi olmayan alt problemler incelenmeden elenir.

Yöntemin adı iki temel işleminden gelir. Dört adımda şöyle özetlenir:

1. **Dallanma** (*branching*): Problem daha küçük alt problemlere ayrılır. Sırt çantasında "eşya 1 alınsın mı?" sorusu iki alt problem doğurur: eşya 1'in alındığı ve alınmadığı durumlar.
2. **Sınırlama** (*bounding*): Her alt problem için ulaşılabilecek en iyi değer tahmin edilir. Bu tahmine **sınır** (*bound*) denir.
3. **Budama** (*pruning*): Sınırı, o ana kadar bulunan en iyi çözümden daha iyi olmayan alt problemler elenir. Elenen alt problemin altındaki hiçbir çözüm üretilmez.
4. **Tekrar:** İşlem kalan bütün alt problemler için sürdürülür. Gereksiz dallar budandıkça arama kısalır.



Şekil 10.1: Dallanıp budanan bir ağaç. Kırmızı çarpıyla işaretli düğümlerin sınırı yetersizdir; bu düğümler budanır. Budanan bir düğümün alt ağacı (soluk kesikli çizgiler) hiç üretilmez. Yeşil halka en iyi çözümü gösterir.

Şekil 10.1 yöntemi bir ağaç üzerinde gösterir. Kökten başlanır ve her düğümde bir karar verilerek aşağı inilir. Sağ üstteki düğüm budanmıştır. Altındaki yedi düğüm soluk çizilmiştir, çünkü algoritma onları hiç üretmez. Bu tasarruf, ağacın derinliği arttıkça katlanarak büyür. Kökün hemen altında budanan bir düğüm ağacın neredeyse yarısını tek hamlede eler.

Yöntem geniş bir alanda kullanılır. Gezgin satıcı, sırt çantası, çizelgeleme, kutu paketleme ve tam sayılı programlama bunların başında gelir. Endüstride üretim planlama, havayolu ekip ataması ve lojistik rotalama gibi problemleri çözen ticari en iyileme yazılımlarının çekirdeğinde de dal sınır vardır.

10.1.3 Temel kavramlar

Dal sınır algoritmaları bir ağaç üzerinde çalışır ve bu ağacın düğümleri çeşitli durumlarda bulunur. Kavramların adlarını baştan koymak, sonraki izlemeleri okumayı kolaylaştırır.

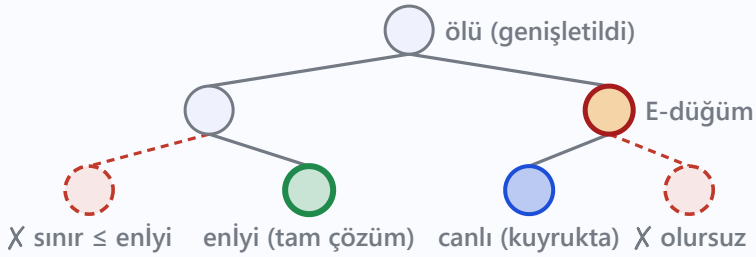
Tanım 10.2

Durum uzayı ağacı (*state-space tree*), bir problemin bütün kısmi ve tam çözümlerini gösteren ağaçtır. Kök hiçbir kararın verilmediği durumdur. Ağacın her düzeyi bir karara karşılık gelir; örneğin *i.* düzeyde "eşya *i* alınsın mı?" sorusu yanıtlanır. Yapraklar bütün kararların verildiği tam çözümlerdir.

Durum uzayı ağacı programda bütünüyle kurulmaz. Algoritma düğümleri gerektikçe üretir ve işi biten düğümleri unutur. Bu yüzden düğümlerin durumunu adlandırmak gerekir:

- **Dallanma ve E-düğüm.** Bir düğümün çocuklarını, yani alt problemlerini üretmeye dallanma denir. O anda çocukları üretilen düğüme **E-düğüm** (*expanding node*, genişletilen düğüm) denir.
- **Sınır.** Düğümün altındaki hiçbir çözümün geçemeyeceği iyimser değerdir. En büyüklemede bir üst sınır, en küçüklemede bir alt sınırdır.
- **Canlı düğüm** (*live node*). Üretilmiş ama henüz genişletilmemiş, hâlâ umut vaat eden düğümdür. Canlı düğümlerin hepsi bir kuyruk yapısında bekler. Bu yapı sıradan bir kuyruk, bir yığın ya da öncelikli kuyruk olabilir.
- **Ölü düğüm** (*dead node*). Genişletilmiş, olursuz bulunmuş ya da sınırı yetersiz olduğu için budanmış düğümdür. Ölü bir düğüm bir daha ele alınmaz.
- **enİyi** (*incumbent*). Şimdiye kadar bulunan en iyi tam çözümün değeridir. Budama kararları bu değere göre verilir. Arama ilerledikçe `enIyi` yalnız iyileşir: en büyüklemede artar, en küçüklemede azalır.

Bir çözüm problemin bütün kısıtlarını sağlıyorsa **olurlu** (*feasible*) çözümdür. Kısıtlardan birini ihlal eden düğüme olursuz düğüm denir. Örneğin ağırlığı kapasiteyi aşan bir sırt çantası olursuzdur.



Şekil 10.2: Durum uzayı ağacında düğüm türleri. Üstteki ölü düğüm genişletilmiştir. Turuncu düğüm o anda genişletilen E-düğümdür. Mavi düğüm kuyrukta bekleyen canlı düğümdür. Yeşil düğüm bir tam çözümdür ve `enIyi` 'yi belirler. Kırmızı kesikli düğümler budanmıştır: biri sınırı `enIyi` 'yi geçemediği, öbürü olursuz olduğu için.

Şekil 10.2 bu türleri tek bir ağaçta gösterir. Bir düğümün yaşam döngüsü şöyledir. Düğüm üretilir ve sınırı hesaplanır. Sınırı umut vaat ediyorsa canlı düğüm olarak kuyruğa girer, etmiyorsa hemen budanır. Sırası geldiğinde kuyruktan çıkar ve E-düğüm olur. Çocukları üretildikten sonra ölü düğüme dönüşür.

10.2 Paradigma

Dal sınır algoritmalarının hepsi aynı iskeleti izler. Problemden probleme değişen üç şey vardır: düğümün nasıl temsil edildiği, çocukların nasıl üretildiği ve sınırın nasıl hesaplandığı. Bu kesim önce iskeleti verir, sonra sınırın, budama kurallarının ve arama stratejisinin seçimini tartışır.

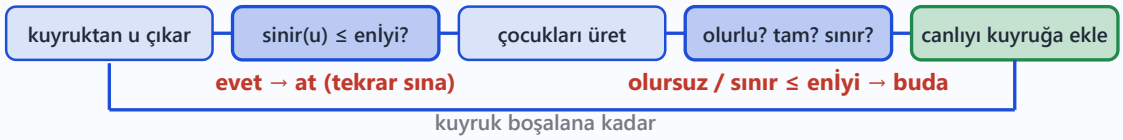
10.2.1 Genel şema

Aşağıdaki kod, bir en büyükleme problemi için dal sınır iskeletidir. Java'ya yakın bir sözde koddur: `kuyruk`, `sinir()`, `olurluMu()`, `tamCozumMu()` ve `deger()` her problemde başka bir anlam kazanır.

```
int enIyi = -∞;
kuyruk.ekle(kök);
while (!kuyruk.bosMu()) {
  Dugum u = kuyruk.cikar();   // strateji
  if (sinir(u) <= enIyi) continue;
  for (Dugum c : u.cocuklar()) {
    if (!olurluMu(c)) continue;
    if (tamCozumMu(c) && deger(c) > enIyi)
      enIyi = deger(c);       // yeni aday
    if (sinir(c) > enIyi)
      kuyruk.ekle(c);         // canlı düğüm
  }
}
return enIyi;
```

Kod bir döngüden ibarettir. Döngünün her turu beş adımdan oluşur:

1. **Seç:** Kuyruktan bir canlı düğüm çıkarılır. Hangi düğümün çıkacağını arama stratejisi belirler: ilk giren (FIFO), son giren (LIFO) ya da sınırı en iyi olan.
2. **Tekrar sına:** `if (sinir(u) <= enIyi) continue;` satırı düğümü bir kez daha sınar. Düğüm kuyruğa girerken umut vaat ediyordu. Ama kuyrukta beklerken `enIyi` büyümüş olabilir. Sınırı artık yetersizse düğüm hiç genişletilmeden atılır.
3. **Dallan:** `for` döngüsü düğümün çocuklarını üretir. Olursuz çocuklar `continue` ile hemen elenir.
4. **Güncelle:** Çocuk bir tam çözümse ve değeri `enIyi`'den büyükse `enIyi` yükseltilir. Bu çözüm yeni adaydır.
5. **Sınırla:** Yalnız sınırı `enIyi`'yi geçebilen çocuklar kuyruğa konur. Öbürleri budanır.



Şekil 10.3: Dal sınır döngüsü. Kuyruktan bir düğüm çıkarılır ve sınırı tekrar sınanır; yetersizse atılır. Çocuklar üretilir; olursuz ya da sınırı yetersiz olanlar budanır, kalanlar kuyruğa eklenir. Döngü kuyruk boşalana kadar sürer.

Döngü kuyruk boşalınca durur. O anda `enIyi` en iyi çözümün değeridir. Bu iddianın gerekçesi basittir. Her düğüm ya genişletilmiş ya da budanmıştır. Budanan düğümlerin altındaki çözümler `enIyi`'yi geçemez, çünkü sınır bunu garanti eder. Genişletilen düğümlerin altındaki çözümler ise tek tek incelenmiştir. Dolayısıyla `enIyi`'den iyi hiçbir çözüm atlanmamıştır.

Kod en büyükleme için yazılmıştır. **En küçüklemede** yönler ters çevrilir. Sınır bir alt sınırdır, `enIyi` başta $+\infty$'dur ve *alt(c) ≥ enIyi* olan düğüm budanır. Gezgin satıcı ve atama problemleri bu biçimde çözülür.

İskelette çözümün kendisi değil yalnız değeri tutulur. Hangi eşyaların alındığı ya da turun hangi sırayla gezildiği de isteniyorsa her düğüm kendi kısmi çözümünü taşır. Bunun için bir `boolean[]` dizisi ya da bit maskesi yeterlidir. `enIyi` güncellenirken bu bilgi de kopyalanır.

10.2.2 Sınır nasıl seçilir?

Dal sınırın bütün başarısı sınır fonksiyonuna bağlıdır. İyi bir sınırın dört özelliği vardır. Bunlardan biri zorunludur, öbür üçü verimlilikle ilgilidir.

Tanım 10.3

Bir en büyükleme probleminde *u* düğümünün sınırı, *u*'nun alt ağacındaki her olurlu çözümün değerinden büyük ya da ona eşitse sınır **geçerlidir**: *sinir(u) ≥ deger(x)*, her *x* için. En küçüklemede koşul tersine döner: *alt(u) ≤ deger(x)*. Geçerli bir sınıra **iyimser** sınır da denir, çünkü dalın gerçekte olabileceğinden daha iyi göründüğünü varsayar.

- **Geçerli (iyimser) olmalıdır.** Bu özellik zorunludur. Sınır gerçek değerin altında kalırsa (en büyüklemede) optimumu içeren dal "yeterince iyi değil" diye budanabilir. Algoritma hata vermeden yanlış bir sonuç döndürür.
- **Sıkı olmalıdır.** Sınır gerçek değere ne kadar yakınsa o kadar çok dal budanır. Sonsuz büyük bir "sınır" geçerlidir ama hiçbir şeyi budamaz; algoritma kaba kuvvete döner.
- **Ucuz hesaplanmalıdır.** Sınır her düğümde hesaplanır. Çok pahalı bir sınır, budadığı düğümlerden daha fazla zaman harcatabilir.
- **Gevşetme ile kurulur.** Sınır elde etmenin genel yolu **gevşetmedir** (*relaxation*). Problemin zor kısıtlarından biri kaldırılır ve geriye kalan kolay problem çözülür. Kısıt kaldırılınca olurlu çözümler kümesi büyür. Bu yüzden gevşetilmiş problemin en iyi değeri asıl problemin en iyi değerinden kötü olamaz; geçerli bir sınırdır.

Gevşetmenin üç örneği bu notta ayrıntılı işlenir. Sırt çantasında "eşya bölünemez" kısıtı kaldırılır ve kesirli sırt çantası çözülür. Tam sayılı programlamada "değişkenler tam sayı olmalı" kısıtı kaldırılır ve sıradan doğrusal program çözülür. Gezgin satıcıda "çıkılan kenarlar tek bir tur oluşturmalı" kısıtı kaldırılır ve her şehirden en ucuz kenarla çıkılır.

Sıkılık ile ucuzluk arasında bir denge vardır. En küçüklemede alt sınır, en iyi senaryoda elde edilebilecek en küçük değerdir. Zayıf bir alt sınır neredeyse hiç budamaz ve arama süresini uzatır. Çok güçlü bir alt sınır ise her düğümde pahalı bir hesap gerektirir. Seçim, budanan düğüm sayısı ile düğüm başına yapılan iş arasındaki dengeye göre yapılır.

Sık yapılan hata

Kötümser bir değeri sınır olarak kullanmak. Sırt çantasında yalnız tam sığan eşyaların kazancını toplamak akla yatkın görünür. Ama bu değer bir olurlu çözümün kazancıdır, yani bir alt sınırdır; üst sınır değildir. Eşyalar (20, 2), (30, 5), (35, 7), (12, 3) ve kapasite 9 olsun. Kökte tam sığanlar 20 + 30 = 50 verir, oysa optimum 55'tir (birinci ve üçüncü eşya). `enIyi` 50 olduğu anda bu "sınır" optimumun dalını keser ve algoritma 50'yi döndürür.

10.2.3 Bir düğüm ne zaman budanır?

Dal sınır algoritmalarında bir düğüm üç nedenle budanır. Kuralların üçü de aynı soruyu yanıtlar: bu düğümün altında `enIyi` 'den iyi bir çözüm olabilir mi?

1. **Olursuzluk.** Düğüm bir kısıtı ihlal ediyorsa altında hiçbir geçerli çözüm yoktur. Örnekler: ağırlık kapasiteyi aşmıştır, bir şehre ikinci kez gidilmiştir, doğrusal programın çözümü yoktur.
2. **Sınır.** En büyüklemede *sinir(u) ≤ enIyi*, en küçüklemede *alt(u) ≥ enIyi* ise altındaki çözümler en iyiyi geçemez. Düğüm olurlu olsa bile budanır.
3. **Tamlık.** Düğüm tam bir çözümse (yapraksa) ya da gevşetmenin çözümü zaten olurluysa dallanmaya gerek kalmaz. Örneğin tam sayılı programlamada doğrusal programın çözümü tam sayı çıkmışsa o düğümün altında daha iyi bir tam sayı çözüm yoktur. Değeri `enIyi` ile karşılaştırılır ve düğüm kapanır.

Eşitlik durumuna dikkat edilmelidir. *sinir(u) = enIyi* olan düğüm de budanır. Altında en fazla `enIyi` kadar iyi bir çözüm vardır ve o değerde bir çözüm zaten elde bulunur. Yalnız en iyi çözümün değeri değil bütün en iyi çözümler isteniyorsa eşitlikte budanmaz; bu, ayrı bir problemdir.

Geri izleme yalnız birinci kuralı ve tamlığı kullanır. Dal sınırı ondan ayıran şey ikinci kuraldır: düğüm olurlu olsa bile "yeterince iyi değilse" kesilir. Bu yüzden geri izlemeyle yazılmış bir aramaya tek bir sınır satırı eklemek onu dal sınıra dönüştürür.

10.2.4 Arama stratejileri

Genel şemada "kuyruktan bir canlı düğüm çıkarılır" denmişti. Hangi canlı düğümün çıkacağı, kuyruk yapısının türüne bağlıdır. Üç yaygın strateji vardır (Tablo 10.2).

Strateji	Kuyruk yapısı	Nasıl gezer?	Artı / eksi
FIFO dal sınır	kuyruk (`ArrayDeque`)	düzey düzey, genişlik öncelikli	sığ çözümleri erken bulur; canlı düğüm sayısı üstel büyüyebilir
LIFO dal sınır	yığın (ya da özyineleme)	derinlik öncelikli	bellek derinlikle orantılı; erken bir tam çözüm `enIyi`'yi hızla yükseltir
En düşük maliyet (LC, en iyi önce)	öncelikli kuyruk (`PriorityQueue`)	sınırı en umut verici düğüm önce	çoğunlukla en az düğümü genişletir; kuyruk bellek tüketir, her işlem *O(log k)*

Tablo 10.2: Canlı düğüm seçme stratejileri.

FIFO (*first in, first out*, ilk giren ilk çıkar) stratejisinde canlı düğümler sıradan bir kuyrukta bekler. Ağaç düzey düzey gezilir; bu, genişlik öncelikli aramadır. Kökün çocukları, sonra torunları, sonra onların çocukları genişletilir. Sığ düzeylerdeki çözümler erken bulunur. Ama derin düzeylere inmeden önce bütün sığ düğümler kuyrukta birikir. Dallanma derecesi 2 olan bir ağaçta *k.* düzey 2^k düğüm içerir; budama zayıfsa kuyruk üstel büyür.

LIFO (*last in, first out*, son giren ilk çıkar) stratejisinde canlı düğümler bir yığında bekler ya da arama doğrudan özyinelemeyle yazılır. Ağaç derinlik öncelikli gezilir. Yığında yalnız geçerli yoldaki düğümlerin henüz denenmemiş kardeşleri bulunur. Bu yüzden bellek, derinlik ile dallanma derecesinin çarpımıyla sınırlıdır. Bir başka yararı da kısa sürede bir yaprağa inmesidir. Erken bulunan bir tam çözüm `enIyi`'ye bir değer verir ve budama daha ilk adımlarda başlar.

En düşük maliyet (*least cost*, LC) stratejisinde canlı düğümler bir öncelikli kuyrukta bekler. Her adımda sınırı en umut verici olan düğüm, yani en büyüklemede sınırı en büyük, en küçüklemede sınırı en küçük düğüm genişletilir. Bu stratejiye **en iyi önce arama** (*best-first search*) da denir. En umut verici dala önce bakıldığı için iyi bir çözüm erken bulunur ve kalan düğümlerin çoğu budanır. Bedeli öncelikli kuyruğun bellek tüketimi ve her ekleme ile çıkarmanın $O(\log k)$ sürmesidir; k, kuyruktaki düğüm sayısıdır.

Üç strateji de sınır geçerli olduğu sürece aynı en iyi değeri bulur. Strateji yalnız iş miktarını, yani üretilen ve genişletilen düğüm sayısını, ve belleği değiştirir. Bu farklar sırt çantası örneğinde sayılarla görülecektir.

10.2.5 En düşük maliyetli arama ve maliyet tahmini

LC aramanın genel biçiminde her düğüme bir tahmini maliyet atanır:

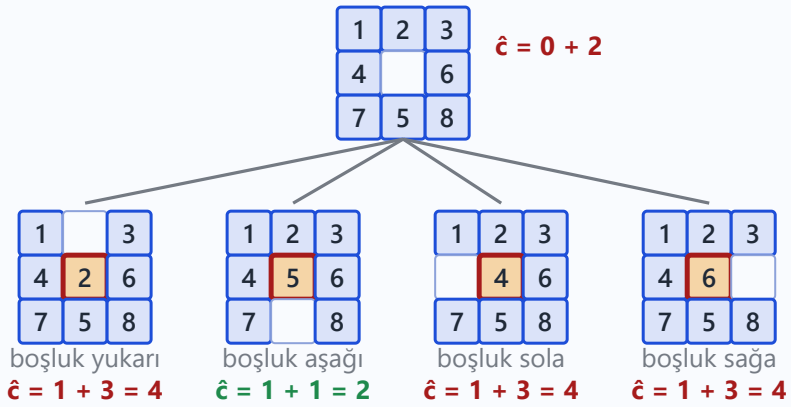
> $$\hat{c}(x) = f(x) + \hat{g}(x)$$

Burada $f(x)$, kökten x'e kadar ödenmiş maliyettir; örneğin x'in derinliği ya da o ana kadar gidilen yolun uzunluğu. $\hat{g}(x)$ ise x'ten bir çözüme kalan maliyetin tahminidir. Canlı düğümlerden $\hat{c}$'si en küçük olan genişletilir.

İki uç seçim, öbür stratejileri verir. $\hat{g} = 0$ alınır ve f düğümün derinliği olursa hep en sığ düğüm seçilir; arama FIFO'ya yaklaşır. $f = 0$ alınırsa yalnız kalan maliyet tahminine bakılır; arama, hedefe en yakın görünen dala dalan açgözlü bir derinlik aramasına yaklaşır. İkisinin toplamı, ödenmiş maliyeti ve kalan tahmini birlikte tartar.

Tahminin iyi bir özelliği, gerçek kalan maliyetten hiçbir zaman büyük olmamasıdır. Böyle bir tahmine **kabul edilebilir sezgisel** (*admissible heuristic*) denir. Bu özellik, dal sınırdaki geçerli sınırın karşılığıdır: kalan maliyet olduğundan az gösterilir, yani tahmin iyimserdir.

Klasik örnek **8 bulmacasıdır**. 3 × 3'lük bir çerçevede 1'den 8'e kadar numaralı taşlar ve bir boşluk vardır. Bir hamlede boşluğa komşu bir taş boşluğa kaydırılır. Hedef, taşları 1 2 3 / 4 5 6 / 7 8 düzenine getirmektir; boşluk sağ alt köşededir. Bu problemde $f(x)$ o ana kadar yapılan hamle sayısı, $\hat{g}(x)$ ise yerinde olmayan taş sayısıdır. Her yanlış yerdeki taş en az bir hamle ister. Bu yüzden $\hat{g}$ kalan hamle sayısını hiçbir zaman fazla tahmin etmez.



Şekil 10.4: 8 bulmacasında LC arama. Başlangıç durumunda 5 ve 8 yanlış yerdedir: $\hat{c} = 0 + 2$. Dört çocuktan "boşluk aşağı" $\hat{c} = 1 + 1 = 2$ ile en küçüktür; öbür üçünün $\hat{c}$ değeri 4'tür. Turuncu: hareket eden taş.

Şekil 10.4 aramanın ilk adımını gösterir. Başlangıç durumu 1 2 3 / 4 □ 6 / 7 5 8'dir (□ boşluk). Yalnız 5 ve 8 yanlış yerdedir; kökün $\hat{c}$ değeri $0 + 2 = 2$'dir. Kök genişletilince dört çocuk oluşur:

- **Boşluk yukarı:** 2 aşağı iner; 1 □ 3 / 4 2 6 / 7 5 8. Yanlış taşlar 2, 5, 8: $\hat{c} = 1 + 3 = 4$.
- **Boşluk aşağı:** 5 yukarı çıkar; 1 2 3 / 4 5 6 / 7 □ 8. Yalnız 8 yanlış: $\hat{c} = 1 + 1 = 2$.
- **Boşluk sola:** 4 sağa kayar; 1 2 3 / □ 4 6 / 7 5 8. Yanlış taşlar 4, 5, 8: $\hat{c} = 4$.
- **Boşluk sağa:** 6 sola kayar; 1 2 3 / 4 6 □ / 7 5 8. Yanlış taşlar 6, 5, 8: $\hat{c} = 4$.

En küçük $\hat{c}$'li çocuk "boşluk aşağı"dır ve o genişletilir. Onun çocuklarından "boşluk sağa" 8'i sola kaydırır ve hedefe ulaşır: $\hat{c} = 2 + 0 = 2$. Çözüm 2 hamledir ve yalnız 2 genişletme yapılmıştır. FIFO ise $\hat{c}$ değerlerine bakmaz. Birinci düzeydeki düğümleri üretim sırasıyla genişletir ve "boşluk yukarı" gibi umutsuz bir dala da zaman harcar. Yol bulmadaki **A* algoritması** bu fikrin ağırlıklı çizgelerdeki karşılığıdır; ilişki notun sonunda ayrıca tartışılır.

10.2.6 Geri izleme ile karşılaştırma

Dal sınır ile **geri izleme** (*backtracking*) aynı ağacı gezer ve kodları birbirine çok benzer. İkisi de kısmi çözümleri genişletir ve umutsuz dalları keser. Farkları neyi aradıklarında ve neye göre kestiklerindedir (Tablo 10.3).

Özellik	Geri izleme	Dal sınır
Problem türü	karar ve arama: bir ya da tüm geçerli çözümler (n-vezir, Sudoku)	optimizasyon: en iyi çözüm (en büyük kazanç, en kısa tur)
Budama ölçütü	olurluluk: kısıt ihlal ediliyor mu?	olurluluk + sınır: `enIyi` 'yi geçebilir mi?
Gezinme	her zaman derinlik öncelikli (özyineleme)	derinlik öncelikli, genişlik öncelikli ya da en iyi önce
Ek bilgi	yalnız kısmi çözüm	kısmi çözüm + sınır fonksiyonu + `enIyi`
Bellek	*O(derinlik)*	derinlik öncelikli sürümde *O(derinlik)*; FIFO ve LC'de kuyruk boyu kadar

Özellik	Geri izleme	Dal sınır
En kötü durum	üstel	yine üstel; pratikte çok daha az düğüm

Tablo 10.3: Geri izleme ve dal sınır.

Geri izleme n-vezir problemini çözerken bir vezirin başka bir veziri tehdit edip etmediğine bakar. Tehdit varsa dalı keser. Bütün çözümler ya da bir çözüm aranır; çözümler arasında "daha iyi" diye bir sıralama yoktur. Dal sınır ise çözümler arasında bir sıralama olduğunda anlamlıdır. Bir dal hiçbir kısıtı ihlal etmese bile, içinden çıkabilecek en iyi çözüm elde bulunandan kötüyse kesilir.

Bu farkın kod karşılığı tek bir satırdır. Gezgin satıcı ya da maksimum klik için yazılmış ve sınır testi içermeyen bir arama aslında geri izlemedir; bütün ağacı gezer. `if (altSinir(...) < enIyi)` gibi tek bir sınır satırı eklenince arama dal sınıra dönüşür.

10.2.7 Avantajlar ve dezavantajlar

Yöntemin güçlü yanları şunlardır:

- Karmaşık optimizasyon problemlerinde kesin en iyi çözümü bulur. Yaklaşık yöntemlerin aksine sonucun en iyi olduğu kanıtlıdır.
- Arama alanını daraltır ve gereksiz olasılıkları eler. İyi bir sınırla ağacın çok küçük bir kesri gezilir.
- En iyi rotayı ya da seçimi bulmayı gerektiren problemler için doğal bir çerçevedir.
- Çözümleri ağaç olarak temsil edilebilen çok çeşitli problemlere uygulanır.
- Erken durdurulabilir. Arama yarıda kesilirse elde bir çözüm (`enIyi`) ve kuyruktaki en iyi sınır bulunur. İkisinin farkı, eldeki çözümün en iyiden en fazla ne kadar uzak olduğunu söyler. Bu bilgi bir kalite garantisidir.

Zayıf yanları da vardır:

- En kötü durumda süre yine üsteldir. Çok fazla olasılık içeren ve sınırın iyi budamadığı durumlarda süre kabul edilemez ölçüde uzar.
- FIFO ve en iyi önce stratejilerinde bellek de üstel büyüyebilir.
- Başarı iyi bir sınır tahminine bağlıdır; zayıf bir sınır neredeyse hiç budamaz.
- Her probleme özgü bir sınır ve dallanma kuralı tasarlamak gerekir. Uygulaması geri izlemeden karmaşıktır.

10.3 0/1 sırt çantası

Yöntemin bütün parçalarını görmek için ilk uygulama olarak 0/1 sırt çantası problemi seçilir. Problem tanıdıktır, durum uzayı ağacı ikilidir ve sınırı kolay hesaplanır. Aynı örnek üç farklı stratejiyle çalıştırılacak ve sonuçlar karşılaştırılacaktır.

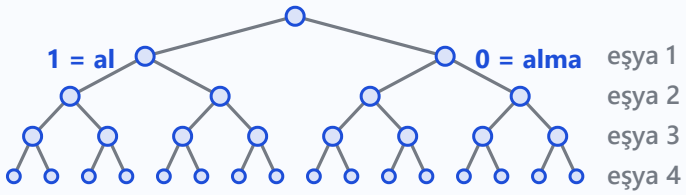
10.3.1 Problem ve durum uzayı ağacı

Problemi kısaca hatırlatmak yeterlidir. n eşya vardır; i. eşyanın kazancı p_i, ağırlığı w_i'dir. Çantanın kapasitesi W'dir. Her eşya için bir karar verilir: $x_i \in \{0, 1\}$. Eşya ya bütünüyle alınır (1) ya hiç alınmaz (0); bölünemez. Amaç

> $\Sigma\, w_i\, x_i \le W$ koşuluyla $\Sigma\, p_i\, x_i$ en büyük

olacak biçimde kararları vermektir. Bu notta sırt çantasının dinamik programlama çözümü kullanılmaz. Dinamik programlama kapasite W'ye bağlı bir tablo kurar ve $O(nW)$ sürer. W çok büyükse ya da ağırlıklar tam sayı değilse bu tablo kurulamaz; dal sınır ise ağırlıkların büyüklüğünden etkilenmez.

Dal sınır için eşyalar önce **kazanç/ağırlık oranına** göre azalan sırada dizilir. Bu sıra, sınır hesabının geçerli olması için gereklidir. Durum uzayı ağacı ikilidir. Kök hiçbir kararın verilmediği durumdur. i. düzeyde i. eşyanın kararı verilir: sol dal "eşya i alındı" (1), sağ dal "alınmadı" (0). Dört eşyalı bir problemde ağacın beş düzeyi vardır ve tam ağaç $2^5 - 1 = 31$ düğüm içerir. Bunların 16'sı yapraktır; her yaprak bir alt kümeye karşılık gelir.



Şekil 10.5: Dört eşyalı sırt çantasının durum uzayı ağacı. Her düzeyde bir eşyanın kararı verilir; sol dal 1 = al, sağ dal 0 = alma. Tam ağaçta 31 düğüm ve 16 yaprak vardır.

Bu notta kullanılan örnek Tablo 10.4'te verilmiştir. Kapasite $W = 16$'dır. Eşyalar oranlarına göre zaten azalan sıradadır: 20, 6, 5, 2.

Eşya	p_i	w_i	p_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

Tablo 10.4: Sırt çantası örneği, $W = 16$.

Optimum, eşya 1 ile eşya 3'ün alınmasıdır: kazanç $40 + 50 = 90$, ağırlık $2 + 10 = 12$. Kaba kuvvet bu sonucu 16 alt kümenin hepsini deneyerek bulur. Dal sınır ise 31 düğümün tamamını üretmeden aynı sonuca ulaşır. Ne kadar az düğüm üretildiği, kullanılan stratejiye bağlıdır.

10.3.2 Üst sınır: kesirli sırt çantası gevşetmesi

Sırt çantası bir en büyükleme problemidir; her düğüm için bir üst sınır gerekir. Bu sınır gevşetmeyle kurulur. “Eşya bölünemez” kısıtı kaldırılınca problem **kesirli sırt çantasına** (*fractional knapsack*) dönüşür. Kesirli sırt çantasında bir eşyanın istenen bir kesri alınabilir ve kazancın da aynı kesri elde edilir. Bu problemi açgözlü yöntem en iyi biçimde çözer: eşyalar oran sırasıyla, sığdığı kadar tam alınır; ilk sığmayan eşyanın sığan kesri alınır ve çanta dolar.

Bir düğümün sınırı aynı fikirle hesaplanır. u düğümünde verilmiş kararlar sabittir: u'nun kazancı k, ağırlığı a, düzeyi i olsun. Henüz karar verilmemiş eşyalar $i+1, i+2, \ldots$ oran sırasıyla, sığdığı kadar tam eklenir. İlk sığmayan eşya m olsun; onun sığan kesri eklenir:

$$sinir(u) = k + \Sigma_{j=i+1}^{m-1} p_j + (W - a - \Sigma_{j=i+1}^{m-1} w_j) \cdot p_m / w_m$$

Bütün kalan eşyalar sığıyorsa m yoktur ve son terim eklenmez. Düğümün kendisi olursuzsa, yani $a > W$ ise sınır 0 alınır. Bu, düğümün hiçbir zaman `enIyi`'yi geçemeyeceğini ve kuyruğa girmeyeceğini garanti eder.

Teorem 10.1 (Kesirli sınır geçerlidir)

Eşyalar p_i/w_i'ye göre azalan sıradaysa, her u düğümü için $sinir(u)$, u'nun alt ağacındaki her olurlu 0/1 çözümün kazancından büyük ya da ona eşittir.

İspat. u'nun alt ağacındaki herhangi bir olurlu 0/1 çözüm, u'da verilmiş kararları korur ve kalan eşyalardan bir alt küme seçer. Bu seçim aynı zamanda kesirli problemin bir çözümüdür: her eşyanın kesri 0 ya da 1'dir. Dolayısıyla 0/1 çözümün kazancı, kalan eşyalar ve kalan kapasite $W - a$ için kurulan kesirli problemin en iyi değerini geçemez. Açgözlü yöntem, eşyalar oran sırasında olduğu için kesirli problemin en iyi değerini verir. Kalan kapasitenin her birimi, kalan eşyalar arasında birim başına en yüksek kazancı veren eşyaya harcanır. Bu değere k eklenince $sinir(u)$ elde edilir. ■

İspatın iki dayanağı vardır: her 0/1 çözümün bir kesirli çözüm olması ve açgözlü yöntemin kesirli problemde en iyi olması. İkinci dayanak, oran sırasına bağlıdır. Eşyalar oran sırasında değilse açgözlü yöntem kesirli problemde de en iyiyi bulamaz. Bu durumda hesaplanan değer gerçek optimumun altında kalabilir ve sınır geçerliliğini yitirir.

Kökün sınırı örnekte şöyle hesaplanır. Eşya 1 ve 2 tam sığar: $40 + 30 = 70$, ağırlık 7. Eşya 3'ün ağırlığı 10'dur ve kalan 9 birime sığmaz. Eşya 3'ün $9/10$'u alınır: $9 \cdot 50/10 = 45$. Böylece $sinir(kök) = 70 + 45 = 115$ olur. Bu sayının anlamı şudur: dört eşyadan hangi alt küme seçilirse seçilsin kazanç 115'i geçemez. Gerçek optimum 90'dır; sınır iyimserdir ama geçerlidir.

Sık yapılan hata

Eşyaları oran sırasına dizmeyi unutmak. Eşyalar (10, 5), (40, 2), (30, 5), (50, 10) sırasıyla verilmiş ve sınır bu sırayla hesaplanmış olsun. Kökte ilk üç eşya tam sığar (ağırlık 12, kazanç 80); kalan 4 birim için eşya 4'ün *4/10*'u alınır: *80 + 20 = 100*. Bu sayı 115'ten küçüktür ve gerçek kesirli optimumu yansıtmaz. Bu örnekte 100 yine 90'dan büyüktür, ama başka bir örnekte sırasız hesap optimumun altına düşebilir ve doğru dal budanır.

10.3.3 Kod izleme: sinir(kök)

Sınır fonksiyonunun Java karşılığı aşağıdadır. Diziler sıfırdan başlar: `p[0]` ve `w[0]` eşya 1'e aittir. `n` eşya sayısı, `W` kapasitedir. `Dugum` sınıfı `seviye`, `kazanc`, `agirlik` ve `sinir` alanlarını taşır. Kökün düzeyi −1'dir; henüz hiçbir eşyanın kararı verilmemiştir.

```
double sinir(Dugum u) {
  if (u.agirlik > W) return 0;   // olursuz
  double sonuc = u.kazanc;
  int toplamAg = u.agirlik;
  int j = u.seviye + 1;
  while (j < n && toplamAg + w[j] <= W) {
    toplamAg += w[j];
    sonuc += p[j];
    j++;
  }
  if (j < n)                // kesirli parça
    sonuc += (W - toplamAg) * (double) p[j] / w[j];
  return sonuc;
}
```

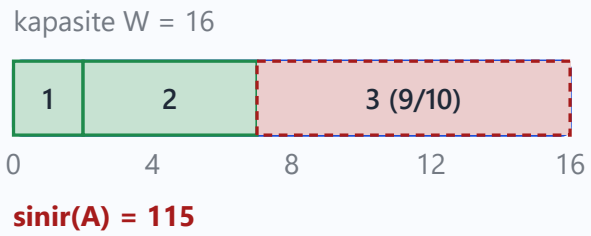
Satırların görevi şöyledir:

- `if (u.agirlik > W) return 0;` olursuz düğümü yakalar. Ağırlık kapasiteyi aşmışsa düğümün altında hiçbir olurlu çözüm yoktur; 0 dönen sınır onun kuyruğa girmesini engeller.
- `sonuc` düğümün kendi kazancıyla, `toplamAg` kendi ağırlığıyla başlar. `j`, kararı verilmemiş ilk eşyayı gösterir.
- `while` döngüsü, sığdığı sürece eşyaları tam ekler. Koşulun iki parçası vardır: eşya kalmalıdır (`j < n`) ve eşya *j* kalan kapasiteye tam sığmalıdır.
- Döngü bittiğinde ya eşya kalmamıştır ya da `j`, ilk sığmayan eşyadır. İkinci durumda `(W - toplamAg) * (double) p[j] / w[j]` kadar kesirli kazanç eklenir. `(double)` dönüşümü tam sayı bölmesini önler.

Fonksiyon bir düğüm için en fazla *n* eşyaya bakar; süresi *O(n)*'dir. Ek bellek kullanmaz. Tablo 10.5 kök için çağrının izlemesidir.

Adım	Sınama	`j`	`toplamAg`	`sonuc`
Giriş	*0 ≤ 16*: olurlu	0	0	0
Döngü 1	*0 + w[0] = 2 ≤ 16*: eşya 1 tam sığar	1	2	40
Döngü 2	*2 + w[1] = 7 ≤ 16*: eşya 2 tam sığar	2	7	70
Döngü sonu	*7 + w[2] = 17 > 16*: eşya 3 sığmaz	2	7	70
Kesir	*j = 2 < 4*: *(16 − 7) · 50/10 = 45*	2	7	115

Tablo 10.5: `sinir(kök)` izlemesi (*W = 16*; kök: düzey −1, kazanç 0, ağırlık 0).



Şekil 10.6: Kökün sınırı kapasite çubuğu olarak. Eşya 1 (2 birim) ve eşya 2 (5 birim) tam girer; kalan 9 birim eşya 3'ün *9/10*'uyla doldurulur. *sinir(A) = 115*.

Şekil 10.6 aynı hesabı bir çubuk üzerinde gösterir. Çubuğun yeşil kısmı tam alınan eşyalardır, taralı kısım kesirli parçadır. Kesirli parça 0/1 problemde alınamaz; sınırın gerçek değerden büyük olmasının nedeni budur.

Örnek 10.1

Eşyalar (*p*, *w*) = (20, 2), (30, 5), (35, 7), (12, 3) ve *W = 9* olsun. Oranlar 10, 6, 5, 4'tür; eşyalar sıralıdır.

1. **Kök.** Eşya 1 ve 2 tam sığar: *20 + 30 = 50*, ağırlık 7. Eşya 3 sığmaz (*7 + 7 = 14 > 9*); kalan 2 birim için *2 · 35/7 = 10*. *sinir(kök) = 60*.
2. **"Eşya 1 alınmadı" düğümü** (düzey 0, kazanç 0, ağırlık 0). Eşya 2 tam sığar: 30, ağırlık 5. Eşya 3 sığmaz; kalan 4 birim için *4 · 35/7 = 20*. Sınır 50'dir.
3. **"Eşya 1 alındı, eşya 2 alınmadı" düğümü** (düzey 1, kazanç 20, ağırlık 2). Eşya 3 tam sığar: *20 + 35 = 55*, ağırlık 9. Kapasite dolmuştur; eşya 4 için kesir 0'dır. Sınır 55'tir.

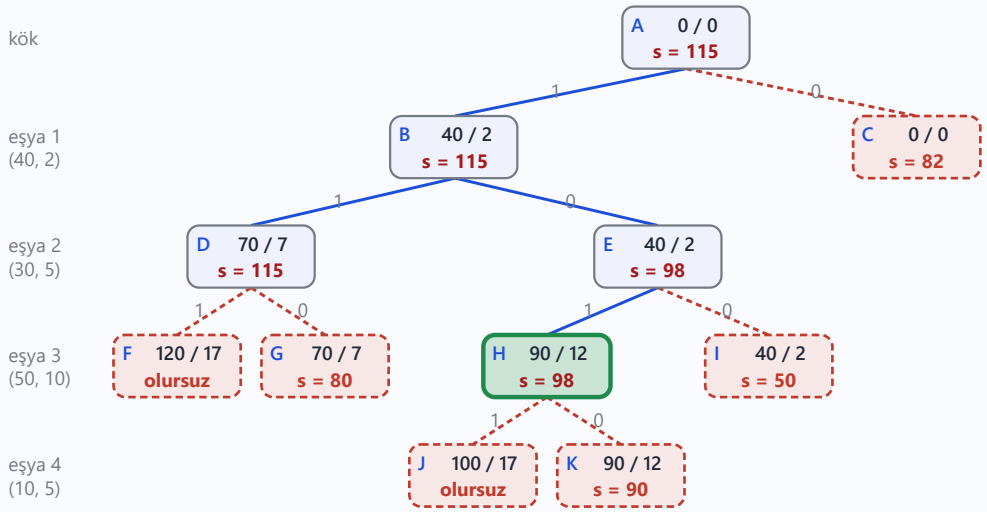
Optimum eşya 1 ve 3'tür: kazanç 55, ağırlık 9. Oran sırasıyla açgözlü seçim ise 1 ve 2'yi alır ve 50'de kalır. Optimum bulunduktan sonra "eşya 1 alınmadı" düğümünün sınırı *50 ≤ 55* olur ve düğüm budanır: eşya 1'siz hiçbir çözüm 55'i geçemez.

10.3.4 En iyi önce dal sınır adım adım

Şimdi asıl örnek ($W = 16$) en iyi önce stratejisiyle çözülür. Her düğüm bir harfle adlandırılır ve "kazanç / ağırlık" çifti ile sınırı (s) birlikte verilir. Kuyruk bir öncelikli kuyruktur; her adımda sınırı en büyük düğüm çıkar. Bir çocuk üretildiğinde, olurluysa ve kazancı `enIyi`'den büyükse `enIyi` hemen güncellenir. Sırt çantasında her olurlu düğüm, alınan eşyalardan oluşan geçerli bir çözümdür; güncelleme için yaprağa inmek gerekmez.

1. **Başlangıç.** Kök A: hiçbir eşya seçilmedi; 0 / 0, $s = 115$. `enIyi` = 0. Kuyruk: A (115).
2. **A çıkarılır ve dallanır.** B (eşya 1 alındı): 40 / 2, olurlu; `enIyi` = 40. B'nin sınırı kökünküyle aynıdır, çünkü kökün sınırı zaten eşya 1'i tam alıyordu: $s = 115$. C (eşya 1 yok): 0 / 0. Sınır hesabında eşya 2 ve eşya 3 tam sığar: $30 + 50 = 80$, ağırlık $5 + 10 = 15$. Eşya 4 sığmaz; kalan 1 birim için $1 \cdot 10/5 = 2$ eklenir. $s = 82$. Kuyruk: B (115), C (82).
3. **B çıkarılır ve dallanır.** D (eşya 2 alındı): 70 / 7; `enIyi` = 70, $s = 115$. E (eşya 2 yok): 40 / 2; eşya 3 tam sığar (90, ağırlık 12), eşya 4'ün $4/5$'i eklenir (8): $s = 98$. Kuyruk: D (115), E (98), C (82).
4. **D çıkarılır ve dallanır.** F (eşya 3 alındı): ağırlık $7 + 10 = 17 > 16$; olursuz, sınırı 0, budanır. G (eşya 3 yok): 70 / 7; eşya 4 tam sığar (80, ağırlık 12), başka eşya yok: $s = 80$. $80 > 70$ olduğu için G kuyruğa girer. Kuyruk: E (98), C (82), G (80).
5. **E çıkarılır ve dallanır.** H (eşya 3 alındı): 90 / 12, olurlu; `enIyi` = 90. H'nin sınırı $90 + 4 \cdot 2 = 98$'dir ve kuyruğa girer. I (eşya 3 yok): 40 / 2; eşya 4 tam sığar: $s = 50 \le 90$, budanır. Kuyruk: H (98), C (82), G (80).
6. **H çıkarılır ve dallanır.** J (eşya 4 alındı): ağırlık 17; olursuz. K (eşya 4 yok): 90 / 12, son eşya; $s = 90 \le 90$, budanır. Kuyruk: C (82), G (80).
7. **C ve G atılır.** C çıkarılır: $82 \le 90$, tekrar sınamada atılır. G çıkarılır: $80 \le 90$, atılır. Kuyruk boşalır.

Sonuç `enIyi` = 90'dır ve H düğümünde bulunmuştur: eşya 1 ve 3. Tam ağacın 31 düğümünden yalnız 11'i üretilmiştir: A'dan K'ye kadar. Bunların 5'i genişletilmiştir: A, B, D, E, H. Kuyruk hiçbir anda 3 düğümü aşmamıştır.



Şekil 10.7: En iyi önce dal sınırın son durumu. Her kutuda düğüm harfi, "kazanç / ağırlık" ve sınır (*s*) vardır. Kırmızı kesikli kutular olursuz ya da sınırla budanmış düğümlerdir. Yeşil kutu H en iyi çözümdür: kazanç 90, ağırlık 12. Soldaki etiketler her düzeyde kararı verilen eşyayı gösterir.

Şekil 10.7 aramanın bitmiş ağacını gösterir. İzlemenin iki anı öğreticidir. Birincisi 4. adımdır. D'nin sınırı 115'tir ama D'nin "al" çocuğu olursuzdur ve "alma" çocuğunun sınırı 80'e düşer. Sınır, karar verildikçe gerçeğe yaklaşır. İkincisi son adımdır. C ve G kuyruğa girdikleri anda umut vaat ediyordu: C girdiğinde `enIyi` 40, G girdiğinde 70'ti. Sıraları geldiğinde `enIyi` 90 olmuştu. Tekrar sınama olmasaydı ikisi de genişletilirdi; sonuç değişmezdi ama boşuna iş yapılırdı.

> **Sık yapılan hata**
>
> `enIyi` 'yi yalnız yapraklarda güncellemek. Sırt çantasında her olurlu düğüm bir çözümdür: seçilen eşyalar çantaya konur, kalanlar alınmaz. Yukarıdaki izlemede `enIyi` 2. adımda 40'a, 3. adımda 70'e, 5. adımda 90'a çıkar. Güncelleme yaprağa kadar bekletilseydi `enIyi` uzun süre 0'da kalırdı ve I gibi düğümler budanamazdı.

10.3.5 Java kodu: dalSinir()

En iyi önce dal sınırın Java kodu aşağıdadır. Öncelikli kuyruk olarak `PriorityQueue` kullanılır.

```
int dalSinir() {
  PriorityQueue<Dugum> kuyruk = new PriorityQueue<>(
      (a, b) -> Double.compare(b.sinir, a.sinir));
  Dugum kok = new Dugum(-1, 0, 0);
  kok.sinir = sinir(kok);
  kuyruk.add(kok);  int enIyi = 0;
  while (!kuyruk.isEmpty()) {
    Dugum u = kuyruk.poll();           // en büyük sınır
    if (u.sinir <= enIyi || u.seviye == n - 1) continue;
    int i = u.seviye + 1;
    // sol çocuk: i. eşyayı al
    Dugum al = new Dugum(i, u.kazanc + p[i],
                         u.agirlik + w[i]);
    if (al.agirlik <= W && al.kazanc > enIyi)
      enIyi = al.kazanc;
    al.sinir = sinir(al);
    if (al.sinir > enIyi) kuyruk.add(al);
    // sağ çocuk: i. eşyayı alma
    Dugum alma = new Dugum(i, u.kazanc, u.agirlik);
    alma.sinir = sinir(alma);
    if (alma.sinir > enIyi) kuyruk.add(alma);
  }
  return enIyi;
}
```

Kodun önemli noktaları şunlardır:

- **Karşılaştırıcı terstir.** `PriorityQueue` varsayılan olarak en küçük öğeyi önce verir. `(a, b) -> Double.compare(b.sinir, a.sinir)` karşılaştırıcısı sırayı tersine çevirir; böylece `poll()` sınırı en büyük düğümü döndürür.
- **Tekrar sınama.** `if (u.sinir <= enIyi || u.seviye == n - 1) continue;` satırı iki şeyi yakalar. Birincisi, kuyrukta beklerken eskiyen düğümlerdir (örnekte C ve G). İkincisi, son düzeydeki düğümlerdir; onların çocuğu yoktur. Sınırın kendisi kuyrukta değişmez; değişen `enIyi`'dir.
- **Sol çocuk.** `al` düğümü *i.* eşyayı ekler. Ağırlık kapasiteyi aşmıyor ve kazanç `enIyi`'den büyükse `enIyi` güncellenir. Sonra sınırı hesaplanır ve sınır `enIyi`'yi geçiyorsa kuyruğa girer. Olursuz bir `al` düğümünün sınırı 0'dır; kuyruğa hiç girmez.
- **Sağ çocuk.** `alma` düğümü ebeveyninin kazanç ve ağırlığını aynen taşır. Ebeveyn olurluysa `alma` da olurludur ve kazancı `enIyi`'yi değiştirmez. Bu yüzden yalnız sınırı denetlenir.
- **Strateji değişikliği.** FIFO için `ArrayDeque` ile `add()` / `poll()`, LIFO için `push()` / `pop()` kullanılır. Kodun geri kalanı aynıdır.

Kod yalnız en iyi değeri döndürür. Seçilen eşyalar da isteniyorsa `Dugum` sınıfına bir `boolean[]` dizisi ya da bir bit maskesi eklenir. Her çocuk ebeveyninin dizisini kopyalar ve kendi kararını işler. `enIyi` güncellenirken bu dizi de saklanır.

Maliyet. En kötü durumda tam ağacın bütün düğümleri üretilir: $O(2^n)$ düğüm. Her düğümde sınır hesabı $O(n)$, kuyruk işlemi $O(\log k)$ sürer. En kötü süre $O(n \cdot 2^n)$'dir. Bellek kuyruktaki düğüm sayısıyla orantılıdır ve en kötü durumda o da üsteldir. Pratikte iyi bir sınırla üretilen düğüm sayısı bunun çok altında kalır.

10.3.6 FIFO dal sınır: kod izleme

Aynı örnek bu kez FIFO kuyrukla çözülür. Kod, öncelikli kuyruk yerine `ArrayDeque` kullanır. Çocuklar `cocuk(u, al)` yardımcı metoduyla üretilir; metot çocuğu kurar ve sınırını da hesaplar. Döngü önce "al" (`al = 1`), sonra "alma" (`al = 0`) çocuğunu üretir.

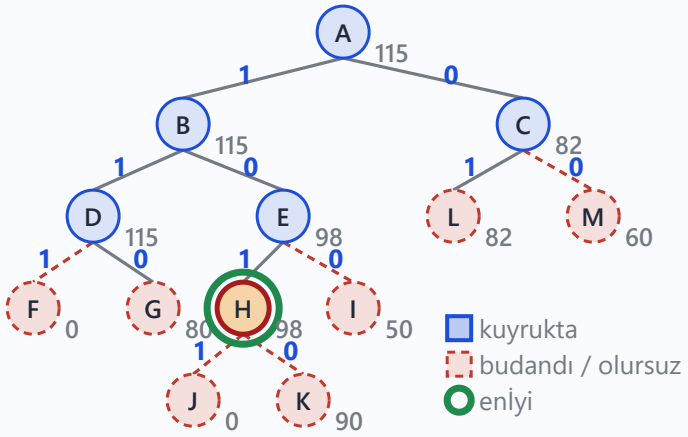
```
int fifoDalSinir() {
  ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
  kuyruk.add(kok);  int enIyi = 0;
  while (!kuyruk.isEmpty()) {
    Dugum u = kuyruk.poll();          // ilk giren
    if (u.sinir <= enIyi || u.seviye == n - 1) continue;
    for (int al = 1; al >= 0; al--) { // önce al
      Dugum c = cocuk(u, al);         // sinir(c) dahil
      if (c.agirlik <= W && c.kazanc > enIyi)
        enIyi = c.kazanc;               // yeni aday
      if (c.sinir > enIyi) kuyruk.add(c);
    }
  }
  return enIyi;
}
```

Bu kodda `kok` önceden kurulmuş ve sınırı hesaplanmıştır. Döngünün yapısı en iyi önce koduyla aynıdır; tek fark, `poll()` 'un kuyruğa ilk giren düğümü döndürmesidir. Tablo 10.6 izlemenin her adımını verir. Harfler üretim sırasını izlemez; C'nin çocukları L ve M diye adlandırılmıştır.

Çıkan	"Al" çocuğu	"Alma" çocuğu	`enIyi`	Kuyruk (baştan sona)
—	—	—	0	A
A (115)	B 40/2, 115 → kuyruğa	C 0/0, 82 → kuyruğa	40	B, C
B (115)	D 70/7, 115 → kuyruğa	E 40/2, 98 → kuyruğa	70	C, D, E
C (82)	L 30/5, 82 → kuyruğa	M 0/0, 60 ≤ 70: budandı	70	D, E, L
D (115)	F 120/17: olursuz	G 70/7, 80 → kuyruğa	70	E, L, G
E (98)	H 90/12, 98 → kuyruğa	I 40/2, 50 ≤ 90: budandı	90	L, G, H
L (82)	82 ≤ 90: tekrar sınamada atılır		90	G, H
G (80)	80 ≤ 90: tekrar sınamada atılır		90	H
H (98)	J 100/17: olursuz	K 90/12, 90 ≤ 90: budandı	90	boş

Tablo 10.6: FIFO dal sınır izlemesi ($W = 16$). Düğümler "kazanç / ağırlık, sınır" biçimindedir.

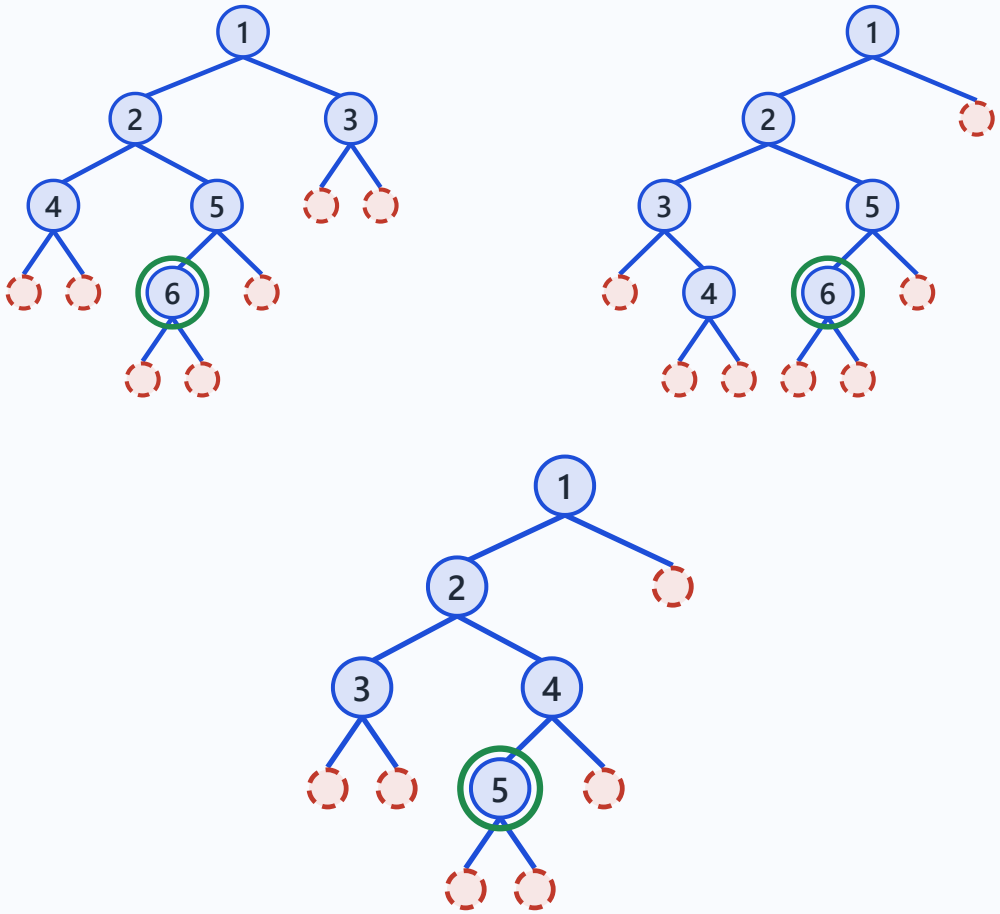
Sonuç yine 90'dır. FIFO 13 düğüm üretmiş ve 6 düğüm genişletmiştir: A, B, C, D, E, H. Kuyruk en çok 3 düğüm tutmuştur. En iyi önce stratejisiyle farkın kaynağı C düğümüdür. En iyi önce, C'nin sınırı (82) kuyruktaki öbür düğümlerden küçük olduğu için C'yi sona bırakmıştı; o sıraya gelindiğinde `enIyi` 90 olmuş ve C hiç genişletilmemişti. FIFO ise C'yi düzey sırasına göre üçüncü olarak genişletir. O anda `enIyi` yalnız 70'tir ve C'nin sınırı 82 olduğu için C atılamaz. C'nin genişletilmesi iki fazla düğüm (L ve M) üretir ve bir fazla genişletme demektir. L de sonunda boşuna kuyrukta bekler.



Şekil 10.8: FIFO dal sınırın son durumu. Düğümlerin yanındaki sayılar sınırdır. Mavi düğümler kuyruğa girmiş canlı düğümlerdir; kırmızı kesikli düğümler budanmış ya da olursuzdur. Yeşil halka en iyi çözüm H'dir (90).

10.3.7 Üç stratejinin karşılaştırılması

Aynı örnek LIFO stratejisiyle de çözülebilir. LIFO'da canlı düğümler bir yığında bekler. Çocuklar yığına "önce al çıksın" diye konursa, yani önce "alma", sonra "al" çocuğu eklenirse arama hep sol daldan derine iner. A'dan sonra B, sonra D genişletilir. D'nin "al" çocuğu olursuzdur; "alma" çocuğu G (70/7, 80) genişletilir. G'nin "al" çocuğu eşya 4'ü ekler: 80/12. Bu olurlu bir çözümdür ve `enIyi` 80'e çıkar. Arama sonra E'ye döner ve H'de 90'ı bulur. Sonunda yığında kalan C (82) atılır. LIFO da 13 düğüm üretir ve 6 düğüm genişletir: A, B, D, G, E, H.



Şekil 10.9: Aynı örnekte üç strateji: solda FIFO (13 düğüm, 6 genişletme), ortada LIFO (13 düğüm, 6 genişletme), sağda en iyi önce (11 düğüm, 5 genişletme). Numaralar genişletme sırasıdır. Küçük daireler üretilip genişletilmeyen düğümlerdir; kırmızı kesikliler budanmıştır. Halka optimumu (90) gösterir.

Şekil 10.9 üç aramayı yan yana gösterir; Tablo 10.7 sayıları özetler.

Yöntem	Üretilen düğüm	Genişletilen	En büyük kuyruk
Kaba kuvvet (tam ağaç)	31	15	—
FIFO dal sınır	13	6	3
LIFO dal sınır (önce “al”)	13	6	3
En iyi önce (LC)	11	5	3

Tablo 10.7: Aynı sırt çantası örneğinde yöntemlerin iş miktarı.

Tablodan üç sonuç çıkar. Birincisi, üç strateji de aynı optimumu bulur; strateji yalnız iş miktarını etkiler. İkincisi, en iyi önce bu örnekte en az düğümü üretmiştir. Bu genel bir eğilimdir ama garanti değildir. Üçüncüsü, küçük örnekte bellek farkı görünmez: üçünde de kuyruk en çok 3 düğüm tutar. Büyük örneklerde fark açılır. FIFO ve en iyi önce stratejilerinde kuyruk üstel büyüyebilir; LIFO'da yığın derinlikle orantılı kalır.

Çocukların yığına konma sırası LIFO'da önemlidir. Sıra tersine çevrilirse, yani “alma” çocuğu önce çıkarsa, arama önce eşya 1'in alınmadığı dala dalar. O dalda iyi çözümler yoktur; `enIyi` önce 40, sonra 50, 60, 80 diye yavaşça yükselir. Bu sürüm 21 düğüm üretir ve 10 düğüm genişletir. Derinlik öncelikli aramada iyi bir çözümü erken bulmak bütün aramayı hızlandırır. Bu yüzden çocuklar, umut vaat eden dal önce çıkacak biçimde sıralanır.

Örnek 10.2

Örnek 10.1'deki eşyalar (20, 2), (30, 5), (35, 7), (12, 3) ve $W = 9$ en iyi önce stratejisiyle çözülür.

1. Kök A: 0 / 0, $s = 60$. Kuyruk: A (60).
2. A genişletilir. B (eşya 1 alındı): 20 / 2, $s = 60$; `enIyi` = 20. C (eşya 1 yok): $s = 50$. Kuyruk: B (60), C (50).
3. B genişletilir. D (eşya 2 alındı): 50 / 7, $s = 60$; `enIyi` = 50. E (eşya 2 yok): 20 / 2, $s = 55$. Kuyruk: D (60), E (55), C (50).
4. D genişletilir. F (eşya 3 alındı): ağırlık 14 > 9, olursuz. G (eşya 3 yok): 50 / 7; eşya 4'ün *2/3*'ü: $s = 50 + 8 = 58$. Kuyruk: G (58), E (55), C (50).
5. G genişletilir. H (eşya 4 alındı): ağırlık 10 > 9, olursuz. I (eşya 4 yok): $s = 50 \le 50$, budanır. Kuyruk: E (55), C (50).
6. E genişletilir. J (eşya 3 alındı): 55 / 9; `enIyi` = 55. J'nin sınırı 55'tir; $55 \le 55$, kuyruğa girmez. K (eşya 3 yok): $s = 32 \le 55$, budanır. Kuyruk: C (50).
7. C çıkarılır: $50 \le 55$, atılır. Kuyruk boş.

Sonuç 55'tir (eşya 1 ve 3). 11 düğüm üretilmiş, 5'i genişletilmiştir. C eklendiğinde `enIyi` 20'ydi; çıkarıldığında 55 olmuştu. 6. adımda J'nin değeri `enIyi` 'ye yazılır ama J kuyruğa girmez. J'nin sınırı değerine eşittir; altında daha iyi bir çözüm yoktur.

10.4 Gezgin satıcı problemi

Gezgin satıcı problemi (*travelling salesman problem*, TSP) bir en küçükleme problemidir. Kısaca hatırlatmak gerekirse: n şehir ve her şehir çifti arasındaki uzaklık verilir. Her şehri tam bir kez ziyaret edip başlangıca dönen en kısa tur aranır. Uzaklıklar bir d matrisinde tutulur; $d[i][j]$, i'den j'ye gitmenin maliyetidir. $d[i][j] = d[j][i]$ ise problem simetriktir. Problem NP-zordur ve kaba kuvvet $(n - 1)!$ tur dener. Bu kesimde aynı dört şehirli örnek üç farklı alt sınırla incelenir.

10.4.1 Durum uzayı ve en ucuz çıkış kenarı sınırı

Tur her zaman şehir 0'dan başlar. Durum uzayı ağacının her düğümü 0'dan başlayan bir **kısmi yoldur**. Kök yalnız 0'ı içerir. Bir düğümün çocukları, yolun son şehrinden henüz ziyaret edilmemiş bir şehre gidilerek elde edilir. $n - 1$ derinliğindeki düğümler bütün şehirleri içerir; son şehirden 0'a dönü-

lünce tur tamamlanır.

Problem en küçükleme olduğu için her düğüme bir **alt sınır** gerekir: o kısmi yolu tamamlayan hiçbir turun maliyeti bu değerin altına inemez. En basit alt sınır şu gözleme dayanır. Turu tamamlamak için bulunulan son şehirden ve henüz ziyaret edilmemiş her şehirden tam bir kez çıkılmak zorundadır. Her çıkış, o şehrin en ucuz kenarı kadar tutar. Şehir *v*'nin en ucuz çıkış kenarının maliyeti `enKucuk[v]` olsun. Bu değerler arama başlamadan bir kez hesaplanır. Alt sınır şöyledir:

> $$alt(yol) = maliyet(yol) + enKucuk[\text{son şehir}] + \Sigma_{v \text{ ziyaret edilmemiş}}\, enKucuk[v]$$

Bu sınır bir gevşetmedir. Asıl problemde çıkılan kenarlar tek bir tur oluşturmalıdır. Gevşetmede bu koşul kaldırılır: her şehir kendi en ucuz kenarıyla çıkar, kenarların bir tur oluşturup oluşturmadığına bakılmaz. Kaldırılan kısıt yüzünden gevşetmenin değeri hiçbir turun maliyetinden büyük olamaz.

Teorem 10.2 (En ucuz çıkış sınırı geçerlidir)

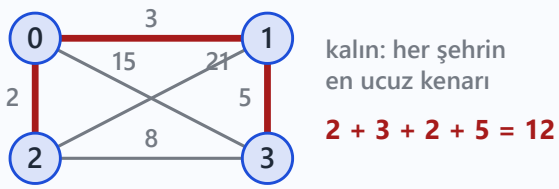
0'dan başlayan her kısmi yol için, bu yolu tamamlayan her turun maliyeti *alt(yol)*'dan büyük ya da ona eşittir.

İspat. Yolu tamamlayan herhangi bir tur, yolun kenarlarını aynen içerir; bunların toplamı *maliyet(yol)*'dur. Turun geri kalanında son şehirden bir kez ve ziyaret edilmemiş her şehirden bir kez çıkılır; başka çıkış yoktur. Şehir *v*'den çıkan kenarın maliyeti en az `enKucuk[v]`'dir. Bu eşitsizlikler toplanınca turun maliyeti en az *alt(yol)* olur. ■

Bu notta kullanılan dört şehirli simetrik matris Tablo 10.8'te verilmiştir. Son sütun her satırın en küçüğüdür, yani `enKucuk` dizisidir.

	0	1	2	3	en küçük
0	—	3	2	15	2
1	3	—	21	5	3
2	2	21	—	8	2
3	15	5	8	—	5

Tablo 10.8: Dört şehirli simetrik uzaklık matrisi *d* ve `enKucuk`.



Şekil 10.10: Dört şehir ve kenar uzunlukları. Kalın kenarlar her şehrin en ucuz çıkış kenarıdır: 0-2, 1-0, 2-0, 3-

1. Kökün alt sınırı *2 + 3 + 2 + 5 = 12*'dir.

Kökte yol yalnız 0'dır ve maliyeti 0'dır. Son şehir 0'dır ve 1, 2, 3 ziyaret edilmemiştir. Alt sınır *0 + 2 + 3 + 2 + 5 = 12* olur. Şekil 10.10'da kalın çizilen dört kenar bir tur oluşturmaz: 0-2 kenarı iki kez seçilmiştir ve 3'e hiç girilmez. Gevşetmenin değeri bu yüzden gerçek turdan küçüktür.

Yol 0 → 1 için sınır şöyledir. Yolun maliyeti *d[0][1] = 3*'tür. Son şehir 1'den çıkılacaktır: `enKucuk[1]` = 3. Ziyaret edilmemiş şehirler 2 ve 3'tür: *2 + 5*. Toplam *3 + 3 + 7 = 13*'tür. Sınır kökünkinden büyüktür; karar verildikçe sınır gerçeğe yaklaşır.

10.4.2 Kod izleme: tsp(0, 0, 1)

Gezgin satıcı için dal sınır, sınırlı bir derinlik öncelikli arama olarak yazılır. Bu bir LIFO dal sınırdır; yığın görevini özyineleme üstlenir.

```
void tsp(int sehir, int maliyet, int sayac) {
  if (sayac == n) {           // tur tamam
    int tur = maliyet + d[sehir][0];
    if (tur < enIyi) enIyi = tur;
    return;
  }
  for (int s = 0; s < n; s++) {
    if (ziyaret[s]) continue;
    int yeni = maliyet + d[sehir][s];
    ziyaret[s] = true;
    if (altSinir(s, yeni) < enIyi) // sınır
      tsp(s, yeni, sayac + 1);
    ziyaret[s] = false;      // geri al
  }
}
```

Metodun parametreleri yolun son şehri (`sehir`), yolun maliyeti (`maliyet`) ve yoldaki şehir sayısıdır (`sayac`). `ziyaret` dizisi yoldaki şehirleri işaretler; `enIyi` başta sonsuzdur. İlk çağrıdan önce `ziyaret[0] = true` yapılır ve `tsp(0, 0, 1)` çağrılır. Satırların görevi şöyledir:

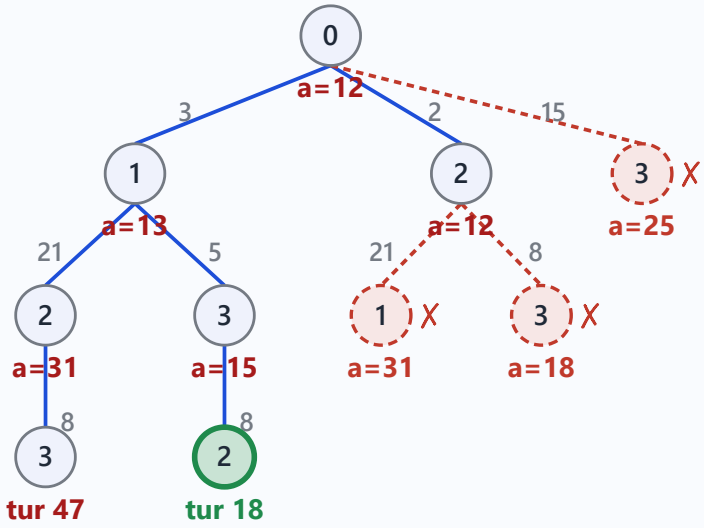
- `if (sayac == n)` taban durumdur. Bütün şehirler yoldadır; son şehirden 0'a dönülür ve tur tamamlanır. Tur `enIyi` 'den kısaysa `enIyi` güncellenir.
- `for` döngüsü dallanmadır. Ziyaret edilmemiş her şehir `s` bir çocuktur. Ziyaret edilmiş şehirler `continue` ile atlanır; bu, olursuzluk kuralıdır.
- `ziyaret[s] = true;` şehri yola ekler. Sınır hesabı bu işaretlemeden sonra yapılır; böylece `altSinir` şehir `s` 'yi "ziyaret edildi" görür ve onu yalnız son şehir olarak sayar.
- `if (altSinir(s, yeni) < enIyi)` sınır kuralıdır. Alt sınır `enIyi` 'den küçükse dala girilir; değilse dal budanır.
- `ziyaret[s] = false;` geri almadır. Döngü bir sonraki şehre geçmeden yol eski hâline döner.

Tablo 10.9 aramanın bütün adımlarını sırayla verir.

Adım	Kenar	`yeni`	`alt`	`enIyi`	Karar
1	0 → 1	*0 + 3 = 3*	*3 + 3 + 2 + 5 = 13*	∞	dallanılır
2	1 → 2	*3 + 21 = 24*	*24 + 2 + 5 = 31*	∞	dallanılır
3	2 → 3	*24 + 8 = 32*	*32 + 5 = 37*	∞	dallanılır
4	tur 0 → 1 → 2 → 3 → 0 = *32 + 15 = 47*			47	ilk aday
5	1 → 3	*3 + 5 = 8*	*8 + 5 + 2 = 15*	47	dallanılır
6	3 → 2	*8 + 8 = 16*	*16 + 2 = 18*	47	dallanılır
7	tur 0 → 1 → 3 → 2 → 0 = *16 + 2 = 18*			18	yeni en iyi
8	0 → 2	*0 + 2 = 2*	*2 + 2 + 3 + 5 = 12*	18	dallanılır
9	2 → 1	*2 + 21 = 23*	*23 + 3 + 5 = 31*	18	budanır
10	2 → 3	*2 + 8 = 10*	*10 + 5 + 3 = 18*	18	budanır
11	0 → 3	*0 + 15 = 15*	*15 + 5 + 3 + 2 = 25*	18	budanır

Tablo 10.9: `tsp(0, 0, 1)` izlemesi.

İzlemede dikkat çeken birkaç nokta vardır. İlk tur (47) kötüdür ama `enIyi`'ye sonlu bir değer verir. Hemen ardından 18 bulunur. 8. adımda 0 → 2 dalının sınırı 12'dir; *12 < 18* olduğu için dala girilir. Bu dalın iki çocuğu budanır. 10. adımda sınır `enIyi`'ye eşittir: *18 ≥ 18*. Bu daldaki en iyi tur en fazla 18 olabilir ve 18'lik bir tur zaten bulunmuştur. 11. adımda kökün son çocuğu 0 → 3 doğrudan budanır ve altındaki iki tur hiç denenmez.



Şekil 10.11: `tsp` aramasının bitmiş ağacı. Her düğümün altındaki *a* değeri alt sınırdır. Kırmızı kesikli düğümler budanmıştır. Yeşil düğüm en kısa turu (18) tamamlar.

Sonuç en kısa tur 0 → 1 → 3 → 2 → 0'dır ve uzunluğu *3 + 5 + 8 + 2 = 18*'dir. Tam durum uzayı ağacında *1 + 3 + 6 + 6 = 16* düğüm vardır. Dal sınır, kök dahil 10 düğüm üretmiş ve `tsp` metodunu 7 kez çağırmıştır. Budamasız arama 16 düğümün hepsini üretir ve metodu 16 kez çağırır.

Kodda en iyi turun kendisi saklanmaz. Saklamak için bir `int[] yol` dizisi tutulur. Döngüde dala girmeden önce `yol[sayac] = s` yazılır. Taban durumda tur `enIyi`'den kısaysa `enIyiYol = yol.clone()` ile kopyalanır. Kopya gereklidir, çünkü `yol` dizisi aramanın geri kalanında değişmeye devam eder.

10.4.3 Kod izleme: altSinir(3, 8)

Sınır fonksiyonu kısadır:

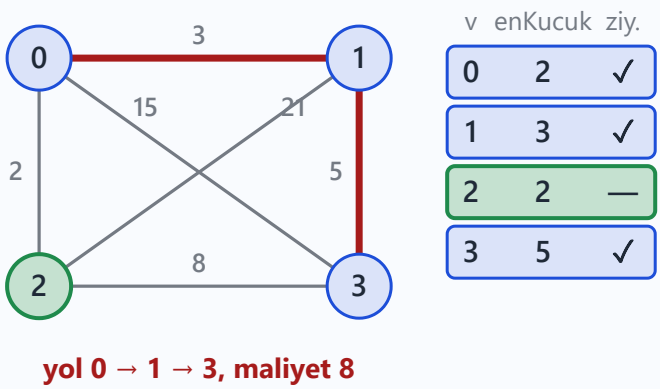
```
int altSinir(int son, int maliyet) {
  int alt = maliyet + enKucuk[son];
  for (int v = 0; v < n; v++)
    if (!ziyaret[v])       // gidilmemiş
      alt += enKucuk[v];
  return alt;
}
```

`alt` önce yolun maliyeti ile son şehrin en ucuz çıkışını toplar. Döngü bütün şehirleri gezer ve ziyaret edilmemiş her şehrin `enKucuk` değerini ekler. Ziyaret edilmiş bir şehir iki türlüdür. Ya yolun içindedir ve çıkış kenarı zaten maliyete katılmıştır, ya da son şehirdir ve çıkışı en başta eklenmiştir. Her iki durumda da yeniden eklenmemelidir. Fonksiyon *O(n)* sürer.

`tsp` izlemesinin 5. adımındaki çağrı `altSinir(3, 8)`'dir. Yol 0 → 1 → 3'tür, maliyeti *3 + 5 = 8*'dir ve `ziyaret` = {0, 1, 3}'tür.

Adım	Açıklama	`alt`
Başlangıç	*8 +* `enKucuk[3]` *= 8 + 5*: son şehir 3'ten bir kez daha çıkılacak	13
v = 0	ziyaret edildi; çıkış kenarı (0 → 1) maliyette	13
v = 1	ziyaret edildi; çıkış kenarı (1 → 3) maliyette	13
v = 2	gidilmedi; en az `enKucuk[2]` = 2 ile çıkılmalı	15
v = 3	ziyaret edildi; çıkışı başta eklendi	15

Tablo 10.10: `altSinir(3, 8)` izlemesi.



Şekil 10.12: `altSinir(3, 8)`. Kırmızı kenarlar yol 0 → 1 → 3'tür (maliyet 8). Sağdaki tablo her şehrin `enKucuk` değerini ve ziyaret durumunu gösterir; yalnız şehir 2 sınıra katkı verir.

Fonksiyon 15 döndürür. Bu daldaki her tur en az 15 tutar. Gerçek tur 0 → 1 → 3 → 2 → 0'dır ve 18 tutar. Arama sırasında `enIyi` 47 olduğu için *15 < 47* sağlanır ve dala girilir.

Örnek 10.3

Aynı matriste 0 → 3 → 1 yolunun alt sınırı nedir? Yolun maliyeti *15 + 5 = 20*'dir. Son şehir 1'dir: `enKucuk[1]` = 3. Ziyaret edilmemiş tek şehir 2'dir: `enKucuk[2]` = 2. Sınır *20 + 3 + 2 = 25*'tir. *25 ≥ 18* olduğu için optimum bulunduktan sonra bu yol budanır. Arama zaten kökte 0 → 3'ü (sınır 25) budadığı için bu düğüme hiç ulaşılmaz.

10.4.4 Daha sıkı sınır: iki en kısa kenar

En ucuz çıkış sınırı her şehirden yalnız bir kenar sayar. Oysa bir turda her şehre bir kenarla girilir ve bir kenarla çıkılır. Simetrik problemde giriş ve çıkış ayrımı yoktur: her şehir tam olarak iki tur kenarına değer. Şehrin bu iki kenarının toplamı, ona değen en kısa iki kenarın toplamından küçük olamaz. Bu gözlem tüm şehirler için toplanırsa her tur kenarı iki kez sayılır: bir kez bir ucunda, bir kez öbür ucunda. Böylece daha sıkı bir alt sınır elde edilir:

$$\text{tur} \geq \tfrac{1}{2} \cdot \Sigma_v \text{ (v'ye değen en kısa iki kenarın toplamı)}$$

Teorem 10.3 (İki en kısa kenar sınırı)

Simetrik bir gezgin satıcı probleminde her turun maliyeti, her şehre değen en kısa iki kenarın toplamlarının toplamının yarısından küçük olamaz.

İspat. Bir tur T olsun. Her v şehri T'nin tam iki kenarına değer; bu iki kenarın toplamına $t(v)$ denirse $\Sigma_v\, t(v) = 2 \cdot \mathit{maliyet}(T)$ olur, çünkü her kenar iki ucunda birer kez sayılır. v'ye değen en kısa iki kenarın toplamı $e(v)$ olsun. T'nin v'ye değen iki kenarı da v'ye değen kenarlardır; toplamları $e(v)$'den küçük olamaz: $t(v) \geq e(v)$. Toplanınca $2 \cdot \mathit{maliyet}(T) \geq \Sigma_v\, e(v)$ elde edilir. ■

Kökte bu sınırı hesaplayan kod şöyledir:

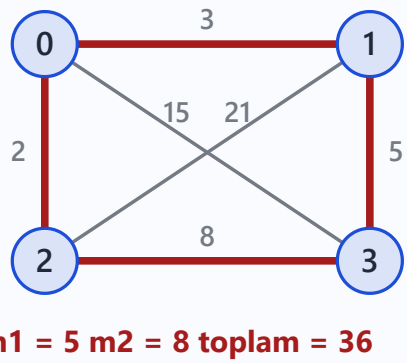
```
double ikiKenarSinir() {
  int toplam = 0;
  for (int v = 0; v < n; v++) {
    int m1 = INF, m2 = INF;     // en küçük iki
    for (int u = 0; u < n; u++) {
      if (u == v) continue;
      if (d[v][u] < m1) { m2 = m1; m1 = d[v][u]; }
      else if (d[v][u] < m2) m2 = d[v][u];
    }
    toplam += m1 + m2;
  }
  return toplam / 2.0;  // her kenar iki kez
}
```

Dış döngü şehirleri gezer. İç döngü, şehre değen kenarlar arasından en küçük iki değeri `m1` ≤ `m2` olacak biçimde bulur. Yeni bir kenar `m1`'den küçükse eski `m1` ikinci sıraya kayar ve yeni kenar birinci olur. Yalnız `m2`'den küçükse ikinci sıraya yazılır. Bu, iki öğeli bir "en küçükler listesi"nin güncellenmesidir. Toplam her kenarı iki kez saydığı için sonuç 2.0'a bölünür; `double` bölme, toplam tek sayıysa yarımı korur. Süre $O(n^2)$'dir.

v	Kenarlar	`m1`	`m2`	`toplam`
0	3, 2, 15	2	3	5
1	3, 21, 5	3	5	13
2	2, 21, 8	2	8	23
3	15, 5, 8	5	8	36

Tablo 10.11: `ikiKenarSinir()` izlemesi.

$v = 0$ satırı iç döngünün nasıl çalıştığını gösterir. Başta `m1` = `m2` = ∞'dur. $u = 1$'de $3 < \infty$: `m1` = 3 olur, eski `m1` (∞) `m2`'ye kayar. $u = 2$'de $2 < 3$: `m2` = 3, `m1` = 2 olur. $u = 3$'te 15 ikisinden de büyüktür; değişiklik yoktur. Toplama $2 + 3 = 5$ eklenir. Sonunda toplam 36'dır ve fonksiyon $36 / 2 = 18$ döndürür.



Şekil 10.13: Son şehir 3'ün en kısa iki kenarı (5 ve 8) ve dört şehrin seçtiği kenarlar. Kalın kenarlar 0-1, 1-3, 3-2, 2-0'dır ve tam da en kısa turu oluşturur. Toplam 36, sınır 18'dir.

Sonuç dikkat çekicidir: sınır 18'dir ve en kısa turun uzunluğuna eşittir. Şekil 10.13'te görüldüğü gibi her şehrin seçtiği iki kenar birlikte gerçek bir tur oluşturur. Gevşetmenin çözümü olurlu çıkmıştır; bu yüzden sınır sıkıdır. Kökte sınır optimuma eşit olduğu için, ilk tam tur bulunur bulunmaz kalan bütün dallar *alt ≥ enIyi* koşuluyla budanır. Kısmi yollarda aynı fikir, yoldaki kenarlar sabitlenerek uygulanır. Yoldaki bir şehrin kenarlarından biri ya da ikisi zaten bellidir. En kısa kenarlar yalnız eksik kalanlar için aranır.

10.4.5 İndirgenmiş maliyet matrisi

Üçüncü bir alt sınır, maliyet matrisinin satır ve sütunlarına bakar. Fikir şudur. Her tur, matrisin her satırından tam bir hücre (bir şehirden çıkış) ve her sütunundan tam bir hücre (bir şehre giriş) kullanır. Bir satırın bütün hücrelerinden aynı sayı çıkarılırsa her turun maliyeti tam o sayı kadar azalır. Turların sıralaması değişmez. Satırlardan ve sütunlardan çıkarılan sayıların toplamı, her turun maliyetinden düşülmüştür. Kalan matris negatif olmadığı için bu toplam bir alt sınırdır.

İşlem şöyle yapılır. Köşegen ∞ yapılır; bir şehirden kendine gidilmez. Her satırdan o satırın en küçüğü çıkarılır. Ardından her sütundan o sütunun en küçüğü çıkarılır. Sonuçta her satırda ve her sütunda en az bir 0 bulunur. Bu matrise **indirgenmiş maliyet matrisi** (*reduced cost matrix*) denir.

```
int indirge(int[][] m) {
  int toplam = 0;
  for (int i = 0; i < n; i++) {       // satırlar
    int mn = satirMin(m, i);
    if (mn > 0 && mn < INF) {
      satirdanCikar(m, i, mn); toplam += mn;
    }
  }
  for (int j = 0; j < n; j++) {       // sütunlar
    int mn = sutunMin(m, j);
    if (mn > 0 && mn < INF) {
      sutundanCikar(m, j, mn); toplam += mn;
    }
  }
  return toplam;
}
```

`satirMin`, `satirdanCikar`, `sutunMin` ve `sutundanCikar` basit yardımcı metotlardır. Çıkarma yalnız ∞ olmayan hücrelere uygulanır. `mn > 0` koşulu sıfır çıkarmayı atlar. `mn < INF` koşulu ise bütünüyle ∞ olan satır ve sütunları atlar; dallanmada bu tür satırlar oluşur. Metot matrisi yerinde değiştirir ve çıkarılan toplamı döndürür. İki geçiş de $O(n^2)$ sürer.

Adım	En küçük	Sonuç	`toplam`
Satır 0: ∞, 3, 2, 15	2	∞, 1, 0, 13	2
Satır 1: 3, ∞, 21, 5	3	0, ∞, 18, 2	5
Satır 2: 2, 21, ∞, 8	2	0, 19, ∞, 6	7
Satır 3: 15, 5, 8, ∞	5	10, 0, 3, ∞	12
Sütun 0, 1, 2	0	değişiklik yok	12
Sütun 3: 13, 2, 6, ∞	2	11, 0, 4, ∞	14

Tablo 10.12: `indirge()` izlemesi (dört şehirli örnek).

Satır indirgemesi bittiğinde toplam 12'dir. Bu sayı en ucuz çıkış kenarı sınırının aynısıdır; satır en küçükleri tam olarak `enKucuk` değerleridir. Sütun adımı yeni bilgi katar. Satır indirgemesinden sonra sütun 3'te hiç 0 yoktur: her şehrin en ucuz çıkışı 3'e gitmez. Oysa bir turda 3'e mutlaka girilir ve bu giriş en az 2 tutar. Sütun adımı “her şehre bir kez girilir” kısıtını da sınıra katar ve sınırı 14'e çıkarır. Tablo 10.13 indirgenmiş matrisin son hâlidir.

	0	**1**	**2**	**3**
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4

	0	1	2	3
3	10	0	3	∞

Tablo 10.13: İndirgenmiş matris (çıkarılan toplam 14). Her satırda ve sütunda en az bir 0 vardır.

Bu yöntemle dallanma bir kenar seçerek yapılır. Ebeveyn düğümün indirgenmiş matrisinde $i \to j$ kenarı seçilsin. Çocuğun matrisi şöyle kurulur. i satırının ve j sütununun bütün hücreleri ∞ yapılır: i'den başka bir çıkış ve j'ye başka bir giriş olamaz. *(j, i)* hücresi de ∞ yapılır; j'den hemen i'ye dönmek erken kapanan bir döngü oluşturur. Matris yeniden indirgenir. Çocuğun sınırı şöyle hesaplanır:

> *sınır(çocuk) = sınır(ebeveyn) + indirgenmiş c[i][j] + yeni indirgemenin toplamı*

Örnek 10.4

Kökün sınırı 14'tür. Kökten üç dal açılır:

1. **0 → 1.** İndirgenmiş *c[0][1] = 1*. Satır 0, sütun 1 ve *(1, 0)* ∞ yapılır. Kalan satırlar: satır 1 = ∞, ·, 18, 0 (en küçük 0); satır 2 = 0, ·, ∞, 4 (0); satır 3 = 10, ·, 3, ∞ (3). Satır 3'ten 3 çıkarılır: 7, ·, 0, ∞. Sütunların hepsinde 0 vardır. Yeni indirgeme 3'tür. Sınır *14 + 1 + 3 = 18*.
2. **0 → 2.** İndirgenmiş *c[0][2] = 0*. Satır 0, sütun 2 ve *(2, 0)* ∞ yapılır. Satır 2 = ∞, 19, ·, 4; en küçüğü 4'tür ve çıkarılır. Öbür satır ve sütunlarda 0 vardır. Sınır *14 + 0 + 4 = 18*.
3. **0 → 3.** İndirgenmiş *c[0][3] = 11*. Satır 0, sütun 3 ve *(3, 0)* ∞ yapılır. Satırların hepsinde 0 kalır. Sütun 2 = 18, ∞, 3 (satır 0 hariç); en küçüğü 3'tür. Sınır *14 + 11 + 3 = 28*.

0 → 3 dalının sınırı 28'dir. Optimum 18 bulunduğu anda bu dal budanır. 0 → 1 ve 0 → 2 dallarının sınırı tam 18'dir ve her ikisi de en kısa turu içerir: 0 → 1 → 3 → 2 → 0 ve onun ters yönü 0 → 2 → 3 → 1 → 0.

İndirgenmiş matris yöntemi, Little ve arkadaşlarının 1963'te yayımladığı makaleye dayanır. Yöntemin önemli bir üstünlüğü vardır: simetri gerektirmez. *d[i][j] ≠ d[j][i]* olan asimetrik problemlerde de geçerlidir. Tek yönlü yollar ya da yokuş yukarı ve aşağı farklı yakıt harcayan araçlar asimetrik problemlere örnektir. İki en kısa kenar sınırı ise simetriye dayanır; asimetrik problemde doğrudan kullanılamaz.

10.4.6 Üç sınırın karşılaştırması

Aynı dört şehirli örnekte üç alt sınır kökte üç farklı değer verir (Şekil 10.14).



Şekil 10.14: Dört şehirli örnekte kökün üç alt sınırı ve optimum: en ucuz çıkış kenarı 12, indirgenmiş matris 14, iki en kısa kenar 18, optimum tur 18.

Sınır	Düğüm başına maliyet	Neyi kullanır?	Not
En ucuz çıkış kenarı	$O(n)$ (`enKucuk[]` önceden hesaplanır)	her şehirden bir çıkış	en ucuz, en gevşek
İndirgenmiş matris	$O(n^2)$	her şehre bir giriş + bir çıkış	asimetrik problemde de geçerli
İki en kısa kenar	$O(n^2)$	her şehre değen iki kenar	simetrik problem için

Tablo 10.14: Üç TSP alt sınırının sıkılığı ve maliyeti.

Sıkı sınır daha çok budar ama her düğümde daha pahalıdır. En ucuz çıkış sınırı `enKucuk` dizisi sayesinde $O(n)$'de hesaplanır. Öbür ikisi her düğümde matrisin tamamına bakar. Sıralama her örnekte aynı olmak zorunda değildir. İndirgenmiş matris sınırı ile iki en kısa kenar sınırı farklı bilgileri kullanır ve başka bir örnekte sıraları değişebilir. En ucuz çıkış sınırı ise ikisinden de büyük olamaz. İndirgemenin satır adımı tek başına onu verir. İki kenar sınırında da her şehrin iki kenarının ortalaması en ucuz kenarından küçük değildir.

Büyük örneklerde daha da sıkı sınırlar kullanılır. **1-ağaç** sınırı, bir şehir dışarıda bırakılarak kalanlar için minimum kapsayan ağaç kurar ve dışarıdaki şehri en kısa iki kenarıyla bağlar. Her tur bir 1-ağaçtır; bu yüzden en küçük 1-ağaç geçerli bir alt sınırdır. Doğrusal programlama gevşetmesi ise bugün binlerce şehirli örnekleri kesin çözen yazılımların temelidir. Hangi sınırın seçileceği, budanan düğüm sayısı ile düğüm başına yapılan iş arasındaki dengeye göre belirlenir.

10.4.7 Daha büyük örnek: 6 şehir

Dört şehirli örnekte budamanın etkisi sınırlıdır; tam ağaç zaten küçüktür. Etkiyi görmek için altı şehirli simetrik bir örnek alınır (Tablo 10.15).

	0	1	2	3	4	5	en küçük
0	—	29	20	21	16	31	16
1	29	—	15	29	28	40	15
2	20	15	—	15	14	25	14
3	21	29	15	—	12	18	12

	0	1	2	3	4	5	en küçük
4	16	28	14	12	—	22	12
5	31	40	25	18	22	—	18

Tablo 10.15: Altı şehirli simetrik uzaklık matrisi.

Kökün en ucuz çıkış sınırı *16 + 15 + 14 + 12 + 12 + 18 = 87*'dir. En kısa tur 0 → 1 → 2 → 3 → 5 → 4 → 0'dır: *29 + 15 + 15 + 18 + 22 + 16 = 115*. Aynı uzunlukta başka turlar da vardır; arama ilk bulduğunu saklar. Kök sınırı ile optimum arasındaki fark büyüktür (87 ile 115); en ucuz çıkış sınırı bu örnekte gevşektir.

Sürüm	Üretilen düğüm	`tsp` çağrısı
Budamasız (geri izleme)	326	326
En ucuz çıkış sınırı	210	130

Tablo 10.16: Altı şehirli örnekte iş miktarı.

Budamasız arama, durum uzayı ağacının bütün düğümlerini üretir: *1 + 5 + 20 + 60 + 120 + 120 = 326*. Son düzeydeki 120 düğüm, *(6 − 1)! = 120* turun her biridir. Sınırlı arama 210 düğüm üretir; yani 210 kez `altSinir` hesaplanır (kök dahil). Bunların yalnız 130'unda dala girilir ve `tsp` çağrılır. Çağrı sayısı 326'dan 130'a, yani yaklaşık beşte ikisine iner. Daha sıkı bir sınır bu sayıları belirgin biçimde düşürür. Örneğin iki en kısa kenar sınırı bu matriste kökte 100,5 verir; 87'den çok daha yakındır.

10.5 Atama problemi

Atama problemi, dal sınırın en küçükleme yönünü küçük ve kolay izlenen bir örnekte göstermek için uygundur. Kod yapısı gezgin satıcı aramasına çok benzer; sınır ise daha da basittir.

10.5.1 Problem ve alt sınır

> **Tanım 10.4**
>
> **Atama problemi** (*assignment problem*): *n* kişi ve *n* iş verilir; *c[i][j]*, *i.* kişinin *j.* işi yapma maliyetidir. Her kişiye farklı bir iş verilerek toplam maliyet en küçüklenir.

Her atama, işlerin bir permütasyonudur; *n!* olası atama vardır. Durum uzayı ağacının *i.* düzeyinde *i.* kişiye henüz kimseye verilmemiş bir iş seçilir. Kök hiçbir kişiye iş verilmemiş durumdur; yapraklar tam atamalardır.

Alt sınır şu gözleme dayanır: henüz iş verilmemiş her kişi, en az kendi satırının en küçük maliyetini öder. Bu bir gevşetmedir. “Her işe tek bir kişi” kısıtı kaldırılmıştır; iki kişi aynı ucuz işi seçebilir. İş çakışmaları yok sayıldığı için sınır gerçek maliyetin üstüne çıkamaz. Sınırı her düğümde hızlı hesaplamak için satır en küçüklerinin sonek toplamları önceden tutulur:

> $$kalanMin[i] = \Sigma_{r \geq i}\, min_j\, c[r][j], \quad kalanMin[n] = 0$$

i. kişiye iş verildikten sonra yolun maliyeti `yeni` ise o düğümün alt sınırı *yeni + kalanMin[i + 1]*'dir. Hesap *O(1)* sürer. *alt ≥ enIyi* ise dal kesilir.

kişi \ iş	0	1	2	en küçük
0	5	8	4	4
1	7	3	6	3
2	2	9	5	2

Tablo 10.17: Üç kişilik atama örneği (satır: kişi, sütun: iş).

Örnekte satır en küçükleri 4, 3, 2'dir. Sonek toplamları *kalanMin = [9, 5, 2, 0]* olur. Kökün alt sınırı 9'dur: hangi atama yapılırsa yapılsın toplam 9'un altına inemez.

Atama probleminin polinom zamanlı bir algoritması da vardır: **Macar yöntemi** (*Hungarian method*) problemi $O(n^3)$ sürede çözer. Bu yüzden saf atama problemi için dal sınır gerekmez; burada yöntemi göstermek için kullanılır. Ek kısıtlı türevlerde ise durum değişir. Örneğin bazı işler aynı kişiye ya da birbirine bağlı iki kişiye verilemiyorsa Macar yöntemi doğrudan uygulanamaz. Bu tür problemlerde dal sınır gerçekten gerekir. Dallanmada Macar yönteminin sonucu da alt sınır olarak kullanılabilir.

10.5.2 Kod izleme: ata(0, 0)

```
void ata(int i, int maliyet) {
  if (i == n) {                     // tam atama
    enIyi = Math.min(enIyi, maliyet);
    return;
  }
  for (int j = 0; j < n; j++) {
    if (kullanildi[j]) continue;
    int yeni = maliyet + c[i][j];
    if (yeni + kalanMin[i + 1] >= enIyi)
      continue;                     // sınır
    kullanildi[j] = true;
    ata(i + 1, yeni);
    kullanildi[j] = false;          // geri al
  }
}
```

Metodun yapısı `tsp` metoduyla aynıdır. `i` sıradaki kişi, `maliyet` o ana kadar yapılan atamaların toplamıdır. `kullanildi` dizisi verilmiş işleri işaretler; olursuzluk kuralı budur. Sınır testi kısmi atamaya iş eklemeden önce yapılır. Dal kesilirse `kullanildi` hiç değişmez; bu yüzden geri alma gerekmez. Taban durumda `Math.min` ile `enIyi` güncellenir.

Adım	Atama	`yeni`	alt	`enIyi`	Karar
1	k0 → iş 0	5	*5 + 5 = 10*	∞	dallanılır
2	k1 → iş 1 (iş 0 dolu)	8	*8 + 2 = 10*	∞	dallanılır
3	k2 → iş 2 (tek boş iş)	13	13	13	tam atama
4	k1 → iş 2	11	*11 + 2 = 13*	13	kesilir
5	k0 → iş 1	8	*8 + 5 = 13*	13	kesilir
6	k0 → iş 2	4	*4 + 5 = 9*	13	dallanılır
7	k1 → iş 0	11	*11 + 2 = 13*	13	kesilir
8	k1 → iş 1	7	*7 + 2 = 9*	13	dallanılır
9	k2 → iş 0	9	9	9	tam atama

Tablo 10.18: `ata(0, 0)` izlemesi (*kalanMin = [9, 5, 2, 0]*, başta `enIyi` = ∞).

İzlemenin özü şudur. İlk tam atama (iş 0, 1, 2) 13 tutar ve `enIyi` olur. Ardından üç dal, sınırı 13'e eşit olduğu için kesilir. 5. adımdaki kesme önemlidir: k0 → iş 1 dalının altında iki tam atama vardır ve hiçbiri denenmez. 6. adımda k0 → iş 2 dalı sınırı 9 ile açılır. Bu dalda k1 → iş 1 ve k2 → iş 0 atamaları *4 + 3 + 2 = 9* verir. Sonuç kökün alt sınırına eşittir; daha iyi bir atama olamayacağı bu eşitlikten hemen anlaşılır.



Şekil 10.15: Aramanın sonunda seçilen hücreler: k0 → iş 2 (4), k1 → iş 1 (3), k2 → iş 0 (2). Toplam 9, kökün alt sınırına eşittir.

Arama toplam 7 çağrı yapmıştır: `ata(0, 0)`, dört ara düğüm ve iki tam atama ile birlikte. Budamasız arama *3! = 6* atamanın hepsini gezer; bunun için *1 + 3 + 6 + 6 = 16* çağrı gerekir. Dört kişilik bir örnekte fark daha büyüktür. Tablo 10.19'daki matriste satır en küçükleri 3, 2, 3, 2'dir ve kökün alt sınırı 10'dur.

kişi \ iş	0	1	2	3
0	7	3	8	4

kişi \ iş	0	1	2	3
1	5	6	2	9
2	4	8	5	3
3	6	2	7	5

Tablo 10.19: Dört kişilik atama örneği.

En iyi atama k0 → iş 3 (4), k1 → iş 2 (2), k2 → iş 0 (4), k3 → iş 1 (2) olup toplamı 12'dir. Dal sınır 21 çağrıda yalnız 4 tam atamaya ulaşır. Budamasız arama *4! = 24* tam atamanın hepsini dener ve 65 çağrı yapar.

10.6 Tam sayılı doğrusal programlama

Dal sınırın endüstrideki en yaygın kullanımı tam sayılı doğrusal programlamadır. Yöntem bu alanda 1960'ta Land ve Doig'in makalesiyle ortaya konmuştur. Bugün üretim planlama, çizelgeleme ve dağıtım problemlerini çözen ticari yazılımlar aynı fikrin gelişmiş sürümlerini kullanır.

10.6.1 Problem ve LP gevşetmesi

> **Tanım 10.5**
>
> **Tam sayılı doğrusal programlama** (*integer linear programming*, ILP), doğrusal bir amaç fonksiyonunun doğrusal eşitsizlik kısıtları altında en büyüklendiği ya da en küçüklendiği problemdir. Bütün değişkenlerin tam sayı olması istenir.

Tam sayı koşulu olmasaydı problem sıradan bir **doğrusal program** (*linear program*, LP) olurdu. Doğrusal programlar simpleks yöntemiyle pratikte çok hızlı, iç nokta yöntemleriyle de polinom sürede çözülür. İki değişkenli küçük örneklerde çözüm grafikle bulunur: kısıtlar düzlemde bir çokgen (olurlu bölge) tanımlar ve en iyi değer bu çokgenin köşelerinden birinde elde edilir. Tam sayı koşulu ise problemi NP-zor yapar.

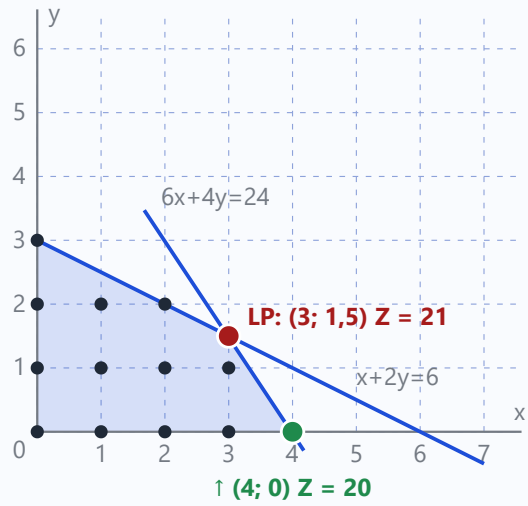
Dal sınır bu iki gerçeği birleştirir:

- **Sınır:** Tam sayı koşulu kaldırılır ve **LP gevşetmesi** (*LP relaxation*) çözülür. Gevşetmenin olurlu bölgesi bütün tam sayı noktaları içerir. Bu yüzden en büyüklemede LP optimumu, ILP optimumu için bir üst sınırdır.
- **Dallanma:** LP çözümünde bir değişken kesirliyse, örneğin *y = 1,5* ise, iki alt problem kurulur: birine *y ≤ 1*, öbürüne *y ≥ 2* kısıtı eklenir. Aradaki kesirli bölge *1 < y < 2* atılır. Bu bölgede tam sayı nokta olmadığı için hiçbir tam sayı çözüm kaybolmaz.

- **Tamlık:** Bir düğümün LP çözümü zaten tam sayıysa o düğümün altında daha iyi bir tam sayı çözüm yoktur. Bu çözüm bir aday olur ve düğüm kapanır.

Örnek problem şudur:

> en büyük $Z = 5x + 4y$; $6x + 4y \le 24$, $x + 2y \le 6$, $x, y \ge 0$ tam sayı



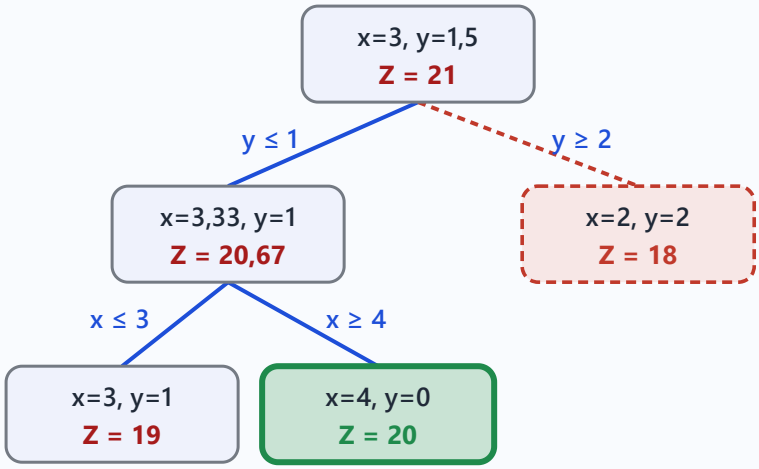
Şekil 10.16: Gölgeli alan LP gevşetmesinin olurlu bölgesidir; noktalar tam sayı çözümlerdir. LP optimumu (3; 1,5) noktasındadır ve $Z = 21$'dir. Tam sayı optimum (4; 0) noktasındadır ve $Z = 20$'dir.

LP gevşetmesinin optimumu iki kısıt doğrusunun kesişimindedir. $6x + 4y = 24$ ve $x + 2y = 6$ birlikte çözülür. İkinci denklem 2 ile çarpılıp birinciden çıkarılınca $4x = 12$, yani $x = 3$ bulunur; buradan $y = 1{,}5$ çıkar. Değer $Z = 15 + 6 = 21$'dir. Öbür köşeler (0; 0), (4; 0) ve (0; 3)'tür; değerleri 0, 20 ve 12'dir. Şekil 10.16 bölgeyi ve iki optimumu gösterir.

10.6.2 LP gevşetmeleriyle dallanma

Dal sınır ağacı adım adım şöyle kurulur:

1. **Kök.** LP optimumu (3; 1,5), $Z = 21$. Bu değer üst sınırdır: hiçbir tam sayı çözüm 21'i geçemez. y kesirlidir; y üzerinde dallanılır.
2. **Dal $y \le 1$.** Yeni LP'de $y = 1$ sınırına dayanılır. $6x + 4 \le 24$'ten $x \le 3{,}33$, $x + 2 \le 6$'dan $x \le 4$ gelir. LP optimumu (3,33; 1), $Z = 20{,}67$'dir. x kesirlidir; $x \le 3$ ve $x \ge 4$ dalları açılır.
3. **Dal $y \le 1$, $x \le 3$.** LP optimumu (3; 1), $Z = 19$. Çözüm tam sayıdır: ilk aday, `enIyi` = 19.
4. **Dal $y \le 1$, $x \ge 4$.** $6 \cdot 4 + 4y \le 24$ kısıtı $y = 0$'ı zorlar. LP optimumu (4; 0), $Z = 20$. Tam sayıdır ve $20 > 19$: `enIyi` = 20.
5. **Dal $y \ge 2$.** $x + 2y \le 6$ kısıtı $x \le 2$'yi zorlar. LP optimumu (2; 2), $Z = 18$. $18 \le 20$ olduğu için dal budanır. Bu dalın LP çözümü tam sayı olsa bile `enIyi`'yi geçemez.



Şekil 10.17: ILP dal sınır ağacı. Her düğümde LP gevşetmesinin çözümü ve değeri yazılıdır. $y \geq 2$ dalı $Z = 18 \leq 20$ ile budanır. Yeşil düğüm tam sayı optimumdur: $x = 4, y = 0, Z = 20$.

Optimum $x = 4, y = 0, Z = 20$'dir. Bu örnek, LP çözümünü yuvarlamanın neden yetmediğini de gösterir. (3; 1,5) aşağı yuvarlanırsa (3; 1) elde edilir ve $Z = 19$ olur; en iyi değildir. Yukarı yuvarlanırsa (3; 2) elde edilir: $6 \cdot 3 + 4 \cdot 2 = 26 > 24$, olursuzdur. Gerçek optimum (4; 0) LP noktasının yakınında değildir; ancak dallanmayla bulunur.

Sık yapılan hata

Kesirli değişkende yanlış dallanmak. $y = 2{,}6$ için doğru alt problemler $y \leq 2$ ve $y \geq 3$'tür. $y = 2$ ve $y = 3$ diye dallanmak öbür bütün y değerlerini kaybeder. $y \leq 2{,}6$ ve $y \geq 2{,}6$ diye dallanmak, var olan LP çözümünü iki alt problemde de bırakır ve arama ilerlemez. $y \leq 3$ ve $y \geq 2$ ise örtüşen iki alt problem verir; aynı noktalar iki kez aranır.

İkinci bir örnek tamlık kuralını gösterir: en büyük $Z = 3x + 2y$; $x + y \leq 4, x \leq 3, y \leq 2$. Kaba kuvvet $0 \leq x \leq 3, 0 \leq y \leq 2$ ızgarasındaki 12 noktanın hepsini dener ve (3; 1) noktasında $Z = 11$'i bulur. LP gevşetmesinin optimumu da (3; 1)'dir: x sınırına (3) ve $x + y \leq 4$ doğrusuna dayanan köşe. Çözüm zaten tam sayı olduğu için dal sınır kökte biter; tek bir LP çözülmüştür.

Gerçek yazılımlar bu temel şemayı birkaç yönde geliştirir. Hangi kesirli değişkende dallanılacağı akıllıca seçilir. LP gevşetmesine, tam sayı noktaları kesmeden kesirli çözümü dışarıda bırakan ek kısıtlar (kesme düzlemleri) eklenir. Bu birleşime dal ve kesme (*branch and cut*) denir. Ayrıca bir alt problemin LP'si, ebeveynin çözümünden başlanarak birkaç adımda yeniden çözülür.

10.7 Maksimum klik

Son uygulama bir çizge problemidir. Klik problemi sosyal ağlarda sıkı bağlı grupları, biyolojide birbirleriyle etkileşen protein kümelerini ve kodlama kuramında hata düzelten kodları bulmakta kullanılır.

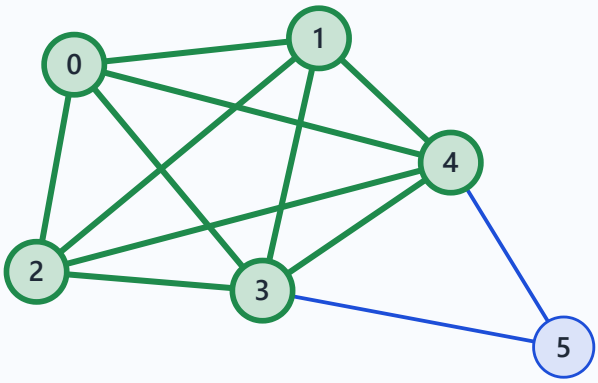
10.7.1 Problem ve sınır

Tanım 10.6

Yönsüz bir çizgede her iki düğümü birbirine komşu olan düğüm kümesine **klik** (*clique*) denir. **Maksimum klik** (*maximum clique*) problemi, en çok düğüm içeren kliği bulma problemidir.

Problem NP-zordur. Arama, mevcut kliği K adım adım büyütür. K'ye eklenebilecek düğümlere **aday** denir. Bir düğüm aday olabilmek için K'deki bütün düğümlere komşu olmalıdır. Aynı kliğin farklı sıralarla tekrar tekrar üretilmesini önlemek için düğümler artan numara sırasıyla eklenir: K'ye son eklenen düğümden büyük numaralı düğümler aday olabilir.

Sınır en büyükleme sınırıdır ve çok basittir. K en iyi durumda bütün adaylarla büyür. Bu yüzden bu dalda bulunabilecek en büyük klik $|K| + |aday|$ düğümü geçemez. $|K| + |aday| \le |enİyi|$ ise mevcut klik ne kadar büyütülse de `enIyi` 'yi geçemez; dal budanır. Sınır geçerlidir, çünkü dal içindeki her klik K ile adayların bir alt kümesinin birleşimidir.



Şekil 10.18: Altı düğümlü örnek çizge. 0, 1, 2, 3 ve 4 numaralı düğümlerin her çifti komşudur; en büyük klik {0, 1, 2, 3, 4}'tür. Düğüm 5 yalnız 3 ve 4'e komşudur.

```
void klik(List<Integer> K, int bas) {
  if (K.size() > enIyi.size())
    enIyi = new ArrayList<>(K);
  List<Integer> aday = new ArrayList<>();
  for (int v = bas; v < n; v++)
    if (uygun(v, K)) aday.add(v);  // K'ye komşu
  if (K.size() + aday.size() <= enIyi.size())
    return;                        // sınır
  for (int v : aday) {
    K.add(v);
    klik(K, v + 1);
    K.remove(K.size() - 1);        // geri al
  }
}
```

Metodun satırları şöyle çalışır:

- İlk iki satır `enIyi`'yi günceller. Bir klik her an olurlu bir çözümdür; yaprağa inmek gerekmez. `new ArrayList<>(K)` bir kopya üretir, çünkü `K` aramanın geri kalanında değişir.
- `for` döngüsü adayları toplar. `uygun(v, K)`, v'nin `K`'deki her düğüme komşu olup olmadığına bakar. Döngü `bas`'tan başlar; daha küçük numaralar zaten denenmiştir.
- `if (K.size() + aday.size() <= enIyi.size()) return;` sınır kuralıdır.
- Son döngü dallanmadır. Her aday sırayla `K`'ye eklenir, arama `v + 1`'den sürer ve aday geri çıkarılır.

Bir çağrıda aday listesi $O(n \cdot |K|)$ sürede kurulur; komşuluk bir matriste tutulursa her `uygun` denetimi $O(|K|)$'dir. Sınır satırı olmadan metot sıradan bir geri izlemedir ve bütün klikleri gezer.

10.7.2 Kod izleme: klik({}, 0)

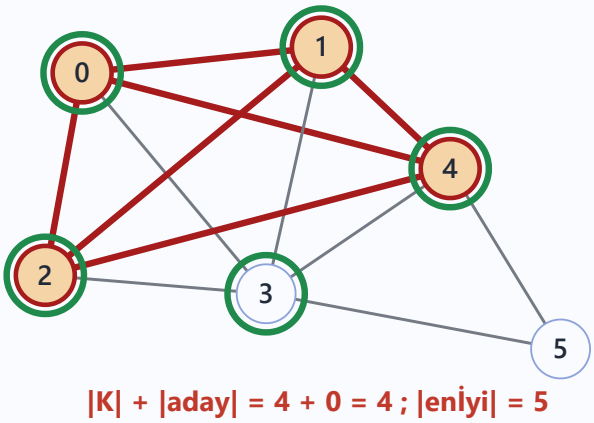
Arama boş klik ve `bas` = 0 ile başlar. Tablo 10.20 çağrıları sırayla verir.

Çağrı	K	Adaylar	$\lvert K\rvert + \lvert aday\rvert$	`enIyi`	Karar
1	{}	{0, 1, 2, 3, 4, 5}	0 + 6	{}	devam
2	{0}	{1, 2, 3, 4}	1 + 4	{0}	devam
3	{0, 1}	{2, 3, 4}	2 + 3	{0, 1}	devam
4	{0, 1, 2}	{3, 4}	3 + 2	{0, 1, 2}	devam
5	{0, 1, 2, 3}	{4}	4 + 1	{0, 1, 2, 3}	devam
6	{0, 1, 2, 3, 4}	{}	5 + 0	{0, 1, 2, 3, 4}	aday yok
7	{0, 1, 2, 4}	{}	4 + 0 ≤ 5	5 düğüm	budanır
8	{0, 1, 3}	{4}	3 + 1 ≤ 5	5 düğüm	budanır
9	{0, 1, 4}	{}	3 + 0 ≤ 5	5 düğüm	budanır
10–12	{0, 2}, {0, 3}, {0, 4}	{3, 4}, {4}, {}	2 + 2, 2 + 1, 2 + 0	5 düğüm	budanır

Çağrı	*K*	Adaylar	*|K| + |aday|*	`enIyi`	Karar
13–17	{1}, {2}, {3}, {4}, {5}	{2, 3, 4}, {3, 4}, {4, 5}, {5}, {}	4, 3, 3, 2, 1	5 düğüm	budanır

Tablo 10.20: `klik({}, 0)` izlemesi.

İlk beş çağrı hep ilk adayı ekleyerek derine iner ve 6. çağrıda beş düğümlü kliğe ulaşır. Bu, çizgenin en büyük kliğidir. Bundan sonra açılan her dalın sınırı en fazla 5'tir ve hepsi hemen budanır. Örneğin 8. çağrıda *K = {0, 1, 3}*'tür; aday yalnız 4'tür. En iyi durumda dört düğümlü bir klik elde edilir ve bu 5'i geçemez. Kökte kalan adaylar 1–5 de tek tek denenir ve hepsi aynı biçimde kesilir. Örneğin *K = {3}* iken adaylar 4 ve 5'tir. 3, 4, 5 bir üçgen oluşturur ama sınır *1 + 2 = 3 ≤ 5*'tir.



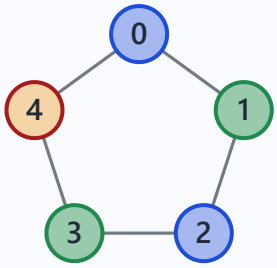
Şekil 10.19: 7. çağrı: *K = {0, 1, 2, 4}* (turuncu), aday yok. Yeşil halkalar en iyi kliği (5 düğüm) gösterir. *|K| + |aday| = 4 + 0 ≤ 5*; dal budanır.

Sonuç {0, 1, 2, 3, 4}'tür. Sınırlı arama 17 çağrı yapar. Sınır satırı silinince arama bütün klikleri gezer ve 36 çağrı yapar. Bu sayı, boş küme dahil çizgedeki kliklerin sayısıdır: tam beşli kısımda $2^5 = 32$ klik, düğüm 5'i içeren {5}, {3, 5}, {4, 5}, {3, 4, 5} dört klik.

10.7.3 Daha sıkı sınır: boyama

|K| + |aday| sınırı her adayın kliğe girebileceğini varsayar. Adaylar arasında komşu olmayan çiftler varsa bu varsayım çok iyimserdir. Daha sıkı bir sınır **çizge boyamadan** gelir. Bir kliğin her düğüm çifti komşudur; bu yüzden bir klikteki düğümlerin hepsi farklı renk almak zorundadır. Adaylar *k* renkle düzgün boyanabiliyorsa, adaylardan en çok *k* düğüm aynı kliğe girebilir. Yeni sınır şöyledir: *|K| + renk(aday) ≤ |enİyi|* ise dal budanır.

En az renk sayısını bulmak da NP-zordur. Ama en iyi boyama gerekmez. Açgözlü boyama düğümleri sırayla gezer ve her düğüme komşularında kullanılmamış en küçük rengi verir. $O(aday^2)$ sürer. Fazla renk kullanabilir; bu yalnız sınırı gevşetir, geçersiz kılmaz. Sınır yine de *|aday|*'dan büyük olamaz, çünkü her düğüm en kötü durumda kendi rengini alır.



3 renk:
{0, 2} {1, 3} {4}

|aday| = 5
renk sınırı = 3

Şekil 10.20: Beş adaylı bir halka. Açgözlü boyama 3 renk kullanır: {0, 2}, {1, 3}, {4}. *|aday| = 5*, renk sınırı 3'tür.

Şekil 10.20'deki örnekte adaylar beş düğümlü bir halka oluşturur; *K* boştur ve *|enİyi| = 3*'tür. Eski sınır *0 + 5 = 5 > 3* verir ve dal açılır. Açgözlü boyama 0'a renk 1, 1'e renk 2, 2'ye renk 1, 3'e renk 2 verir. 4 hem 3'e hem 0'a komşudur; renk 3'ü alır. Renk sınırı *0 + 3 ≤ 3* verir ve dal budanır. Halkadaki en büyük klik zaten yalnız 2 düğümlüdür. En hızlı kesin klik algoritmalarının çoğu bu fikre dayanır. Tomita'nın MCQ ve MCS algoritmaları adayları renk sırasıyla dener ve renk sınırıyla budar.

10.8 Diğer problemler ve A* ile ilişki

10.8.1 Dal sınırla çözülen diğer problemler

Yöntem, çözümü bir dizi kararla kurulan hemen her optimizasyon problemine uygulanabilir. Her uygulamada sorulacak iki soru aynıdır: dallanma nasıl yapılır ve hangi gevşetme geçerli bir sınır verir?

- **Hamilton yolu.** Her düğümü tam bir kez ziyaret eden yol aranır. Ağırlıklı sürümü gezgin satıcıya çok benzer ve benzer sınırlarla çözülür.
- **Mantıksal doyurulabilirlik** (SAT). Değişkenlere doğruluk değeri atanır; çelişen kısmi atamalar budanır. Bu, DPLL algoritmasının özüdür. En büyük SAT sürümünde, mümkün olduğu kadar çok tümcenin doğru olması istenir; sınır, kalan atamalarla sağlanabilecek en fazla tümce sayısıdır.
- **Çizelgeleme.** İşler makinelere dağıtılır; en büyük yük (*makespan*) ya da toplam gecikme en küçüklenir. Alt sınır, kalan işlerin en iyi durumda yaratacağı yüktür; örneğin kalan iş toplamının makinelere eşit bölünmesi.
- **Kutu paketleme.** Eşyalar en az sayıda kutuya yerleştirilir. Alt sınır *⌈Σ boyut / kutu kapasitesi⌉*'dir: kutular kusursuz doldurulsa bile bu kadar kutu gerekir.
- **Maksimum kapsama.** En fazla öğeyi kapsayan *k* küme seçilir. Üst sınır, kalan en büyük kümelerin ekleyebileceği öğe sayısıdır.
- **Atama problemi** ve ek kısıtlı türevleri. Alt sınır, atanmamış her satırın en küçük maliyetidir.

Dal sınır NP-zor problemler için anlamlıdır. Maksimum akış, en kısa yol ya da minimum kapsayan ağaç gibi polinom zamanlı algoritması bilinen problemlerde dal sınıra gerek yoktur. Bu problemlerde polinom algoritma her zaman tercih edilir; dal sınır en kötü durumda üstel sürer.

10.8.2 Dal sınır, A* ve derinlik öncelikli dal sınır

En iyi önce dal sınır ile yol bulmadaki A* algoritması aynı fikrin iki yüzüdür. Tablo 10.21 üç yakın yöntemi karşılaştırır.

Özellik	En iyi önce dal sınır (LC)	A*	Derinlik öncelikli dal sınır
Seçim	en iyi sınır ($\hat{c}$)	en küçük $f = g + h$	yığın ya da özyineleme
Sınır / tahmin	geçerli (iyimser) sınır	kabul edilebilir h (fazla tahmin etmez)	geçerli sınır + `enIyi`
Hedef	optimizasyon (en iyi yaprak)	başlangıçtan hedefe en kısa yol	optimizasyon
Bellek	canlı düğümler: üstel olabilir	açık liste: üstel olabilir	*O(derinlik)*
İlk tam çözüm	genellikle geç ama iyi	bulunan ilk hedef optimumdur	erken; her an elde bir çözüm var

Tablo 10.21: En iyi önce dal sınır, A* ve derinlik öncelikli dal sınır.

A*, ağırlıklı bir çizgede yol arayan LC dal sınırdır. $g(x)$ başlangıçtan x'e ödenen maliyettir; LC aramadaki $f(x)$'in karşılığıdır. $h(x)$ ise x'ten hedefe kalan maliyetin bir alt sınırıdır; $\hat{g}(x)$'in karşılığıdır. 8 bulmacasındaki yanlış taş sayısı böyle bir tahmindir. h kabul edilebilirse, kuyruktan çıkarılan ilk hedef düğüm en kısa yolu verir. Gerekçe dal sınırdakiyle aynıdır. O hedefin maliyeti, kuyruktaki her düğümün f değerinden küçük ya da ona eşittir. f değerleri ise o düğümlerden geçen her yolun alt sınırıdır.

Derinlik öncelikli dal sınır, belleğin kısıtlı olduğu durumların yöntemidir. Bellek derinlikle sınırlıdır ve aramanın her anında elde bir çözüm bulunur. Bedeli, kötü bir dala girildiğinde iyi çözümün geç bulunmasıdır. İkisinin arası da vardır. IDA* (*iterative deepening A**), derinlik öncelikli aramayı bir f eşiğiyle çalıştırır. Eşiği aşan düğümleri keser ve arama başarısız olursa eşiği yükseltip yeniden başlar. Bellek derinlikle sınırlı kalır, en iyi önce aramanın sırası da büyük ölçüde korunur.

10.9 Tasarım paradigmalarının karşılaştırması

Dal sınır, algoritma tasarım paradigmaları arasında en son ele alınanlardandır ve ötekilerin bazı fikirlerini birleştirir. Tablo 10.22 altı paradigmayı aynı ölçütlerle yan yana koyar.

Paradigma	Karar	Alt problemler / gezinme	Ne zaman?	Örnek
Kaba kuvvet	bütün adaylar üretilir, sonra sınanır	tam arama uzayı	uzay çok küçükse	8^8 vezir yerleşimi
Böl ve fethet	sabit bölme kuralı; böl, çöz, birleştir	bağımsız; yukarıdan aşağı	alt problemler örtüşmüyorsa	birleştirme sıralaması, Strassen
Açgözlü	o an en iyi görünen seçilir; geri dönülmez	seçimden sonra tek alt problem	açgözlü seçim özelliği kanıtlıysa	Dijkstra, Huffman, Kruskal
Dinamik programlama	bütün seçenekler denenir, en iyisi saklanır	örtüşen; çoğunlukla aşağıdan yukarı	optimal alt yapı, polinom sayıda alt problem	Fibonacci, 0/1 sırt çantası, en uzun ortak alt dizi

Paradigma	Karar	Alt problemler / gezinme	Ne zaman?	Örnek
Geri izleme	kısmi çözüm genişletilir; geçersizse geri alınır	durum uzayı ağacında derinlik öncelikli	kısıt sağlama, bütün çözümler	n-vezir, Sudoku
Dal sınır	geri izleme + sınırla budama	derinlik öncelikli, genişlik öncelikli ya da en iyi önce	NP-zor optimizasyon, kesin sonuç	0/1 sırt çantası, gezgin satıcı

Tablo 10.22: Tasarım paradigmalarının birikimli karşılaştırması.

Bir problem için paradigma seçerken sorular belli bir sırayla sorulur. Alt problemler bağımsız mı? Öyleyse böl ve fethet uygundur. Alt problemler örtüşüyor ve sayıları polinom mu? Dinamik programlama uygundur. Yerel en iyi seçim küresel en iyiye götürüyor ve bu kanıtlanabiliyor mu? Açgözlü yaklaşım uygundur. Bunların hiçbiri yoksa ve problem NP-zorsa iki seçenek kalır. Kesin sonuç isteniyorsa geri izleme ya da dal sınır kullanılır. Polinom sürede, kalitesi garantili ama yaklaşık bir sonuç yetiyorsa yaklaşım algoritmasına başvurulur.

Aynı problem birden çok paradigmayla çözülebilir. 0/1 sırt çantası buna iyi bir örnektir. Dinamik programlama onu $O(nW)$'de çözer; kapasite küçükse en iyi seçimdir. Kapasite çok büyükse ya da ağırlıklar kesirliyse dal sınır öne çıkar. Açgözlü yöntem ise 0/1 sürümde en iyiyi garanti etmez; yalnız kesirli sürümde doğrudur. Dal sınır bu açgözlü yöntemi bir sınır hesabı olarak içine alır. Paradigmalar birbirine rakip değildir; çoğu zaman birbirinin içinde parça olarak kullanılır.

10.10 Sık yapılan hatalar

- **Geçersiz (kötümser) sınır kullanmak.** Sırt çantasında yalnız tam sığan eşyaların kazancını toplamak bir üst sınır değildir; bir olurlu çözümün değeridir. Optimumu içeren dal budanabilir ve algoritma hata vermeden yanlış sonuç döndürür.
- **Yönü karıştırmak.** En büyüklemede *sınır ≤ enİyi*, en küçüklemede *alt ≥ enİyi* ise budanır. Tersini yazmak ya optimumu kaçırır ya da hiç budamaz. En küçüklemede `enIyi`'nin başta $+\infty$, en büyüklemede $-\infty$ ya da 0 olması gerektiği de unutulmamalıdır.
- **Kuyruktan çıkınca tekrar sınamamak.** Düğüm eklendiğinde umut vaat ediyordu. Beklerken `enIyi` büyüdüyse artık gereksizdir. Sırt çantası örneğinde C ve G böyle düğümlerdir. Tekrar sınama olmazsa sonuç değişmez ama gereksiz düğümler genişletilir.
- **Eşyaları sıralamamak.** Kesirli sınır, eşyalar p_i/w_i oranına göre azalan sırada değilse geçerli bir üst sınır vermeyebilir.
- **`enIyi`'yi yalnız yapraklarda güncellemek.** Sırt çantasında ve klikte her olurlu düğüm bir çözümdür. Erken güncelleme `enIyi`'yi hızla yükseltir ve daha çok budama sağlar.
- **Dal sınırı geri izlemeyle aynı sanmak.** Sınır testi içermeyen kod, örneğin budamasız TSP araması, dal sınır değildir; bütün ağacı gezer.
- **Geri almayı unutmak.** Özyinelemeli sürümlerde `ziyaret[s] = false`, `kullanildi[j] = false` ya da `K.remove(...)` satırı unutulursa sonraki dallar yanlış durumla çalışır ve olurlu çözümler kaybolur.

- `enIyi`**'yi kopyalamadan saklamak.** Klik aramasında `enIyi = K` yazmak, değişmeye devam eden listeyi saklar. Arama bittiğinde `enIyi` boş liste gösterir. Kopya (`new ArrayList<>(K)`, `yol.clone()`) şarttır.

10.11 Özet

- Dal sınır, optimizasyon problemlerinin çözüm uzayını bir durum uzayı ağacı olarak gezer. Döngü şöyledir: bir canlı düğüm seçilir, sınırı tekrar sınanır, düğüm dallanır, çocukların sınırı hesaplanır ve umutsuz olanlar budanır. `enIyi`, o ana kadar bulunan en iyi tam çözümün değerini tutar.
- Sınır geçerli (iyimser), sıkı ve ucuz olmalıdır. Geçerlilik zorunludur; sıkılık ve ucuzluk verimliliği belirler. Sınır çoğunlukla gevşetmeyle kurulur. En büyüklemede üst, en küçüklemede alt sınır kullanılır.
- Düğüm üç nedenle budanır: olursuzluk, yetersiz sınır (*sinır ≤ enİyi* ya da *alt ≥ enİyi*) ve tamlık (yaprak ya da tam sayı çıkan LP çözümü). Dal sınırı geri izlemeden ayıran sınır kuralıdır.
- FIFO genişlik öncelikli gezer ve kuyruk kullanır. LIFO derinlik öncelikli gezer, yığın ya da özyineleme kullanır ve az bellek ister. LC (en iyi önce) öncelikli kuyrukla en umut verici düğümü seçer ve çoğunlukla en az düğümü üretir. Üçü de aynı optimumu bulur.
- Sırt çantasında kesirli sırt çantası gevşetmesi üst sınır verir. Örnekte *sinir(kök) = 115*, optimum 90'dır. En iyi önce 31 düğümün 11'ini, FIFO ve LIFO 13'ünü üretir.
- Gezgin satıcıda en ucuz çıkış kenarı sınırı, yolun maliyetine son şehrin ve gidilmemiş şehirlerin en ucuz çıkışlarını ekler. Dört şehirde 16 düğüm yerine 10 düğüm üretilir. İndirgenmiş matris (14) ve iki en kısa kenar (18) daha sıkıdır ama düğüm başına $O(n^2)$ sürer.
- Atamada alt sınır, kalan kişilerin satır en küçüklerinin toplamıdır. Tam sayılı programlamada LP gevşetmesi üst sınır verir ve kesirli bir değişkende iki alt probleme dallanılır. Klikte *|K| + |aday| ≤ |enİyi|* sınırı kullanılır; boyama daha sıkı bir sınır verir.
- Yöntem en kötü durumda üsteldir. Pratikteki başarısı sınırın kalitesine bağlıdır. Polinom algoritması olan problemlerde kullanılmaz.
- Bağımsız alt problemler böl ve fethete, örtüşen alt problemler dinamik programlamaya, kanıtlı açgözlü seçim açgözlü yaklaşıma götürür. NP-zor bir problemde kesin sonuç için geri izleme ya da dal sınır kullanılır.

10.12 Kavram listesi

dal sınır *branch and bound* · dallanma *branching* · sınır *bound* · budama *pruning* · durum uzayı ağacı *state-space tree* · olurlu *feasible* · gevşetme *relaxation* · üst sınır *upper bound* · en iyi önce arama *best-first search* · en düşük maliyetli (LC) arama *least-cost search* · derinlik öncelikli dal sınır *depth-first branch and bound* · kabul edilebilir sezgisel *admissible heuristic*

A* algoritması *A* search* · geri izleme *backtracking* · 0/1 sırt çantası *0/1 knapsack* · kesirli sırt çantası *fractional knapsack* · gezgin satıcı problemi *travelling salesman problem* · indirgenmiş maliyet matrisi *reduced cost matrix* · atama problemi *assignment problem* · Macar yöntemi *Hungarian method* · tam sayılı doğrusal programlama *integer linear programming* · LP gevşetmesi *LP relaxation* · maksimum klik *maximum clique* · çizge boyama *graph coloring* · NP-zor *NP-hard*

10.13 Kendini Dene

1. Eşyalar $(p, w) = (30, 3), (40, 5), (45, 9), (10, 4)$ ve $W = 12$ olsun. Kökün kesirli üst sınırı nedir? En iyi önce dal sınır hangi düğümleri genişletir, kaç düğüm üretir ve optimum nedir?

   Cevap

   Oranlar 10, 8, 5, 2,5'tir; eşyalar sıralıdır. Kökte eşya 1 ve 2 tam sığar (70, ağırlık 8); eşya 3'ün $4/9$'u alınır: $4 \cdot 45/9 = 20$. *sinir(kök)* $= 90$. Kök genişletilir: B (eşya 1 alındı) 30/3, sınır 90, `enIyi` = 30; C (eşya 1 yok) sınır $40 + 7 \cdot 5 = 75$. B genişletilir: D 70/8, sınır 90, `enIyi` = 70; E 30/3, sınır 75. D genişletilir: "al" çocuğu 115/17 olursuzdur; G (eşya 3 yok) 70/8, sınır $70 + 10 = 80$. G genişletilir: H (eşya 4 alındı) 80/12, `enIyi` = 80; H'nin sınırı 80'dir ve kuyruğa girmez. I (eşya 4 yok) sınır 70, budanır. Kuyruktaki E (75) ve C (75) atılır. 4 genişletme (A, B, D, G), 9 düğüm. Optimum 80'dir: eşya 1, 2 ve 4.

2. Aynı örnek FIFO kuyrukla çözülürse kaç düğüm üretilir, kaç düğüm genişletilir? Fark nereden gelir?

   Cevap

   FIFO 13 düğüm üretir ve 6 düğüm genişletir: A, B, C, D, E ve G'nin karşılığı olan düğüm. Sonuç yine 80'dir. FIFO, C'yi (sınır 75) düzey sırasıyla erkenden genişletir; o anda `enIyi` yalnız 70'tir ve C atılamaz. E de genişletilir; çocuğu 75/12 olursuz değildir ve `enIyi` 'yi 75'e çıkarır. En iyi önce ise C ve E'yi sona bırakır; o sırada `enIyi` 80 olmuştur ve ikisi de atılır.

3. Simetrik uzaklıklar $d(0,1) = 10$, $d(0,2) = 15$, $d(0,3) = 20$, $d(1,2) = 35$, $d(1,3) = 25$, $d(2,3) = 30$ olsun. Kökte en ucuz çıkış kenarı, indirgenmiş matris ve iki en kısa kenar sınırları nedir? En kısa tur kaç

birimdir?

Cevap

En ucuz çıkışlar 10, 10, 15, 20'dir: sınır 55. İndirgemede satır en küçükleri *10 + 10 + 15 + 20 = 55*'tir. Satır indirgemesinden sonra sütun 2'de en küçük 5, sütun 3'te 10 kalır: sınır *55 + 5 + 10 = 70*. İki en kısa kenar toplamları 25, 35, 45, 45'tir: *150 / 2 = 75*. En kısa tur 0 → 1 → 3 → 2 → 0'dır: *10 + 25 + 30 + 15 = 80*. Sıra yine en ucuz çıkış ≤ indirgenmiş matris ≤ iki kenar ≤ optimumdur. En ucuz çıkış sınırıyla `tsp` araması bu örnekte 16 düğümün 15'ini üretir; gevşek sınır neredeyse hiç budamaz.

4. Maliyet matrisi [[4, 7, 3], [2, 6, 1], [3, 9, 4]] olan atama problemi için `kalanMin` nedir? Optimum atama nedir ve `ata(0, 0)` kaç çağrı yapar?

Cevap

Satır en küçükleri 3, 1, 3'tür: *kalanMin = [7, 4, 3, 0]*; kökün alt sınırı 7'dir. Arama sırayla 14, 13 ve 11 maliyetli tam atamalar bulur. Optimum k0 → iş 1 (7), k1 → iş 2 (1), k2 → iş 0 (3) olup toplamı 11'dir. Arama 12 çağrı yapar; budamasız arama 16 çağrı yapar. Bu örnekte ilk dal (k0 → iş 0) kötü olduğu için budama geç başlar.

5. En büyük $Z = 3x + 5y$; $x + 2y \le 7$, $3x + y \le 9$, $x, y \ge 0$ tam sayı. LP gevşetmeleriyle dal sınır uygulanınca (önce y, sonra x kesirliyse onda dallanılarak) ağaç nasıl oluşur?

Cevap

Kök: iki doğrunun kesişimi (2,2; 2,4), $Z = 18{,}6$; y kesirlidir. Dal $y \le 2$: (2,33; 2), $Z = 17$; x kesirlidir. Onun altında $x \le 2$: (2; 2), $Z = 16$, tam sayı, `enIyi` = 16. $x \ge 3$: (3; 0), $Z = 9 \le 16$, budanır. Dal $y \ge 3$: (1; 3), $Z = 18$, tam sayı, `enIyi` = 18. Optimum $x = 1$, $y = 3$, $Z = 18$'dir. LP çözümünü aşağı yuvarlamak (2; 2) ve 16 verirdi.

6. Kenarları 0–1, 0–2, 0–3, 1–2, 1–3, 2–3, 3–4, 3–5, 4–5 olan çizgede en büyük klik nedir? `klik` metodu sınırla ve sınırsız kaç çağrı yapar?

Cevap

En büyük klik {0, 1, 2, 3}'tür (4 düğüm). Sınırlı arama 13, sınırsız arama 22 çağrı yapar. Sınırsız çağrı sayısı, boş küme dahil kliklerin sayısıdır: {0, 1, 2, 3}'ün 16 alt kümesi ile {4}, {5}, {3, 4}, {3, 5}, {4, 5}, {3, 4, 5}. Sınırlı aramada {0, 1, 2, 3} bulunduktan sonra örneğin *K = {3}*, aday = {4, 5} dalı *1 + 2 ≤ 4* ile kesilir.

7. Sırt çantası sınırında kesirli parça yerine kalan bütün eşyaların kazancı eklenirse sınır geçerli olur

mu? *W = 16* örneğinde kökte ne verir?

> **Cevap**
>
> Geçerlidir: hiçbir 0/1 çözüm, kalan eşyaların hepsini almaktan fazla kazandıramaz. Ama kapasiteyi hiç hesaba katmadığı için çok gevşektir. Kökte *40 + 30 + 50 + 10 = 130* verir; kesirli sınır 115'tir. Daha gevşek sınır daha az budar; sonuç doğru kalır ama daha çok düğüm üretilir.

8. En iyi önce sırt çantası kodunda karşılaştırıcı yanlışlıkla `Double.compare(a.sinir, b.sinir)` yazılırsa sonuç doğru kalır mı? Ne değişir?

> **Cevap**
>
> Sonuç doğru kalır. Budama kuralı ve sınır geçerli olduğu sürece hangi sırayla gezilirse gezilsin optimum kaybolmaz. Değişen sıradır: kuyruktan her adımda sınırı *en küçük* düğüm çıkar. Arama en umutsuz dallarla uğraşır ve `enIyi` geç yükselir. Bu yüzden daha çok düğüm genişletilir; strateji artık “en iyi önce” değil, “en kötü önce” olur.

10.14 İleri okuma

- E. Horowitz, S. Sahni, S. Rajasekaran, *Computer Algorithms* (2. baskı, 2008), Bölüm 8: LC arama ve $\hat{c} = f + \hat{g}$ maliyet fonksiyonu burada ayrıntılı işlenir. FIFO ve LC dal sınırın sırt çantası ve gezgin satıcı uygulamaları bu nottakine yakın bir dille verilir.
- A. Levitin, *Introduction to the Design and Analysis of Algorithms* (3. baskı, 2012), Bölüm 12.2: atama, sırt çantası ve gezgin satıcı için dal sınırı kısa ve görsel örneklerle anlatır; geri izlemeyle birlikte okunmaya uygundur.
- J. D. C. Little, K. G. Murty, D. W. Sweeney, C. Karel, “An Algorithm for the Traveling Salesman Problem” (1963): “dal sınır” adını ilk kullanan ve indirgenmiş maliyet matrisi yöntemini tanıtan makale.
- R. Carraghan, P. M. Pardalos, “An Exact Algorithm for the Maximum Clique Problem” (1990): *|K| + |aday|* sınırına dayanan klasik klik algoritması; renk sınırlı sonraki algoritmaların başlangıç noktasıdır.

10.15 Kaynaklar

1. R. Carraghan, P. M. Pardalos, “An Exact Algorithm for the Maximum Clique Problem”, *Operations Research Letters* 9(6), 1990.
2. E. Horowitz, S. Sahni, S. Rajasekaran, *Computer Algorithms*, 2. baskı, Silicon Press, 2008 (Bölüm 8: Branch-and-Bound).

3. A. H. Land, A. G. Doig, “An Automatic Method of Solving Discrete Programming Problems”, *Econometrica* 28(3), 1960.

4. A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3. baskı, Pearson, 2012 (Bölüm 12.2).

5. J. D. C. Little, K. G. Murty, D. W. Sweeney, C. Karel, “An Algorithm for the Traveling Salesman Problem”, *Operations Research* 11(6), 1963.

6. R. Neapolitan, *Foundations of Algorithms*, 5. baskı, Jones & Bartlett, 2015 (Bölüm 6).

KISIM IV · BÖLÜM 11

Yaklaşım Algoritmaları

Bir kargo firması her sabah yüzlerce adrese uğrayacak araçların rotasını çıkarır. Bir fabrika gün başında işleri makinelere dağıtır. Bir ağ yöneticisi her bağlantıyı izleyecek kadar gözlem noktası yerleştirmek ister. Bu problemlerin hepsinde en iyi çözümü bulan bilinen her algoritma, girdi büyüdükçe üstel süreye çıkar. Oysa kararın birkaç saniyede verilmesi gerekir. Yaklaşım algoritmaları bu çıkmaza bir orta yol sunar. En iyi çözümden vazgeçerler, ama buldukları çözümün en iyiden ne kadar kötü olabileceğini kanıtla sınırlarlar. Not önce yaklaşım oranı kavramını ve yaklaşım algoritmasını sezgisel yöntemden ayıran farkı tanıtır. Ardından gezgin satıcı problemi için en yakın komşu sezgiseli, minimum kapsayan ağaca dayalı 2-yaklaşım ve 2-opt yerel araması işlenir. Makine çizelgelemede liste çizelgeleme ve en uzun iş önce kuralı işlenir. Düğüm kapsamada kenar seçen 2-yaklaşım ve 3-SAT'tan indirgeme, sırt çantasında ise oran sıralı açgözlü kural ve yaklaşım şeması ele alınır. Son kesim, sayısal yakınsamanın tipik bir örneği olan kuvvet yinelemesini anlatır. Her algoritma için Java kodu, adım adım izleme, oran kanıtı ve oranın sıkı olduğunu gösteren örnekler verilir.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum11/src`

ÖĞRENME ÇIKTILARI

Kavram
Yaklaşım algoritmasını, yaklaşım oranını ve yaklaşım şemalarını (PTAS, FPTAS) tanımlamak; yaklaşım algoritmasını sezgisel ve metasezgisel yöntemlerden ayırmak.

Gezgin satıcı
En yakın komşu sezgiselini, metrik TSP için minimum kapsayan ağaca dayalı 2-yaklaşımı ve 2-opt yerel aramasını adım adım uygulamak; genel TSP'de sabit oranlı bir yaklaşımın neden olmadığını açıklamak.

Çizelgeleme ve kapsama
Liste çizelgeleme, en uzun iş önce kuralı ve düğüm kapsama 2-yaklaşımını Java ile izlemek; oranlarını kanıtlamak ve sıkı örnekler kurmak; 3-SAT'tan düğüm kapsamaya indirgemeyi açıklamak.

Sayısal yakınsama
Kuvvet yinelemesiyle baskın özdeğeri ve özvektörü hesaplamak; yakınsama hızının $|\lambda_2/\lambda_1|$ oranına nasıl bağlı olduğunu göstermek.

11.1 Yaklaşım algoritmaları

Şimdiye kadar incelenen algoritmaların çoğu problemin en iyi çözümünü bulur. Sıralama algoritması diziyi tam olarak sıralar, en kısa yol algoritması gerçekten en kısa yolu verir. Bu algoritmaların hepsi polinom sürede çalışır: girdi iki katına çıktığında süre sabit bir katsayıyla büyür. Ne var ki uygulamada karşılaşılan birçok en iyileme probleminde böyle bir algoritma bilinmiyor. Bu problemlerde en iyi çözümü bulmanın bilinen tek yolu, olası çözümlerin çok büyük bir kısmını denemektir. Çözüm sayısı ise girdiyle birlikte üstel ya da faktöriyel hızla artar.

11.1.1 Kesin çözüm neden pahalıdır?

Gezgin satıcı problemi (TSP), düğüm kapsama, makine çizelgeleme ve sırt çantası bu türden problemlerdir. Dördü de **NP-zor** problemler sınıfındadır. Karmaşıklık sınıflarını kısaca anımsamak yararlı olur. P, polinom sürede çözülebilen karar problemlerinin sınıfıdır. NP ise önerilen bir çözümün doğruluğu polinom sürede denetlenebilen karar problemlerinin sınıfıdır. Bir problem, NP'deki her problem ona polinom sürede indirgenebiliyorsa NP-zordur; hem NP-zor hem de NP'nin içindeyse NP-tamdır. P = NP olup olmadığı bilinmiyor, ama yaygın kanı ikisinin farklı olduğu yönündedir. P ≠ NP ise hiçbir NP-zor problem için polinom zamanlı kesin bir algoritma yoktur.

Bu sonucun pratik anlamı açıktır: yüz şehirlik bir rotanın ya da bin işlik bir çizelgenin en iyi çözümünü makul sürede garanti eden bir yöntem beklenmemelidir. Önde üç seçenek vardır. Küçük girdilerle yetinip üstel süre kabullenilebilir; dal sınır ve dinamik programlama bu süreyi azaltır ama ortadan kaldırmaz. Hızlı ama hiçbir şey vaat etmeyen bir sezgisel kullanılabilir. Üçüncü seçenek bu notun konusudur: polinom sürede çalışan ve sonucunun en iyiden ne kadar uzak olabileceğini kanıtla sınırlayan bir algoritma.

Tanım 11.1

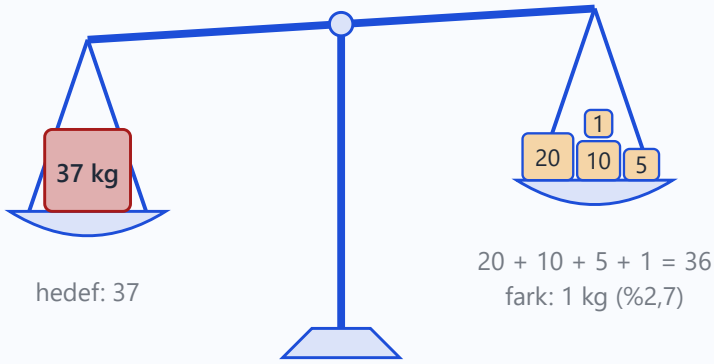
Yaklaşım algoritması (*approximation algorithm*), bir en iyileme problemi için polinom sürede çalışan ve her girdide, en iyi çözüme kanıtlanmış bir oran içinde yakın olan uygun bir çözüm üreten algoritmadır.

Tanımın üç parçası da önemlidir. Algoritma *polinom sürede* çalışır; aksi hâlde kesin çözüm aramaktan bir farkı kalmaz. Ürettiği çözüm *uygundur*, yani problemin bütün kısıtlarını sağlar: tur her şehre uğrar, çizelge her işi bir makineye verir, kapsama her kenarı kapsar. Son olarak çözümün kalitesi *her girdi için* kanıtlanmış bir sınırın içindedir. Birkaç denemede iyi sonuç vermesi yetmez. Algoritma en kötü girdide bile bu sınırı aşmamalıdır.

Bu bir takastır. Optimallikten biraz vazgeçilir; karşılığında hem verimlilik hem de çözüm kalitesi için bir güvence elde edilir. Hesaplama kaynakları, yani zaman ve bellek kısıtlı olduğunda bu takas çoğu zaman en akılcı seçimdir. Üstelik kanıtlanan oran bir en kötü durum sınırıdır. Pratikteki girdilerde yaklaşım algoritmalarının bulduğu çözümler çoğunlukla en iyiye bu sınırdan çok daha yakındır.

11.1.2 Terazi benzetmesi

İki kefeli bir terazinin sol kefesinde 37 kilogramlık bir yük olsun. Sağ kefeye konabilecek ağırlıklar 20, 10, 5 ve 1 kilogramdır ve her biri yalnız bir kez kullanılabilir. Bu ağırlıklarla 37 kilogram tam olarak kurulamaz: hepsinin toplamı bile 36 kilogramdır. Elde edilebilecek en yakın denge 36 kilogramdır ve aradaki fark 1 kilogram, yani yükün yaklaşık %2,7'sidir.



Şekil 11.1: Terazi benzetmesi. Solda 37 kilogramlık yük, sağda 20 + 10 + 5 + 1 = 36 kilogram. Tam denge kurulamaz; fark 1 kilogramdır (%2,7).

Şekil 11.1 bu durumu gösterir. Benzetmenin taşıdığı fikir şudur: kesin dengeyi, yani optimumu aramak yerine hızlıca kurulan ve hatası sınırlı olan bir denge kabul edilir. Yaklaşım algoritması da bunu yapar. Bulduğu çözüm en iyi olmayabilir, ama aradaki farkın en fazla ne kadar olabileceği önceden bellidir. Terazide fark yükün %3'ünü geçmiyorsa “en fazla %3 hata” denebilir. Yaklaşım algoritmalarında bu sınır bir oranla ifade edilir.

11.1.3 Yaklaşım oranı

Bir girdi için algoritmanın bulduğu çözümün maliyeti C, en iyi (optimal) çözümün maliyeti C^* olsun. Problem bir en küçükleme problemiyse $C \geq C^*$'dır; en büyükleme problemiyse $C \leq C^*$'dır. İki durumu tek bir ölçüyle karşılaştırmak için büyük olan değer küçük olana bölünür.

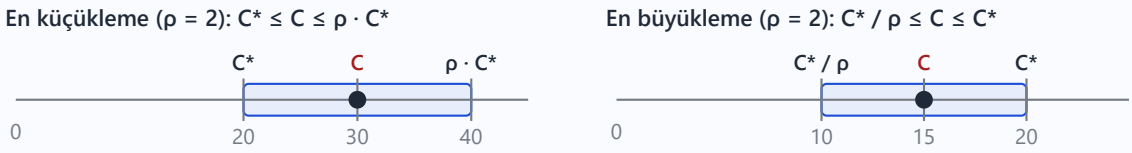
Tanım 11.2

Bir algoritma, *n* boyutlu her girdi için

> max(C / C*, C* / C) ≤ ρ(n)

koşulunu sağlıyorsa bir **ρ(n)-yaklaşım algoritmasıdır**. *ρ(n)*'ye algoritmanın **yaklaşım oranı** (*approximation ratio*) denir. Oran her zaman *ρ ≥ 1*'dir; *ρ = 1* ise algoritma en iyi çözümü bulur.

Tanımdaki *max* iki durumu birleştirir. En küçüklemede $C \geq C^*$ olduğu için büyük oran C / C^*'dır ve koşul $C \leq \rho \cdot C^*$ biçimini alır. Algoritmanın bulduğu maliyet, en iyi maliyetin en fazla ρ katıdır. Örneğin 2-yaklaşım bir düğüm kapsama algoritması en fazla en küçük kapsamanın iki katı düğüm seçer. En büyüklemede $C \leq C^*$ olduğu için büyük oran C^* / C'dir ve koşul $C \geq C^* / \rho$ olur. Algoritmanın bulduğu değer en iyi değerin en az $1/\rho$'sidir. Örneğin 2-yaklaşım bir sırt çantası algoritması en iyi değerin en az yarısını garanti eder.



Şekil 11.2: Sayı doğrusunda yaklaşım oranı (*ρ = 2*). Solda en küçükleme: *C* = 20* ise *C* 20 ile 40 arasındadır. Sağda en büyükleme: *C* = 20* ise *C* 10 ile 20 arasındadır.

Şekil 11.2 iki durumu sayı doğrusunda gösterir. En küçüklemede *C*'nin bulunabileceği aralık *[C*, ρ · C*]*, en büyüklemede *[C* / ρ, C*]*'dır. Şekildeki örnekte *C* = 20* ve *ρ = 2* alınmıştır. En küçükleme yapan bir algoritma 30 bulmuşsa oran bu girdi için *30 / 20 = 1,5*'tir. En büyükleme yapan bir algoritma 15 bulmuşsa oran *20 / 15 ≈ 1,33*'tür. İkisi de 2 sınırının içindedir.

Örnek 11.1

Bir en küçükleme probleminde algoritma 30 maliyetli bir çözüm buluyor, en iyi çözümün maliyeti 20'dir. Bu girdi için oran *C / C* = 30 / 20 = 1,5*'tir. Algoritma bir 2-yaklaşım algoritmasıysa bu sonuç beklenen sınırın içindedir, çünkü *30 ≤ 2 · 20 = 40*'tır.

Ters yönden bakılabilir. 2-yaklaşım bir algoritma 30 bulduysa en iyi maliyet hakkında ne söylenebilir? *C ≤ 2 · C** eşitsizliğinden *C* ≥ 15* çıkar. Ayrıca en iyi çözüm, bulunan çözümden kötü olamaz: *C* ≤ 30*. Demek ki en iyi maliyet 15 ile 30 arasındadır. Yaklaşım oranı, en iyi çözüm bilinmeden onun için bir alt sınır da verir.

Oranın n'ye bağlı yazılması bilinçlidir. Bazı algoritmaların oranı sabittir: düğüm kapsama için 2, metrik TSP için 1,5. Bazılarınınki girdiyle büyür: en yakın komşu sezgiselinin metrik TSP'deki oranı $\Theta(\log n)$'dir. Sabit oran çok daha güçlü bir garantidir. Girdi ne kadar büyürse büyüsün, algoritma en iyiden sabit bir kattan fazla uzaklaşmaz.

Sık yapılan hata

Oranı ters yazmak. En küçüklemede koşul $C \le \rho \cdot C^*$, en büyüklemede $C \ge C^* / \rho$'dir. En büyüklemede $C \ge \rho \cdot C^*$ yazmak, algoritmanın en iyiden daha iyi bir değer bulmasını istemek olur; bu imkânsızdır. ρ her iki türde de 1'den büyük ya da 1'e eşittir. Bazı kaynaklar en büyükleme için 1'den küçük bir oran kullanır (“optimumun en az 0,5'i”), ama bu not sunumdaki gibi her zaman $\rho \ge 1$ yazar.

11.1.4 Yaklaşım algoritması, sezgisel ve yaklaşım şemaları

Zor bir en iyileme problemine uygulanabilecek yöntemler, verdikleri garantiye göre sıralanabilir. Tablo 11.1 bu sıralamayı özetler.

Yöntem	Garanti	Örnek
Kesin algoritma	En iyi çözüm; NP-zor problemlerde üstel süre	Kaba kuvvet TSP: $(n-1)!/2$ tur
Yaklaşım algoritması	Polinom süre ve kanıtlanmış oran ρ	Düğüm kapsama: $\rho = 2$
PTAS	Her sabit $\varepsilon > 0$ için $(1+\varepsilon)$ oranı; süre n'de polinom, $1/\varepsilon$'da üstel olabilir	Öklid TSP (Arora, 1998)
FPTAS	$(1+\varepsilon)$ oranı; süre hem n'de hem $1/\varepsilon$'da polinom	Sırt çantası: $O(n^3/\varepsilon)$
Sezgisel	Garanti yok; pratikte çoğunlukla iyi	En yakın komşu, 2-opt
Metasezgisel	Garanti yok; probleme bağlı olmayan arama çerçevesi	Genetik algoritma, karınca kolonisi

Tablo 11.1: Zor en iyileme problemleri için yöntem türleri ve verdikleri garanti.

Kesin algoritma en güçlü garantiyi verir, ama NP-zor problemlerde bedeli üstel süredir. Yaklaşım algoritması polinom sürede kalır ve kalitesini bir oranla güvenceye alır. Bazı problemlerde kullanıcı istediği doğruluğu bir ε sayısıyla seçer ve algoritma $(1+\varepsilon)$ oranlı bir çözüm döndürür. Böyle bir algoritma ailesine **yaklaşım şeması** (*approximation scheme*) denir.

Tanım 11.3

Her sabit $\varepsilon > 0$ için $(1+\varepsilon)$-yaklaşım veren ve süresi n'de polinom olan algoritma ailesine **polinom zamanlı yaklaşım şeması** (*polynomial-time approximation scheme*, PTAS) denir. Süre ayrıca $1/\varepsilon$'da da polinomsa aileye **tam polinom zamanlı yaklaşım şeması** (*fully polynomial-time approximation scheme*, FPTAS) denir.

İki şema arasındaki fark ilk bakışta ince görünür, ama pratikte büyüktür. Süresi $O(n^{1/\varepsilon})$ olan bir PTAS, her sabit ε için polinomdur. Yine de $\varepsilon = 0{,}1$ seçildiğinde süre n^{10} olur ve bu, yüz elemanlık bir girdide bile kullanılamaz. Süresi $O(n^3/\varepsilon)$ olan bir FPTAS'ta ise $\varepsilon = 0{,}1$ süreyi yalnız on katına çıkarır. Sırt çantası FPTAS'ı bulunan bir problemdir. Düzlemdeki noktalar için Öklid TSP'nin PTAS'ı vardır; Arora ve Mitchell bunu birbirinden bağımsız olarak göstermiştir.

Tablonun alt kısmındaki yöntemler hiçbir garanti vermez. **Sezgisel** (*heuristic*) yöntem, probleme özgü bir sağduyu kuralıyla hızlıca iyi bir çözüm üretir. En yakın komşu ve 2-opt TSP'nin bilinen sezgiselleridir. **Metasezgisel** (*metaheuristic*) yöntem ise belirli bir probleme bağlı olmayan genel bir arama çerçevesidir; benzetimli tavlama, genetik algoritma ve karınca kolonisi bu gruptadır. Bu yöntemler pratikte çoğunlukla çok iyi sonuç verir, ama en kötü durumda en iyiden ne kadar uzaklaşabilecekleri bilinmez.

11.1.5 Uygulamalar ve teknikler

Yaklaşım algoritmalarının kullanıldığı problemler günlük hayatın içindedir. **Gezgin satıcı problemi** kargo dağıtımında ve devre kartlarında delik delme sırasında görülür. **Sırt çantası** bütçe ve yük planlamada, **iş çizelgeleme** makine ve işlemci planlamada kullanılır. **Düğüm kapsama** ise bir ağa en az sayıda izleme noktası yerleştirir. Büyük matrislerin özdeğerlerine yinelemeyle yakınsayan sayısal yöntemler de aynı başlık altında anılır; Google'ın PageRank hesabı bunun en bilinen örneğidir.

Bu problemlere uygulanan yaklaşım teknikleri birkaç ana fikirde toplanır:

- **Basitleştirme:** problemin daha kolay bir sürümü çözülür ve sonucu asıl probleme uyarlanır. Tam sayı kısıtını kaldıran doğrusal programlama gevşetmesi (*LP relaxation*) ve TSP için alt sınır veren minimum kapsayan ağaç bu türdendir.
- **Açgözlü seçim:** her adımda o an en iyi görünen seçenek alınır. En yakın komşu, en uzun iş önce ve oran sıralı sırt çantası örnektir.
- **Yerel arama:** mevcut çözüm küçük değişikliklerle iyileştirilir; daha iyi komşu çözüm kalmayınca durulur. 2-opt bunun örneğidir.
- **Birincil–eşlenik (primal–dual) yöntem:** doğrusal programın birincil ve eşlenik (*dual*) çözümleri birlikte büyütülür. Kapsama problemlerinde sık kullanılır.
- **Rastgelelik:** rastgele yuvarlama ya da örnekleme ile beklenen değerde oran garantisi elde edilir.
- **Metasezgiseller:** benzetimli tavlama, genetik algoritma, karınca kolonisi. Garanti vermezler, ama esnek ve güçlüdürler.

Bu tekniklerin ortak bir noktası vardır. Oran kanıtlarının hemen hepsi en iyi çözümün bilinmeyen değeri için bir *alt sınır* kurar. En iyi tur en az minimum kapsayan ağaç kadar uzundur; en iyi çizelge en az en uzun iş kadar sürer; en küçük kapsama en az bir eşleştirmedeki kenar sayısı kadar düğüm içerir. Algoritmanın sonucu bu alt sınırla karşılaştırılır. En iyi çözüm hiç hesaplanmadan oran böyle kanıtlanır. Bu notun bütün oran kanıtları aynı kalıba uyar.

11.2 Gezgin satıcı problemi

Gezgin satıcı problemi, yaklaşım algoritmalarının en çok çalışıldığı problemdir. Tanımı bir cümleye sığar, ama en iyi çözümü bulmak şaşırtıcı ölçüde zordur. Bu kesim önce problemi tanımlar ve kesin çözümün maliyetini hesaplar. Ardından iki yaklaşık yöntemi inceler: en yakın komşu sezgiselini ve minimum kapsayan ağaca dayalı 2-yaklaşımı.

11.2.1 Problemin tanımı

Tanım 11.4

Gezgin satıcı problemi (*travelling salesman problem*, TSP): n şehir ve her i, j şehir çifti arasındaki uzaklık d_{ij} verilir. Her şehre tam bir kez uğrayan ve başladığı şehre dönen turlar arasında toplam uzaklığı en küçük olanı bulmak istenir. $d_{ij} = d_{ji}$ ise problem **simetrik TSP** adını alır.

TSP bir en iyileme problemidir: amaç toplam uzaklığı en küçük yapmaktır. Aynı zamanda kombinatoryal bir problemdir, çünkü çözüm şehirlerin gezilme sırasıdır, yani bir permütasyondur. Çizge diliyle söylenirse simetrik TSP, kenarları ağırlıklı bir tam çizgede en hafif **Hamilton çevrimini** arar. Hamilton çevrimi, her düğümden tam bir kez geçip başlangıca dönen çevrimdir.

Problem sık sık en kısa yol problemiyle karıştırılır. İki şehir arasındaki en kısa yol Dijkstra algoritmasıyla polinom sürede bulunur. TSP ise *bütün* şehirlerden geçen en kısa kapalı turu ister ve NP-zordur. Yüzlerce şehir için en iyi rotayı bulmak pratikte çok zordur.

Bu kesimde sunumdaki beş şehirli örnek kullanılır. Şehirler 0'dan 4'e numaralıdır ve uzaklıklar Tablo 11.2'deki simetrik matriste verilmiştir.

d	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

Tablo 11.2: Beş şehrin uzaklık matrisi *d*.

Matris simetriktir: *d[1][3] = d[3][1] = 25* gibi. Köşegen sıfırdır, çünkü bir şehrin kendisine uzaklığı yoktur. Bir turun uzunluğu, ardışık şehirler arasındaki uzaklıkların ve son şehirden ilk şehre dönüşün toplamıdır. Örneğin $0 \to 1 \to 2 \to 3 \to 4 \to 0$ turunun uzunluğu $10 + 35 + 30 + 40 + 25 = 140$'tır.

11.2.2 Kesin çözümün maliyeti

En doğrudan kesin yöntem, kaba kuvvetle bütün turları denemektir. n şehrin $n!$ sıralaması vardır, ama bunların çoğu aynı turu farklı yazar. Tur kapalı olduğu için başlangıç şehri sabitlenebilir: $0 \to 1 \to 3 \to 2 \to 4 \to 0$ ile $1 \to 3 \to 2 \to 4 \to 0 \to 1$ aynı turdur. Geriye $(n-1)!$ sıra kalır. Simetrik TSP'de bir tur ile onun tersten okunuşu da aynı uzunluktadır. Bu yüzden farklı tur sayısı $(n-1)!/2$'dir.

n	$(n-1)!/2$	Süre
5	12	anında
10	181.440	≈ 0,2 milisaniye
15	$\approx 4{,}4 \cdot 10^{10}$	≈ 44 saniye
20	$\approx 6{,}1 \cdot 10^{16}$	≈ 1,9 yıl
25	$\approx 3{,}1 \cdot 10^{23}$	$\approx 10^{7}$ yıl

Tablo 11.3: Simetrik TSP'de farklı tur sayısı ve saniyede 10^9 tur denendiğinde gereken süre.

Tablo 11.3 faktöriyelin ne kadar hızlı büyüdüğünü gösterir. Beş şehirde yalnız 12 tur vardır ve hepsi elle bile denenebilir. On şehirde tur sayısı 181.440'a çıkar; bir bilgisayar bunu bir anda bitirir. Yirmi şehirde ise saniyede bir milyar tur deneyen bir bilgisayar bile yaklaşık iki yıl çalışır. Yirmi beş şehirde gereken süre yaklaşık on milyon yıldır. Şehir sayısını beş artırmak, süreyi milyonlarca kat büyütür.

Daha akıllı bir kesin yöntem dinamik programlamadır. Held–Karp algoritması, "başlangıçtan çıkıp S kümesindeki şehirlerin hepsine uğrayan ve j'de biten en kısa yol" değerlerini tablolar ve $O(n^2 \cdot 2^n)$ sürede çalışır. 25 şehirde bu yaklaşık $2 \cdot 10^{10}$ işlemdir ve faktöriyelden çok daha iyidir. Yine de süre ve $O(n \cdot 2^n)$ bellek üsteldir; altmış şehirde bu yöntem de kullanılamaz.

11.2.3 En yakın komşu sezgiseli

En yakın komşu (*nearest neighbor*) sezgiseli, bir insanın haritaya bakmadan uygulayacağı kuralı izler: her zaman henüz gidilmemiş en yakın şehre gidilir. Adımları şöyledir:

1. Bir başlangıç şehri seçilir ve ziyaret edildi olarak işaretlenir.
2. Bulunulan şehirden ziyaret edilmemiş en yakın şehre gidilir. Bu bir açgözlü seçimdir.
3. Bütün şehirler ziyaret edilene kadar ikinci adım tekrarlanır.
4. Son şehirden başlangıç şehrine dönülür ve tur kapanır.

Sunumdaki Java kodu bu adımları doğrudan uygular:

```
int[] enYakinKomsu(int[][] d, int bas) {
  int n = d.length, toplam = 0, simdiki = bas;
  boolean[] ziyaret = new boolean[n];
  int[] tur = new int[n + 1];
  tur[0] = bas; ziyaret[bas] = true;
  for (int k = 1; k < n; k++) {
    int enYakin = -1, enAz = Integer.MAX_VALUE;
    for (int j = 0; j < n; j++)
      if (!ziyaret[j] && d[simdiki][j] < enAz) {
        enAz = d[simdiki][j]; enYakin = j; }
    // en yakın şehre git
    tur[k] = enYakin; ziyaret[enYakin] = true;
    toplam += enAz; simdiki = enYakin;
  }
  // başlangıca dön
  tur[n] = bas; toplam += d[simdiki][bas];
  return tur;
}
```

Kod, uzaklık matrisini `d` ve başlangıç şehrini `bas` olarak alır. `ziyaret` dizisi her şehrin turda olup olmadığını tutar. `tur` dizisinin $n + 1$ gözü vardır, çünkü başlangıç şehri hem ilk hem son gözde yer alır. `simdiki` o an bulunulan şehri, `toplam` o ana kadarki yol uzunluğunu gösterir.

Dış döngü $k = 1$'den $n - 1$'e kadar döner ve her turda bir şehir ekler. İç döngü `simdiki` şehrin satırını tarar. Ziyaret edilmemiş ve şimdiye kadarki en küçük uzaklıktan daha yakın olan her şehir `enYakin` adayı olur. Karşılaştırmanın `<` ile yapılması bir eşitlik kuralı da getirir: aynı uzaklıkta iki şehir varsa indisi küçük olan seçilir. İç döngü bitince en yakın şehir tura yazılır, işaretlenir, uzaklığı toplama eklenir ve yeni `simdiki` olur. Dış döngüden çıkınca son şehirden başlangıca dönüş kenarı eklenir.

Maliyet hesabı kısadır. Dış döngü $n - 1$ kez, iç döngü her seferinde n kez döner. İç döngünün gövdesi sabit süredir. Toplam süre $O(n^2)$'dir. Ek bellek `ziyaret` ve `tur` dizileri için $O(n)$'dir. Girdinin kendisi zaten n^2 boyutlu bir matris olduğu için bu süre girdiyi okuma süresiyle aynı düzeydedir. Daha hızlısı beklenemez.

11.2.4 Kod izleme: beş şehir

`enYakinKomsu(d, 0)` çağrısı Tablo 11.2'deki matrisle çalıştırılsın. Başlangıçta `ziyaret[0] = true`, `tur = [0]` ve `toplam = 0`'dır. Tablo 11.4 her dış döngü adımını gösterir.

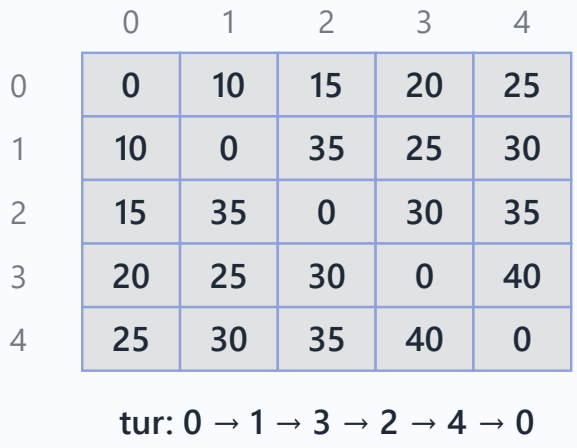


Şekil 11.3: En yakın komşu izlemesinin başlangıcı. Uzaklık matrisi; tur yalnız 0 şehrinden oluşur.

`k`	`simdiki`	**Taranan (ziyaret edilmemiş) uzaklıklar**	`enYakin` (`enAz`)	`toplam`	`tur`
1	0	*d[0][1] = 10, d[0][2] = 15, d[0][3] = 20, d[0][4] = 25*	1 (10)	10	0, 1
2	1	*d[1][2] = 35, d[1][3] = 25, d[1][4] = 30*	3 (25)	35	0, 1, 3
3	3	*d[3][2] = 30, d[3][4] = 40*	2 (30)	65	0, 1, 3, 2
4	2	*d[2][4] = 35*	4 (35)	100	0, 1, 3, 2, 4
dönüş	4	*d[4][0] = 25*	—	125	0, 1, 3, 2, 4, 0

Tablo 11.4: `enYakinKomsu(d, 0)` izlemesi.

İlk adımda 0. satır taranır. Dört aday arasında en küçük uzaklık 10'dur ve 1 şehrine aittir. Tur *0 → 1* olur, toplam 10'dur. İkinci adımda 1. satırda yalnız ziyaret edilmemiş 2, 3 ve 4 sütunlarına bakılır; 0 sütunu işaretli olduğu için atlanır. En küçük değer 25'tir, şehir 3 seçilir ve toplam 35 olur. Üçüncü adımda 3. satırdan 2 (30) ve 4 (40) arasında 2 seçilir; toplam 65'tir. Dördüncü adımda tek aday kalmıştır: 4. Toplam *65 + 35 = 100* olur. Son olarak 4'ten 0'a dönülür ve *d[4][0] = 25* eklenir.



Şekil 11.4: İzlemenin sonu. Tur *0 → 1 → 3 → 2 → 4 → 0*, uzunluk 125.

Sonuç *0 → 1 → 3 → 2 → 4 → 0* turudur ve uzunluğu 125'tir (Şekil 11.4). Doğrulama betiği aynı kodu beş başlangıç şehrinin hepsiyle çalıştırmıştır. Her başlangıçta bulunan turun uzunluğu yine 125'tir. Örneğin 2'den başlanırsa tur *2 → 0 → 1 → 3 → 4 → 2* olur: *15 + 10 + 25 + 40 + 35 = 125*.

11.2.5 Kaba kuvvetle karşılaştırma

Beş şehirde *(5 − 1)!/2 = 12* farklı tur vardır. Hepsi denenince en yakın komşunun sonucu değerlendirilebilir. Tablo 11.5 on iki turu ve uzunluklarını verir.

Tur	Uzunluk	Tur	Uzunluk
0 → 1 → 3 → 2 → 4 → 0	125	0 → 1 → 2 → 3 → 4 → 0	140
0 → 1 → 3 → 4 → 2 → 0	125	0 → 1 → 2 → 4 → 3 → 0	140
0 → 1 → 4 → 2 → 3 → 0	125	0 → 2 → 1 → 3 → 4 → 0	140
0 → 1 → 4 → 3 → 2 → 0	125	0 → 2 → 1 → 4 → 3 → 0	140
0 → 2 → 3 → 1 → 4 → 0	125	0 → 3 → 1 → 2 → 4 → 0	140
0 → 2 → 4 → 1 → 3 → 0	125	0 → 3 → 2 → 1 → 4 → 0	140

Tablo 11.5: Beş şehirli örneğin on iki turu (başlangıç 0'a sabitlenmiş, ters yön aynı sayılmış).

Tablo iki ilginç sonuç verir. Birincisi, turların yarısı 125, öbür yarısı 140 uzunluğundadır. En iyi uzunluk 125'tir ve altı farklı tur bu değere ulaşır. En yakın komşunun bulduğu tur bunlardan biridir; yani bu örnekte sezgisel en iyi sonucu bulmuştur. Bu bir şans eseridir ve genel bir durum değildir. İkincisi, matris **üçgen eşitsizliğini** sağlamaz: *d[1][2] = 35*'tir, ama 1'den 0 üzerinden 2'ye gitmek yalnız *d[1][0] + d[0][2] = 10 + 15 = 25* tutar. Doğrudan yol dolambaçlı yoldan uzundur. Böyle bir matrise metrik değil denir. Bu özellik, oran garantileri için belirleyicidir.

En yakın komşunun güçlü ve zayıf yanları şöyle özetlenebilir. Çok basittir ve $O(n^2)$ sürede çalışır. Rastgele düzlem örneklerinde bulduğu turlar ortalama olarak en iyinin yaklaşık %25 üzerindedir. Yerel arama için iyi bir başlangıç turu sağlar. Buna karşılık sabit bir yaklaşım oranı yoktur. Rosenkrantz, Stearns ve Lewis, üçgen eşitsizliği sağlansa bile oranın $\Theta(\log n)$ düzeyine çıkabildiğini göstermiştir: üst sınır $O(\log n)$'dir ve bu sınıra yaklaşan girdiler kurulabilir.

11.2.6 Genel TSP'de sabit oran yoktur

En yakın komşunun sabit bir oranı yoktur; ama belki daha akıllı bir algoritmanın vardır. Genel TSP için, yani uzaklıklar üzerinde hiçbir koşul yokken, bu umut boşunadır. P ≠ NP ise genel TSP için polinom zamanlı hiçbir sabit oranlı yaklaşım algoritması yoktur. Kanıt, Hamilton çevrimi problemine dayanır. Bir çizgede Hamilton çevrimi olup olmadığını belirleme problemi NP-tamdır.

Teorem 11.1 (Genel TSP'nin yaklaşılamazlığı)

P ≠ NP ise hiçbir sabit $\rho \geq 1$ için genel TSP'nin polinom zamanlı bir ρ-yaklaşım algoritması yoktur.

İspat. Kanıt çelişkiyle yapılır. Polinom zamanlı bir ρ-yaklaşım algoritması A olduğu varsayılsın. n düğümlü herhangi bir G çizgesi verilsin. Bu çizgeden şöyle bir TSP örneği kurulur: şehirler G'nin düğümleridir; G'de kenarla bağlı çiftlerin uzaklığı 1, bağlı olmayan çiftlerin uzaklığı $\rho n + 1$'dir. Kurulum $O(n^2)$ sürer.

G'de bir Hamilton çevrimi varsa bu çevrim, bütün kenarları 1 uzunluğunda olan bir turdur. En iyi tur n uzunluğundadır. A'nın turu en fazla ρn uzunluğundadır. Bu tur tek bir ağır kenar bile içeremez, çünkü o kenar tek başına $\rho n + 1 > \rho n$ eder. Demek ki A'nın turu yalnız G'nin kenarlarını kullanır ve G'nin bir Hamilton çevrimidir.

G'de Hamilton çevrimi yoksa her tur en az bir ağır kenar içerir ve uzunluğu ρn'den büyüktür. A'nın turu da öyledir.

Öyleyse A'yı çalıştırıp bulduğu turun uzunluğunun ρn'yi aşıp aşmadığına bakmak, G'de Hamilton çevrimi olup olmadığını polinom sürede söyler. Hamilton çevrimi NP-tam olduğu için bu, P = NP demektir. Varsayımla çelişir. □

Kurgunun gücü ağır kenarlardan gelir: tek bir ağır kenar bile oranı bozar. Ancak bu kurgu üçgen eşitsizliğini ağır biçimde ihlal eder; ağır u–v kenarı $\rho n + 1$ iken ortak bir komşu üzerinden yol yalnız 2'dir. Uzaklıkların üçgen eşitsizliğini sağladığı TSP'de kurgu geçersizdir ve sabit oranlı algoritmalar vardır.

11.2.7 Metrik TSP için MST tabanlı 2-yaklaşım

Tanım 11.5

Her i, j, k şehri için $d_{ij} \leq d_{ik} + d_{kj}$ eşitsizliği sağlanıyorsa TSP örneğine **metrik TSP** denir. Bu eşitsizliğe **üçgen eşitsizliği** (*triangle inequality*) denir: iki şehir arasında doğrudan gitmek, üçüncü bir şehre uğramaktan hiçbir zaman daha uzun değildir.

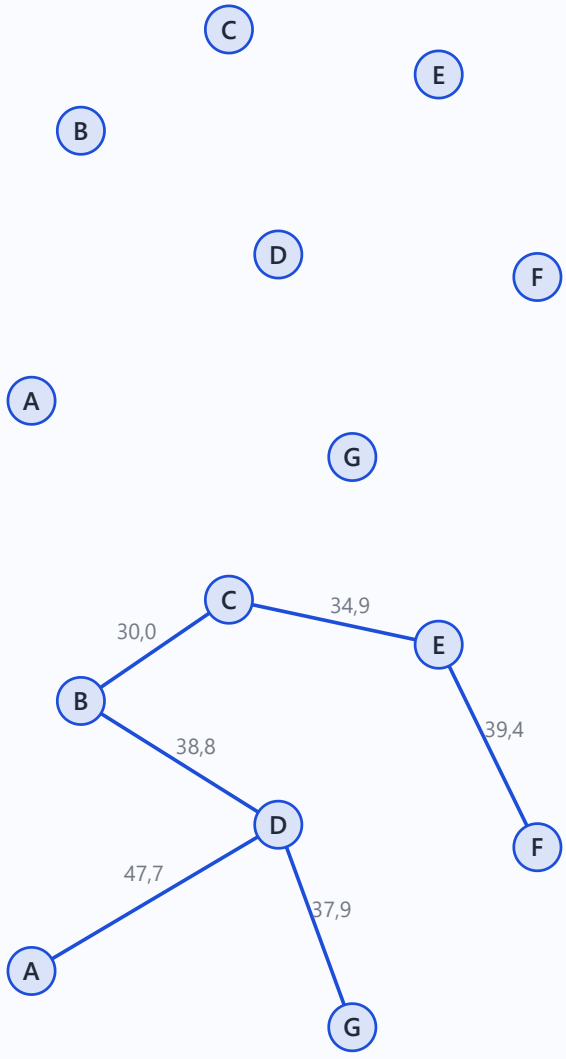
Gerçek hayattaki uzaklıkların çoğu metriktir. Düzlemdeki noktalar arasındaki Öklid uzaklığı, bir yol ağındaki en kısa yol uzaklıkları ve uçuş süreleri bu türdendir. Metrik TSP hâlâ NP-zordur, ama artık sabit oranlı bir algoritma mümkündür. En basiti minimum kapsayan ağaca (*minimum spanning tree*, MST) dayanır. Minimum kapsayan ağaç, bir çizgenin bütün düğümlerini bağlayan ağaçların toplam ağırlığı en küçük olanıdır. Prim ya da Kruskal algoritmasıyla $O(n^2)$ ya da $O(E \log V)$ sürede bulunur.

Algoritmanın üç adımı vardır:

1. Şehirlerin tam çizgesinde bir minimum kapsayan ağaç T kurulur.

2. *T*, bir kök şehirden başlayarak derinlik öncelikli gezilir. Her kenar bir kez gidişte, bir kez dönüşte kullanılır. Bu kapalı yürüyüşün uzunluğu *2 · w(T)*'dir.
3. Yürüyüşte daha önce görülmüş şehirler atlanır (**kısayol**). Geriye şehirlerin önce-kök (*preorder*) sırası kalır; sona başlangıç şehri eklenince tur kapanır.

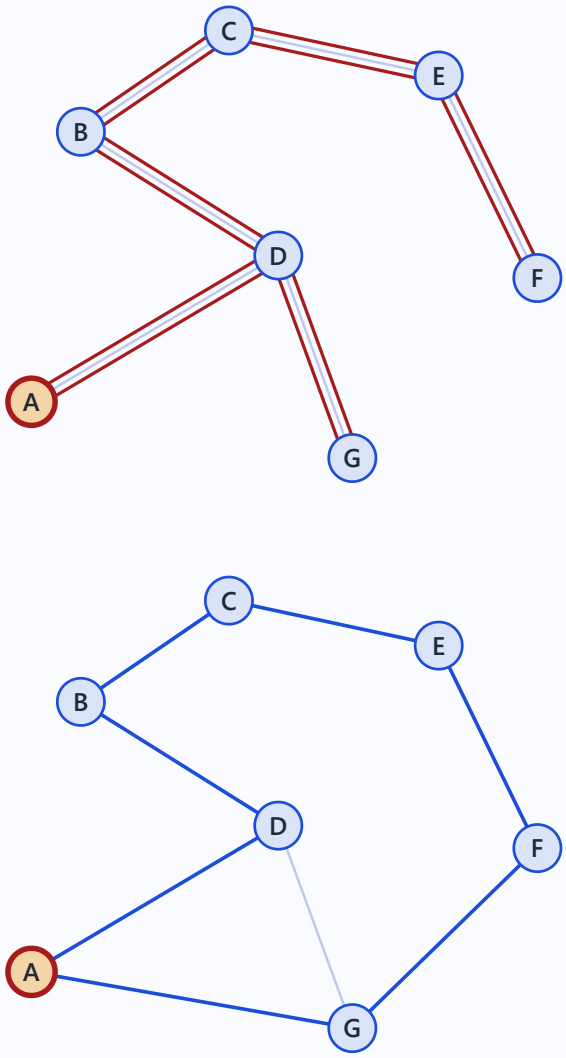
Sunumdaki örnekte düzlemde yedi şehir vardır ve uzaklıklar Öklid uzaklığıdır. Doğrulama betiği şehirlerin koordinatlarını şekildeki çizimden geri hesaplamıştır: A (10, 20), B (18, 68), C (42, 86), D (50, 46), E (76, 78), F (92, 42), G (62, 10). Bütün uzaklıklar bu koordinatlardan bulunur.



Şekil 11.5: Solda yedi şehir. Sağda Prim algoritmasıyla kurulan minimum kapsayan ağaç; kenar ağırlıkları Öklid uzaklıklarıdır ve toplamları *w(T) = 228,8*'dir.

Prim algoritması A'dan başlatılırsa kenarlar şu sırayla eklenir: A–D (47,7), D–G (37,9), D–B (38,8), B–C (30,0), C–E (34,9), E–F (39,4). Toplam ağırlık *w(T) = 228,8*'dir (Şekil 11.5). Ağacın neden en iyi tur için bir alt sınır olduğu kolayca görülür. En iyi turdan herhangi bir kenar silinirse bütün şehirleri bağ-

layan bir yol kalır. Yol da bir kapsayan ağaçtır. Minimum kapsayan ağaç, bütün kapsayan ağaçların en hafifi olduğu için bu yoldan uzun olamaz. Kenar ağırlıkları negatif olmadığından $w(T) \le OPT$ çıkar.



Şekil 11.6: Solda ağacın iki kez gezilmesi; her kenar bir gidiş, bir dönüş için kullanılır ve yürüyüş $2 \cdot w(T) = 457{,}6$ uzunluğundadır. Sağda kısayollu tur $A \to D \to B \to C \to E \to F \to G \to A$, uzunluk 287,7.

İkinci adımda ağaç A'dan derinlik öncelikli gezilir. Sunumdaki gezinme D'nin çocuklarından önce B'yi, sonra G'yi ziyaret eder: $A \to D \to B \to C \to E \to F \to E \to C \to B \to D \to G \to D \to A$. Her ağaç kenarı iki kez kullanılır ve yürüyüşün uzunluğu $2 \cdot 228{,}8 = 457{,}6$'dır. Bu yürüyüş geçerli bir tur değildir, çünkü D, B, C ve E birden çok kez geçilir.

Üçüncü adımda tekrar eden şehirler atlanır. Yürüyüş soldan okunur ve ilk kez görülen her şehir tura yazılır: A, D, B, C, E, F; sonra E, C, B, D atlanır; G yazılır; D ve A atlanır. Sona A eklenince tur $A \to D \to B \to C \to E \to F \to G \to A$ olur. Uzunluğu Tablo 11.6'da adım adım hesaplanmıştır.

Kenar	Uzunluk	Açıklama
A → D	47,7	ağaç kenarı
D → B	38,8	ağaç kenarı
B → C	30,0	ağaç kenarı
C → E	34,9	ağaç kenarı
E → F	39,4	ağaç kenarı
F → G	43,9	kısayol: F → E → C → B → D → G yürüyüşü (181,0) yerine
G → A	53,0	kısayol: G → D → A yürüyüşü (85,6) yerine
Toplam 287,7		

Tablo 11.6: Kısayollu turun kenarları. Ağaç kenarı olmayan iki kenar kısayoldur.

Kısayolların yürüyüşü hiç uzatmadığı tablodan görülür. F'den G'ye yürüyüş beş ağaç kenarından geçer ve 181,0 tutar; doğrudan F–G kenarı ise 43,9'dur. Bunu garanti eden üçgen eşitsizliğidir: bir ara şehri atlamak, yani $d_{ik} + d_{kj}$ yerine d_{ij} kullanmak, uzunluğu artırmaz. Atlama tekrar tekrar uygulanınca birden çok ara şehir de atlanabilir.

Teorem 11.2 (MST tabanlı 2-yaklaşım)

Metrik TSP'de, minimum kapsayan ağacın önce-kök sırasıyla kurulan tur en fazla en iyi turun iki katı uzunluğundadır: $tur \le 2 \cdot OPT$.

İspat. Kanıt iki eşitsizliğin birleşimidir. Birincisi alt sınırdır: en iyi turdan bir kenar silinince bir kapsayan yol kalır, bu yol bir kapsayan ağaçtır ve ağırlığı en az $w(T)$'dir. Silinen kenarın ağırlığı negatif olmadığından $w(T) \le OPT$'dir.

İkincisi üst sınırdır. Ağacın iki kez gezilmesi her kenarı tam iki kez kullanır ve uzunluğu $2 \cdot w(T)$'dir. Tur bu yürüyüşten kısayollarla elde edilir. Her kısayol, iki şehir arasındaki bir alt yürüyüşü doğrudan kenarla değiştirir. Üçgen eşitsizliği tekrar tekrar uygulanınca doğrudan kenar, alt yürüyüşten uzun olamaz. Demek ki $tur \le 2 \cdot w(T)$'dir. İki eşitsizlik birleşince $tur \le 2 \cdot w(T) \le 2 \cdot OPT$ elde edilir. □

Örnekte kaba kuvvet $6!/2 = 360$ turun hepsini dener. En iyi tur $A \to B \to C \to E \to F \to G \to D \to A$'dır ve uzunluğu 282,5'tir. Algoritmanın bulduğu 287,7, en iyinin yalnız yaklaşık 1,02 katıdır. Garanti 2'dir, ama pratik sonuç çok daha iyidir. Kanıttaki sınırlar da kontrol edilebilir: $w(T) = 228{,}8 \le 282{,}5 = OPT$ ve $287{,}7 \le 457{,}6 = 2 \cdot w(T)$.

Örnek 11.2

Derinlik öncelikli gezinmede bir düğümün çocuklarının hangi sırayla ziyaret edileceği algoritmada belirtilmez. Sunumdaki gezinme D'nin çocuklarından B'yi önce alır. Prim algoritması ise G'yi B'den önce ağaca ekler (D–G 37,9, D–B 38,8). Çocuklar ağaca eklenme sırasıyla gezilseydi

yürüyüş $A \to D \to G \to D \to B \to C \to E \to F \to E \to C \to B \to D \to A$ olurdu. Kısayollu tur $A \to D \to G \to B \to C \to E \to F \to A$ çıkar. Uzunluğu $47{,}7 + 37{,}9 + 72{,}8 + 30{,}0 + 34{,}9 + 39{,}4 + 84{,}9 = 347{,}7$'dir. Bu tur sunumdakinden belirgin biçimde uzundur, ama yine de $2 \cdot w(T) = 457{,}6$ sınırının altındadır. Oran $347{,}7 / 282{,}5 \approx 1{,}23$'tür. Garanti çocuk sırasından bağımsızdır; gerçek sonuç ise bu sıraya duyarlıdır.

Algoritmanın süresini ağaç kurma belirler. Tam çizgede Prim algoritması dizi tabanlı uygulamayla $O(n^2)$ sürer. Derinlik öncelikli gezinme ve kısayol adımı $O(n)$'dir. Toplam süre $O(n^2)$'dir; en yakın komşuyla aynı düzeydedir, ama artık kanıtlanmış bir oran vardır.

Sık yapılan hata

Üçgen eşitsizliğini unutmak. Kısayol adımının yürüyüşü uzatmadığı yalnız metrik örneklerde doğrudur. Tablo 11.2'deki matriste $d[1][2] = 35$, $d[1][0] + d[0][2] = 25$'ten büyüktür. Böyle bir örnekte 1'den 2'ye kısayol, yürüyüşü uzatır ve $tur \le 2 \cdot OPT$ sınırı geçerliliğini yitirir. Algoritma yine bir tur üretir, ama garantisi yoktur.

11.2.8 Christofides ve diğer sınırlar

MST tabanlı algoritmanın kaybı tek bir yerden gelir: her ağaç kenarı iki kez yürünür. Christofides 1976'da bu çift yürüyüşü daha ucuz bir yapıyla değiştirdi. Ağaçta derecesi tek olan düğümler bulunur; bunların sayısı her zaman çifttir. Bu düğümler arasında en küçük ağırlıklı mükemmel eşleştirme kurulur. Ağaç ile eşleştirmenin birleşiminde her düğümün derecesi çifttir ve bu yüzden bir Euler turu vardır. Euler turu her kenardan tam bir kez geçen kapalı yürüyüştür. Bu yürüyüşe kısayol uygulanır. Eşleştirmenin ağırlığı en fazla $OPT / 2$'dir; toplam $w(T) + OPT/2 \le 1{,}5 \cdot OPT$ olur. Algoritmanın süresi, eşleştirme adımı yüzünden $O(n^3)$'tür.

Christofides'in 1,5 sınırı kırk yılı aşkın süre geliştirilemedi. 2020'de Karlin, Klein ve Oveis Gharan oranı rastgele bir algoritmayla $3/2 - \varepsilon$'a indirdi; buradaki ε 10^{-36}'dan büyük çok küçük bir sabittir. Pratik bir fark yaratmaz, ama 1,5'in aşılabildiğini gösterir. Tablo 11.7 TSP için bilinen başlıca sonuçları toplar.

Durum	Oran	Açıklama
Genel TSP	yok	P ≠ NP ise hiçbir sabit ρ için polinom zamanlı ρ-yaklaşım yoktur; olsaydı Hamilton çevrimi problemini çözerdi.
Metrik TSP, MST tabanlı	2	MST + önce-kök gezinme + kısayol; $O(n^2)$.
Metrik TSP, Christofides	1,5	MST + tek dereceli düğümlerde en küçük mükemmel eşleştirme + Euler turu (1976); $O(n^3)$.
Öklid TSP	$1 + \varepsilon$	Düzlemdeki noktalar için PTAS vardır (Arora, Mitchell).

Durum	Oran	Açıklama
En yakın komşu	$\Theta(\log n)$	Metrik durumda bile sabit oran garanti etmez.

Tablo 11.7: TSP için üçgen eşitsizliğine göre bilinen yaklaşım sonuçları.

11.3 Yerel arama: 2-opt

En yakın komşu ve MST tabanlı algoritma bir turu sıfırdan kurar. Yerel arama ise elde zaten bir tur olduğunu varsayar ve onu adım adım iyileştirir. Başlangıç turu rastgele olabilir ya da başka bir sezgiselden gelebilir.

11.3.1 Yerel arama fikri

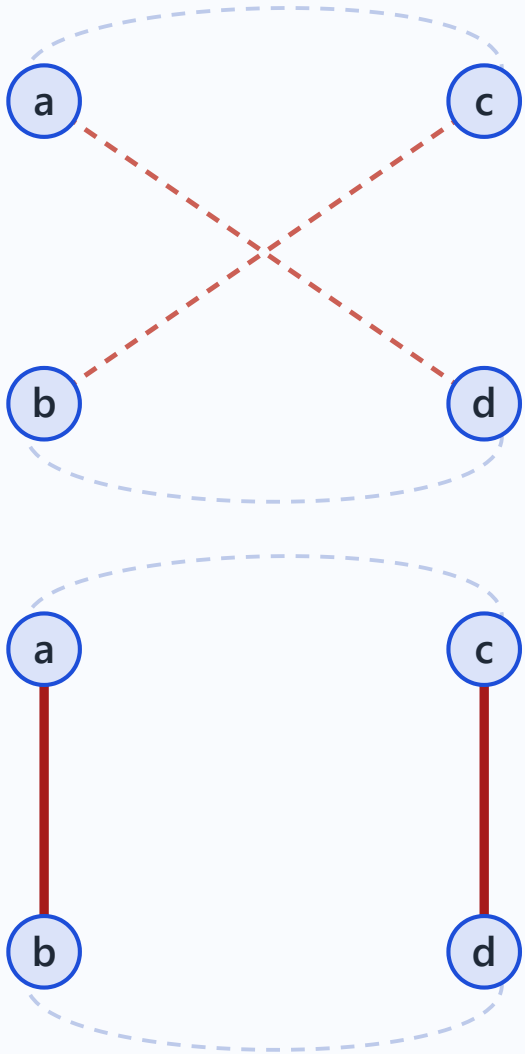
Tanım 11.6

Yerel arama (*local search*), bir çözümden başlayıp onu küçük değişikliklerle iyileştiren arama yöntemidir. Bir çözüme tek bir değişiklikle ulaşılabilen çözümlere onun **komşuları** denir. Hiçbir komşusu kendisinden daha iyi olmayan çözüme **yerel optimum** (*local optimum*) denir.

Yerel arama bir dağcının sisli havada tırmanmasına benzer. Dağcı yalnız bulunduğu noktanın çevresini görür. Hangi adım daha yükseğe çıkarıyorsa o adımı atar; hiçbir yönde yükselme kalmayınca durur. Durduğu yer bir tepedir, ama dağın en yüksek tepesi olmayabilir. Yerel arama da yerel bir optimumda durur. Bu nokta genel optimum (*global optimum*), yani bütün çözümlerin en iyisi olmayabilir.

11.3.2 2-opt hamlesi ve kazancı

Bir turdan iki kenar çıkarılsın: (a, b) ve (c, d). Tur iki parçaya ayrılır. Parçaları yeniden bir tur hâline getirmenin, eski bağlantılardan farklı tek bir yolu vardır: a'yı c'ye, b'yi d'ye bağlamak. Bu bağlantı yapılınca $b \dots c$ arasındaki alt yol ters yönde gezilir. Bu değişikliğe **2-opt hamlesi** denir.



Şekil 11.7: 2-opt hamlesi. Solda *(a, b)* ve *(c, d)* kenarları çaprazlanıyor (kesikli). Sağda bunların yerine *(a, c)* ve *(b, d)* konmuştur; aradaki yol ters çevrilmiştir. Noktalı yay turun geri kalanını gösterir.

Hamlenin turu kısaltıp kısaltmadığı yalnız dört kenara bakılarak anlaşılır. Tersine çevrilen alt yolun iç kenarları aynı kalır, çünkü simetrik TSP'de bir kenarın uzunluğu yönüne bağlı değildir. Değişen yalnız çıkan iki kenar ile giren iki kenardır. Hamlenin kazancı şöyle yazılır:

$$\Delta = d(a, c) + d(b, d) - d(a, b) - d(c, d)$$

Δ < 0 ise yeni tur eskisinden *|Δ|* kadar kısadır ve hamle yapılır. *Δ ≥ 0* ise hamle turu kısaltmaz. Kazanç *O(1)* sürede hesaplanır. Tur uzunluğu yeniden toplanmaz.

Öklid düzleminde 2-opt'un geometrik bir yorumu vardır: kesişen iki kenar her zaman kaldırılabilir. *(a, b)* ve *(c, d)* doğru parçaları bir *X* noktasında kesişsin. Üçgen eşitsizliğine göre *d(a, c) ≤ d(a, X) + d(X, c)* ve *d(b, d) ≤ d(b, X) + d(X, d)*'dir. Sağ tarafların toplamı tam olarak *d(a, b) + d(c, d)*'dir. Demek ki *Δ ≤ 0*'dır ve noktalar aynı doğru üzerinde değilse eşitsizlik kesindir. Sonuç olarak 2-opt yerel optimumu olan bir Öklid turunun hiçbir kenarı bir başkasını kesmez.

Örnek 11.3

Noktalar *a (0, 0)*, *b (4, 3)*, *c (4, 0)*, *d (0, 3)* olsun ve turda *(a, b)* ile *(c, d)* kenarları bulunsun. Bu iki kenar dikdörtgenin köşegenleridir ve birbirini keser. Her biri $\sqrt{(16 + 9)} = 5$ uzunluğundadır. Yeni kenarlar *(a, c)* ve *(b, d)* dikdörtgenin 4 uzunluğundaki yatay kenarlarıdır. Kazanç $\Delta = 4 + 4 - 5 - 5 = -2$'dir. Tur 2 birim kısalır ve hamle yapılır.

2-opt hamlesi yapacak hiçbir kenar çifti kalmayınca tur bir **2-opt yerel optimumudur**. Bu tur çoğunlukla iyi bir turdur, ama genel optimum olması gerekmez. Daha iyi bir tura ulaşmak için iki kenardan fazlasını aynı anda değiştirmek gerekebilir.

11.3.3 Kod ve maliyet

Sunumdaki Java kodu “ilk iyileştiren hamle” stratejisini izler. Turu tarar ve kazancı negatif olan ilk hamleyi hemen uygular:

```
void ikiOpt(int[] tur, double[][] d) {
  int n = tur.length;
  boolean iyilesti = true;
  while (iyilesti) {
    iyilesti = false;
    for (int i = 0; i < n - 1; i++)
      for (int j = i + 2; j < n; j++) {
        int a = tur[i], b = tur[i + 1];
        int c = tur[j], e = tur[(j + 1) % n];
        double fark = d[a][c] + d[b][e]
                    - d[a][b] - d[c][e];
        if (fark < -1e-9) {   // kısalıyor
          tersCevir(tur, i + 1, j);
          iyilesti = true;
        }
      }
  }
}
```

Tur, şehir numaralarından oluşan `tur` dizisinde tutulur ve kapalıdır: son şehirden ilk şehre bir kenar vardır. `iyilesti` bayrağı bir taramada en az bir hamle yapılıp yapılmadığını izler. Dış `while` döngüsü, bir tarama hiç hamle yapmadan bitene kadar sürer.

İç döngüler kenar çiftleri üzerinde gezinir. `i`. kenar `tur[i]` ile `tur[i + 1]` arasındaki kenardır; `j`. kenar `tur[j]` ile `tur[(j + 1) % n]` arasındadır. Mod işlemi son kenarın, yani son şehirden ilk şehre dönen kenarın da hesaba katılmasını sağlar. `j` 'nin `i + 2` 'den başlaması, komşu iki kenarın seçilmesini önler. Komşu kenarların ortak bir ucu vardır ve onlarla yapılan bir 2-opt hamlesi turu değiş-

tirmez. Yine de `i = 0` ve `j = n - 1` durumunda iki kenar turun başında birleşir: `e = tur[0] = a` olur ve kazanç her zaman sıfır çıkar. Kod bu durumu ayrıca dışlamaz; kazanç sıfır olduğu için hamle yapılmaz.

Kazanç `fark` değişkeninde hesaplanır. Karşılaştırma `fark < 0` yerine `fark < -1e-9` ile yapılır. Uzaklıklar `double` olduğu için yuvarlama hataları sıfır olması gereken bir farkı -10^{-16} gibi çok küçük bir negatif sayı yapabilir. Böyle bir "hayalet kazanç", algoritmanın aynı iki tur arasında sonsuza kadar gidip gelmesine yol açabilir. Küçük eşik bunu önler.

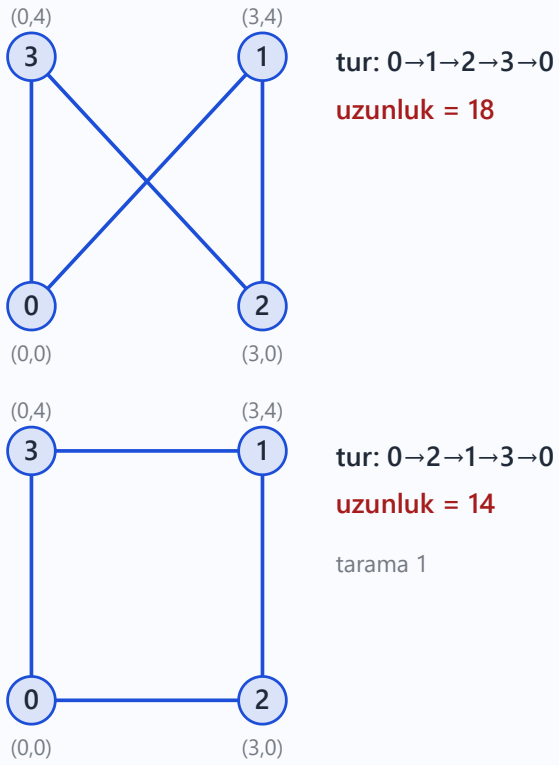
Hamle yapılınca `tersCevir(tur, i + 1, j)` çağrılır. Bu yardımcı metot sunumda verilmez; `tur[i + 1]` ile `tur[j]` arasındaki öğeleri, iki sınır dahil, yerinde ters çevirir. İki uçtan ortaya doğru yer değiştiren bir döngüyle yazılabilir:

```
void tersCevir(int[] t, int i, int j) {
  while (i < j) {
    int x = t[i]; t[i] = t[j]; t[j] = x;
    i++; j--;
  }
}
```

Maliyet üç düzeyde incelenir. Bir tarama $n(n - 3)/2$ civarında, yani $O(n^2)$ kenar çiftine bakar. Her çiftin kazancı $O(1)$'de hesaplanır. Hamle yapılan çiftte ters çevirme $O(n)$ sürer. Kaç tarama yapılacağı ise önceden bilinmez. En kötü durumda tarama sayısı üstel olabilir; kuramsal olarak kötü girdiler kurulabilir. Pratikte ise birkaç düzine tarama çoğunlukla yeter.

11.3.4 Kod izleme: dört nokta

Sunumdaki izleme dört noktayla çalışır: 0 (0, 0), 1 (3, 4), 2 (3, 0) ve 3 (0, 4). Noktalar 3 × 4 boyutunda bir dikdörtgenin köşeleridir. Uzaklıklar Öklid uzaklığıdır: yatay kenarlar 3, dikey kenarlar 4, köşegenler 5 birimdir. Başlangıç turu $0 \to 1 \to 2 \to 3 \to 0$'dır. Bu tur iki köşegeni kullanır: $d(0, 1) + d(1, 2) + d(2, 3) + d(3, 0) = 5 + 4 + 5 + 4 = 18$.



Şekil 11.8: Dört noktada 2-opt. Solda başlangıç turu *0 → 1 → 2 → 3 → 0*, uzunluk 18; (0, 1) ve (2, 3) kenarları çaprazlanıyor. Sağda ilk hamleden sonraki tur *0 → 2 → 1 → 3 → 0*, uzunluk 14.

Tarama	`i, j`	`a, b, c, e`	`fark`	Sonuç
1	0, 2	0, 1, 2, 3	*3 + 3 − 5 − 5 = −4*	hamle: `tur = [0, 2, 1, 3]`, uzunluk 14
1	0, 3	0, 2, 3, 0	*4 + 3 − 3 − 4 = 0*	komşu kenarlar (`e = a`); hamle yok
1	1, 3	2, 1, 3, 0	*5 + 5 − 4 − 4 = 2*	hamle yok; `iyilesti = true`
2	0, 2	0, 2, 1, 3	*5 + 5 − 3 − 3 = 4*	hamle yok
2	0, 3	0, 2, 3, 0	*4 + 3 − 3 − 4 = 0*	hamle yok
2	1, 3	2, 1, 3, 0	*5 + 5 − 4 − 4 = 2*	hamle yok; `iyilesti = false`, döngü biter

Tablo 11.8: `ikiOpt` izlemesi, dört nokta (*n = 4*).

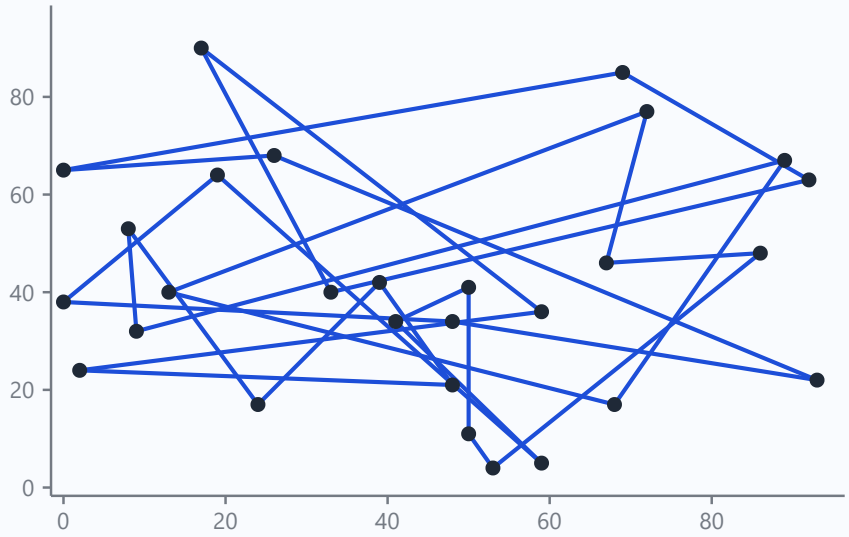
Tablo 11.8 izlemeyi verir. *n = 4* olduğu için `i` 0 ile 2 arasında, `j` de `i + 2` ile 3 arasında değişir. Geçerli çiftler (0, 2), (0, 3) ve (1, 3)'tür. İlk taramada (0, 2) çifti *(0, 1)* ve *(2, 3)* köşegenlerini çıkarıp *(0, 2)* ve *(1, 3)* kenarlarını önerir. Bu kenarlar 3 birimlik kısa kenarlardır: *fark = 3 + 3 − 5 − 5 = −4*. Hamle yapılır, `tersCevir(tur, 1, 2)` ile 1 ve 2 yer değiştirir ve tur `[0, 2, 1, 3]` olur. Yeni uzunluk *3 + 4 + 3 + 4 = 14*'tür.

Tarama aynı tur üzerinde sürer. (0, 3) çiftinde iki kenar ortak uçludur (`e = a`) ve fark sıfırdır. (1, 3) çifti köşegenleri geri getirir ve *+2* verir. İlk taramada hamle yapıldığı için tarama yinelenir; ikinci taramada hiçbir çift negatif kazanç vermez ve döngü biter.

Sonuç `[0, 2, 1, 3]` turu ve 14 uzunluğudur. Bu tur dikdörtgenin çevresidir. Dört noktada yalnız üç farklı tur vardır (*3!/2 = 3*): çevre (14) ve köşegen kullanan iki tur (18 ve 16). Bu örnekte 2-opt yerel optimumu aynı zamanda genel optimumdur.

11.3.5 Yirmi sekiz şehirde 2-opt

Daha büyük bir örnek, 2-opt'un gücünü daha açık gösterir. Sunumda 28 şehir 0–100 aralığında rastgele yerleştirilmiş ve rastgele bir başlangıç turu seçilmiştir. Tur kenarları sürekli birbirini keser ve uzunluk 1269,8'dir. Doğrulama betiği şehir koordinatlarını sunumdaki çizimden okumuş ve aynı `ikiOpt` kodunu çalıştırmıştır. Sunumdaki bütün değerler bu çalıştırmayla birebir tutar.

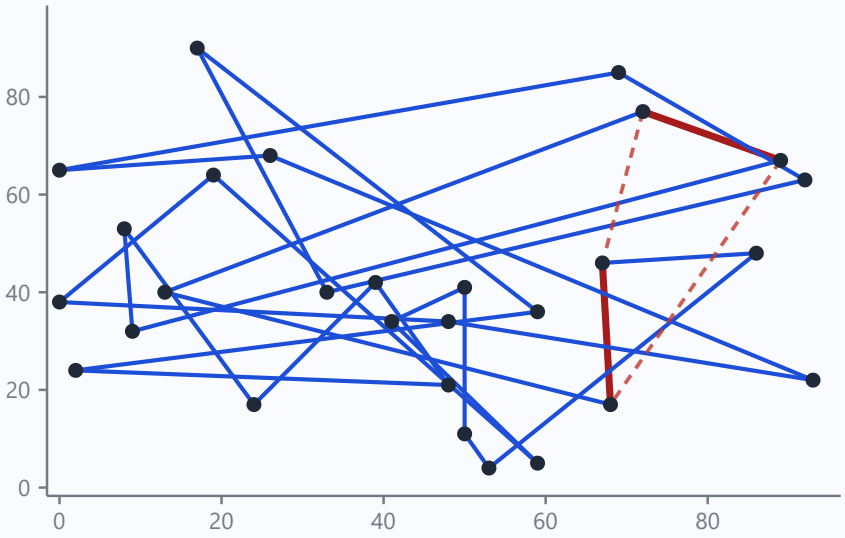


Şekil 11.9: 28 şehir ve rastgele başlangıç turu; uzunluk 1269,8. Kenarlar birbirini 66 noktada keser.

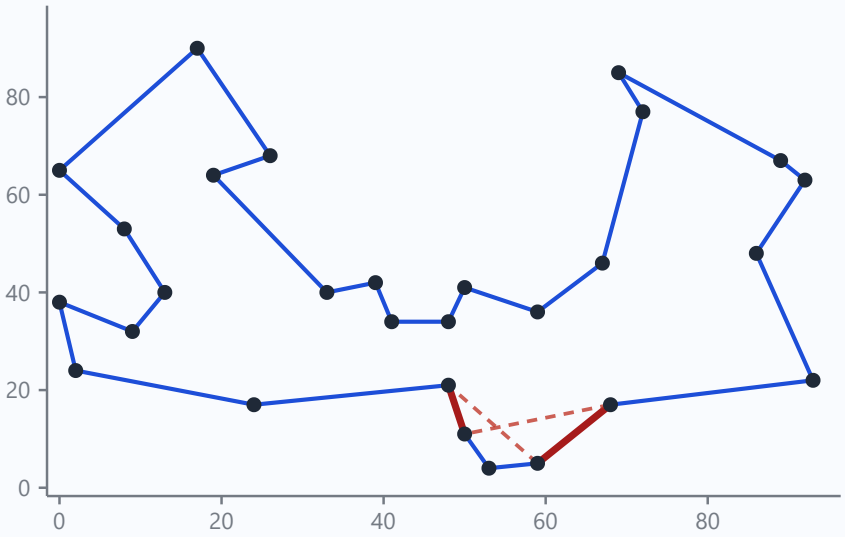
Hamle	Kazanç Δ	Tur uzunluğu
başlangıç	—	1269,8
1	−36,9	1232,9
2	−8,9	1224,0
4	−27,6	1187,3
8	−32,6	1111,7
14	−14,9	1005,2
22	−37,0	852,8
32	−11,8	595,9
48 (son)	−13,2	430,8

Tablo 11.9: 28 şehirde ilk iyileştiren hamle stratejisiyle 2-opt; seçilmiş hamlelerden sonra kazanç ve tur uzunluğu.

Tablo 11.9'daki değerler iki şey gösterir. Birincisi, tek tek hamlelerin kazancı küçüktür. Tablodaki kazançlar 9 ile 37 birim arasındadır; 48 hamlenin en büyüğü bile (70,4) o anki tur uzunluğunun %10'unu geçmez. İlk iyileştiren hamle stratejisi büyük kazancı aramaz, bulduğu ilk kazancı alır. İkincisi, bu küçük kazançlar birikir. 48 hamleden sonra tur 430,8'e iner. Bu, başlangıç uzunluğunun yaklaşık %34'üdür.



Şekil 11.10: İlk iyileştiren hamle. Kesikli kenarlar çıkarılır, kalın kenarlar eklenir; $\Delta = -36{,}9$, yeni uzunluk 1232,9.



Şekil 11.11: 48. ve son hamle. Artık hiçbir 2-opt hamlesi turu kısaltmaz; tur yerel optimumdur, uzunluk 430,8. Kesişen kenar kalmamıştır.

Son turda kesişen kenar yoktur (Şekil 11.11). Bu, 2-opt hamlesinin geometrik yorumundan beklenen sonuçtur: 2-opt yerel optimumu olan bir Öklid turunun kenarları birbirini kesmez. Ancak kesişme olmaması turun en iyi olduğu anlamına gelmez. Kesişmesiz birçok tur vardır ve bunların uzunlukları birbirinden farklıdır.

Strateji ve başlangıç turu sonucu etkiler. Doğrulama betiği iki farklı deneme daha yapmıştır. “En iyi iyileştiren hamle” stratejisi her taramada bütün çiftlere bakıp kazancı en büyük olanı uygular. Aynı rastgele turdan başladığında 23 hamlede durur ve 441,9 uzunluğunda bir yerel optimum bulur. En yakın komşu turundan (518,6) başlayan ilk iyileştiren hamle stratejisi ise yalnız 10 hamlede durur ve 449,7'ye ulaşır. Bu örnekte rastgele başlangıç daha uzun sürmüş ama daha iyi bir yerel optimuma varmıştır. Genel bir kural çıkarılamaz. Yerel aramanın sonucu başlangıca ve stratejiye bağlıdır; pratikte farklı başlangıçlarla birkaç kez çalıştırılıp en iyisi alınır.

11.3.6 Yerel optimumdan kaçış: metasezgiseller

Yerel aramanın temel sorunu, ilk yerel optimumda takılmasıdır. Yalnız iyileştiren hamleleri kabul eden bir algoritma, bir çukura düştüğünde oradan çıkamaz. Metasezgiseller bu sorunu farklı yollardan aşar: bazen kötüleştiren hamleleri de kabul eder, aynı anda birçok çözümü birlikte geliştirir ya da geçmişten öğrenir.



Şekil 11.12: Maliyet manzarası. Yalnız iyileştiren hamleleri kabul eden yerel arama soldaki çukurda kalır. Benzetimli tavlama, kötüleştiren bir hamleyi $e^{-\Delta/T}$ olasılıkla kabul ederek tepeyi aşabilir ve genel optimuma ulaşabilir.

Benzetimli tavlama (*simulated annealing*) adını metallerin tavlanmasından alır. Sıcak metal yavaşça soğutulursa atomları düzenli, enerjisi düşük bir yapıya yerleşir. Algoritma her adımda rastgele bir komşu seçer. Komşu daha iyiyse hamle yapılır. Daha kötüyse, yani maliyeti $\Delta > 0$ kadar artırıyorsa, hamle $e^{-\Delta/T}$ olasılıkla yine de kabul edilir. Buradaki T, “sıcaklık” denen bir parametredir ve yavaşça düşürülür. Tipik bir ayar $T_0 = 100$ ile başlar ve her adımda $T \leftarrow 0{,}997 \cdot T$ yapar.

Sıcaklık T	Δ / T	Kabul olasılığı
100	0,05	≈ 0,95
10	0,5	≈ 0,61

Sıcaklık T	Δ / T	Kabul olasılığı
1	5	≈ 0,0067

Tablo 11.10: $\Delta = 5$ kadar kötüleştiren bir hamlenin kabul olasılığı $e^{-\Delta/T}$.

Tablo 11.10 sıcaklığın etkisini gösterir. Yüksek sıcaklıkta kötüleştiren hamleler de neredeyse her zaman kabul edilir ve arama manzarada serbestçe dolaşır. Sıcaklık düştükçe hamleler seçicileşir. $T = 1$'de 5 birim kötüleştiren bir hamle yaklaşık 150 denemede bir kabul edilir. Sonunda algoritma neredeyse yalnız iyileştiren hamleleri kabul eden bir yerel aramaya dönüşür. Soğuma hızı da hesaplanabilir. $100 \cdot 0{,}997^{k} < 1$ eşitsizliği $k > \ln 100 / (-\ln 0{,}997) \approx 1532{,}8$ verir; sıcaklık 1533 adımda 1'in altına iner.

Genetik algoritma (*genetic algorithm*) bir tur popülasyonuyla çalışır. Kısa turlar daha sık seçilir, seçilen turlar çaprazlama (*crossover*) ile birleştirilir ve mutasyon çeşitliliği korur. Tipik bir ayar 100 bireylik popülasyon, rastgele kesim noktalı çaprazlama ve 0,05 mutasyon olasılığıdır.

Karınca kolonisi optimizasyonu (*ant colony optimization*) gerçek karıncalardan esinlenir. Kısa tur kuran yapay karıncalar kenarlara daha çok feromon bırakır; sonraki karıncalar feromonu yoğun kenarları tercih eder ve feromon zamanla buharlaşır.

Üç yöntem de TSP, araç rotalama, iş çizelgeleme ve atama problemlerinde yaygın olarak kullanılır. Hiçbiri yaklaşım oranı garanti etmez. Sonuç başlangıç çözümüne, parametrelere ve rastgele sayı üretecinin tohumuna duyarlıdır.

> **Sık yapılan hata**
>
> Metasezgiselin sonucunu kesin sanmak. Sıcaklık yeterince yavaş, logaritmik bir hızla düşürülürse benzetimli tavlamanın genel optimuma olasılıkla yakınsadığı kuramsal olarak bilinir. Pratikteki her soğuma planı ise sonludur ve garanti vermez. Aynı algoritma farklı tohum ve parametrelerle birkaç kez çalıştırılır ve en iyi sonuç alınır. Bulunan sonucun en iyiye ne kadar yakın olduğunu söylemek için bir alt sınır gerekir; örneğin minimum kapsayan ağacın ağırlığı.

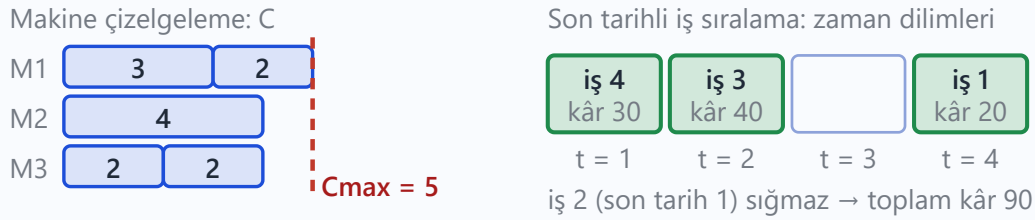
11.4 İş çizelgeleme

İş çizelgeleme (*job scheduling*), belirli kaynaklarla (işçiler, makineler, işlemciler) işlerin en verimli biçimde planlanmasıdır. Amaç, işlerin tamamlanma süresini en aza indirmek ya da kâr, gecikme ve son tarih gibi ölçütlere göre en iyi planı bulmaktır. Bu kesim iki problemi ayırır ve yaklaşım algoritmalarına konu olanını ayrıntılı inceler.

11.4.1 İki çizelgeleme problemi

Makine çizelgeleme probleminde n iş ve m özdeş makine vardır. j. işin süresi p_j'dir. Her iş tek bir makinede, kesintisiz çalışır. Bir makinenin yükü, ona verilen işlerin sürelerinin toplamıdır. Son işin bittiği an, yani en yüklü makinenin yükü C_{max} ile gösterilir. Bu değere **en büyük yük** (*makespan*) denir. Amaç C_{max}'ı en küçük yapmaktır. Problem $m \geq 2$ için NP-zordur; $m = 2$ durumu bile, işleri toplamları eşit iki kümeye bölme probleminin bir sürümüdür.

Son tarihli iş sıralama probleminde her iş bir birim sürer ve bir kârı ile bir son tarihi vardır. Son tarihinden önce biten işlerin toplam kârı en büyük olmalıdır. Bu problem, işleri kâra göre azalan sırayla son tarihinden önceki en geç boş dilime koyan açgözlü bir algoritmayla en iyi biçimde çözülür.



Şekil 11.13: İki çizelgeleme problemi. Solda üç makineye dağıtılmış işler ve en büyük yük $C_{max} = 5$. Sağda dört zaman dilimine yerleştirilmiş birim süreli işler; son tarihi 1 olan iş 2 sığmaz ve toplam kâr 90'dır.

11.4.2 Liste çizelgeleme

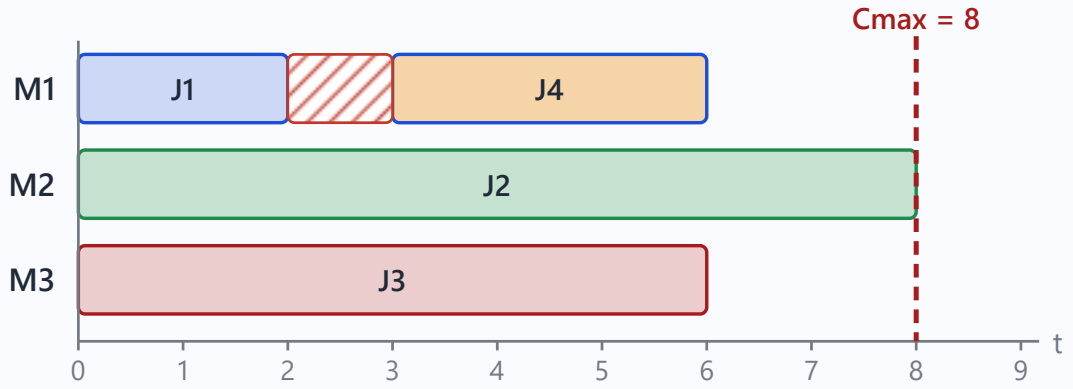
Tanım 11.7

Liste çizelgeleme (*list scheduling*), işleri verilen sırayla ele alır ve sıradaki işi o an en erken boşalan, yani en az yüklü makineye verir. Algoritma Graham'a (1966) dayanır.

Liste çizelgeleme çevrim içi bir algoritmadır: bir işi atarken sonra gelecek işleri bilmesi gerekmez. Bu yüzden işlerin zaman içinde geldiği ortamlarda doğrudan kullanılabilir. Sunumdaki örnek bu durumu gösterir. Üç özdeş makine M1, M2 ve M3 vardır. J1 (süre 2), J2 (süre 8) ve J3 (süre 6) $t = 0$'da hazırdır; J4 (süre 3) $t = 3$'te gelir.

Olay	Karar	Gerekçe
J1 (2), $t = 0$	M1, [0, 2]	Bütün makineler boş; ilk makine seçilir.
J2 (8), $t = 0$	M2, [0, 8]	M1, $t = 2$'ye kadar meşgul.
J3 (6), $t = 0$	M3, [0, 6]	M1 $t = 2$'ye, M2 $t = 8$'e kadar meşgul.
$t = 2$	M1 boşta	Bekleyen iş yok; M1 $t = 3$'e kadar boş kalır.
J4 (3), $t = 3$	M1, [3, 6]	M1 boş; M2 $t = 8$'e, M3 $t = 6$'ya kadar meşgul.

Tablo 11.11: Varış zamanlı liste çizelgeleme örneği.



Şekil 11.14: Liste çizelgelemenin sonucu. M1'de J1 ve J4 arasında bir birimlik boşluk (taralı) vardır; en büyük yük M2'nin yüküdür: $C_{max} = 8$.

Bütün işler atandıktan sonra en büyük yük 8'dir ve M2'ye aittir (Şekil 11.14). Bu çizelge en iyidir, çünkü hiçbir çizelge en uzun işten önce bitemez. J2 tek başına 8 birim sürer; en iyi en büyük yük de en az 8'dir. Genelde ise liste çizelgeleme en iyiyi bulmaz; yalnız $2 - 1/m$ oranını garanti eder.

En az yüklü makine bir dizi taranarak bulunursa süre $O(nm)$, makineler bir en küçük heap'te (*min-heap*) tutulursa $O(n \log m)$ olur.

11.4.3 Liste çizelgelemenin oranı

Oran kanıtı, en iyi çözüm için iki alt sınıra dayanır. Bunlardan biri iş yüküyle ilgilidir. Toplam iş $\Sigma\, p_j$'dir ve m makineye dağıtılmalıdır. En iyi çizelgede bile bir makinenin yükü en az ortalama yük kadardır: $OPT \geq \Sigma\, p_j / m$. Öbürü en uzun işle ilgilidir. Her iş bir makinede kesintisiz çalışır; en uzun iş tek başına $\max p_j$ sürer: $OPT \geq \max p_j$.

Teorem 11.3 (Graham)

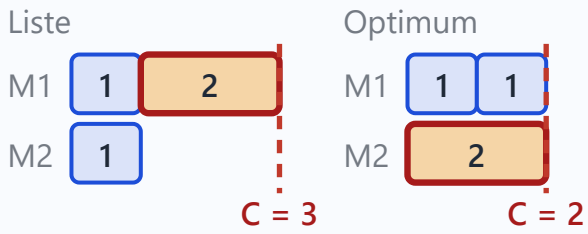
Bütün işler başta hazırsa liste çizelgeleme, işlerin sırası ne olursa olsun, $C_{max} \leq (2 - 1/m) \cdot OPT$ koşulunu sağlayan bir çizelge üretir.

İspat. Çizelgede en son biten iş j olsun ve s_j anında başlamış olsun. $C_{max} = s_j + p_j$'dir. j atandığı anda en az yüklü makineye gitmiştir; demek ki o anda bütün makinelerin yükü en az s_j'dir. Başka bir deyişle s_j'den önce bütün makineler meşguldü; boş bir makine olsaydı j daha erken başlardı. Bu meşguliyet j dışındaki işlerden gelir: $m \cdot s_j \leq \Sigma\, p - p_j$, yani $s_j \leq (\Sigma\, p - p_j) / m$.

$$C_{max} = s_j + p_j \leq \Sigma\, p / m + (1 - 1/m) \cdot p_j \leq OPT + (1 - 1/m) \cdot OPT = (2 - 1/m) \cdot OPT$$

Son adımda iki alt sınır kullanılır: $\Sigma\, p / m \leq OPT$ ve $p_j \leq \max p \leq OPT$. □

Oranın sıkı olduğu, yani daha iyi bir sınırın kanıtlanamayacağı bir örnekle gösterilir. $m = 2$ makine ve sırasıyla 1, 1, 2 süreli üç iş olsun. Liste çizelgeleme ilk işi M1'e, ikinciyi M2'ye verir; üçüncü iş için iki makinenin yükü de 1'dir ve iş M1'e gider. Yükler 3 ve 1 olur; $C_{max} = 3$'tür. En iyi çizelge iki kısa işi bir makineye, uzun işi öbürüne verir ve 2'de biter. Oran $3/2 = 2 - 1/2$'dir; teoremdeki sınıra tam olarak ulaşılmıştır.



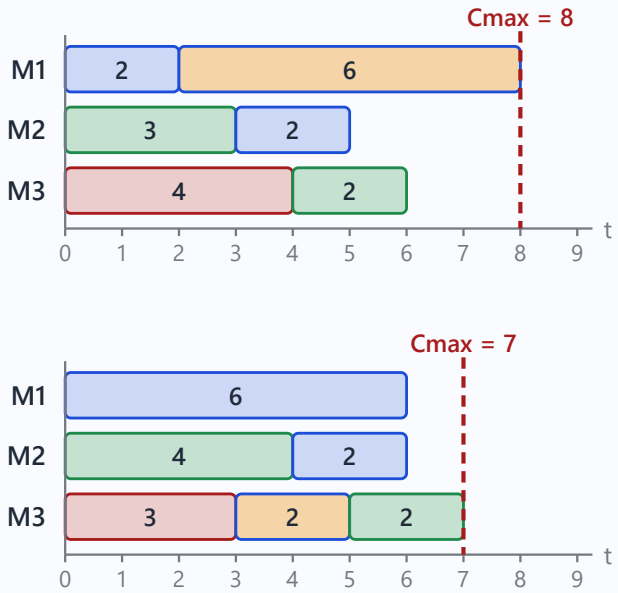
Şekil 11.15: Sıkı örnek, $m = 2$ ve işler 1, 1, 2. Liste çizelgeleme $C = 3$, en iyi çizelge $C = 2$ bulur.

Örnek her m için genelleştirilebilir: önce $m(m - 1)$ adet 1 birimlik iş, sonra tek bir m birimlik iş gelsin. Kısa işler makinelere eşit dağılır ve her makinenin yükü $m - 1$ olur. Uzun iş bunlardan birine eklenir: $C_{max} = 2m - 1$. En iyi çizelge uzun işi tek başına bir makineye verir ve kısa işleri kalan $m - 1$ makineye m'şer dağıtır: $OPT = m$. Oran $(2m - 1)/m = 2 - 1/m$'dir. $m = 3$ için altı adet 1 birimlik iş ve bir 3 birimlik iş, $5/3$ oranını verir.

11.4.4 En uzun iş önce (LPT)

Sıkı örnekler aynı zayıflığı gösterir: uzun bir iş sona kalınca, öbür makineler çoktan bitmişken tek bir makine çalışmaya devam eder. Çözüm basittir: uzun işler başa alınır. **En uzun iş önce** (*longest processing time first*, LPT) kuralı, işleri süreye göre azalan sıraya koyar ve sonra liste çizelgeleme uygular. Sona kalan işler kısadır; dengesizliği düzeltirler.

Sunumdaki örnekte işler 2, 3, 4, 6, 2, 2 ve $m = 3$'tür. Toplam iş 19'dur; en iyi en büyük yük en az $\lceil 19/3 \rceil = 7$'dir. Verilen sırayla liste çizelgeleme önce 2, 3 ve 4'ü üç makineye dağıtır. 6 birimlik iş en az yüklü makineye, yani 2 birim yüklü M1'e gider ve onun yükü 8 olur. Kalan iki kısa iş M2 ve M3'e eklenir. $C_{max} = 8$'dir. LPT sırası 6, 4, 3, 2, 2, 2'dir. Uzun işler önce üç makineye dağılır; kısa işler sonra en az yüklü makinelere eklenir ve yükler 6, 6, 7 olur. $C_{max} = 7$'dir ve alt sınıra eşit olduğu için en iyidir.



Şekil 11.16: Aynı işler, iki sıra. Solda verilen sırayla liste çizelgeleme, $C_{max} = 8$. Sağda LPT sırası 6, 4, 3, 2, 2, 2, $C_{max} = 7 = OPT$.

Teorem 11.4 (Graham, 1969)

LPT kuralı $C_{max} \le (4/3 - 1/(3m)) \cdot OPT$ koşulunu sağlayan bir çizelge üretir. Süresi sıralama yüzünden $O(n \log n)$'dir.

İspat. Kanıtın ana hattı şöyledir. Sınırı aşan bir karşı örnek olduğu varsayılır ve iş sayısı en az olan böyle bir örnek alınır. Bu örnekte en son biten iş j, en kısa iştir. Aksi hâlde j'den sonra gelen işler silinir, C_{max} değişmez, OPT artmaz ve daha küçük bir karşı örnek bulunur.

Liste çizelgeleme kanıtındaki eşitsizlik buraya da uygulanır: $C_{max} \le \Sigma\, p / m + (1 - 1/m) \cdot p_j \le OPT + (1 - 1/m) \cdot p_j$. Eğer $p_j \le OPT/3$ ise sağ taraf $OPT + (1 - 1/m) \cdot OPT/3 = (4/3 - 1/(3m)) \cdot OPT$'dir ve sınır sağlanır. Öyleyse karşı örnekte $p_j > OPT/3$'tür. j en kısa iş olduğu için bütün işler $OPT/3$'ten uzundur. En iyi çizelgede bir makineye en fazla iki iş sığar. Bu durumda LPT'nin en iyi çizelgeyi bulduğu doğrudan gösterilebilir: en uzun m iş birer makineye konur, kalanlar en uzun işin yanına en kısa iş gelecek biçimde eşleştirilir. Çelişki elde edilir. □

LPT sınırı da sıkıdır. $m = 3$ ve işler 5, 5, 4, 4, 3, 3, 3 olsun. LPT yükleri önce 5, 5, 4 yapar. Sonraki 4 ve iki 3 yükleri 8, 8, 8'e getirir; son 3 bir makineyi 11'e çıkarır. En iyi çizelge $\{5, 4\}$, $\{5, 4\}$, $\{3, 3, 3\}$ ile 9'da biter: oran $11/9 = 4/3 - 1/9$. Aynı kalıp her m için kurulur: $2m - 1, 2m - 1, \ldots, m + 1, m + 1$ ve üç tane m. Doğrulama betiği $m = 2, 3, 4$ için LPT'nin $4m - 1$, en iyinin $3m$ bulduğunu denetlemiştir.

LPT çevrim içi değildir: işleri sıralamak için hepsini önceden bilmelidir. Bunun karşılığında garantisi çok daha güçlüdür. Liste çizelgelemenin sınırı m arttıkça 2'ye, LPT'ninki $4/3$'e yaklaşır; $m = 2$'de değerler 1,5 ve $7/6 \approx 1{,}17$, $m = 4$'te 1,75 ve 1,25'tir. Makine çizelgelemenin bir PTAS'ı da vardır

(Hochbaum ve Shmoys, 1987): her $1 + \varepsilon$ oranına polinom sürede ulaşılabilir, ama süre $1/\varepsilon$'da üsteldir.

11.4.5 LPT kodu ve izleme

Sunumdaki Java kodu LPT'yi en basit biçimde yazar:

```
int lpt(Integer[] isler, int m) {
  Arrays.sort(isler, Comparator.reverseOrder());
  int[] yuk = new int[m];
  for (int p : isler) {
    int k = 0;              // en az yüklü
    for (int i = 1; i < m; i++)
      if (yuk[i] < yuk[k]) k = i;
    yuk[k] += p;
  }
  return Arrays.stream(yuk).max().getAsInt();
}
```

İlk satır işleri `Comparator.reverseOrder()` ile büyükten küçüğe sıralar. Bu karşılaştırıcı nesnelerle çalıştığı için dizi `int[]` değil `Integer[]` türündedir. `yuk` dizisi her makinenin o anki yükünü tutar ve başta sıfırdır. Döngü her işi ele alır. İç döngü en az yüklü makinenin indisini `k`'de bulur. Karşılaştırma kesin küçüktür (`<`); bu yüzden yükler eşitse indisi küçük olan makine seçilir. İş o makinenin yüküne eklenir. Sonunda yüklerin en büyüğü döndürülür.

Kodun ilk satırı silinirse geriye liste çizelgeleme kalır: işler verildiği sırayla en az yüklü makineye gider. İki algoritma arasındaki tek fark bu sıralamadır. Kodda en az yüklü makine dizi taranarak bulunur ve her iş için $O(m)$ sürer. Toplam süre $O(n \log n + nm)$'dir. Makineler bir heap'te tutulursa süre $O(n \log n + n \log m) = O(n \log n)$'ye iner.

Adım	`p`	`yuk` (önce)	En az yüklü `k`	`yuk` (sonra)
1	6	[0, 0, 0]	0 (eşitlikte ilk makine)	[6, 0, 0]
2	4	[6, 0, 0]	1 (yük 0)	[6, 4, 0]
3	3	[6, 4, 0]	2 (yük 0)	[6, 4, 3]
4	2	[6, 4, 3]	2 (yük 3)	[6, 4, 5]
5	2	[6, 4, 5]	1 (yük 4)	[6, 6, 5]
6	2	[6, 6, 5]	2 (yük 5)	[6, 6, 7]

Tablo 11.12: `lpt({2, 3, 4, 6, 2, 2}, 3)` izlemesi; sıralama sonrası işler 6, 4, 3, 2, 2, 2.



Şekil 11.17: LPT izlemesinin sonu. Makine yükleri 6, 6 ve 7; $C_{max} = 7$.

Tablo 11.12 kodun her adımını gösterir. İlk adımda bütün yükler sıfırdır. İç döngü `k = 0` ile başlar ve hiçbir yük 0'dan küçük olmadığı için değişmez; 6 birinci makineye gider. İkinci ve üçüncü adımlarda boş makineler sırayla doldurulur. Dördüncü adımda en az yüklü makine 3 yüklü M3'tür (`k = 2`). Beşinci adımda yükler 6, 4, 5'tir ve 2 birimlik iş 4 yüklü makineye gider. Son adımda yükler 6, 6, 5'tir ve son iş 5 yüklü makineye eklenir. En büyük yük 7'dir. Alt sınır *⌈19/3⌉ = 7* olduğu için bu çizelge en iyidir.

11.5 Düğüm kapsama

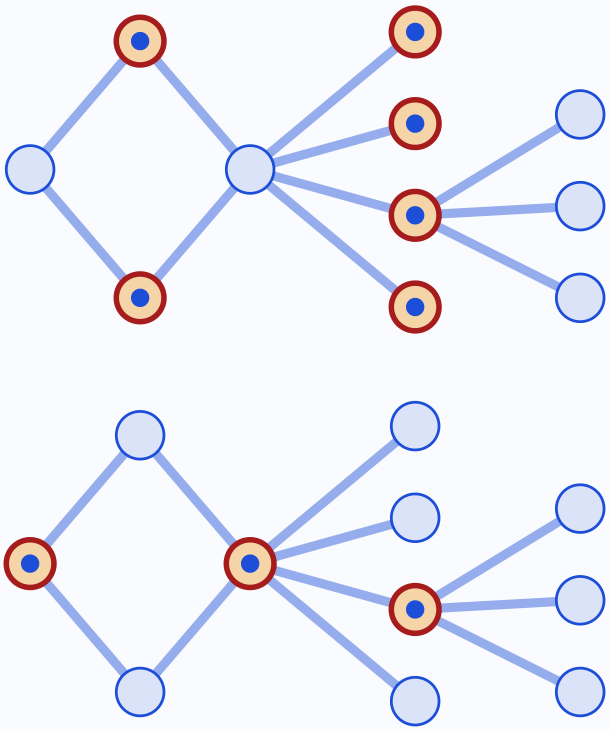
Bir kampüs ağında her bağlantının izlenmesi istensin. İzleme yazılımı bir yönlendiriciye kurulunca o yönlendiriciye bağlı bütün hatları görür. En az kaç yönlendiriciye yazılım kurulmalıdır? Bu soru, çizge kuramının klasik düğüm kapsama problemidir.

11.5.1 Problemin tanımı

Tanım 11.8

G = (V, E) bir yönsüz çizge olsun. Her kenarın en az bir ucunu içeren $C \subseteq V$ kümesine **düğüm kapsama** (*vertex cover*) kümesi denir. Bir kenarın uçlarından biri kümedeyse kenar *kapsanmıştır*. En küçük düğüm kapsama problemi, eleman sayısı en küçük kapsama kümesini arar.

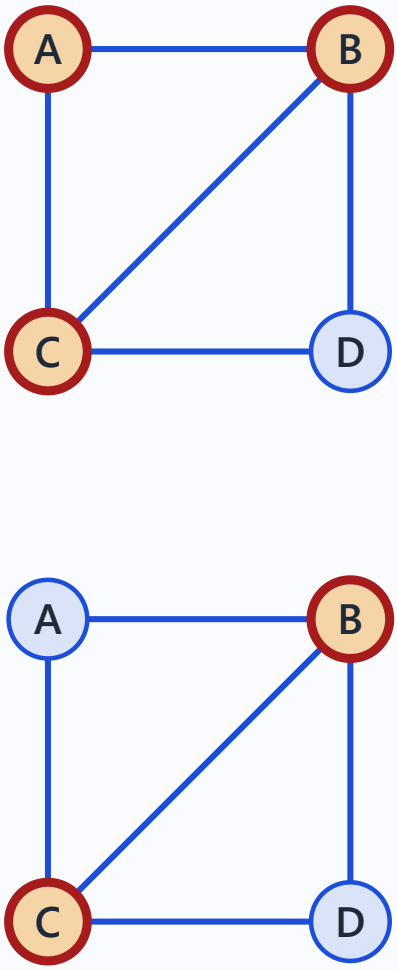
Kapsama kümesi bir bekçi düzeni gibi düşünülebilir: her düğümdeki bekçi o düğüme değen kenarları görür ve her kenarı en az bir bekçi görmelidir. Problemin karar sürümü, yani “*k* düğümlük bir kapsama var mı?” sorusu, NP-tamdır.



Şekil 11.18: Aynı çizgede iki kapsama. Solda geçerli bir kapsama, boyut 6: her kenarın en az bir ucunda bekçi vardır. Sağda en küçük kapsama, boyut 3.

Şekil 11.18'deki çizgede soldaki altı düğümlük küme geçerlidir, ama sağdaki üç düğüm aynı işi görür. En soldaki düğüm dörtgenin iki kenarını, ortadaki merkez düğüm dörtgenin öbür iki kenarını ve sağa açılan dört kenarı, sağdaki iç düğüm de üç yaprağa giden kenarları kapsar. Üçten azının yetmediği de gösterilebilir: dörtgenin bir kenarı, merkezden sağa giden bir kenar ve bir yaprak kenarı ortak ucu olmayan üç kenardır. Her biri ayrı bir bekçi ister.

Sunumdaki ikinci örnek dört düğümlüdür. Düğümler A, B, C, D, kenarlar *(A, B), (A, C), (B, C), (B, D), (C, D)*'dir. *{A, B, C}* geçerli bir kapsamadır: her kenarın en az bir ucu kümededir. Ama en küçük değildir, çünkü *{B, C}* de beş kenarın hepsini kapsar. Tek düğüm yetmez; örneğin B, *(A, C)* kenarını kapsamaz. En iyi değer 2'dir.

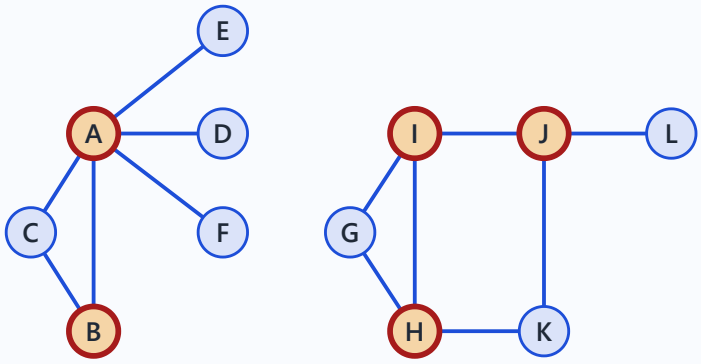


Şekil 11.19: Dört düğümlü çizge. Solda *{A, B, C}*, geçerli ve boyutu 3. Sağda *{B, C}*, en küçük ve boyutu 2.

11.5.2 Çözüm yaklaşımları

Kesin çözüm, 2^n alt kümenin her birini $O(m)$ sürede denetler: toplam $O(2^n \cdot m)$. Dal sınır ve doğrusal programlama bu aramayı hızlandırır, ama üstel süreyi ortadan kaldırmaz.

Yaklaşık çözümlerde iki açgözlü fikir öne çıkar: en yüksek dereceli düğümü seçmek ve kapsanmamış bir kenarın iki ucunu birden almak. İlki doğal görünür ama garantisi zayıftır; ikincisi savurgan görünür, ama 2 oranını garanti eder.

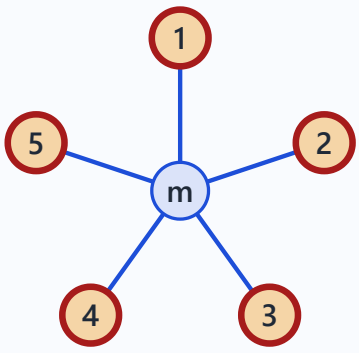


Şekil 11.20: A'dan L'ye 12 düğümlü çizge. En küçük kapsamalardan biri *{A, B, H, I, J}*, 5 düğüm.

Şekil 11.20'deki çizge izlemede kullanılır. Kenarları *(A, B), (A, C), (A, D), (A, E), (A, F), (B, C), (G, H), (G, I), (H, I), (H, K), (I, J), (J, K), (J, L)*'dir. Doğrulama betiği bütün alt kümeleri denemiştir. En küçük kapsama 5 düğümlüdür ve dört farklı en iyi çözüm vardır: *{A, B, H, I, J}*, *{A, B, G, H, J}*, *{A, C, H, I, J}* ve *{A, C, G, H, J}*. Beşten azının yetmediği şöyle görülür: A, B, C üçgeni en az iki düğüm ister; G, H, I üçgeni de öyle. *(J, L)* kenarı da bu iki üçgenden ayrı bir düğüm ister.

11.5.3 Tek uç eklemek neden yetmez?

Akla gelen ilk algoritma şudur. Başlangıçta kapsama kümesi boştur. Kapsanmamış bir kenar seçilir ve uçlarından biri kümeye eklenir. Bütün kenarlar kapsanana kadar bu adım tekrarlanır. Algoritma her zaman geçerli bir kapsama üretir, ama hangi ucun ekleneceğine dair bir kural koymadıkça kalitesi hakkında hiçbir şey söylenemez.



Şekil 11.21: Yıldız çizge. Her kenarda yaprak ucu seçilirse 5 düğüm seçilir; en iyi çözüm yalnız merkezdir, *{m}*.

Şekil 11.21 kötü durumu gösterir. Merkez *m* beş yaprağa bağlıdır. Algoritma her kenarda yaprak ucunu seçerse kenarlar birer birer kapsanır ve sonunda beş yaprağın hepsi seçilir. Oysa yalnız merkezi seçmek bütün kenarları kapsar. *n* yapraklı bir yıldızda bu algoritmanın oranı *n*'dir ve girdiyle sınırsız büyür.

Doğru 2-yaklaşım küçük ama belirleyici bir değişiklik yapar. Kapsanmamış bir *(u, v)* kenarı seçilir ve *u* ile *v*'nin *ikisi de* kümeye eklenir. *u* ya da *v*'ye değen bütün kenarlar kapsanmış sayılır. Kenar kalmayana kadar bu adımlar tekrarlanır. Yıldızda bu algoritma ilk kenarın iki ucunu, yani merkezi ve bir yaprağı seçer. Bütün kenarlar kapsanır; sonuç 2 düğümdür ve en iyinin iki katıdır.

11.5.4 Kenar seçen 2-yaklaşım: kod ve izleme

Sunumdaki Java kodu kenarları bir dizide alır ve onları sırayla işler:

```
Set<Character> dugumKapsama(char[][] kenarlar) {
  Set<Character> kapsama = new HashSet<>();
  for (char[] k : kenarlar) {
    char u = k[0], v = k[1];
    if (!kapsama.contains(u) && !kapsama.contains(v)) {
      kapsama.add(u); kapsama.add(v); }  // iki uç
  }
  return kapsama;
}
```

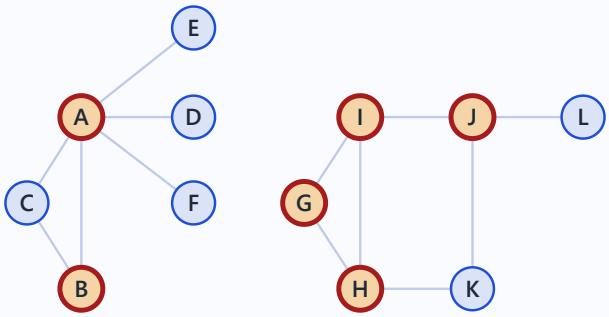
Her kenar iki karakterlik bir dizidir: `k[0]` ve `k[1]` uçlardır. Kapsama bir `HashSet`'te tutulur; `contains` ve `add` işlemleri ortalama *O(1)* sürer. Döngü her kenarda iki ucun da kapsamada olmadığını denetler. Öyleyse kenar henüz kapsanmamıştır ve iki ucu birden eklenir. Uçlardan biri zaten kapsamadaysa kenar kapsanmıştır ve atlanır. Kodda "kapsanmış sayma" işlemi ayrıca yazılmaz; bir kenar, uçlarından biri kümeye girdiği anda kendiliğinden kapsanmış olur ve sırası geldiğinde atlanır.

Maliyet açıktır. Her kenar bir kez incelenir ve her inceleme ortalama sabit sürer. Toplam süre *O(V + E)*'dir. Bellek kapsama kümesi için *O(V)*'dir.

Kenar	Durum	Kapsama (sonra)
(A, B)	iki uç da dışarıda → A ve B eklenir	{A, B}
(A, C)	A kapsamada → atlanır	{A, B}
(A, D), (A, E), (A, F)	A kapsamada → atlanır	{A, B}
(B, C)	B kapsamada → atlanır	{A, B}
(G, H)	iki uç da dışarıda → G ve H eklenir	{A, B, G, H}
(G, I)	G kapsamada → atlanır	{A, B, G, H}
(H, I), (H, K)	H kapsamada → atlanır	{A, B, G, H}
(I, J)	iki uç da dışarıda → I ve J eklenir	{A, B, G, H, I, J}
(J, K), (J, L)	J kapsamada → atlanır	{A, B, G, H, I, J}

Tablo 11.13: `dugumKapsama` izlemesi, 12 düğümlü çizge.

Tablo 11.13, kodu Şekil 11.20'deki çizgede çalıştırır. Kenarlar yukarıdaki sırayla verilmiştir. İlk kenar *(A, B)*'nin iki ucu da dışarıdadır; A ve B eklenir. A'ya ya da B'ye değen sonraki beş kenar atlanır. *(G, H)* ikinci bağımsız kenardır; G ve H eklenir. *(G, I)*, *(H, I)* ve *(H, K)* atlanır. *(I, J)*'nin iki ucu da dışarıdadır; I ve J eklenir. Kalan iki kenarı J kapsar.



Şekil 11.22: İzlemenin sonu. Kapsama *{A, B, G, H, I, J}*, 6 düğüm. Seçilen *(A, B)*, *(G, H)*, *(I, J)* kenarları ortak ucu olmayan bir eşleştirmedir.

Sonuç *{A, B, G, H, I, J}* kümesidir ve 6 düğüm içerir. En iyi değer 5 olduğu için oran *6/5 = 1,2*'dir; garanti edilen 2'nin çok altındadır. Algoritmanın seçtiği üç kenar, *(A, B)*, *(G, H)* ve *(I, J)*, ortak ucu olmayan kenarlardır. Ortak ucu olmayan kenarlardan oluşan kümeye **eşleştirme** (*matching*) denir. Oran kanıtının anahtarı bu eşleştirmedir.

11.5.5 Oran neden en fazla 2'dir?

Teorem 11.5 (Kenar seçen 2-yaklaşım)

Kenar seçen algoritmanın bulduğu C kümesi geçerli bir düğüm kapsamadır ve $|C| \le 2 \cdot OPT$'dir.

İspat. Geçerlilik kolaydır: bir kenar işlendiğinde ya uçlarından biri zaten kümededir ya da iki ucu birden eklenir. Her iki durumda da kenar kapsanır.

Oran için algoritmanın iki ucunu eklediği kenarlar M kümesi olsun. Bir kenar M'ye ancak iki ucu da henüz kümede değilken girer. Bu yüzden M'deki iki kenarın ortak ucu olamaz; M bir eşleştirmedir. Şimdi herhangi bir kapsama, örneğin en iyi kapsama C^* ele alınsın. C^*, M'deki her kenarı kapsamak zorundadır. M'deki kenarların ortak ucu olmadığı için bir düğüm bunlardan en fazla birini kapsayabilir. Demek ki C^*, M'deki her kenar için ayrı bir düğüm içerir: $OPT = |C^*| \ge |M|$. Algoritma tam olarak $|C| = 2\,|M|$ düğüm seçer. Birleştirince $|C| = 2\,|M| \le 2 \cdot OPT$ çıkar. □

Başka yaklaşımlar da vardır. **En yüksek dereceli düğümü seçen açgözlü** algoritma doğal görünür. Her adımda en çok kapsanmamış kenara değen düğümü alır. Şekil 11.20'deki çizgede, eşit derecelerde alfabetik sıra izlenirse, bu algoritma sırasıyla A, H, J, B, G'yi seçer ve 5 düğümle en iyiyi bulur.

Ancak bu algoritmanın oranı yalnız *O(log n)* ile sınırlanabilir ve oranın gerçekten logaritmik büyüdüğü çizgeler kurulabilir. İyi bir sezgiseldir, ama sabit oranlı bir yaklaşım algoritması değildir.

Doğrusal programlama ile yuvarlama ikinci bir 2-yaklaşım verir. Her düğüme $x_v \in \{0, 1\}$ değişkeni atanır ve her kenar için $x_u + x_v \geq 1$ kısıtı yazılır; $\Sigma\, x_v$ en küçüklenir. Tam sayı koşulu kaldırılıp $0 \leq x_v \leq 1$ yazılırsa problem bir doğrusal programa, yani **LP gevşetmesine** dönüşür ve polinom sürede çözülür. Gevşetmenin değeri *OPT*'tan küçük ya da ona eşittir. Ardından $x_v \geq 1/2$ olan düğümler seçilir. Her kenar kısıtında iki değişkenden en az biri *1/2*'ye ulaştığı için sonuç geçerli bir kapsamadır. Seçilen her düğümün değeri en az *1/2* olduğu için eleman sayısı gevşetme değerinin en fazla iki katıdır.

2'den belirgin biçimde iyi sabit bir oran bilinmiyor. Benzersiz Oyunlar Varsayımı (*Unique Games Conjecture*) doğruysa her $\varepsilon > 0$ için $2 - \varepsilon$ oranlı polinom zamanlı bir algoritmanın olmadığı gösterilmiştir (Khot ve Regev, 2008). Basit görünen kenar seçen algoritma, bu varsayım altında en iyi olası sabit orana ulaşır.

> **Sık yapılan hata**
>
> Kenarın yalnız bir ucunu eklemek. Bu, algoritmayı daha az savurgan gösterir; ama yıldız çizgede oran *n*'ye çıkar. İki ucu da eklemek, en iyi çözümle karşılaştırma yapılabilmesinin koşuludur: eşleştirme kanıtı ancak seçilen kenarların ortak ucu olmadığında işler. Tersine, iki ucu eklenen kenarlar bir eşleştirme olmasaydı alt sınır kurulamazdı.

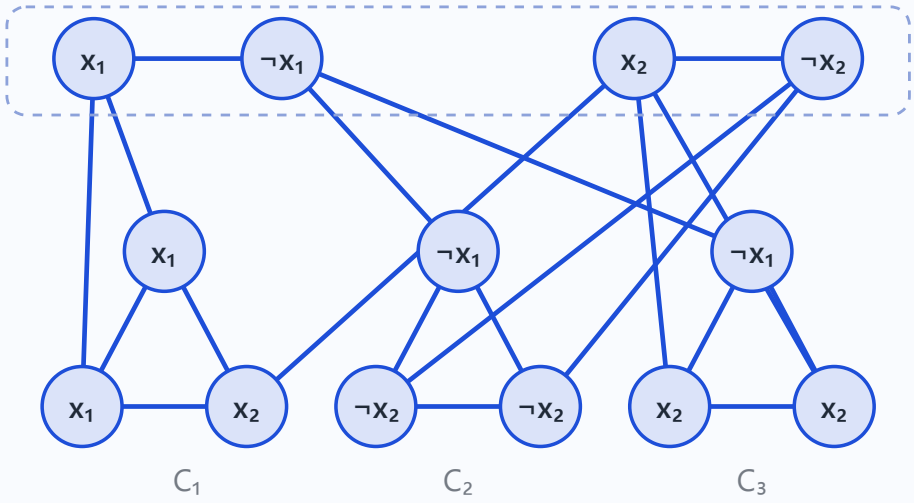
11.5.6 NP-tamlık: 3-SAT'tan indirgeme

Düğüm kapsamanın karar sürümünün NP-tam olduğu bir indirgemeyle gösterilir. İndirgeme, bilinen bir NP-tam problemin her örneğini polinom sürede düğüm kapsamanın bir örneğine dönüştürür ve iki örneğin yanıtı aynı olur. Kaynak problem olarak 3-SAT kullanılır. 3-SAT'ta bir mantık formülü, her biri üç değişmezden (*literal*) oluşan tümcelerin “ve” ile bağlanmasıdır; her tümce değişmezlerin “veya” ile bağlanmasıdır. Değişmez bir değişken ya da onun değilidir. Soru, formülü doğru yapan bir değer ataması olup olmadığıdır.

İndirgeme, *l* değişkenli ve *m* tümceli bir formülden şu çizgeyi kurar:

- Her x_i değişkeni için iki düğüm, x_i ve $\neg x_i$, ve aralarında bir kenar. Bu kenarı kapsamak için iki düğümden biri seçilmelidir; seçim değişkenin değerini belirler.
- Her tümce için üç düğümlü bir üçgen; düğümler tümcenin değişmezleriyle etiketlenir. Bir üçgeni kapsamak için üç düğümünden en az ikisi seçilmelidir.
- Her üçgen düğümü, aynı etiketli değişken düğümüne bir kenarla bağlanır.

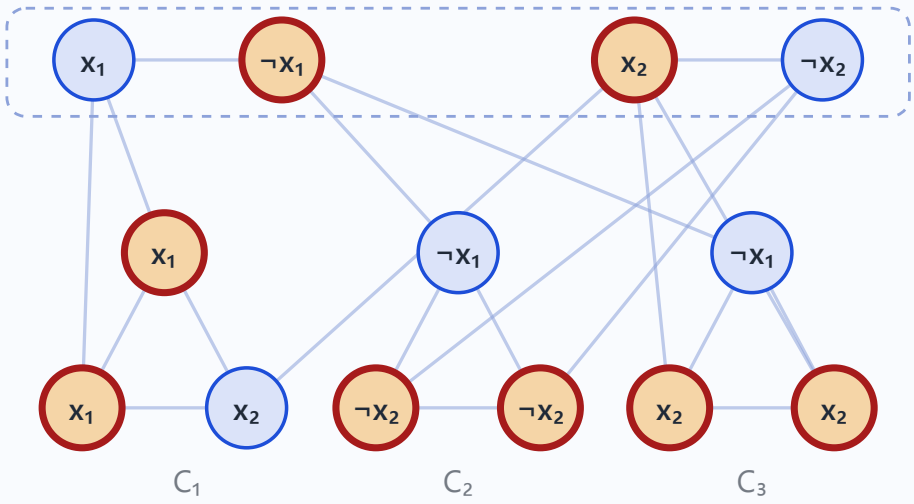
Aranan kapsama boyutu $k = l + 2m$'dir: her değişken kenarından bir düğüm, her üçgenden iki düğüm. Çizgede $2l + 3m$ düğüm ve $l + 6m$ kenar vardır; kurulum polinom süredir.



Şekil 11.23: $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$ için kurulan çizge. Üst kutuda iki değişken kenarı, altta üç tümce üçgeni; $l = 2$, $m = 3$, $k = 8$.

Sunumdaki örnek formül $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$'dir. İki değişken ve üç tümce vardır; çizgede $4 + 9 = 13$ düğüm ve $2 + 9 + 9 = 20$ kenar bulunur. Aranan boyut $k = 2 + 2 \cdot 3 = 8$'dir.

Formülü sağlayan atama $x_1 = yanlış$, $x_2 = doğru$'dur. Kapsama bu atamadan şöyle kurulur. x_1 yanlış olduğu için değişken kenarından $\neg x_1$ seçilir; x_2 doğru olduğu için x_2 seçilir. Her tümcede doğru bir değişmez vardır. O değişmezin üçgen düğümü dışarıda bırakılır ve öbür iki düğüm seçilir. Dışarıda kalan düğümün bağlantı kenarını ise değişken tarafında seçilmiş düğüm kapsar. C_1'de doğru değişmez x_2'dir; iki x_1 düğümü seçilir. C_2'de doğru değişmez $\neg x_1$'dir; iki $\neg x_2$ düğümü seçilir. C_3'te doğru değişmez $\neg x_1$'dir; iki x_2 düğümü seçilir.



Şekil 11.24: Atamadan kurulan kapsama: değişken kenarlarından $\neg x_1$ ve x_2, her üçgenden doğru değişmezin dışındaki iki düğüm. 8 düğüm, 20 kenarın hepsini kapsar.

Sonuçta 8 düğüm seçilmiştir ve 20 kenarın hepsi kapsanır (Şekil 11.24). Doğrulama betiği bu çizgede en küçük kapsamanın gerçekten 8 olduğunu ve formülü sağlayan tek atamanın x_1 = *yanlış*, x_2 = *doğru* olduğunu bütün olasılıkları deneyerek doğrulamıştır.

Ters yön de geçerlidir. $k = l + 2m$ boyutlu bir kapsama, her değişken kenarından tam bir, her üçgenden tam iki düğüm içerir. Bu yapılar en az bir ve en az iki düğüm ister ve toplam tam olarak k'dir. Değişken tarafında seçilen düğümler bir atama tanımlar. Her üçgende dışarıda kalan düğümün bağlantı kenarı kapsanmış olmalıdır; bu yalnız değişken tarafındaki eşi seçilmişse mümkündür. Demek ki o değişmez atamada doğrudur ve tümce sağlanır. Sonuç: φ, ancak ve ancak çizgede $l + 2m$ boyutlu bir düğüm kapsama varsa sağlanabilirdir.

11.6 Sırt çantası

0/1 sırt çantası probleminde n eşya vardır. i. eşyanın değeri v_i, ağırlığı w_i'dir. Çantanın kapasitesi W'dir. Toplam ağırlığı W'yi aşmayan ve toplam değeri en büyük olan eşya kümesi aranır. Her eşya ya bütünüyle alınır ya da hiç alınmaz. Problem NP-zordur. Dinamik programlamayla $O(nW)$ sürede çözülür, ama bu süre W'nin sayısal değerine bağlıdır. W girdide $\log W$ bitle yazıldığı için süre girdinin boyutunda üsteldir.

11.6.1 Oran sıralı açgözlü kural

En doğal açgözlü kural, eşyaları birim ağırlık başına değerlerine göre sıralamaktır. Eşyalar v_i / w_i oranına göre azalan sıraya konur ve sırayla incelenir. Çantaya sığan eşya alınır, sığmayan atlanır. Sıralama baskın maliyettir; süre $O(n \log n)$'dir.

Eşya	Değer	Ağırlık	Oran	Karar
6	20	7	2,86	alındı (ağırlık 7)
3	8	4	2,00	sığmaz (7 + 4 = 11)
7	15	8	1,88	sığmaz
2	5	3	1,67	alındı (ağırlık 10)
5	10	6	1,67	sığmaz
1	3	2	1,50	sığmaz
4	6	5	1,20	sığmaz

Tablo 11.14: Kapasite $W = 10$ için oran sıralı açgözlü kural.

Tablo 11.14 sunumdaki yedi eşyalı örneği verir. En yüksek oranlı eşya 6'dır ($20/7 \approx 2{,}86$) ve alınır; çantada 7 kilogram vardır. Sıradaki eşya 3, 4 kilogramdır ve sığmaz. Eşya 7 de sığmaz. Eşya 2, 3 kilogramdır ve çantayı tam 10 kilograma tamamlar. Bundan sonra hiçbir eşya sığmaz. Açgözlü toplam

değer *20 + 5 = 25*'tir. Doğrulama betiği $2^7 = 128$ alt kümenin hepsini denemiştir; en iyi değer de 25'tir ve yine eşya 2 ile 6'dan oluşur. Bu örnekte açgözlü kural en iyiyi bulur. Eşya 2 ile eşya 5'in oranı eşittir (5/3 = 10/6); tabloda eşya 2 önce gelir.

Eşyalar bölünebilseydi, yani problem kesirli sırt çantası olsaydı, bu kural her zaman en iyi sonucu verirdi: sığmayan ilk eşyanın bir parçası alınır ve çanta tam dolar. 0/1 sürümünde ise kuralın hiçbir garantisi yoktur.

Örnek 11.4

Kapasite *W = 10*, iki eşya olsun: (değer 2, ağırlık 1) ve (değer 10, ağırlık 10). Oranlar 2 ve 1'dir. Açgözlü kural önce birinci eşyayı alır. Çantada 9 kilogram yer kalır ve ikinci eşya sığmaz. Açgözlü değer 2'dir. En iyi çözüm yalnız ikinci eşyayı alır ve 10 değerine ulaşır. Oran *10 / 2 = 5*'tir. İkinci eşyanın değeri ve ağırlığı *W* yapılırsa oran *W / 2* olur; *W* büyüdükçe sınırsız kötüleşir.

11.6.2 İyileştirilmiş kural: 2-yaklaşım

Kötü örneğin kaynağı tek bir değerli eşyadır. Oran sıralaması onu geride bırakır ve küçük eşyalar çantada yer kaplayınca o eşya artık sığmaz. Düzeltme basittir. Açgözlü değer, tek başına çantaya sığan en değerli eşyanın değeriyle karşılaştırılır ve büyük olan alınır:

```
sonuç = max( açgözlü değer, sığan en değerli tek eşya )
```

Kötü örnekte bu kural *max(2, 10) = 10* bulur ve en iyiye ulaşır. Genel garanti de 2'dir.

Teorem 11.6 (İyileştirilmiş açgözlü kural)

Ağırlığı *W*'yi aşan eşyalar baştan atılmış olsun. Açgözlü değer *G*, sığan en değerli tek eşyanın değeri v_{max} ise $\max(G, v_{max}) \geq OPT / 2$'dir.

İspat. Eşyalar oran sırasına göre *1, 2, ..., n* diye numaralansın. Sırayla alınırken çantaya sığmayan ilk eşya *k* olsun. Kesirli sırt çantasının en iyi değeri $v_1 + \dots + v_{k-1}$'e *k*. eşyanın bir parçasının eklenmesiyle bulunur. 0/1 çözüm de kesirli bir çözümdür; bu yüzden

$$OPT \leq v_1 + \dots + v_{k-1} + v_k$$

yazılabilir. Açgözlü kural ilk $k - 1$ eşyanın hepsini almıştır ve sonra sığan başka eşyalar da alabilir: $G \geq v_1 + \dots + v_{k-1}$. *k*. eşya tek başına sığdığı için $v_k \leq v_{max}$'tır. Birleştirince $OPT \leq G + v_{max} \leq 2 \cdot \max(G, v_{max})$ çıkar. Sığmayan hiçbir eşya yoksa, yani *k* bulunmuyorsa, açgözlü kural bütün eşyaları alır ve *G = OPT*'tur. □

Örnek 11.5

$W = 15$ ve eşyalar (değer, ağırlık) olarak E1 (30, 10), E2 (20, 8), E3 (18, 7), E4 (6, 4) olsun. Oranlar 3,00; 2,50; 2,57 ve 1,50'dir. Oran sırası E1, E3, E2, E4'tür. E1 alınır (10 kg). E3 sığmaz (17 kg), E2 sığmaz (18 kg), E4 sığar (14 kg). Açgözlü değer 36'dır. Sığan en değerli tek eşya E1'dir (30); iyileştirilmiş kural 36 bulur. En iyi çözüm E2 ile E3'ü alır: tam 15 kg ve 38 değer. Oran $38/36 \approx 1{,}06$'dır. Kanıttaki üst sınır da denetlenebilir: k burada E3'tür ve $OPT = 38 \le 30 + 18 = 48$'dir. Kesirli değer $30 + (5/7) \cdot 18 \approx 42{,}86$'dır ve o da 38'den büyüktür.

11.6.3 Tam polinom zamanlı yaklaşım şeması

Sırt çantası, 2'den çok daha iyi bir oranı kabul eden problemlerdendir. Her $\varepsilon > 0$ için en iyi değerin en az $(1 - \varepsilon)$'unu bulan ve süresi $O(n^3/\varepsilon)$ olan bir algoritma vardır. Bu, bir tam polinom zamanlı yaklaşım şemasıdır (FPTAS).

Fikir değerleri ölçeklemektir. Sırt çantasının değerlere göre kurulan bir dinamik programı vardır: her x değeri için "toplam değeri tam x olan en hafif eşya kümesinin ağırlığı" tablolanır. Tablonun boyu değerlerin toplamı kadardır ve süre $O(n \cdot \Sigma\, v_i)$ olur. Değerler büyükse bu süre büyüktür. FPTAS değerleri bir $K = \varepsilon \cdot v_{max} / n$ sayısına böler ve aşağı yuvarlar: $v'_i = \lfloor v_i / K \rfloor$. Ölçeklenmiş değerlerin her biri en fazla n/ε'dur ve toplamları en fazla n^2/ε'dur. Dinamik program bu küçük değerlerle $O(n \cdot n^2/\varepsilon) = O(n^3/\varepsilon)$ sürede çalışır ve ölçeklenmiş problemin en iyi çözümünü bulur.

Yuvarlamanın verdiği kayıp sınırlıdır. Her eşyanın değeri en fazla K kadar kaybedilir; en iyi küme en fazla n eşya içerdiği için toplam kayıp en fazla $nK = \varepsilon \cdot v_{max} \le \varepsilon \cdot OPT$'tur. Bu yüzden bulunan kümenin gerçek değeri en az $(1 - \varepsilon) \cdot OPT$'tur. Sunumdaki özet tablosu bu oranı en küçükleme biçimine benzeterek $1 + \varepsilon$ diye yazar; iki yazım küçük ε için aynı garantiyi anlatır.

11.7 Kuvvet yinelemesi

Şimdiye kadarki algoritmalar kombinatoryal problemlere yaklaşık çözümler üretti. Sayısal hesaplamada ise "yaklaşım" başka bir biçimde görülür: kesin değeri sonlu adımda bulmak yerine, her adımda ona biraz daha yaklaşan bir dizi üretilir. Bu davranışa **yakınsama** (*convergence*) denir. Yakınsamanın tipik örneği, bir matrisin en büyük özdeğerini bulan kuvvet yinelemesidir.

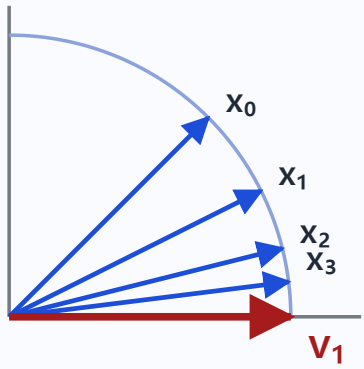
11.7.1 Baskın özdeğer ve özvektör

Kare bir A matrisi için $A \cdot v = \lambda \cdot v$ eşitliğini sağlayan sıfırdan farklı v vektörüne **özvektör**, λ sayısına da ona ait **özdeğer** denir. Özvektör, matrisle çarpılınca yönü değişmeyen, yalnız λ katına uzayan ya da kısalan vektördür. $n \times n$ bir matrisin en fazla n özdeğeri vardır. Mutlak değeri en büyük olana **baskın özdeğer** denir ve λ_1 ile gösterilir; ona ait özvektör v_1'dir.

Tanım 11.9

Kuvvet yinelemesi (*power iteration*), sıfırdan farklı bir x_0 vektörünü A matrisiyle tekrar tekrar çarpıp her adımda normalleştirerek baskın özvektör v_1'e ve baskın özdeğer λ_1'e yakınsayan yinelemeli yöntemdir.

Bütün özdeğerleri kesin olarak hesaplayan yöntemler $O(n^3)$ sürer. Matris binlerce satırlıysa bu kabul edilebilir; milyarlarca satırlıysa değildir. Kuvvet yinelemesi yalnız baskın özdeğeri ve özvektörü yaklaşık olarak verir, ama her adımı yalnız bir matris–vektör çarpımıdır. Bu çarpım yoğun bir matriste $O(n^2)$, seyrek bir matriste sıfır olmayan öğe sayısı kadar sürer. Dev boyutlu seyrek matrislerde kullanılabilen tek yöntem çoğunlukla budur.



Şekil 11.25: *$A = [[2, 0], [0, 1]]$* ve *$x_0 = (1, 1)$* için birim vektörler x_0, x_1, x_2, x_3 adım adım *$v_1 = (1, 0)$* yönüne döner.

En basit örnek köşegen bir matristir. *$A = [[2, 0], [0, 1]]$* matrisinin özdeğerleri 2 ve 1, özvektörleri *$(1, 0)$* ve *$(0, 1)$*'dir. *$x_0 = (1, 1)$* alınsın. Her çarpımda birinci bileşen 2 katına çıkar, ikinci bileşen aynı kalır: $A^k\, x_0 = (2^k, 1)$. Normalleştirilmiş vektörün yönü giderek *$(1, 0)$*'a döner (Şekil 11.25). İkinci bileşenin birinciye oranı 2^{-k}'dir. Hata her adımda yarıya iner. Bu oranın 10^{-3}'ün altına inmesi için $2^k > 1000$ gerekir; $2^9 = 512$, $2^{10} = 1024$ olduğundan $k = 10$ adım yeter.

11.7.2 Adımlar ve yakınsamanın nedeni

Algoritmanın adımları şöyledir:

1. Bir A matrisi belirlenir ve sıfırdan farklı bir başlangıç vektörü x_0 seçilir. Rastgele seçmek yaygındır.
2. Vektör matrisle çarpılır: $y = A \cdot x_k$.
3. Yeni vektörün normu, yani uzunluğu 1'e getirilir: $x_{k+1} = y \,/\, \|y\|$.
4. İkinci ve üçüncü adımlar, istenen sayıda ya da vektördeki değişim yeterince küçülene kadar tekrarlanır.

Sonuçta x_k, v_1'e yakınsar. Özdeğer tahmini için iki yol vardır. Normalleştirmeden önceki uzunluk $\|A \cdot x_k\|$, $|\lambda_1|$'e yakınsar. Simetrik matrislerde daha hızlı yakınsayan tahmin **Rayleigh bölümüdür**: $\|x_k\| = 1$ iken $\lambda \approx x_k^{T} A\, x_k$.

Yöntemin neden çalıştığı, başlangıç vektörünün özvektörler cinsinden yazılmasıyla görülür. A'nın özvektörleri bir taban oluştursun ve $x_0 = c_1 v_1 + c_2 v_2 + \dots + c_n v_n$ olsun. A ile her çarpım her bileşeni kendi özdeğeriyle çarpar:

$$A^k x_0 = \lambda_1^{\,k} \left[c_1 v_1 + c_2 (\lambda_2/\lambda_1)^k v_2 + \dots + c_n (\lambda_n/\lambda_1)^k v_n \right]$$

$|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$ ise köşeli parantezin içindeki $(\lambda_i/\lambda_1)^k$ çarpanlarının hepsi geometrik olarak sıfıra iner. Geriye $c_1 v_1$ kalır. Normalleştirme baştaki $\lambda_1^{\,k}$ çarpanını siler ve vektör v_1 yönünde birim vektöre yaklaşır. Sönmesi en yavaş olan bileşen v_2'ninkidir. Bu yüzden hata her adımda yaklaşık $|\lambda_2/\lambda_1|$ katına iner. Oran 0,1 ise her adımda bir doğru basamak kazanılır; oran 0,99 ise bir basamak için yaklaşık 230 adım gerekir.

Formül iki koşulu da açıkça gösterir. Birincisi $c_1 \neq 0$ olmalıdır; başlangıç vektörü v_1'e dik seçilirse v_1 bileşeni hiç yoktur ve kuramsal olarak ortaya çıkmaz. Pratikte yuvarlama hataları küçük bir v_1 bileşeni yaratır, ama yakınsama gecikir. Rastgele seçilen bir vektörün tam dik olma olasılığı sıfırdır. İkincisi $|\lambda_1| > |\lambda_2|$ olmalıdır. Mutlak değerce eşit iki özdeğer varsa, örneğin $\lambda_1 = 3$ ve $\lambda_2 = -3$, ikinci bileşen sönmez ve vektör iki yön arasında gidip gelebilir.

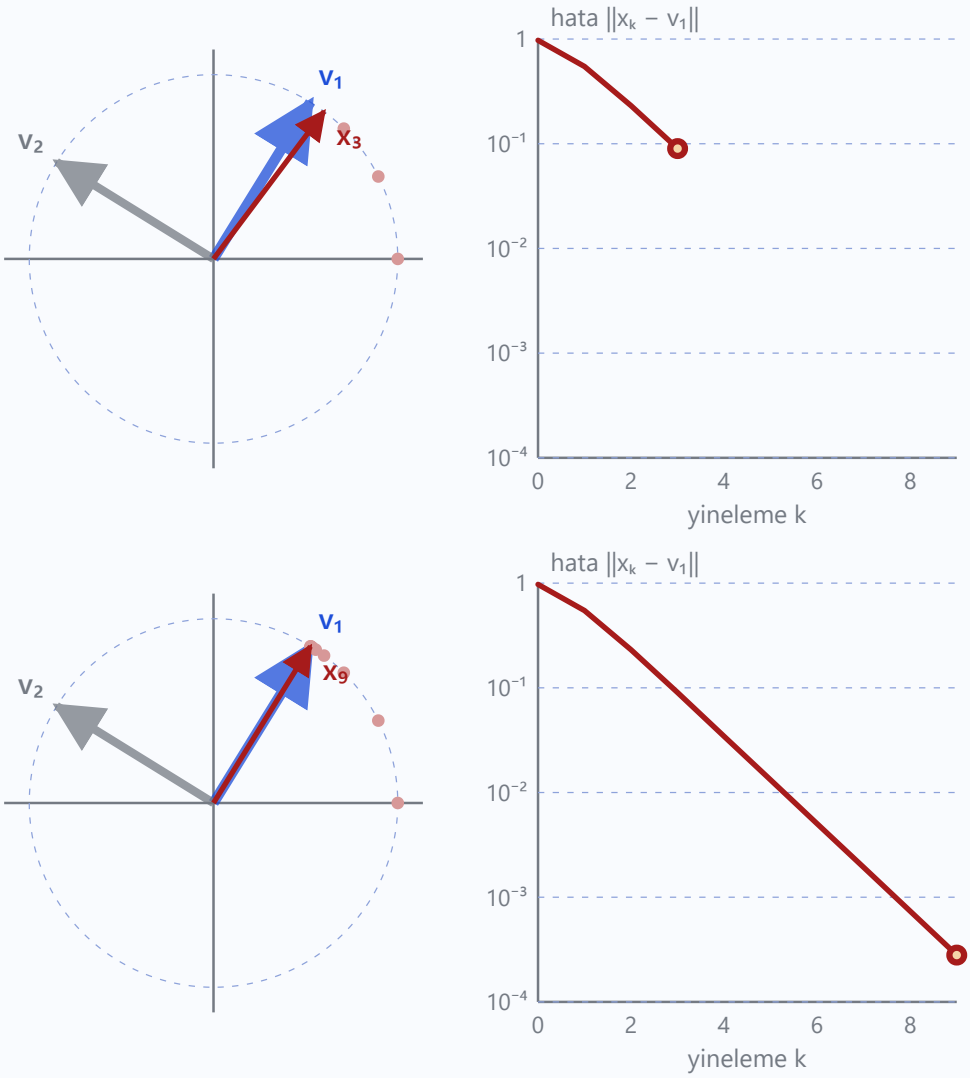
11.7.3 Adım adım yakınsama

Sunumdaki örnek simetrik bir matristir: $A = [[2, 1], [1, 3]]$. Karakteristik denklem $(2 - \lambda)(3 - \lambda) - 1 = \lambda^2 - 5\lambda + 5 = 0$'dır ve kökleri $\lambda_1 = (5 + \sqrt{5})/2 \approx 3{,}618$ ile $\lambda_2 = (5 - \sqrt{5})/2 \approx 1{,}382$'dir. Baskın özvektör, uzunluğu 1 olacak biçimde $v_1 \approx (0{,}5257;\ 0{,}8507)$'dir. Oran $\lambda_2/\lambda_1 \approx 0{,}382$'dir. Başlangıç vektörü $x_0 = (1, 0)$'dır ve hatası $\|x_0 - v_1\| \approx 0{,}974$'tür.

k	$A \cdot x_{k-1}$	x_k	λ tahmini	Hata
0	—	(1, 0)	—	0,974
1	(2,000; 1,000)	(0,894; 0,447)	3,000	0,5465
2	(2,236; 2,236)	(0,707; 0,707)	3,500	0,2313
3	(2,121; 2,828)	(0,600; 0,800)	3,600	0,0899
4	(2,000; 3,000)	(0,5547; 0,8321)	3,61538	$3{,}4 \cdot 10^{-2}$
5	(1,942; 3,051)	(0,5369; 0,8437)	3,61765	$1{,}3 \cdot 10^{-2}$
6	(1,917; 3,068)	(0,5300; 0,8480)	3,61798	$5{,}0 \cdot 10^{-3}$
7	(1,908; 3,074)	(0,5274; 0,8496)	3,61803	$1{,}9 \cdot 10^{-3}$
8	(1,904; 3,076)	(0,5264; 0,8503)	3,61803	$7{,}3 \cdot 10^{-4}$
9	(1,903; 3,077)	(0,5260; 0,8505)	3,618034	$2{,}8 \cdot 10^{-4}$

Tablo 11.15: $A = [[2, 1], [1, 3]]$, $x_0 = (1, 0)$ ile kuvvet yinelemesi. λ, Rayleigh bölümüdür; hata $\|x_k - v_1\|$.

Tablo 11.15 ilk dokuz adımı verir. İlk adımda $A \cdot (1, 0) = (2, 1)$'dir. Uzunluğu $\sqrt{5} \approx 2{,}236$'dır ve bölününce $x_1 = (0{,}894;\ 0{,}447)$ çıkar. Rayleigh bölümü $x_1^T A\, x_1 = 3{,}0$'dır. İkinci adımda $A \cdot x_1 = (2{,}236;\ 2{,}236)$ olur; vektör tam köşegen yöne döner ve $\lambda \approx 3{,}5$'tir. Üçüncü adımda $x_3 = (0{,}6;\ 0{,}8)$ ve $\lambda \approx 3{,}6$ bulunur. Sonraki adımlarda λ tahmini hızla 3,618'e oturur.



Şekil 11.26: Üstte üçüncü, altta dokuzuncu yineleme. Solda tahmin vektörü x_k'nın v_1'e yaklaşması, sağda log ölçekli hata grafiği. Hata grafiği bir doğruya oturur: yakınsama doğrusal (geometrik) hızdadır.

Hata sütunu yakınsama hızını açıkça gösterir. Ardışık iki hatanın oranı ilk adımlarda 0,42 ve 0,39'dur; dördüncü adımdan sonra $0{,}382 \approx \lambda_2/\lambda_1$'e oturur. Hata her adımda yaklaşık 0,38 katına iner. Log ölçekli grafikte bu davranış eğimi $\log_{10} 0{,}382 \approx -0{,}42$ olan bir doğrudur (Şekil 11.26). Sayısal analizde böyle bir yakınsamaya doğrusal ya da geometrik yakınsama denir. Her adım yaklaşık 0,42 ondalık basamak kazandırır; bir basamak için iki buçuk adım gerekir. Özdeğer tahmini özvektörden daha hızlı yakınsar: simetrik matriste Rayleigh bölümünün hatası, vektör hatasının karesiyle orantılıdır. Dokuzuncu adımda vektör hatası $2{,}8 \cdot 10^{-4}$ iken λ altı basamağa kadar doğrudur.

11.7.4 Java kodu ve 3 × 3 örnek

Sunumdaki Java kodu algoritmayı sabit sayıda adımla çalıştırır:

```
double[] kuvvet(double[][] a, int adim) {
  int n = a.length;
  double[] x = new double[n];
  Arrays.fill(x, 1.0);     // x₀ ≠ 0
  for (int k = 0; k < adim; k++) {
    double[] y = new double[n];
    for (int i = 0; i < n; i++)  // y = A·x
      for (int j = 0; j < n; j++)
        y[i] += a[i][j] * x[j];
    double norm = 0;
    for (double yi : y) norm += yi * yi;
    norm = Math.sqrt(norm);
    for (int i = 0; i < n; i++)  // ‖x‖ = 1
      x[i] = y[i] / norm;
  }
  return x;
}
```

Metot matrisi `a` ve adım sayısını `adim` olarak alır. Başlangıç vektörü bütün bileşenleri 1 olan vektördür. Bu seçim basittir ve çoğu matriste $c_1 \neq 0$ koşulunu sağlar. Dış döngünün her turunda önce $y = A \cdot x$ çarpımı iki iç içe döngüyle hesaplanır; `y[i]`, A'nın `i`. satırı ile `x`'in iç çarpımıdır. Sonra `y`'nin Öklid normu, bileşenlerin karelerinin toplamının karekökü olarak bulunur. Son olarak `x`, `y / norm` yapılır. Metot son `x`'i döndürür; `norm` değeri döndürülmez, ama λ_1 tahmini olarak kullanılabilir.

Her adımda çarpım n^2 çarpma ve toplama yapar; normalleştirme $O(n)$ sürer. `adim` adımlık toplam süre $O(adim \cdot n^2)$'dir. Bellek `x` ve `y` için $O(n)$'dir; matrisin kendisi değiştirilmez. Kod bir durma ölçütü kullanmaz. Pratik uygulamalarda döngü, ardışık iki `x` arasındaki fark bir eşiğin altına inince durdurulur.

Sunumdaki örnek matris şudur:

> A = [[0,511; 0,32; 0,12], [0,211; 0,62; 0,12], [0,311; 0,12; 0,16]], x_0 = (1, 1, 1)

Bu matrisin özdeğerleri yaklaşık 0,9008; 0,3 ve 0,0902'dir. Oran $\lambda_2/\lambda_1 = 0{,}3 / 0{,}9008 \approx 0{,}33$'tür; yakınsama hızlıdır. Matrisin bir özelliği de dikkat çeker. Birinci ve ikinci satırın ilk iki öğesinin toplamı aynıdır ($0{,}511 + 0{,}32 = 0{,}211 + 0{,}62 = 0{,}831$) ve üçüncü öğeleri eşittir. Bu yüzden x'in ilk iki bileşeni eşitse $A \cdot x$'inkiler de eşit kalır. $x_0 = (1, 1, 1)$ bu özelliği taşıdığı için yinelemenin her adımında $x_1 = x_2$'dir.

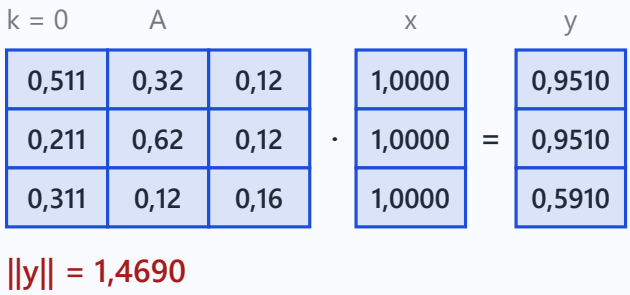
`k`	`y = A · x`	`norm` = $\|y\|$	**Yeni** `x`
0	(0,9510; 0,9510; 0,5910)	1,4690	(0,6474; 0,6474; 0,4023)
1	(0,5862; 0,5862; 0,3434)	0,8974	(0,6533; 0,6533; 0,3827)
2	(0,5888; 0,5888; 0,3428)	0,9005	(0,6539; 0,6539; 0,3807)

`k`	`y = A · x`	`norm` = ‖y‖	**Yeni** `x`
3	(0,5890; 0,5890; 0,3427)	0,9008	(0,6539; 0,6539; 0,3805)
4–7	(0,5891; 0,5891; 0,3427)	0,9008	(0,6539; 0,6539; 0,3805)

Tablo 11.16: `kuvvet(A, adim)` izlemesi, 3 × 3 örnek. Kodda döngü değişkeni `k` 0'dan başlar.

İlk adımda `y[0]`, birinci satırın `x` ile çarpımıdır: *0,511 · 1 + 0,32 · 1 + 0,12 · 1 = 0,951*. `y[1]` de aynı nedenle 0,951'dir, `y[2]` ise *0,311 + 0,12 + 0,16 = 0,591*'dir. Norm $\sqrt{(0{,}951^2 + 0{,}951^2 + 0{,}591^2)} \approx 1{,}4690$'dır. Bölününce $x = (0{,}6474;\ 0{,}6474;\ 0{,}4023)$ elde edilir. Bu vektörün uzunluğu 1'dir; sayılar artık ne büyür ne söner.

İkinci adımda $y = (0{,}5862;\ 0{,}5862;\ 0{,}3434)$'tür. Bileşenlerin oranı ilk adımdakine çok yakındır; yön az değişmiştir. Norm 0,8974'tür. Üçüncü adımda norm 0,9005'e, dördüncüde 0,9008'e çıkar ve orada kalır. Normun $\lambda_1 = 0{,}9008$'e yakınsadığı görülür. Vektör dördüncü adımdan sonra dört basamakta değişmez: $v_1 \propto (0{,}6539;\ 0{,}6539;\ 0{,}3805)$.



Şekil 11.27: İlk yinelemede $y = A \cdot x$ çarpımı ve norm: $y = (0{,}9510;\ 0{,}9510;\ 0{,}5910)$, $\|y\| = 1{,}4690$.

Sunumdaki izleme `adim = 3` ile durur ve $x \approx (0{,}6539;\ 0{,}6539;\ 0{,}3807)$ döndürür; üçüncü bileşen bir adım sonra 0,3805'e oturur. Adım sayısı istenen doğruluğa göre seçilmelidir: hatanın 10^{-d} katına inmesi için yaklaşık $d \,/\, \log_{10}(\lambda_1/|\lambda_2|)$ adım gerekir. Burada oran yaklaşık 3'tür ve her adım yarım basamaktan biraz az ($\log_{10} 3 \approx 0{,}48$) kazandırır.

Sık yapılan hata

Normalleştirmeyi atlamak. `x` her adımda normalleştirilmezse $A^k x_0$ doğrudan hesaplanmış olur. $|\lambda_1| > 1$ ise bileşenler $\lambda_1{}^k$ hızıyla büyür ve birkaç yüz adımda `double` türünün sınırını aşar. $|\lambda_1| < 1$ ise, 3 × 3 örnekteki gibi, bileşenler sıfıra iner ve anlamlı basamaklar kaybolur. Normalleştirme yalnız ölçeği düzeltir; yönü, yani aranan bilgiyi değiştirmez.

11.7.5 Sınırlar ve uygulamalar

Kuvvet yinelemesinin güçlü yanları sadeliğinden gelir. Yalnız matris–vektör çarpımı kullanır. Matrisin kendisini değiştirmez ve ek bellek olarak yalnız birkaç vektör, yani $O(n)$ yer ister. Bu yüzden milyarlarca satırı olan seyrek matrislere de uygulanabilir.

Sınırları da açıktır. Yalnız baskın özdeğeri bulur. $|\lambda_2|$, $|\lambda_1|$'e yakınsa çok yavaş yakınsar; eşitse yakınsamayabilir. Başlangıç vektörü v_1'e dik seçilirse ($c_1 = 0$) kuramsal olarak başarısız olur.

Yöntemin en ünlü uygulaması Google'ın **PageRank** hesabıdır. Bir sayfanın önemi, ona bağlantı veren sayfaların önemine göre belirlenir; bu tanım bir özvektör denklemidir. Sayfaların önem değerleri, web bağlantı matrisinin baskın özvektörüdür. Matrisin milyarlarca satırı vardır ama çok seyrektir ve kuvvet yinelemesi birkaç düzine adımda yeterli doğruluğa ulaşır. Türevleri de yaygındır: ters yineleme en küçük özdeğeri, kaydırmalı yineleme belirli bir değere en yakın özdeğeri bulur; Lanczos ve Arnoldi yöntemleri birkaç özdeğeri birlikte hesaplar. Temel bileşenler çözümlemesinde (PCA) ilk temel bileşen de bu yolla bulunabilir.

11.8 Karşılaştırma ve sık yapılan hatalar

Tablo 11.17 notta incelenen algoritmaları problem, oran ve süre bakımından yan yana koyar.

Problem	Algoritma	Oran	Süre
TSP (genel)	En yakın komşu (sezgisel)	garanti yok	$O(n^2)$
Metrik TSP	MST + önce-kök + kısayol / Christofides	2 / 1,5	$O(n^2)$ / $O(n^3)$
TSP	2-opt yerel arama	garanti yok	tarama başına $O(n^2)$
Makine çizelgeleme	Liste çizelgeleme / LPT	$2 - 1/m$ / $4/3 - 1/(3m)$	$O(n \log m)$ / $O(n \log n)$
Düğüm kapsama	Kenar seç, iki ucu ekle	2	$O(V + E)$
0/1 sırt çantası	*max*(oran sıralı, en iyi tek eşya) / FPTAS	2 / $1 + \varepsilon$	$O(n \log n)$ / $O(n^3/\varepsilon)$
Baskın özdeğer	Kuvvet yinelemesi	hata $\propto \lvert\lambda_2/\lambda_1\rvert^k$	adım başına $O(n^2)$

Tablo 11.17: İncelenen algoritmaların karşılaştırması.

Notun çeşitli yerlerinde ele alınan sık hatalar şunlardır:

- **Sezgiseli yaklaşım algoritması sanmak.** En yakın komşu ve 2-opt iyi sonuç verir, ama oran garantisi yoktur. Yaklaşım algoritması kanıtlanmış bir ρ ister.
- **Oranı ters yazmak.** En küçüklemede $C \le \rho \cdot C^*$, en büyüklemede $C \ge C^* / \rho$. ρ her zaman en az 1'dir.
- **Kapsamada tek uç eklemek.** Kenarın yalnız bir ucunu eklemek yıldız çizgede n kat kötü sonuç verebilir; 2-yaklaşım iki ucu da ekler.
- **Üçgen eşitsizliğini unutmak.** MST tabanlı 2-yaklaşımın kısayol adımı yalnız metrik TSP'de yolu uzatmaz.

- **Metasezgiseli kesin sanmak.** Benzetimli tavlama ve genetik algoritma en iyiye ulaşmayı garanti etmez; farklı tohum ve parametrelerle birkaç kez çalıştırılıp en iyi sonuç alınır.
- **Normalleştirmeyi atlamak.** Kuvvet yinelemesinde `x` her adımda normalleştirilmezse sayılar taşar ($|\lambda_1| > 1$) ya da sıfıra iner.

11.9 Özet

- **Yaklaşım algoritması:** NP-zor bir en iyileme problemi için polinom sürede çalışır ve her girdide $\max(C/C^*, C^*/C) \le \rho$ koşulunu sağlar; en küçüklemede $C \le \rho \cdot C^*$, en büyüklemede $C \ge C^*/\rho$.
- **Yöntem türleri:** kesin algoritma, yaklaşım algoritması, PTAS, FPTAS (süre $1/\varepsilon$'da da polinom), sezgisel ve metasezgisel (garanti yok).
- **TSP:** kaba kuvvet $(n-1)!/2$ tur, Held–Karp $O(n^2\,2^n)$. Genel TSP'de P ≠ NP ise sabit oranlı yaklaşım yoktur. En yakın komşu $O(n^2)$'dir ama oranı $\Theta(\log n)$ olabilir.
- **Metrik TSP:** $w(T) \le OPT$ ve kısayollu tur $\le 2 \cdot w(T)$; yedi şehirde 287,7 / 282,5. Christofides 1,5 oranını verir.
- **2-opt:** kazanç $\Delta = d(a,c) + d(b,d) - d(a,b) - d(c,d)$; yerel optimumda kesişen kenar kalmaz; 28 şehirde 48 hamleyle 1269,8'den 430,8'e. Metasezgiseller yerel optimumdan kaçabilir ama garanti vermez.
- **Makine çizelgeleme:** liste çizelgeleme $2 - 1/m$, LPT $4/3 - 1/(3m)$; alt sınırlar $\Sigma p/m$ ve $\max p$; iki sınır da sıkıdır.
- **Düğüm kapsama:** kapsanmamış bir kenarın iki ucu eklenir; seçilen kenarlar eşleştirmedir, $|C| = 2|M| \le 2 \cdot OPT$, süre $O(V+E)$. 3-SAT indirgemesinde $k = l + 2m$.
- **Sırt çantası:** $\max$(oran sıralı açgözlü, en değerli tek eşya) 2-yaklaşımdır; değer ölçekleme FPTAS verir.
- **Kuvvet yinelemesi:** $x_{k+1} = A\,x_k / \|A\,x_k\| \to v_1$; hata her adımda yaklaşık $|\lambda_2/\lambda_1|$ katına iner; koşullar $c_1 \ne 0$ ve $|\lambda_1| > |\lambda_2|$.

11.10 Kavram listesi

yaklaşım algoritması *approximation algorithm* · yaklaşım oranı *approximation ratio* · yaklaşım şeması *approximation scheme (PTAS / FPTAS)* · sezgisel *heuristic* · metasezgisel *metaheuristic* · gezgin satıcı problemi *travelling salesman problem (TSP)* · Hamilton çevrimi *Hamiltonian cycle* · üçgen eşitsizliği *triangle inequality* · minimum kapsayan ağaç *minimum spanning tree (MST)* · en yakın komşu *nearest neighbor* · yerel arama *local search* · yerel optimum *local optimum* · benzetimli tavlama *simulated annealing* · liste çizelgeleme *list scheduling* · en büyük yük *makespan* · en uzun iş önce *longest processing time first (LPT)* · düğüm kapsama *vertex cover* · eşleştirme *matching*

indirgeme *reduction* | LP gevşetmesi *LP relaxation* | 0/1 sırt çantası *0/1 knapsack* | kuvvet yinelemesi *power iteration* | yakınsama *convergence*

11.11 Kendini Dene

1. Bir 2-yaklaşım algoritması bir en küçükleme probleminde 50 maliyetli bir çözüm buluyor. En iyi maliyet hangi aralıktadır? Aynı sonuç bir en büyükleme probleminde bulunsaydı aralık ne olurdu?

   Cevap

   En küçüklemede $50 \le 2 \cdot OPT$ olduğundan $OPT \ge 25$'tir; en iyi çözüm bulunan çözümden kötü olamayacağı için $OPT \le 50$'dir. Aralık $[25, 50]$'dir. En büyüklemede $50 \ge OPT / 2$ olduğundan $OPT \le 100$'dür ve $OPT \ge 50$'dir. Aralık $[50, 100]$'dür.

2. Dört şehrin uzaklık matrisi satır satır 0: (0 1 15 6), 1: (1 0 7 3), 2: (15 7 0 2), 3: (6 3 2 0) olsun. En yakın komşu 0'dan başlayınca hangi turu bulur? En iyi tur nedir? Matris metrik midir?

   Cevap

   0'dan en yakın şehir 1'dir (1). 1'den ziyaret edilmemişler 2 (7) ve 3 (3); 3 seçilir. 3'ten tek aday 2'dir (2). 2'den 0'a dönülür (15). Tur $0 \to 1 \to 3 \to 2 \to 0$, uzunluk $1 + 3 + 2 + 15 = 21$'dir. Dört şehirde üç farklı tur vardır: $0 \to 1 \to 2 \to 3 \to 0$ (16), $0 \to 1 \to 3 \to 2 \to 0$ (21) ve $0 \to 2 \to 1 \to 3 \to 0$ (31). En iyisi 16'dır. Matris metrik değildir: $d[0][2] = 15 > d[0][1] + d[1][2] = 8$. Ucuz seçimler pahalı dönüş kenarını sona bırakmıştır.

3. Noktalar $a\ (0, 0)$, $b\ (3, 4)$, $c\ (6, 0)$, $d\ (3, -4)$ ve turda (a, b) ile (c, d) kenarları var. 2-opt kazancı nedir; hamle yapılır mı? Kenarlar birbirini kesiyor mu?

   Cevap

   $d(a, b) = d(c, d) = 5$, $d(a, c) = 6$, $d(b, d) = 8$'dir. $\Delta = 6 + 8 - 5 - 5 = 4 > 0$; hamle turu uzatır ve yapılmaz. İki kenar kesişmez: (a, b) eksenin üstünde, (c, d) altındadır. Kesişmeyen kenar çiftlerinde 2-opt kazancı pozitif olabilir; geometrik sonuç yalnız kesişen kenarlar için bir iyileşme garanti eder.

4. Süreleri 1, 2, 3, 4, 5, 6 olan işler $m = 3$ makineye bu sırayla liste çizelgelemeyle veriliyor. Makine

yükleri ve C_{max} nedir? LPT ne bulur? En iyi değer kaçtır?

Cevap

Liste: 1, 2, 3 üç makineye gider (yükler 1, 2, 3). 4 en az yüklü birinci makineye (5), 5 ikinciye (7), 6 üçüncüye (9) gider. Yükler 5, 7, 9; $C_{max} = 9$. LPT sırası 6, 5, 4, 3, 2, 1'dir: 6, 5, 4 üç makineye; 3 en az yüklü üçüncüye (7), 2 ikinciye (7), 1 birinciye (7). Yükler 7, 7, 7. Toplam iş 21 olduğundan alt sınır 7'dir; LPT en iyiyi bulur. Liste çizelgelemenin oranı $9/7 \approx 1{,}29$'dur ve $2 - 1/3 \approx 1{,}67$ sınırının altındadır.

5. Altı düğümlü $1–2–3–4–5–6–1$ çevriminin kenarları $(1,2), (2,3), (3,4), (4,5), (5,6), (6,1)$ sırasıyla kenar seçen 2-yaklaşıma veriliyor. Kaç düğüm seçilir? Kenarlar $(2,3), (5,6), (1,2), (3,4), (4,5), (6,1)$ sırasıyla verilseydi sonuç ne olurdu? En iyi değer nedir?

Cevap

İlk sırada $(1,2)$, $(3,4)$ ve $(5,6)$ seçilir; sonuç 6 düğümdür. İkinci sırada $(2,3)$ ve $(5,6)$ seçilir; kalan dört kenarın birer ucu kapsamadadır ve sonuç $\{2, 3, 5, 6\}$, 4 düğümdür. En iyi kapsama 3 düğümdür, örneğin $\{1, 3, 5\}$. Oranlar 2 ve $4/3$'tür. Kenar sırası sonucu değiştirir, ama garanti her sırada geçerlidir.

6. $W = 50$ ve eşyalar (değer, ağırlık) olarak (60, 10), (100, 20), (120, 30) olsun. Oran sıralı açgözlü kural, iyileştirilmiş kural ve en iyi 0/1 çözüm ne bulur? Kesirli sürümün değeri nedir?

Cevap

Oranlar 6, 5 ve 4'tür. Açgözlü kural (60, 10) ve (100, 20)'yi alır (30 kg); (120, 30) sığmaz (60 kg). Açgözlü değer 160'tır. Sığan en değerli tek eşya 120'dir; iyileştirilmiş kural $\max(160, 120) = 160$ bulur. En iyi 0/1 çözüm (100, 20) ve (120, 30)'u alır: tam 50 kg, 220 değer. Oran $220/160 = 1{,}375 \le 2$'dir. Kesirli sürümde ilk iki eşya ve üçüncünün $2/3$'ü alınır: $60 + 100 + 80 = 240$.

7. $A = [[2, 1], [1, 2]]$ ve $x_0 = (1, 0)$ için kuvvet yinelemesinin ilk adımını yapınız. Özdeğerler 3 ve 1 olduğuna göre hatanın 10^{-6} katına inmesi için yaklaşık kaç adım gerekir?

Cevap

$y = A \cdot x_0 = (2, 1)$, $\|y\| = \sqrt{5} \approx 2{,}2361$, $x_1 = (0{,}8944; 0{,}4472)$. Rayleigh bölümü $x_1^{T} A x_1 = 2{,}8$'dir; gerçek değer 3'tür. Hata her adımda yaklaşık $1/3$ katına iner. $(1/3)^k < 10^{-6}$ için $k > 6 / \log_{10} 3 \approx 12{,}6$, yani 13 adım gerekir.

8. Bütün uzaklıkları 1 ya da 2 olan bir TSP örneği metrik midir? Bu örneğe MST tabanlı 2-yaklaşım

uygulanabilir mi?

Cevap

Evet, metriktir. Her i, j, k için $d_{ij} \le 2$'dir; ayrıca $d_{ik} + d_{kj} \ge 1 + 1 = 2$'dir. Üçgen eşitsizliği her zaman sağlanır. Bu yüzden MST tabanlı algoritma uygulanabilir ve $2 \cdot OPT$ garantisi geçerlidir. Hamilton çevrimi kanıtındaki kurgu ise ağır kenarı $\rho n + 1$ yaptığı için metrik değildir; ağırlıklar 1 ve 2 ile sınırlanınca o kanıt çalışmaz.

11.12 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 35: düğüm kapsama, metrik TSP ve küme kapsama için yaklaşım algoritmaları, Hamilton çevrimi indirgemesinin tam kanıtıyla birlikte.
- D. P. Williamson ve D. B. Shmoys, *The Design of Approximation Algorithms* (2011): açgözlü, yerel arama, doğrusal programlama ve yaklaşım şemaları tekniklerini düzenli bir sırayla anlatır; kitabın elektronik sürümü yazarların sitesinde ücretsizdir.
- V. V. Vazirani, *Approximation Algorithms* (2001): kapsama, sırt çantası FPTAS'ı ve birincil–eşlenik yöntemleri kısa ve kesin kanıtlarla verir.
- G. H. Golub ve C. F. Van Loan, *Matrix Computations* (4. baskı, 2013): kuvvet yinelemesinin yakınsama analizi, ters ve kaydırmalı yineleme, Lanczos yöntemi.

11.13 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 35: Approximation Algorithms).
2. N. Christofides, “Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem”, Rapor 388, Carnegie Mellon Üniversitesi, 1976.
3. G. H. Golub, C. F. Van Loan, *Matrix Computations*, 4. baskı, Johns Hopkins University Press, 2013.
4. R. L. Graham, “Bounds on Multiprocessing Timing Anomalies”, *SIAM Journal on Applied Mathematics* 17(2), 1969.
5. A. R. Karlin, N. Klein, S. Oveis Gharan, “A (Slightly) Improved Approximation Algorithm for Metric TSP”, *Proceedings of the 53rd ACM Symposium on Theory of Computing (STOC)*, 2021.
6. M. Sipser, *Introduction to the Theory of Computation*, 3. baskı, Cengage, 2012 (3-SAT → düğüm kapsama indirgemesi).
7. V. V. Vazirani, *Approximation Algorithms*, Springer, 2001.
8. D. P. Williamson, D. B. Shmoys, *The Design of Approximation Algorithms*, Cambridge University Press, 2011.

KISIM V

Diğer Hesaplama Modelleri

Bölüm 12–14

KISIM V · BÖLÜM 12

Rastgele Algoritmalar

Bir algoritmanın her adımı girdiyle belirlenmek zorunda değildir. Algoritma yazı-tura atabilir, bir zar çevirebilir, diziden rastgele bir öğe seçebilir. Bu küçük değişiklik şaşırtıcı sonuçlar doğurur. Sıralı dizide yavaşlayan hızlı sıralama, pivotunu rastgele seçince hiçbir girdide sürekli yavaş kalmaz. Yüzlerce basamaklı bir sayının asal olup olmadığı, çarpanlarına hiç bakılmadan, milisaniyeler içinde ve yok denecek kadar küçük bir hata olasılığıyla anlaşılır. Bir çizgenin en zayıf kesimi, rastgele kenarlar birleştirilerek bulunur. Not önce rastgele algoritmayı, beklenen süreyi ve gereken olasılık araçlarını tanıtır. Ardından iki temel türü, Las Vegas ve Monte Carlo algoritmalarını ve hata olasılığının tekrarla nasıl küçültüldüğünü anlatır. Sonra örnekler tek tek incelenir. İlk örnekler Fisher–Yates karıştırması, saf karıştırmanın yanlılığı ve BogoSort'tur. Bunları rastgele pivotlu hızlı sıralama, beklenen karşılaştırma sayısı ve rastgele seçme izler. Son örnekler Monte Carlo yöntemiyle π tahmini, Miller–Rabin asallık testi ve Karger minimum kesim algoritmasıdır. Not, sözde rastgele sayı üreteçleriyle biter.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum12/src`

ÖĞRENME ÇIKTILARI

Kavram
Rastgele algoritmayı tanımlamak; beklenen çalışma süresini ortalama durum süresinden ayırmak; Las Vegas ve Monte Carlo algoritmalarının farkını açıklamak.

Karıştırma ve BogoSort
Fisher–Yates karıştırmasını yazmak ve her permütasyonu eşit olasılıkla ürettiğini kanıtlamak; saf karıştırmanın neden yanlı olduğunu saymayla göstermek; BogoSort'un beklenen $\Theta(n \cdot n!)$ süresini hesaplamak.

Rastgele QuickSort
Rastgele pivotlu bölümlemeyi adım adım izlemek; beklenen $O(n \log n)$ ve en kötü $O(n^2)$ süreyi gerekçelendirmek; beklenen karşılaştırma sayısını gösterge değişkenleriyle hesaplamak; rastgele seçmenin beklenen $O(n)$ süresini açıklamak.

Monte Carlo
π tahmini, Miller–Rabin asallık testi ve Karger minimum kesim algoritmasında hata olasılığını hesaplamak ve bağımsız tekrarlarla küçültmek.

12.1 Rastgelelik ve algoritmalar

12.1.1 Şansa dayanan kararlar

Algoritma denince çoğunlukla tamamen belirlenmiş bir adımlar dizisi akla gelir. Aynı girdi verilirse algoritma aynı adımları aynı sırayla atar ve aynı sonucu üretir. Bu tür algoritmalara **deterministik** (belirlenimci) algoritma denir. Bazı problemlerde ise algoritmanın kararlarına bilinçli olarak şans katmak işe yarar. Algoritma bir noktada yazı-tura atar, zar atar ya da bir dizinin öğelerinden birini rastgele seçer ve yoluna bu seçime göre devam eder.

Şans oyunları bu fikrin en tanıdık örneğidir. İki zarın atıldığı bir oyun düşünülsün. Toplam 7 ya da 11 gelirse oyuncu kazanır; gelmezse zarlar yeniden atılır. İki zarın 36 eşit olasılıklı sonucu vardır. Toplamın 7 olduğu sonuçlar (1, 6), (2, 5), (3, 4), (4, 3), (5, 2) ve (6, 1)'dir; sayıları 6'dır. Toplamın 11 olduğu sonuçlar (5, 6) ve (6, 5)'tir; sayıları 2'dir.

> *P(7) = 6/36, P(11) = 2/36, P(kazanma) = 8/36 = 2/9 ≈ 0,22*



Şekil 12.1: İki zar: 3 ve 4, toplam 7. Bu atışta oyuncu kazanır.

Bir atışta kazanma olasılığı yaklaşık 0,22'dir. Oyuncu kazanana kadar kaç kez atış yapar? Bu sayı her oyunda farklıdır: bazen ilk atışta kazanılır, bazen on atış gerekir. Ama ortalaması bellidir. Her atışın kazanma olasılığı aynı ve atışlar birbirinden bağımsız olduğu için beklenen atış sayısı *1 / (2/9) = 4,5*'tir. Bu sonuç geometrik dağılımdan gelir. Sabit tohumla (`random.Random(42)`) oynatılan 200.000 oyunda ortalama atış sayısı 4,502 çıktı; formülle uyumludur.

Zar oyunundaki iki gözlem bütün not boyunca tekrar eder. Birincisi, tek bir çalıştırmanın ne kadar süreceği önceden bilinmez; bilinen, sürenin beklenen değeridir. İkincisi, beklenen değer bir olasılık hesabıyla bulunur ve uzun bir benzetimle doğrulanabilir.

12.1.2 Rastgele algoritma ve beklenen süre

Tanım 12.1

Rastgele algoritma (*randomized algorithm*), girdiye ek olarak rastgele sayılar kullanan ve bazı kararlarını bu sayılara göre veren algoritmadır. Aynı girdiyle yapılan iki çalıştırma farklı adımlar izleyebilir; çalışma süresi ve bazen çıktının kendisi çalıştırmadan çalıştırmaya değişir.

Şekil 12.2 bu yapıyı gösterir. Algoritmanın iki girişi vardır: problemin girdisi ve bir rastgele bit kaynağı. Girdi aynı kalsa bile rastgele bitler değişir; bu yüzden algoritmanın izlediği yol da değişebilir.



Şekil 12.2: Rastgele algoritma: girdi ve rastgele bitler algoritmaya girer, çıktı üretilir. Algoritmanın kararlarında şans payı vardır.

Deterministik bir algoritmanın sabit bir girdi üzerindeki süresi tek bir sayıdır. Rastgele algoritmada ise aynı girdi için süre bir **rastgele değişkendir**: her çalıştırmada farklı bir değer alabilir. Bu yüzden analizde tek bir sayı yerine sürenin **beklenen değeri** $E[T(n)]$ incelenir. Burada önemli bir ayrıntı vardır. Beklenen değer algoritmanın kendi yazı-turaları üzerinden alınır, girdi üzerinden alınmaz. Girdi sabittir ve en kötüsü bile olabilir; olasılık yalnız algoritmanın yaptığı rastgele seçimlerden gelir.

Tanım 12.2

Bir rastgele algoritmanın **beklenen çalışma süresi**, büyüklüğü n olan her girdi için algoritmanın rastgele seçimleri üzerinden alınan beklenen süredir. Algoritmanın beklenen süresi denince bu değerlerin bütün girdiler üzerindeki en büyüğü anlaşılır:

$$T_{bek}(n) = \max_{|x| = n} E[\, T(x) \,]$$

Bu tanım **ortalama durum** analiziyle karıştırılmamalıdır. Ortalama durum analizi deterministik bir algoritmaya uygulanır ve girdiler hakkında bir varsayım yapar: örneğin dizinin her sıralamasının eşit olasılıklı olduğunu varsayar. Varsayım tutmazsa sonuç da geçersiz olur. Gerçek hayatta veriler çoğu

zaman sıralı ya da sıralıya yakın gelir, yani eşit olasılık varsayımı sık sık bozulur. Beklenen süre ise hiçbir girdi varsayımı yapmaz. Her girdi için ayrı ayrı geçerlidir; en kötü girdi için bile.

Örnek 12.1

Pivotu her zaman alt dizinin ilk öğesi olarak seçen hızlı sıralama, girdi rastgele sıralıysa ortalama $O(n \log n)$ sürede çalışır. Bu bir ortalama durum sonucudur. Aynı algoritmaya 10 öğeli sıralı bir dizi verilirse her çalıştırmada tam 45 karşılaştırma yapılır, çünkü bu girdi her seferinde en kötü durumu doğurur. Pivotu rastgele seçen sürüm ise aynı sıralı dizide ortalama yaklaşık 24,4 karşılaştırma yapar ve bu değer girdinin sırasına bağlı değildir. Sıralı dizi, ters sıralı dizi ya da herhangi bir dizilim için beklenen değer aynıdır. 24,4 değeri, beklenen karşılaştırma sayısı formülü $2(n + 1)H_n - 4n$'den gelir.

12.1.3 Neden rastgelelik?

Rastgelelik ilk bakışta bir belirsizlik kaynağı gibi görünür. Yine de algoritma tasarımında dört önemli kazanç sağlar.

Kötü girdiye karşı koruma. Deterministik hızlı sıralama, pivotu ilk öğe seçiyorsa sıralı dizide $O(n^2)$ süreye düşer. Sıralı dizi ise uygulamada çok yaygındır. Bir saldırgan algoritmanın kurallarını biliyorsa en kötü girdiyi bilerek hazırlayabilir. Pivot rastgele seçildiğinde hiçbir girdi sürekli kötü değildir. Kötü durum ancak algoritmanın kendi seçimleri şanssız giderse ortaya çıkar ve bunun olasılığı çok küçüktür.

Basitlik. Rastgele algoritma çoğu zaman bilinen en iyi deterministik algoritmadan çok daha kısa ve anlaşılırdır. Karger'in minimum kesim algoritması birkaç satırlık bir döngüdür. Rastgele seçme algoritması da hızlı sıralamanın küçük bir değişikliğidir. Bu problemlerin deterministik çözümleri ise ya uzun ve incelikli ya da uygulamada yavaştır.

Hız. Miller–Rabin testi yüzlerce basamaklı sayıların asallığını milisaniyeler içinde belirler. Hata olasılığı istenen kadar küçültülebilir. Kriptografide anahtar üretimi bu hıza dayanır.

Örnekleme ve simetri kırma. Bazı işler rastgelelik olmadan ya çok zordur ya da imkânsızdır. Monte Carlo benzetimleri, büyük bir veriden rastgele örneklem alma ve dağıtık sistemlerde birbirinin aynısı olan süreçler arasından bir lider seçme bunlara örnektir. Aynı programı çalıştıran iki özdeş süreç deterministik kurallarla birbirinden ayrılamaz; rastgele bir sayı bu simetriyi bozar.

Bu kazançların bir bedeli vardır. Süre ya da doğruluk için artık kesin değil, olasılıksal bir güvence verilir. “Bu algoritma her zaman $O(n \log n)$'dir” denemez; “beklenen süresi $O(n \log n)$'dir” ya da “yanılma olasılığı 10^{-12}'den küçüktür” denir. Ayrıca algoritmanın iyi bir rastgele sayı üretecine ihtiyacı vardır. Üretecin kusurları analizdeki varsayımları bozar.

12.1.4 Olasılık araçları

Rastgele algoritmaların analizinde birkaç temel olasılık aracı tekrar tekrar kullanılır. Tablo 12.1 bunları özetler; ardından her biri kısa bir örnekle açıklanır.

Kavram	Tanım / kural	Örnek
Beklenen değer	$E[X] = \Sigma\, x \cdot P(X = x)$	bir zarın beklenen değeri 3,5
Doğrusallık	$E[X + Y] = E[X] + E[Y]$ (bağımsızlık gerekmez)	iki zarın toplamı: 7
Gösterge değişkeni	A olursa $X_A = 1$, olmazsa 0; $E[X_A] = P(A)$	QuickSort'ta "i ile j karşılaştırıldı mı?"
Geometrik dağılım	başarı olasılığı p olan bağımsız denemelerde ilk başarıya kadar beklenen deneme $1/p$	BogoSort: $p = 1/n! \to n!$ karıştırma
Bağımsız tekrar	her biri q olasılıkla başarısız k bağımsız denemenin hepsinin başarısız olma olasılığı q^k	Miller–Rabin: $(1/4)^k$
Markov eşitsizliği	$X \geq 0$ ise $P(X \geq c \cdot E[X]) \leq 1/c$	Las Vegas → Monte Carlo

Tablo 12.1: Notta kullanılan olasılık araçları.

Beklenen değer. Bir rastgele değişkenin alabileceği her değer, o değerin olasılığıyla çarpılıp toplanır. Bir zarın her yüzü 1/6 olasılıkla gelir: $E[X] = (1 + 2 + 3 + 4 + 5 + 6) / 6 = 3{,}5$. Beklenen değerin kendisi hiç gözlenmeyebilir; hiçbir zar 3,5 göstermez. Beklenen değer, çok sayıda deneme yapıldığında ortalamanın yaklaştığı değerdir.

Doğrusallık. İki rastgele değişkenin toplamının beklenen değeri, beklenen değerlerinin toplamıdır. İki zarın toplamı için $3{,}5 + 3{,}5 = 7$ bulunur; 36 sonucu tek tek toplamaya gerek kalmaz. Bu kuralın en güçlü yanı bağımsızlık istememesidir. Değişkenler birbirine bağlı olsa bile kural geçerlidir. Hızlı sıralamanın analizi tam olarak bu özelliğe dayanır: karşılaştırmalar birbirinden bağımsız değildir, ama beklenen toplam yine de tek tek beklenen değerlerin toplamıdır.

Gösterge değişkeni. Bir A olayı için X_A, olay gerçekleşirse 1, gerçekleşmezse 0 değerini alır. Beklenen değeri $1 \cdot P(A) + 0 \cdot (1 - P(A)) = P(A)$'dır. Bir şeyin kaç kez olduğu sayılacaksa her olası olay için bir gösterge yazılır, sayı göstergelerin toplamı olur, doğrusallıkla beklenen değer olasılıkların toplamına iner. Hızlı sıralamada "z_i ile z_j karşılaştırıldı mı?" sorusu böyle bir göstergedir.

Geometrik dağılım. Her denemenin başarı olasılığı p olsun ve denemeler bağımsız olsun. İlk başarıya kadar yapılan deneme sayısı X ise $P(X = k) = (1 - p)^{k-1}\, p$'dir ve $E[X] = 1/p$'dir. Bu sonuç kısa bir akıl yürütmeyle de görülür. İlk deneme her durumda yapılır. p olasılıkla iş biter; $1 - p$ olasılıkla süreç baştan başlar ve aynı beklenen değer yeniden harcanır: $E = 1 + (1 - p)\, E$, buradan $E = 1/p$. Zar oyununda $p = 2/9$ ve beklenen atış 4,5'tir.

Bağımsız tekrar. Bir deneme en çok q olasılıkla başarısız oluyorsa, birbirinden bağımsız k denemenin hepsinin başarısız olma olasılığı en çok q^k'dır. q 1'den küçük olduğu sürece bu değer k ile üstel olarak azalır. Monte Carlo algoritmalarının hatası bu yolla küçültülür.

Markov eşitsizliği. Negatif değer almayan bir rastgele değişken, beklenen değerinin c katını en çok $1/c$ olasılıkla aşar. Gerekçesi basittir: X en az $c \cdot E[X]$ olduğunda beklenen değere en az $c \cdot E[X] \cdot P(X \geq c \cdot E[X])$ katkı yapar. Bu katkı $E[X]$'i aşamaz. Eşitsizlik dağılım hakkında hiçbir şey bilmeden yalnız beklenen değerle bir kuyruk sınırı verir; bu yüzden kaba ama her yerde geçerlidir.

Sık yapılan hata

Doğrusallık için bağımsızlık aramak. $E[X + Y] = E[X] + E[Y]$ her zaman doğrudur. Bağımsızlık ancak çarpım için gerekir: $E[XY] = E[X]\,E[Y]$ eşitliği genel olarak yalnız bağımsız değişkenlerde geçerlidir. Hızlı sıralamanın karşılaştırma analizinde bağımlı göstergelerin toplanması bu yüzden sorun değildir.

12.2 Las Vegas ve Monte Carlo

12.2.1 İki tür rastgele algoritma

Rastgele algoritmalar, rastgeleliğin neyi etkilediğine göre iki gruba ayrılır. Birinci grupta sonuç her zaman doğrudur, yalnız süre şansa bağlıdır. İkinci grupta süre sınırlıdır, ama sonuç küçük bir olasılıkla yanlış olabilir. Bu gruplara kumarhaneleriyle ünlü iki şehrin adı verilmiştir.

Tanım 12.3

Las Vegas algoritması (*Las Vegas algorithm*) her zaman doğru sonuç verir; çalışma süresi rastgeledir ve beklenen süresi analiz edilir. **Monte Carlo algoritması** (*Monte Carlo algorithm*) ise önceden belirlenmiş bir süre sınırı içinde biter; sonucu küçük bir olasılıkla yanlış olabilir.

	Las Vegas	Monte Carlo
Doğruluk	her zaman doğru sonuç	küçük bir olasılıkla yanlış sonuç
Süre	rastgele; beklenen süre analiz edilir	sınırlı (deterministik üst sınır)
Rastgele olan	ne kadar süreceği	sonucun doğru olup olmadığı
Örnekler	rastgele QuickSort, QuickSelect, BogoSort, Fisher–Yates	Miller–Rabin, Karger minimum kesim, π tahmini

Tablo 12.2: Las Vegas ve Monte Carlo algoritmalarının karşılaştırması.

Rastgele pivotlu hızlı sıralama bir Las Vegas algoritmasıdır. Pivot ne seçilirse seçilsin, sonunda dizi doğru sıralanır. Şanssız pivotlar yalnız işi uzatır. BogoSort da Las Vegas'tır: durduğunda dizi kesinlikle sıralıdır, ama ne zaman duracağı belli değildir. Fisher–Yates karıştırmasının durumu biraz farklıdır.

Süresi her çalıştırmada tam $n - 1$ takastır; rastgele olan, üretilen permütasyondur. Bu algoritmada "doğru sonuç" tek bir dizi değil, istenen olasılık dağılımıdır ve bu dağılım her zaman tam olarak sağlanır. Bu yüzden o da Las Vegas grubuna konur.

Miller–Rabin testi ise Monte Carlo'dur. Tur sayısı baştan bellidir ve her tur sınırlı sayıda çarpma yapar. Ama "asal" yanıtı küçük bir olasılıkla yanlış olabilir. Karger algoritması da sabit sayıda büzme yapar ve bazen minimum olmayan bir kesim döndürür. π tahmini istenen sayıda nokta atar ve gerçek değere yalnız yaklaşır.

12.2.2 Tek yönlü ve iki yönlü hata

Monte Carlo algoritmaları hatanın yönüne göre de ayrılır. Bir karar probleminde, yani yanıtı "evet" ya da "hayır" olan bir problemde, algoritma yanıtlardan yalnız birinde yanılabiliyorsa hata **tek yönlüdür**. Miller–Rabin bunun tipik örneğidir. Test "bileşik" derse bu kesindir, çünkü sayının asal olmadığını gösteren somut bir kanıt bulmuştur. Yalnız "asal" yanıtı yanlış olabilir. Tek yönlü hatanın büyük bir avantajı vardır: tekrarlar sonucunda tek bir "bileşik" yanıtı bile kesin karar verir.

Hata iki yönde de olabiliyorsa **iki yönlüdür**. π tahmini gibi sayısal benzetimler gerçek değerin hem üstünde hem altında sapabilir. Bu tür algoritmalarda hata payı örnek sayısı arttıkça küçülür, ama tek bir çalıştırmanın sonucunun hangi yönde saptığı bilinmez. Karger algoritmasının hatası da tek yönlüdür, ama yönü farklıdır: bulunan kesim hiçbir zaman minimumdan küçük olamaz, yalnız ondan büyük olabilir. Bu yüzden birkaç çalıştırmanın en küçüğü alınır.

12.2.3 Hata olasılığının küçültülmesi

Monte Carlo algoritmasının hata olasılığı tek çalıştırmada büyük olabilir. Ama algoritma bağımsız rastgele seçimlerle birçok kez çalıştırılırsa hata hızla küçülür. Bu tekniğe **tekrarla güçlendirme** (*amplification*) denir.

> **Teorem 12.1 (Tekrarla güçlendirme)**
>
> Tek yönlü hatalı bir Monte Carlo algoritmasının tek turda yanılma olasılığı en çok q olsun. Algoritma k kez bağımsız olarak çalıştırılır ve turlardan biri kesin yanıtı verirse o yanıt seçilir. Bu birleşik algoritmanın yanılma olasılığı en çok q^k'dır.

İspat. Birleşik algoritma yalnız bütün turlar yanıldığında yanılır. Turlar bağımsız rastgele seçimler kullandığı için bu olasılık tek tek yanılma olasılıklarının çarpımıdır ve en çok $q \cdot q \cdot \ldots \cdot q = q^k$'dır. □

Miller–Rabin'de bileşik bir sayı için bir turun yanılma olasılığı en çok $1/4$'tür. Yirmi turda yanılma olasılığı en çok $4^{-20} \approx 9{,}1 \cdot 10^{-13}$'e iner. Bu, bilgisayar donanımında kendiliğinden bir bit hatası oluşma olasılığından bile küçüktür. Şekil 12.3 sınırın tur sayısıyla nasıl düştüğünü logaritmik ölçekte gösterir: her yeni tur sınırı 4'e böler.



Şekil 12.3: Miller–Rabin hata sınırı 4^{-k}, logaritmik ölçekte. $k = 9$'da sınır $3{,}8 \cdot 10^{-6}$'dır; $k = 10$'da $9{,}5 \cdot 10^{-7}$'ye iner ve 10^{-6}'nın altına ilk kez düşer.

Örnek 12.2

Miller–Rabin'in hata olasılığını 10^{-9}'un altına indirmek için en az kaç tur gerekir? Koşul $4^{-k} < 10^{-9}$, yani $4^k > 10^9$'dur. $4^{14} = 268.435.456$ yetmez. $4^{15} = 2^{30} = 1.073.741.824$ ise 10^9'u aşar. En az 15 tur gerekir. Aynı hesap 10^{-6} için 10 tur verir. Sınır yarıya değil dörtte bire indiği için gereken tur sayısı, istenen ondalık basamak sayısının yaklaşık $\log_4 10 \approx 1{,}66$ katıdır.

Karger algoritmasında tek çalıştırmanın başarı olasılığı en az $2/(n(n-1))$'dir. Bu küçük bir sayıdır; 100 düğümlü bir çizgede yaklaşık $0{,}0002$'dir. Ama algoritma $n(n-1)/2 \cdot \ln n$ kez tekrarlanırsa hepsinin başarısız olma olasılığı en çok $1/n$'ye iner. Başarı olasılığı küçük de olsa, sıfırdan büyük ve bilinen bir alt sınırı varsa tekrar onu istenen düzeye çıkarır.

12.2.4 Türler arası dönüşüm

Las Vegas ve Monte Carlo algoritmaları birbirine dönüştürülebilir. İki dönüşüm de pratikte kullanılır.

Las Vegas'tan Monte Carlo'ya. Beklenen süresi $E[T]$ olan bir Las Vegas algoritması, $2 \cdot E[T]$ adım sonra durdurulur. Algoritma bu sürede bitmişse yanıtı doğrudur. Bitmemişse "yanıt yok" döndürülür. Markov eşitsizliğine göre $P(T \geq 2\,E[T]) \leq 1/2$'dir; yani yeni algoritma en çok 1/2 olasılıkla yanıtsız kalır. Süresi artık kesin olarak sınırlıdır. Sınır $c \cdot E[T]$ alınırsa yanıtsız kalma olasılığı en çok $1/c$'dir.

Örnek 12.3

Dört farklı öğeli bir dizide BogoSort'un karıştırma sayısı, başarı olasılığı $p = 1/24$ olan geometrik dağılıma uyar ve beklenen değeri 24'tür. Algoritma 48 karıştırmadan sonra durdurulsun. Markov eşitsizliği yanıtsız kalma olasılığı için 1/2 sınırını verir. Gerçek olasılık çok daha küçüktür: ilk 48 karıştırmanın hepsinin başarısız olma olasılığı $(23/24)^{48} \approx 0{,}130$'dur. Sabit tohumlu 100.000 denemelik bir benzetim de 0,129 verdi. Markov sınırı doğrudur ama kabadır, çünkü dağılım hakkında yalnız beklenen değeri kullanır.

Monte Carlo'dan Las Vegas'a. Monte Carlo algoritmasının yanıtı hızlıca doğrulanabiliyorsa algoritma, doğrulanan bir yanıt bulunana kadar tekrarlanır. Örneğin bir sayıyı çarpanlarına ayırmaya çalışan rastgele bir algoritmanın bulduğu çarpanlar çarpılarak kolayca denetlenir. Tek denemenin başarı olasılığı p ise geometrik dağılımdan beklenen deneme sayısı $1/p$'dir. Sonuç her zaman doğru olur, yalnız süre rastgeleleşir. $p = 1/3$ için beklenen tekrar 3'tür; tohumu 42 olan 100.000 denemelik benzetimde ortalama 2,994 çıktı.

> **Sık yapılan hata**
>
> **Doğrulanamayan yanıtı "doğru çıkana kadar" tekrarlamak.** Miller–Rabin'in "asal" yanıtı hızlıca doğrulanamaz; asallığın kısa bir kanıtı elde yoktur. Bu yüzden Miller–Rabin bu yolla Las Vegas'a çevrilemez. Dönüşüm yalnız yanıtın kendisi ucuzca denetlenebildiğinde işe yarar.

12.3 Karıştırma

12.3.1 Problem ve doğruluk ölçütü

Bir dizinin öğelerini rastgele bir sıraya sokmak, yani **karıştırmak** (*shuffle*), sık karşılaşılan bir iştir. Kart oyunlarında deste karıştırılır, müzik çalarlar şarkı listesini karışık sırayla çalar, algoritma testlerinde rastgele sıralı test verisi üretilir. Rastgele pivotlu hızlı sıralamanın bir başka biçimi de diziyi önce karıştırıp sonra deterministik hızlı sıralamayı çalıştırmaktır.

Karıştırmanın "doğru" olması ne demektir? Zayıf bir ölçüt, her öğenin her konumda eşit olasılıkla bulunmasıdır. Güçlü ve asıl istenen ölçüt ise n öğenin $n!$ farklı sıralamasının, yani **permütasyonunun**, her birinin tam $1/n!$ olasılıkla üretilmesidir. Böyle bir karıştırmaya **düzgün** karıştırma denir. Güçlü ölçüt zayıf olanı da kapsar; tersi doğru değildir. Örneğin bir yöntem [1, 2, 3] dizisini yalnız üç çevrimsel kaydırmasından birine dönüştürsün: [1, 2, 3], [2, 3, 1] ya da [3, 1, 2]. Üç sonuç eşit olasılıklı olsun. Bu yöntem her öğeyi her konuma 1/3 olasılıkla koyar. Ama altı permütasyonun üçünü hiç üretmez.

n	n! permütasyon
3	6
4	24
10	3.628.800
52	$\approx 8{,}07 \cdot 10^{67}$

Tablo 12.3: Permütasyon sayısı $n!$ çok hızlı büyür.

52 kartlık bir destenin permütasyon sayısı yaklaşık $8 \cdot 10^{67}$'dir. Bu sayı o kadar büyüktür ki iyi karıştırılmış bir destenin dizilişi büyük olasılıkla daha önce hiçbir destede görülmemiştir. Karıştırma algoritması bu dev kümeden eşit olasılıkla seçim yapmalıdır, ama permütasyonları tek tek listeleyemez.

Çözüm, seçimi küçük adımlara bölmektir.

12.3.2 Fisher–Yates karıştırması

Fisher–Yates karıştırması (*Fisher–Yates shuffle*), Knuth karıştırması olarak da anılır. Fikir şudur: dizinin son konumuna gelecek öğe, bütün öğeler arasından rastgele seçilir ve oraya yerleştirilir. Sonra sondan bir önceki konuma gelecek öğe, henüz yerleşmemiş öğeler arasından seçilir. Bu, ilk konuma kadar sürer. Her adımda bir konum kesinleşir ve bir daha değişmez.

```
static void karistir(int[] a) {
  for (int i = a.length - 1; i > 0; i--) {
    int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
    int t = a[i]; a[i] = a[j]; a[j] = t;
  }
}
```

Kodda `zar`, sınıfta bir kez oluşturulan paylaşılan bir `java.util.Random` nesnesidir. Döngü $i = n - 1$'den başlar ve $i = 1$'e kadar iner. Her turda `zar.nextInt(i + 1)`, 0 ile i arasında (ikisi de dahil) düzgün dağılmış bir tam sayı j üretir. Sonra `a[i]` ile `a[j]` takas edilir. $j = i$ de çıkabilir; bu durumda öğe kendisiyle takas edilir ve yerinde kalır. Bu olasılık önemlidir: bir öğenin başladığı yerde kalması da geçerli bir sonuçtur ve karıştırma bu sonucu da doğru olasılıkla üretmelidir.

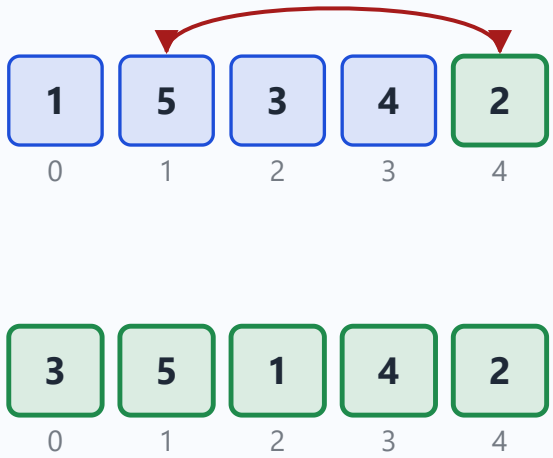
Döngü $i = 0$ için çalışmaz. O noktada tek bir öğe kalmıştır ve seçilecek başka bir şey yoktur. Toplam $n - 1$ takas yapılır ve her takas sabit sürer. Süre her çalıştırmada aynıdır: $\Theta(n)$. Ek bellek $O(1)$'dir; karıştırma dizinin kendi içinde yapılır. Değişen, sürenin değil sonucun kendisidir. Java'nın `Collections.shuffle(liste)` metodu da bu algoritmayı kullanır.

> **Örnek 12.4**
>
> $a = [1, 2, 3, 4, 5]$ dizisi karıştırılsın. Rastgele sayı üreteci sırasıyla $j = 1, 3, 0, 1$ değerlerini versin. Tablo 12.4 her adımı verir.

i	seçim aralığı	j	işlem	dizi
—	—	—	başlangıç	[1, 2, 3, 4, 5]
4	a[0..4], 5 öğe	1	a[4] ↔ a[1]	[1, 5, 3, 4, **2**]
3	a[0..3], 4 öğe	3	j = i, değişmez	[1, 5, 3, **4**, **2**]
2	a[0..2], 3 öğe	0	a[2] ↔ a[0]	[3, 5, **1**, **4**, **2**]
1	a[0..1], 2 öğe	1	j = i, değişmez	[3, **5**, **1**, **4**, **2**]

Tablo 12.4: Fisher–Yates izlemesi, $a = [1, 2, 3, 4, 5]$. Kalın değerler kesinleşmiş konumlardır.

İlk turda 5 öğenin her biri 1/5 olasılıkla son konuma gelebilirdi; seçilen $j = 1$, yani 2 değeridir. Takastan sonra `a[4] = 2` kesinleşir. İkinci turda $j = 3 = i$ çıktığı için 4 değeri yerinde kalır. Üçüncü turda üç öğe arasından `a[0] = 1` seçilir ve `a[2]`'ye gider; eski `a[2] = 3` başa geçer. Son turda yine $j = i$ çıkar. `a[0] = 3` kalan tek öğedir ve kendiliğinden yerleşir. Sonuç [3, 5, 1, 4, 2]'dir. Java ile, aynı j değerlerini döndüren bir üreteçle çalıştırılan kod da bu diziyi verir.



Şekil 12.4: Fisher–Yates izlemesinden iki an. Solda ilk takastan sonra dizi: a[4] = 2 kesinleşti (yeşil). Sağda döngü bitince: [3, 5, 1, 4, 2]; bütün konumlar kesin.

12.3.3 Doğruluğun kanıtı

Fisher–Yates'in doğru olduğu kısa bir sayma akıl yürütmesiyle gösterilir. Algoritma $i = n - 1$ turunda n değerden birini, $i = n - 2$ turunda $n - 1$ değerden birini seçer. Bu böyle devam eder; son tur $i = 1$'de 2 değerden biri seçilir. Olası seçim dizilerinin sayısı $n \cdot (n - 1) \cdot \ldots \cdot 2 = n!$'dir ve her seçim dizisi eşit olasılıklıdır.

Teorem 12.2 (Fisher–Yates düzgündür)

Fisher–Yates karıştırması, farklı öğeli bir dizinin $n!$ permütasyonunun her birini tam $1/n!$ olasılıkla üretir.

İspat. Seçim dizileri ile permütasyonlar arasında birebir bir eşleme vardır. İstenen bir permütasyon verilmiş olsun. Son konumdaki öğe bu permütasyonda bellidir. O öğe, $i = n - 1$ turunda dizinin tek bir konumundadır; o konum j olmak zorundadır. Takastan sonra kalan $n - 1$ öğelik ön kısım için aynı akıl yürütme tekrarlanır. Böylece her permütasyonu üreten tek bir seçim dizisi bulunur. Seçim dizileri $n!$ tanedir ve permütasyonlar da $n!$ tanedir. Her seçim dizisinin olasılığı $(1/n) \cdot (1/(n - 1)) \cdot \ldots \cdot (1/2) = 1/n!$ olduğundan her permütasyonun olasılığı da $1/n!$'dir. □

Kanıt bir şeyi açıkça gösterir: karıştırmanın düzgünlüğü, eşit olasılıklı yol sayısının permütasyon sayısına eşit olmasından gelir. Beş öğeli örnekte olası çekiliş dizisi sayısı $5 \cdot 4 \cdot 3 \cdot 2 = 120 = 5!$'dir; her permütasyon tam bir kez elde edilir. Sabit tohumla (`new Random(42)`) [1, 2, 3] dizisi 600.000 kez karıştırıldı. Altı permütasyonun her biri 99.754 ile 100.315 arasında çıktı. Beklenen değer 100.000'dir ve sapmalar bu kadar denemede beklenen rastgele dalgalanma düzeyindedir.

12.3.4 Saf karıştırma yanlıdır

Karıştırmanın akla ilk gelen yazılışı şudur: her konum, dizinin herhangi bir konumuyla rastgele takas edilir. Kod kısa ve masum görünür.

```
// YANLI: her i için 0..n-1 arasından seç
for (int i = 0; i < n; i++) {
  int j = zar.nextInt(n);
  takas(a, i, j);
}
```

Bu kodda her i için j, Fisher–Yates'teki gibi daralan bir aralıktan değil, her zaman bütün dizinin aralığı olan $0..n-1$'den seçilir. Dizi her turda gerçekten karışır, ama sonuç düzgün değildir. Nedeni sayma ile görülür. n çekilişin her biri n farklı değer alabilir. Eşit olasılıklı yol sayısı n^n'dir. Permütasyon sayısı ise $n!$'dir. Her permütasyon eşit olasılıklı olsaydı her biri tam $n^n / n!$ yolla üretilirdi ve bu sayı bir tam sayı olurdu.

$n = 3$ için $3^3 = 27$ yol ve $3! = 6$ permütasyon vardır. 27, 6'ya bölünmez; dolayısıyla permütasyonlar eşit olasılıklı olamaz. Bu akıl yürütme $n \geq 3$ için genel olarak geçerlidir. $n - 1$ sayısı $n!$'yi böler ama n^n ile ortak böleni yoktur; bu yüzden $n^n / n!$ hiçbir zaman tam sayı olmaz.

sonuç	yol	olasılık	olması gereken	benzetim (600.000)
123	4	4/27 ≈ 0,148	1/6 ≈ 0,167	88.843
132	5	5/27 ≈ 0,185	1/6 ≈ 0,167	110.659
213	5	5/27 ≈ 0,185	1/6 ≈ 0,167	112.175
231	5	5/27 ≈ 0,185	1/6 ≈ 0,167	110.602
312	4	4/27 ≈ 0,148	1/6 ≈ 0,167	89.008
321	4	4/27 ≈ 0,148	1/6 ≈ 0,167	88.713

Tablo 12.5: [1, 2, 3] dizisinde saf karıştırmanın 27 çekiliş dizisi tek tek sayıldı. Son sütun, tohumu 42 olan 600.000 denemelik benzetimin sonucudur.

Tablo 12.5'te 132, 213 ve 231 sonuçları beşer yolla, diğer üçü dörder yolla elde edilir. Beklenen sayılar $600.000 \cdot 5/27 \approx 111.111$ ve $600.000 \cdot 4/27 \approx 88.889$'dur; benzetim bu değerlere çok yakındır. Fark küçük görünebilir, ama sistematiktir: deneme sayısı arttıkça kaybolmaz, tersine daha belirgin hâle gelir. Bir kart oyununda ya da bir istatistik deneyinde bu yanlılık sonuçları bozar.

Saf karıştırma zayıf ölçütü de sağlamaz. Tablodan 3 değerinin konumları okunsun: ilk konuma gelme olasılığı *(4 + 4)/27 = 8/27*, ortadaki konuma *10/27*, son konuma *9/27*'dir. Üçü de 1/3 olmalıydı. Dört öğeli dizide durum daha da karışıktır: $4^4 = 256$ yol 24 permütasyona dağılır. 256/24 ≈ 10,67 tam sayı değildir; Java ile yapılan tam sayımda en az 8, en çok 15 yolla üretilen permütasyonlar çıktı.

> **Sık yapılan hata**
>
> **Karıştırmanın düzgünlüğünü gözle denetlemek.** Saf karıştırmanın çıktıları gözle bakıldığında gayet rastgele görünür. Yanlılık ancak sayımla ya da çok sayıda denemeyle ortaya çıkar. Doğru kalıp `zar.nextInt(i + 1)`'dir: seçim aralığı her turda daralır ve henüz yerleşmemiş öğelerle sınırlı kalır.

12.3.5 Tek karakterlik bir hata: nextInt(i)

Fisher–Yates kodunda `zar.nextInt(i + 1)` yerine yanlışlıkla `zar.nextInt(i)` yazıldığında ne olur? Bu durumda *j* en çok *i − 1* olabilir; `a[i]` her zaman kendinden önceki bir öğeyle takas edilir ve hiçbir öğe yerinde kalamaz. [1, 2, 3] dizisinde *i = 2* turunda *j ∈ {0, 1}*, *i = 1* turunda ise *j = 0* zorunludur. Yalnız *2 · 1 = 2* yol vardır.

i = 2'de j	ilk takastan sonra	i = 1'de j	sonuç
0	[3, 2, 1]	0	[2, 3, 1]
1	[1, 3, 2]	0	[3, 1, 2]

Tablo 12.6: `nextInt(i)` ile [1, 2, 3] dizisinin bütün yolları.

İki sonucun her biri 1/2 olasılıkla çıkar. Diğer dört permütasyon, sıralı [1, 2, 3] dahil, hiç üretilmez. Bu değiştirilmiş yöntemin kendine ait bir adı vardır: **Sattolo algoritması**. Yalnız tek bir çevrimden oluşan *(n − 1)!* permütasyonu eşit olasılıkla üretir. Bazı uygulamalarda tam olarak bu istenir; örneğin her öğenin başka bir öğeye gönderildiği ve kimsenin kendine düşmediği bir döngü kurulacaksa. Ama düzgün bir karıştırma değildir. Tek bir karakterlik fark, çıktının olasılık dağılımını tamamen değiştirir.

12.4 BogoSort

12.4.1 Algoritma

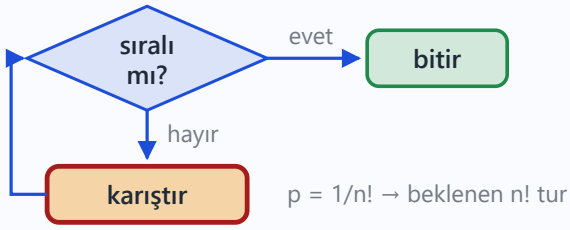
BogoSort, sıralama algoritmaları arasında bilinen en kötü performanslılardan biridir. “Aptal sıralama” ya da permütasyon sıralaması (*permutation sort*) adlarıyla da anılır. Mantığı tek cümleyle anlatılır: liste sıralı olana kadar rastgele karıştırılır. Gerçek uygulamalarda kullanılmaz. Ama Las Vegas algoritmalarının ve geometrik dağılımın en sade örneğidir; bu yüzden öğreticidir.

1. Liste sıralı mı? Evet ise işlem biter.

2. Hayır ise liste rastgele karıştırılır ve birinci adıma dönülür.

```
static void bogoSirala(int[] a) {
  while (!siraliMi(a))
    karistir(a);        // Fisher–Yates
}
static boolean siraliMi(int[] a) {
  for (int i = 0; i + 1 < a.length; i++)
    if (a[i] > a[i + 1]) return false;
  return true;
}
```

`siraliMi` metodu komşu öğe çiftlerini soldan sağa denetler. Bir çiftte soldaki sağdakinden büyükse dizi sıralı değildir ve metot hemen `false` döndürür. Bütün çiftler uygunsa `true` döner. En kötü durumda $n - 1$ karşılaştırma yapar: $O(n)$. `bogoSirala` ise sıralı olmayan diziyi Fisher–Yates ile karıştırır ve denetimi tekrarlar. Karıştırma için doğru bir yöntem kullanılması önemlidir; saf karıştırmayla bazı permütasyonlar daha seyrek çıkar ve analiz geçersizleşir.



Şekil 12.5: BogoSort akışı: sıralıysa biter, değilse karıştırıp yeniden sorar. Her turun başarı olasılığı $p = 1/n!$, beklenen tur sayısı $n!$'dir.

12.4.2 Bir çalıştırma

Örnek 12.5

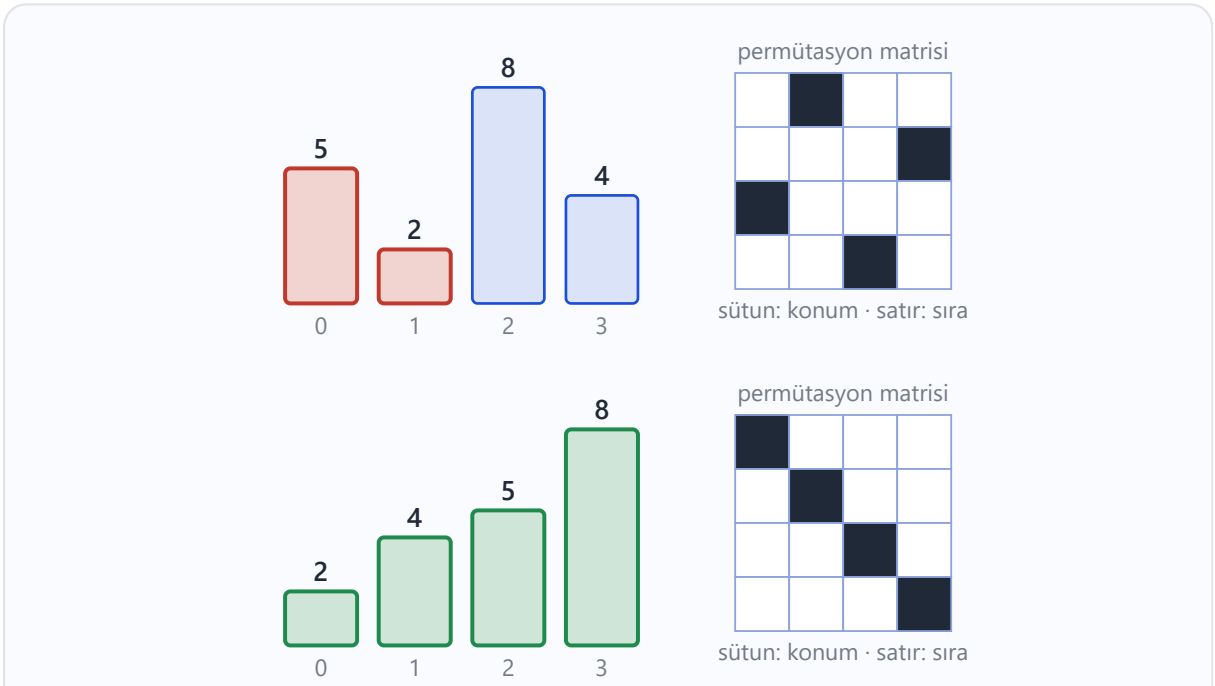
[5, 2, 8, 4] listesi BogoSort ile sıralansın. Her karıştırma Fisher–Yates ile yapılır. Tablo 12.7, tohumlu bir üreteçle yapılan bir çalıştırmanın bütün adımlarını verir. Her satırda `siraliMi` 'nin bulduğu ilk ters sıralı komşu çift gösterilmiştir.

karıştırma	dizi	ilk ters çift	sonuç
0 (başlangıç)	[5, 2, 8, 4]	a[0] = 5 > a[1] = 2	sıralı değil
1	[5, 4, 2, 8]	a[0] = 5 > a[1] = 4	sıralı değil
2	[8, 4, 5, 2]	a[0] = 8 > a[1] = 4	sıralı değil
3	[4, 2, 8, 5]	a[0] = 4 > a[1] = 2	sıralı değil

karıştırma	dizi	ilk ters çift	sonuç
4	[2, 8, 4, 5]	a[1] = 8 > a[2] = 4	sıralı değil
5	[5, 8, 2, 4]	a[1] = 8 > a[2] = 2	sıralı değil
6	[8, 4, 2, 5]	a[0] = 8 > a[1] = 4	sıralı değil
7	[2, 8, 5, 4]	a[1] = 8 > a[2] = 5	sıralı değil
8	[2, 4, 5, 8]	yok	sıralı, algoritma durur

Tablo 12.7: [5, 2, 8, 4] için bir BogoSort çalıştırması.

Bu çalıştırmada 8 karıştırma gerekti. Dört öğenin 24 permütasyonundan yalnız biri sıralıdır; beklenen karıştırma sayısı 24'tür. Bu çalıştırma beklenenden şanslı geçmiştir. Başka bir çalıştırma 3 karıştırmada da bitebilir, 80 karıştırmada da. `siraliMi`'nin çoğu turda ilk ya da ikinci karşılaştırmada durduğuna da dikkat edilmelidir. Rastgele bir dizide ilk çiftin ters olma olasılığı 1/2'dir; bu yüzden denetim ortalamada $n - 1$ karşılaştırmadan çok daha kısa sürer. Karıştırma ise her turda tam $n - 1$ takas yapar.



Şekil 12.6: Solda başlangıç [5, 2, 8, 4]; kırmızı çubuklar ilk ters çifttir. Sağda sekizinci karıştırmadan sonra [2, 4, 5, 8]. Sağdaki permütasyon matrisinde koyu hücre, değerin sırasını (satır) ve konumunu (sütun) gösterir; sıralı dizide köşegen dolar.

12.4.3 Analiz

Öğeler farklı olsun. Fisher–Yates her permütasyonu $1/n!$ olasılıkla ürettiği için bir karıştırmanın sıralı çıkma olasılığı $p = 1/n!$'dir. Karıştırmalar birbirinden bağımsızdır. Sıralı dizi elde edilene kadar yapılan karıştırma sayısı geometrik dağılıma uyar. Beklenen değeri $1/p = n!$'dir.

Her tur bir karıştırma ($O(n)$) ve bir denetim ($O(n)$) içerir. Beklenen süre bu yüzden $\Theta(n \cdot n!)$'dir. Kaynaklarda bu süre bazen $O((n+1)!)$ diye de yazılır. İki ifade aynı büyüklüktedir, çünkü $(n+1)! = (n+1) \cdot n!$'dir ve $n+1$ ile n arasında sabit bir çarpan farkı vardır.

n	beklenen karıştırma n!	süre
5	120	0,12 µs
10	3.628.800	3,6 ms
12	479.001.600	0,48 s
13	6.227.020.800	6,2 s
15	1.307.674.368.000	≈ 22 dakika
20	2.432.902.008.176.640.000	≈ 77 yıl

Tablo 12.8: BogoSort'ta beklenen karıştırma sayısı ve saniyede 10^9 karıştırma yapabilen hayalî bir makinede süre.

Tablo, faktöriyel büyümenin pratikte ne anlama geldiğini gösterir. On iki öğe yarım saniyede sıralanırken on beş öğe yirmi dakikadan fazla, yirmi öğe bir insan ömrü kadar sürer. Beklenen değer formülü benzetimle de doğrulandı. Denemeler sabit tohumla (`new Random(42)`) ve sıralı olmayan karışık bir başlangıçtan yapıldı. Ortalama karıştırma sayıları şöyle çıktı: $n = 4$ için 23,9 (beklenen 24), $n = 5$ için 119,7 (beklenen 120), $n = 6$ için 714,5 (beklenen 720). Her n için 20.000 çalıştırma yapıldı.

En iyi durum. Dizi zaten sıralıysa tek bir denetim yapılır ve algoritma $O(n)$ sürede biter.

En kötü durum. Burada bir üst sınır yoktur. k karıştırmanın hepsinin sırasız çıkma olasılığı $(1 - 1/n!)^k$dır. Bu sayı k büyüdükçe sıfıra yaklaşır ama hiçbir k için sıfır olmaz. Demek ki her k için en az k tur süren bir çalıştırma mümkündür. $O((n+1)!)$ ifadesi yalnız beklenen süre için bir sınırdır, en kötü durum için değildir. Şanssız karıştırmalar sonsuza dek sürebilir; bunun olasılığı sıfıra gider.

Örnek 12.6

Dört öğeli bir dizide BogoSort'un 24 karıştırmadan, yani beklenen değerden sonra hâlâ bitmemiş olma olasılığı nedir? İlk 24 karıştırmanın hepsinin sırasız çıkması gerekir: $(23/24)^{24} \approx 0{,}36$. Çalıştırmaların üçte birinden fazlası beklenen değeri aşar. 100 karıştırmadan sonra bitmemiş olma olasılığı ise $(23/24)^{100} \approx 0{,}014$'tür. Geometrik dağılımın kuyruğu üstel olarak incelir, ama hiçbir zaman kopmaz.

Sık yapılan hata

Beklenen süre sınırını en kötü durum sınırı sanmak. "BogoSort $O((n+1)!)$'dir" cümlesi yalnız beklenen süre için doğrudur. Las Vegas algoritmalarında en kötü durum çoğu zaman sınırsızdır. Analizde hangi sürenin konuşulduğu her zaman açıkça belirtilmelidir.

12.5 Rastgele QuickSort

12.5.1 Pivot seçimi ve etkisi

Hızlı sıralama (*quick sort*, QuickSort) kısaca şöyle çalışır. Diziden bir **pivot** öğesi seçilir. Dizi, pivottan küçük ya da eşit olanlar solda, büyük olanlar sağda kalacak biçimde **bölümlenir** (*partition*) ve pivot iki grubun arasına, kalıcı yerine yerleşir. Sonra iki taraf ayrı ayrı, özyinelemeyle sıralanır. Bölümleme $\Theta(n)$ sürer. Algoritmanın toplam süresi tamamen pivotun ne kadar dengeli böldüğüne bağlıdır.

Pivot her seferinde diziyi iki eşit yarıya bölerse $\log_2 n$ düzey oluşur ve süre $\Theta(n \log n)$ olur. Pivot her seferinde alt dizinin en küçük ya da en büyük öğesiyse parçalardan biri boş, öteki yalnız bir eksik çıkar. Bu durumda n düzey oluşur ve süre $\Theta(n^2)$'ye çıkar. Pivot seçme kuralı bu yüzden algoritmanın en önemli tasarım kararıdır. Tablo 12.9 yaygın stratejileri karşılaştırır.

Pivot stratejisi	Kötü girdi	Beklenen / tipik	En kötü
İlk ya da son öğe	sıralı ya da ters sıralı dizi (çok yaygın)	$O(n \log n)$*	$O(n^2)$
Üçün ortancası	özel olarak hazırlanmış diziler	$O(n \log n)$*	$O(n^2)$
Rastgele öğe	yok: kötülük yalnız şanssızlıktan gelir	$O(n \log n)$, her girdi için	$O(n^2)$
Gerçek ortanca (BFPRT)	yok	$O(n \log n)$	$O(n \log n)$

Tablo 12.9: Pivot stratejileri. (*) Deterministik stratejilerde $O(n \log n)$ yalnız girdi rastgele sıralıysa geçerlidir (ortalama durum).

İlk ya da son öğeyi pivot seçmek en basit kuraldır, ama zaten sıralı ya da ters sıralı dizide en kötü durumu doğurur. Bu girdiler uygulamada çok sık görülür. **Üçün ortancası** (*median-of-three*) kuralı ilk, orta ve son öğenin ortancasını pivot seçer. Sıralı dizide mükemmel çalışır, ama kuralı bilen biri onu yanıltacak özel diziler hazırlayabilir. Pivot olarak alt dizinin gerçek ortancası bulunursa bölme her zaman dengelidir. Ortancayı doğrusal sürede bulan deterministik bir algoritma vardır; geliştiricilerinin baş harfleriyle BFPRT (Blum, Floyd, Pratt, Rivest, Tarjan) diye anılır. Ama sabitleri büyüktür ve uygulamada yavaş kalır.

Rastgele pivotun farkı ince ama önemlidir. Deterministik stratejilerde $O(n \log n)$, girdinin rastgele sıralı olduğu varsayımına dayanır. Rastgele pivotta beklenti algoritmanın kendi seçimleri üzerindendir; bu yüzden her girdi için geçerlidir. Yine de en kötü durum ortadan kalkmaz. Algoritma her seferinde şanssız bir pivot seçerse süre yine $O(n^2)$ olur. Değişen şey, bu durumun artık belirli bir girdiye bağlı olmaması ve olasılığının çok küçük olmasıdır. Uygulamada rastgele pivot ya da üçün ortancası tercih edilir.

Sık yapılan hata

"Rastgele pivot en kötü durumu yok eder." Bu yanlıştır; en kötü durum hâlâ $O(n^2)$'dir. Rastgele pivotun kazandırdığı şey, hiçbir girdinin sürekli kötü olmaması ve kötü durumun ola-

sılığının çok küçük olmasıdır.

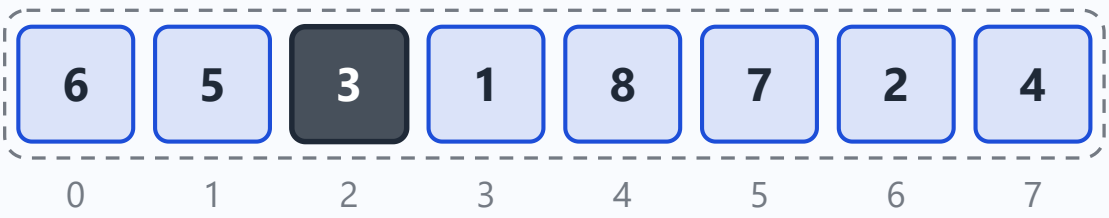
12.5.2 Rastgele pivotla bir çalıştırma

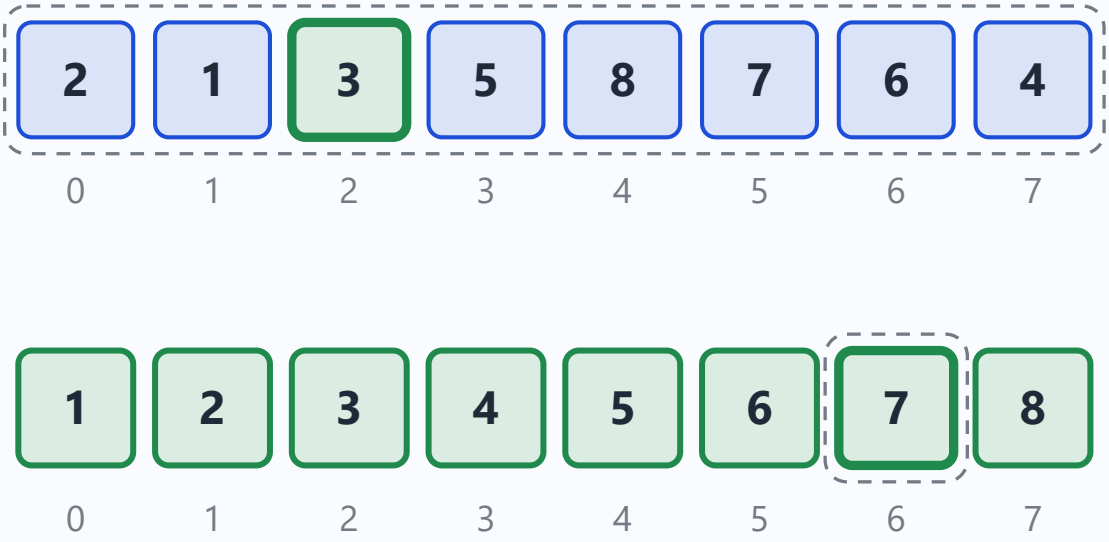
Rastgele pivotlu hızlı sıralama [6, 5, 3, 1, 8, 7, 2, 4] dizisinde çalıştırılsın. Bu çalıştırmada pivotlar sırasıyla 3, 1, 4, 6 ve 8 olarak seçilmiş olsun. İzlemede bölümleme iki işaretçiyle yapılır. *i* soldan ilerler ve pivottan küçük öğeleri atlar; ilk büyük ya da eşit öğede durur. *j* sağdan ilerler ve pivottan büyük öğeleri atlar; ilk küçük ya da eşit öğede durur. *i* < *j* ise iki öğe takas edilir ve arama sürer. İşaretçiler karşılaştığında pivot kalıcı yerindedir. Bu yöntem Hoare'un bölümleme fikrine dayanır. Pivot bu sırada kendisi de yer değiştirebilir; konumu izlenir.

alt dizi	pivot	işaretçiler ve takaslar	sonra dizi	pivotun yeri
a[0..7]	3	i = 0 (6), j = 6 (2): takas; i = 1 (5), j = 3 (1): takas; karşılaşma	[2, 1, 3, 5, 8, 7, 6, 4]	2
a[0..1]	1	i = 0 (2), j = 1 (1): takas; karşılaşma	[1, 2, 3, 5, 8, 7, 6, 4]	0
a[1..1]	—	tek öğe, zaten sıralı	değişmez	—
a[3..7]	4	i = 3 (5), j = 7 (4): takas; karşılaşma	[1, 2, 3, 4, 8, 7, 6, 5]	3
a[4..7]	6	i = 4 (8), j = 7 (5): takas; i = 5 (7), j = 6 (6): takas; karşılaşma	[1, 2, 3, 4, 5, 6, 7, 8]	5
a[4..4]	—	tek öğe	değişmez	—
a[6..7]	8	i, 7'yi atlayıp 8'de durur; j zaten 8'de; takas yok	[1, 2, 3, 4, 5, 6, 7, 8]	7
a[6..6]	—	tek öğe	dizi sıralandı	—

Tablo 12.10: [6, 5, 3, 1, 8, 7, 2, 4] dizisinde rastgele pivotlu hızlı sıralama. Pivotlar 3, 1, 4, 6, 8.

İlk bölümleme ayrıntılı izlensin. Pivot 3'tür ve indis 2'dedir. *i*, `a[0] = 6`'da hemen durur, çünkü 6 pivottan büyüktür. *j* sağdan başlar: 4 pivottan büyüktür, atlanır; `a[6] = 2`'de durur. *0 < 6* olduğu için 6 ile 2 takas edilir ve dizi [2, 5, 3, 1, 8, 7, 6, 4] olur. Arama sürer: *i*, 2'yi atlayıp `a[1] = 5`'te durur; *j*, 6, 7 ve 8'i atlayıp `a[3] = 1`'de durur. 5 ile 1 takas edilir: [2, 1, 3, 5, 8, 7, 6, 4]. Bir sonraki adımda *i*, 1'i atlayıp pivotun kendisinde (indis 2) durur; *j* de 5'i atlayıp yine indis 2'ye gelir. İşaretçiler karşılaşmıştır. Pivot 3 kalıcı yerindedir: solunda yalnız küçükler (2, 1), sağında yalnız büyükler var.





Şekil 12.7: Rastgele pivotlu hızlı sıralamadan üç an. Üstte başlangıç: pivot 3 rastgele seçildi (siyah kutu); kesikli çerçeve etkin alt dizidir. Ortada ilk bölümleme bitti: 3 kalıcı yerinde (indis 2, yeşil). Altta bütün öğeler kesinleşti.

Bu çalıştırmada pivotlar oldukça dengeli düştü. İlk pivot 3, sekiz öğeyi 2 ve 5 öğeli iki parçaya ayırdı. Özyineleme ağacının derinliği 4'te kaldı. Pivot olarak sürekli en küçük öğe seçilseydi derinlik 7 olurdu. Bir pivotun kalıcı yeri, dizide ondan küçük öğelerin sayısına eşittir; bu, hangi bölümleme yöntemi kullanılırsa kullanılsın böyledir. Örneğin 6'dan küçük beş öğe vardır, bu yüzden 6'nın yeri indis 5'tir.

12.5.3 Kod ve bölümleme izlemesi

Aşağıdaki kod rastgele pivotlu hızlı sıralamanın tamamıdır. Bölümleme için **Lomuto** yöntemi kullanılır: rastgele seçilen pivot önce alt dizinin sonuna taşınır, sonra tek bir soldan sağa geçişle küçükler sola toplanır. İzlemedeki iki işaretçili yöntemle aynı işi yapar; pivotlar aynı kalıcı yerlere gider, yalnız aradaki takaslar farklıdır.

```
static Random zar = new Random();

static void hizliSirala(int[] a, int alt, int ust) {
  if (alt < ust) {
    int p = bol(a, alt, ust); // rastgele
    hizliSirala(a, alt, p - 1);
    hizliSirala(a, p + 1, ust);
  }
}

static int bol(int[] a, int alt, int ust) {
  int r = alt + zar.nextInt(ust - alt + 1);
  takas(a, r, ust);             // pivot sona
  int pivot = a[ust], k = alt;
  for (int i = alt; i < ust; i++)
    if (a[i] <= pivot)
      takas(a, i, k++);         // küçükleri sola
  takas(a, k, ust);             // pivot yerine
  return k;
}
```

`hizliSirala` metodu standart özyinelemeli iskelettir. Alt dizide en az iki öğe varsa `bol` çağrılır, dönen pivot konumunun iki yanı ayrı ayrı sıralanır. Rastgelelik yalnız `bol`'un ilk satırındadır: `alt + zar.nextInt(ust - alt + 1)` ifadesi *alt* ile *ust* arasında (ikisi dahil) düzgün dağılmış bir *r* indisi verir. Bu öğe `a[ust]` ile takas edilerek sona taşınır ve pivot olur.

Döngü değişmezi şudur: her turun başında `a[alt..k - 1]` pivottan küçük ya da eşit öğeleri, `a[k..i - 1]` ise pivottan büyük öğeleri içerir. `a[i]` pivottan küçük ya da eşitse `a[k]` ile takas edilir ve *k* bir artar; böylece "küçükler" bölgesi bir öğe büyür. Döngü bitince `a[k]`, büyükler bölgesinin ilk öğesidir. Pivot onunla takas edilerek kalıcı yerine konur ve *k* döndürülür. `bol`, *ust* − *alt* karşılaştırma yapar.

> **Örnek 12.7**
>
> *a = [6, 5, 3, 1, 8, 7, 2, 4]* için `bol(a, 0, 7)` çağrılsın ve `zar.nextInt(8)` 2 döndürsün. Pivot `a[2] = 3`'tür. Tablo 12.11 bölümlemenin her adımını verir.

adım	i	a[i] ≤ 3?	işlem	k	dizi
pivot sona	—	—	a[2] ↔ a[7]	0	[6, 5, 4, 1, 8, 7, 2, 3]
1	0	6: hayır	—	0	değişmez
2	1	5: hayır	—	0	değişmez
3	2	4: hayır	—	0	değişmez
4	3	1: evet	a[3] ↔ a[0]	1	[1, 5, 4, 6, 8, 7, 2, 3]
5	4	8: hayır	—	1	değişmez
6	5	7: hayır	—	1	değişmez

adım	i	a[i] ≤ 3?	işlem	k	dizi
7	6	2: evet	a[6] ↔ a[1]	2	[1, 2, 4, 6, 8, 7, 5, 3]
pivot yerine	—	—	a[2] ↔ a[7]	2	[1, 2, 3, 6, 8, 7, 5, 4]

Tablo 12.11: Lomuto bölümlemesi, $r = 2$ (pivot 3).

Yedi karşılaştırma yapıldı ve `bol` 2 döndürdü. Pivot 3 yine indis 2'ye yerleşti; iki işaretçili izlemede de aynı yere gitmişti. Ama dizinin geri kalanı farklıdır: iki işaretçili yöntem [2, 1, 3, 5, 8, 7, 6, 4] üretmişti, Lomuto ise [1, 2, 3, 6, 8, 7, 5, 4] üretir. Bölümlemenin sözleşmesi yalnız pivotun yerini ve iki tarafın içeriğini belirler; taraflardaki sıra yönteme göre değişir. Kod Java ile çalıştırıldı ve aynı diziyi verdi. 2000 rastgele dizide `hizliSirala`'nın sonucu `Arrays.sort` ile karşılaştırıldı ve hepsi aynı çıktı.

12.5.4 En iyi, en kötü ve beklenen durum



Şekil 12.8: Solda dengeli bölme: $log_2\ n$ düzeyin her biri toplam n iş yapar, süre $O(n\ log\ n)$. Sağda her seferinde en küçük ya da en büyük pivot: n düzey ve $n + (n - 1) + \ldots + 1 = O(n^2)$.

Durum	Yineleme bağıntısı	Süre
En iyi	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n\ log\ n)$
Beklenen (rastgele pivot)	$T(n) = (1/n)\ \Sigma_q\ [T(q) + T(n - 1 - q)] + \Theta(n)$	$\Theta(n\ log\ n)$
En kötü	$T(n) = T(n - 1) + \Theta(n)$	$\Theta(n^2)$

Tablo 12.12: Hızlı sıralamanın üç durumu için yineleme bağıntıları.

Beklenen durum bağıntısı şöyle okunur. Rastgele pivot, alt dizinin n öğesinden her birini $1/n$ olasılıkla seçer. Pivot sıralı hâlde q. öğeyse (0'dan sayılarak) sol parça q, sağ parça $n - 1 - q$ öğelidir. Bağıntı bütün q değerlerinin ortalamasını alır ve bölümlemenin doğrusal maliyetini ekler. Çözümü $\Theta(n\ log\ n)$'dir. Bu bağıntıyı doğrudan çözmek yerine karşılaştırma sayısını gösterge değişkenleriyle hesaplamak daha kolaydır.

Önce bir sezgisel gözlem yapılabilir. Pivot, sıralı hâlde ortadaki yarıya düşen öğelerden biriyse (yani ne en küçük çeyrekte ne en büyük çeyrekte ise) iki parçanın büyüğü en çok *3n/4* öğeli olur. Bu "iyi pivot" olayının olasılığı 1/2'dir. Her iyi pivot, öğenin içinde bulunduğu parçayı en az dörtte bir küçültür. Bir öğenin yolunda en çok $\log_{4/3} n$ iyi pivot olabilir ve iyi pivotlar ortalama iki denemede bir gelir. Her öğe ortalama *O(log n)* bölümlemeye katılır; toplam iş *O(n log n)*'dir.

En kötü durumun olasılığı. Rastgele pivotta en kötü durum için her bölümlemede alt dizinin en küçük ya da en büyük öğesi seçilmelidir. *k* öğeli bir alt dizide bunun olasılığı *2/k*'dir. Seçimler bağımsız olduğundan olasılıklar çarpılır:

> $P(\text{en kötü}) = (2/n) \cdot (2/(n-1)) \cdot \ldots \cdot (2/2) = 2^{n-1} / n!$

$n = 5$ için bu değer $16/120 \approx 0{,}133$'tür; küçük dizilerde en kötü durum hiç de seyrek değildir. Python'da sabit tohumla (`random.Random(42)`) 200.000 kez beş öğeli dizi sıralandı. Karşılaştırma sayısının en büyük değer olan 10'a ulaştığı çalıştırmaların oranı 0,134 çıktı. $n = 10$ için olasılık $1{,}4 \cdot 10^{-4}$'e, $n = 20$ için ise $2^{19} / 20! \approx 2 \cdot 10^{-13}$'e düşer. Gerçek boyutlu dizilerde en kötü durum pratikte hiç görülmez.

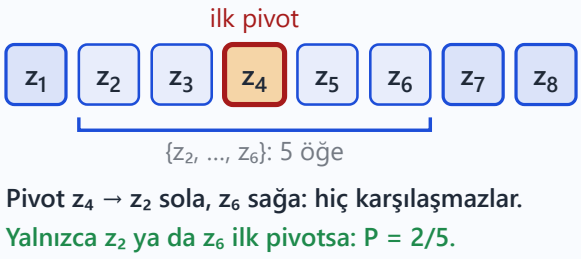
12.5.5 Beklenen karşılaştırma sayısı

Hızlı sıralamanın süresi, yaptığı karşılaştırma sayısıyla orantılıdır: bölümleme döngüsünün her turu tam bir karşılaştırma yapar. Bu sayı *X* olsun. *E[X]* gösterge değişkenleri ve doğrusallıkla hesaplanır.

Öğeler sıralı hâlleriyle adlandırılsın: $z_1 < z_2 < \ldots < z_n$. Burada z_i, dizinin *i*. en küçük öğesidir; dizideki konumuyla ilgisi yoktur. Her $i < j$ çifti için bir gösterge tanımlansın: z_i ile z_j algoritma boyunca karşılaştırılırsa $X_{ij} = 1$, karşılaştırılmazsa 0. İki öğe en fazla bir kez karşılaştırılabilir, çünkü karşılaştırmalar yalnız pivotla yapılır ve pivot o bölümlemeden sonra bir daha hiçbir alt diziye girmez. Bu yüzden $X = \sum_{i<j} X_{ij}$'dir.

> **Teorem 12.3 (Karşılaştırılma olasılığı)**
>
> Rastgele pivotlu hızlı sıralamada z_i ile z_j ($i < j$) tam olarak $2 / (j - i + 1)$ olasılıkla karşılaştırılır.

İspat. $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ kümesi ele alınsın; bu kümede $j - i + 1$ öğe vardır. Bu kümeden hiçbir öğe pivot seçilmediği sürece kümenin bütün öğeleri aynı alt dizide kalır, çünkü dışarıdan seçilen her pivot ya hepsinden küçük ya da hepsinden büyüktür. Kümeden ilk pivot seçildiği an belirleyicidir. Bu pivot z_i ya da z_j ise, öteki öğe o anda aynı alt dizidedir ve pivotla karşılaştırılır. Pivot aradaki öğelerden biriyse z_i sola, z_j sağa düşer; ikisi ayrı parçalarda kalır ve hiç karşılaşmaz. Pivot alt dizinin öğeleri arasından düzgün seçildiği için, kümeden ilk seçilen öğe kümenin her öğesi için eşit olasılıklıdır. İstenen iki öğeden biri olma olasılığı $2 / (j - i + 1)$'dir. □



Şekil 12.9: $\{z_2, \ldots, z_6\}$ kümesinden ilk pivot z_4 seçilirse z_2 sola, z_6 sağa düşer ve hiç karşılaştırılmazlar. Yalnız z_2 ya da z_6 ilk pivot olursa karşılaştırılırlar: olasılık 2/5.

Teoremin iki uç sonucu ilginçtir. Sıralı hâlde komşu iki öğe, yani $j = i + 1$, olasılık $2/2 = 1$ ile, yani her zaman karşılaştırılır. Bu doğaldır: aralarında başka öğe yoktur ve onları ayırabilecek bir pivot bulunamaz. En küçük ile en büyük öğe ise ancak ilk pivot ikisinden biriyse karşılaşır: olasılık $2/n$. Python ile sekiz öğeli rastgele diziler 100.000 kez sıralandı (tohum 42). z_2–z_6 çiftinin karşılaştırılma oranı 0,399 (beklenen 0,4), z_1–z_8 çiftininki 0,251 (beklenen 0,25), komşu z_3–z_4 çiftininki tam 1 çıktı.

Doğrusallıkla beklenen karşılaştırma sayısı göstergelerin olasılıklarının toplamıdır:

> $$E[X] = \Sigma_{i<j}\, 2 / (j - i + 1)$$

Toplam, $k = j - i + 1$ uzaklığına göre gruplanarak kapalı biçime getirilir. Uzaklığı k olan çift sayısı $n - k + 1$'dir ($2 \le k \le n$). Böylece

> $$E[X] = \Sigma_{k=2..n}\, (n - k + 1) \cdot 2/k = 2(n + 1)(H_n - 1) - 2(n - 1) = 2(n + 1)H_n - 4n$$

elde edilir. Burada $H_n = 1 + 1/2 + \ldots + 1/n$, n. harmonik sayıdır ve yaklaşık $\ln n + 0{,}577$'dir. Kaba bir üst sınır da kolayca görülür: her i için iç toplam $2(1/2 + 1/3 + \ldots)$'den küçüktür, dolayısıyla $E[X] < 2n\,H_n$'dir. $2n \ln n = 2 \ln 2 \cdot n \log_2 n$ olduğundan bu sınır yaklaşık $1{,}39\, n \log_2 n$'dir. Rastgele pivotlu hızlı sıralama, mükemmel dengeli bölmenin $n \log_2 n$ karşılaştırmasından yalnız yaklaşık %39 fazla karşılaştırma yapar. Kapalı formülün $n \le 30$ için tam kesirli aritmetikle toplamın kendisine eşit olduğu Python ile denetlendi.

n	$2(n+1)H_n - 4n$	benzetim ortalaması	sıralı girdi, ilk öğe pivot
10	24,4	24,4 (20.000 sıralama)	45
100	647,9	648,0 (20.000)	4.950
1.000	10.985,9	11.003,6 (2.000)	499.500
10.000	155.771,7	155.710,7 (200)	49.995.000
10^6	$\approx 2{,}48 \cdot 10^7$	—	$\approx 5{,}0 \cdot 10^{11}$

Tablo 12.13: Rastgele pivotlu hızlı sıralamada beklenen karşılaştırma sayısı (formül ve `new Random(42)` ile benzetim) ve sıralı girdide ilk öğeyi pivot seçen sürümün karşılaştırma sayısı.

Benzetimde her seferinde sıralı bir dizi (1, 2, ..., n) sıralandı. Bu, deterministik ilk öğe pivotunun en kötü girdisidir; rastgele pivot için ise sıradan bir girdidir. Ortalamalar formüle çok yakındır; küçük farklar sınırlı sayıda tekrardan gelen rastgele dalgalanmadır. Son sütun $n(n-1)/2$'dir: ilk öğe sıralı dizide hep en küçük öğedir, her düzeyde parça yalnız bir küçülür. $n = 1000$'de fark yaklaşık 45 kattır, $n = 10^6$'da ise yaklaşık 20.000 kat.

Sık yapılan hata

Ortalama durum ile beklenen süreyi karıştırmak. Ortalama durum, girdi dağılımı hakkında bir varsayım yapar; varsayım bozulursa güvence de bozulur. Beklenen süre ise her sabit girdi için algoritmanın yazı-turaları üzerinden hesaplanır. Tablo 12.13'teki 24,4 değeri, sıralı dizi dahil 10 öğeli her girdi için geçerlidir.

12.5.6 Bellek ve tekrarlı anahtarlar

Hızlı sıralama diziyi yerinde sıralar; ek bellek yalnız özyineleme yığınıdır. Yığın derinliği özyineleme ağacının yüksekliğidir. Rastgele pivotta beklenen derinlik $O(\log n)$'dir, ama en kötü durumda n'ye çıkabilir. Bunu önlemek için basit bir teknik vardır. Önce küçük parçaya özyinelemeyle inilir, büyük parça ise bir döngüyle işlenir. Küçük parça en çok yarı boyutta olduğu için her özyinelemeli çağrıda boyut en az yarıya iner. Yığın derinliği en kötü durumda da $O(\log n)$'de kalır.

Dizide çok sayıda eşit anahtar varsa Lomuto bölümlemesi kötü davranır. Bütün öğeleri eşit bir dizide her öğe "≤ pivot" koşulunu sağlar, pivot hep sona yerleşir ve parçalardan biri hep boş kalır. Pivot rastgele olsa bile süre $\Theta(n^2)$'dir. Çözüm **üç yollu bölümlemedir**: dizi pivottan küçükler, pivota eşitler ve pivottan büyükler olarak üçe ayrılır. Eşitler bölgesi kalıcı yerindedir ve özyinelemeye girmez.

12.6 Rastgele seçme (QuickSelect)

12.6.1 Problem ve kod

Seçme problemi, dizinin s. en küçük öğesini bulmaktır. Ortanca, en küçük ya da en büyük on öğenin sınırı gibi sorular bu problemin örnekleridir. Diziyi sıralayıp s. konumu okumak $O(n \log n)$ sürer. Ama sıralamanın büyük kısmı gereksizdir: yalnız tek bir öğenin yeri soruluyor. **Rastgele seçme** (*quickselect*) algoritması hızlı sıralamanın bölümlemesini kullanır, ama her seferinde yalnız aranan öğenin bulunduğu tarafa iner.

```
static int sec(int[] a, int alt, int ust, int s) {
  if (alt == ust) return a[alt];
  int p = bol(a, alt, ust);
  if (s == p) return a[p];
  if (s < p) return sec(a, alt, p - 1, s);
  return sec(a, p + 1, ust, s);
}
```

s, 0 tabanlı bir sıra numarasıdır: *s = 0* en küçük öğe, *s = n − 1* en büyük öğedir. Metot önce taban durumunu denetler. Aralıkta tek öğe kalmışsa (*alt = ust*) aranan odur. Değilse aynı rastgele `bol` çağrılır ve pivotun kalıcı yeri *p* öğrenilir. Pivotun kalıcı yeri, sıralı dizideki yeridir. *s = p* ise aranan pivottur. *s < p* ise aranan öğe pivotun solundadır; *s > p* ise sağındadır. Hangi tarafa inilirse inilsin *s* değişmez, çünkü indisler dizinin tamamına göredir. Öteki taraf bir daha hiç incelenmez.

Algoritma da bir Las Vegas algoritmasıdır. Sonuç her zaman doğrudur; rastgele olan yalnız ne kadar bölümleme yapılacağıdır. Bir yan etkisi de vardır: metot diziyi yerinde değiştirir. Dönüşten sonra `a[s]` aranan öğedir; solundakiler ondan küçük ya da eşit, sağındakiler büyüktür.

12.6.2 Kod izleme

Örnek 12.8

a = [6, 5, 3, 1, 8, 7, 2, 4] dizisinde `sec(a, 0, 7, 5)` çağrılsın; aranan 0 tabanlı 5. öğe, yani 6. en küçük öğedir. `bol` sırasıyla 3, 7 ve 5 öğelerini pivot seçsin. Tablo 12.14 çağrıları verir.

çağrı	pivot	bol sonrası dizi	p	karar
sec(a, 0, 7, 5)	3	[1, 2, 3, 6, 8, 7, 5, 4]	2	s = 5 > 2: sağa, [3, 7]
sec(a, 3, 7, 5)	7	[1, 2, 3, 6, 4, 5, 7, 8]	6	s = 5 < 6: sola, [3, 5]
sec(a, 3, 5, 5)	5	[1, 2, 3, 4, 5, 6, 7, 8]	4	s = 5 > 4: sağa, [5, 5]
sec(a, 5, 5, 5)	—	değişmez	—	alt = ust = 5: `return a[5]` = 6

Tablo 12.14: `sec(a, 0, 7, 5)` izlemesi.

İlk çağrıda pivot 3, Lomuto bölümlemesiyle indis 2'ye yerleşir. Aranan konum 5, pivotun sağındadır; bu yüzden yalnız [3, 7] aralığına inilir. Sol taraftaki iki öğe (1 ve 2) bir daha incelenmez. İkinci çağrıda pivot 7'dir; 3..7 aralığında 7'den küçük üç öğe (6, 5, 4) olduğu için 7, indis 6'ya gider. Aranan konum 5 artık pivotun solundadır. Üçüncü çağrıda üç öğelik [6, 4, 5] aralığında pivot 5 seçilir ve indis 4'e yerleşir. Son çağrıda aralıkta tek öğe kalır: *alt = ust = 5*. Metot `a[5] = 6` döndürür ve bu değer çağrı zincirinden olduğu gibi geri gelir.



Şekil 12.10: Üstte ilk çağrı: etkin aralık [0, 7], aranan konum 5 işaretli. Altta sonuç: 5. konumdaki öğe 6. Bu örnekte dizi sonunda tamamen sıralı çıktı; bu bir rastlantıdır, algoritma yalnız gereken parçaları böler.

Java ile, `bol`'un aynı pivotları seçmesini sağlayan betikli bir üreteçle çalıştırılan kod 6 döndürdü; ara diziler de tablodakilerle aynı çıktı. Ayrıca 2000 rastgele dizide ve rastgele s değerlerinde `sec`'in sonucu, sıralanmış dizinin s. öğesiyle karşılaştırıldı; hepsi eşit çıktı.

12.6.3 Analiz

Rastgele seçmenin beklenen süresi $O(n)$'dir. Hızlı sıralamadan farkı, özyinelemenin iki değil tek kola inmesidir. İyi bir pivot, yani ortadaki yarıya düşen bir pivot, aralığı en çok $3/4$'üne indirir ve 1/2 olasılıkla gelir. Ortalama iki bölümlemede bir aralık en az dörtte bir küçülür. Aralık boyutu n'den $3n/4$'e, sonra $9n/16$'ya iner. Her aşamanın beklenen maliyeti en çok iki bölümleme, yani o aşamadaki boyutun iki katı kadardır. Toplam beklenen iş bir geometrik seriyle sınırlanır:

> $E[T(n)] \le 2n\,(1 + 3/4 + (3/4)^2 + \dots) = 2n \cdot 4 = 8n = O(n)$

Bu kaba bir sınırdır. Ortanca aranırken beklenen karşılaştırma sayısının kesin değeri yaklaşık $2(1 + \ln 2)\,n \approx 3{,}39\,n$'dir. Java ile ortanca 2000 kez arandı (tohum 42): $n = 1001$ için ortalama $3{,}33\,n$, $n = 10.001$ için $3{,}39\,n$ karşılaştırma çıktı. Küçük n'deki fark, formüldeki düşük dereceli terimlerden gelir.

En kötü durum yine $O(n^2)$'dir. Pivot her seferinde aralığın bir ucuna düşerse aralık her adımda yalnız bir küçülür. Bölümleme zaten $\Theta(n)$ sürdüğü için $O(\log n)$ gibi bir süre ise imkânsızdır: her öğe en az bir kez okunmalıdır. Seçme için en kötü durumda da doğrusal süre isteniyorsa pivot BFPRT yöntemiyle, yani “ortancaların ortancası” ile seçilir. Uygulamada ise rastgele sürüm daha hızlıdır.

Sık yapılan hata

Seçmenin beklenen süresini $O(n \log n)$ sanmak. Aynı bölümlemeyi kullandığı için hızlı sıralamayla aynı süreyi bekleme eğilimi vardır. Ama seçme her düzeyde yalnız bir parçayla

devam eder ve iş geometrik olarak azalır: *n + 3n/4 + 9n/16 + …* Toplam doğrusaldır.

12.7 Monte Carlo ile π

12.7.1 Yöntem ve kod

Monte Carlo yöntemi, bir sayıyı rastgele örneklerin oranından tahmin eder. En bilinen örneği π sayısının tahminidir. Kenarı 1 olan birim kare ve merkezi karenin sol alt köşesinde, yarıçapı 1 olan çeyrek daire düşünülsün. Karenin alanı 1, çeyrek dairenin alanı *π/4*'tür. Kareye düzgün rastgele bir nokta atılırsa noktanın çeyrek dairenin içine düşme olasılığı alanların oranıdır: *p = π/4 ≈ 0,785*.

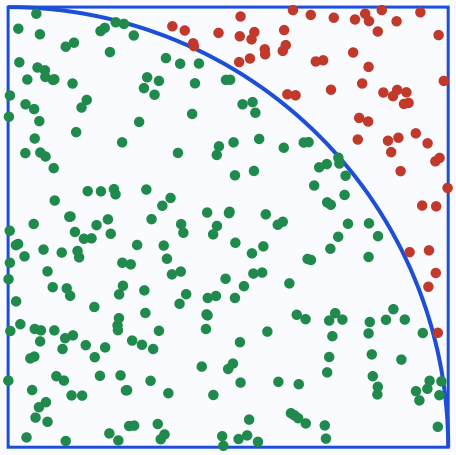
Bir nokta *(x, y)*, ancak orijine uzaklığı en çok 1 ise çeyrek dairenin içindedir: $x^2 + y^2 \le 1$. Karekök almaya gerek yoktur. *n* nokta atılır ve içeri düşenler sayılır. İç nokta oranı *π/4*'ün tahminidir; dolayısıyla

> $\pi \approx 4 \cdot \text{iç} / n$

olur. Yöntem iki yönlü hatalı bir Monte Carlo algoritmasıdır. Süresi tam *n* adımdır, ama sonucu gerçek değerin altında da üstünde de kalabilir.

```
static double piTahmin(int n) {
  int ic = 0;
  for (int t = 0; t < n; t++) {
    double x = zar.nextDouble(), y = zar.nextDouble();
    if (x * x + y * y <= 1) ic++;
  }
  return 4.0 * ic / n;
}
```

Her turda `zar.nextDouble()` iki kez çağrılır ve *[0, 1)* aralığında düzgün dağılmış iki sayı üretir. Nokta çeyrek dairedeyse `ic` bir artar. Döngüden sonra `4.0 * ic / n` döndürülür. Buradaki `4.0` önemlidir: ifadeyi `double` aritmetiğine geçirir. `4 * ic / n` yazılsaydı tam sayı bölmesi yapılır ve kesir kısmı atılırdı; örneğin 6 noktadan 4'ü içerideyse sonuç 2,667 yerine 2 olurdu. Metot *O(n)* sürede ve *O(1)* ek bellekle çalışır.



Şekil 12.11: Birim kareye atılan 300 nokta. 239'u çeyrek dairenin içinde (yeşil): $\pi \approx 4 \cdot 239 / 300 \approx 3{,}187$.

12.7.2 Kod izleme

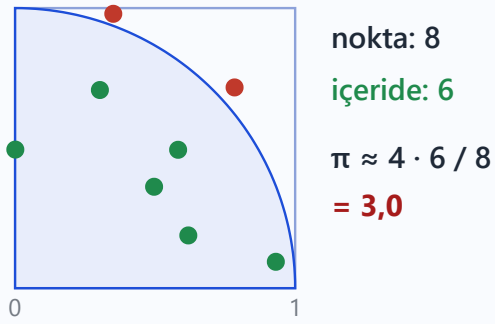
Örnek 12.9

`zar = new Random(2024)` ile `piTahmin(8)` çağrılsın. Tohum sabit olduğu için çalıştırma her seferinde aynı noktaları üretir. Tablo 12.15 sekiz noktanın hepsini verir; koordinatlar dört basamağa yuvarlanmıştır.

t	x	y	$x^2 + y^2$	sonuç	ic
0	0,6185	0,1883	0,4179	içeride	1
1	0,9310	0,0942	0,8756	içeride	2
2	0,5821	0,4941	0,5830	içeride	3
3	0,0008	0,4948	0,2448	içeride	4
4	0,3028	0,7075	0,5921	içeride	5
5	0,7837	0,7167	1,1279	dışarıda	5
6	0,3506	0,9796	1,0825	dışarıda	5
7	0,4963	0,3621	0,3774	içeride	6

Tablo 12.15: `piTahmin(8)`, tohum 2024.

Sekiz noktanın altısı içeride kaldı. Metot $4.0 \cdot 6 / 8 = 3{,}0$ döndürür. Tahmin gerçek değerden yaklaşık 0,14 uzaktır. Bu kadar az noktayla şaşırtıcı değildir: tahminin standart sapması yaklaşık $1{,}64 / \sqrt{8} \approx 0{,}58$'dir. 3,0 tahmini bu ölçüye göre oldukça iyidir. Aynı tohumla aynı kod her zaman 3,0 verir; tohum değiştirilirse başka bir sonuç çıkar.



Şekil 12.12: `piTahmin(8)` bittiğinde: 8 nokta, 6'sı içeride (yeşil), 2'si dışarıda (kırmızı). Tahmin *4 · 6 / 8 = 3,0*.

12.7.3 Hata ve nokta sayısı

Tahminin ne kadar güvenilir olduğu basit bir olasılık hesabıyla bulunur. Her nokta, $p = \pi/4$ olasılıkla 1, aksi hâlde 0 değerini alan bağımsız bir göstergedir. n göstergenin toplamı olan `ic` 'in beklenen değeri np, varyansı $np(1-p)$'dir. Tahmin $4 \cdot ic / n$ olduğu için beklenen değeri tam olarak $4p = \pi$'dir; yöntem ortalamada doğru değeri verir. Standart sapması ise

> $$\sigma = 4 \cdot \sqrt{p(1-p)/n} = 1{,}642 / \sqrt{n}$$

olur. Hata, nokta sayısının karekökü ile ters orantılı azalır. Bunun pratik anlamı ağırdır: hatayı 10'da birine indirmek için nokta sayısı 100 katına çıkmalıdır. Bir ondalık basamak daha doğruluk 100 kat daha fazla iş ister.

n	tahmin	|tahmin − π|	σ = 1,642/√n
100	3,16000	0,0184	0,164
1.000	3,10400	0,0376	0,0519
10.000	3,12240	0,0192	0,0164
100.000	3,14128	0,00031	0,0052
1.000.000	3,13863	0,00296	0,0016
10.000.000	3,14066	0,00093	0,00052

Tablo 12.16: Farklı nokta sayılarıyla π tahmini. Her satırda `zar = new Random(42)` ile baştan başlandı.

Tablo, tek tek çalıştırmaların gürültülü olduğunu gösterir. 100 noktalık tahmin şans eseri çok iyi çıkmıştır; 1.000.000 noktalık tahmin ise 100.000 noktalıktan daha kötüdür. Bu çelişki değildir. σ, tek bir çalıştırmanın hatasını değil, hataların tipik büyüklüğünü verir. Bunu sınamak için 1.000 noktalık tahmin 2000 kez tekrarlandı (tohum 42). Hataların karelerinin ortalamasının karekökü 0,0514 çıktı; formül 0,0519 verir. Normal dağılım yaklaşımıyla hata, çalıştırmaların yaklaşık %95'inde $1{,}96\,\sigma$'nın altında kalır. Hatanın %95 güvenle 0,01'in altında olması için yaklaşık 104.000 nokta, 0,001'in altında olması için yaklaşık 10,4 milyon nokta gerekir.

Örnek 12.10

10.000 noktayla yapılan bir tahminde hata yaklaşık 0,016'dır. Hatayı yaklaşık 0,0016'ya indirmek için kaç nokta gerekir? Hata $1/\sqrt{n}$ ile azaldığından hatayı 10'da birine indirmek için $\sqrt{n}$ 10 kat, n ise 100 kat artmalıdır: 1.000.000 nokta. Nokta sayısını 100.000'e çıkarmak hatayı yalnız $\sqrt{10} \approx 3{,}16$ kat azaltır.

π tahmini, π'yi hesaplamanın iyi bir yolu değildir; çok daha hızlı deterministik formüller vardır. Ama yöntemin kendisi çok değerlidir. Alanı ya da hacmi formülle hesaplanamayan karmaşık şekiller, çok boyutlu integraller, fizikte parçacık benzetimleri ve finansta risk hesapları aynı fikirle yapılır. Monte Carlo yönteminin hatası boyut sayısından bağımsız olarak $1/\sqrt{n}$ ile azalır. Yüksek boyutlu problemlerde bu özellik onu ızgara tabanlı yöntemlerden üstün kılar.

Sık yapılan hata

Tam sayı bölmesi. `4 * ic / n` ifadesinde üç işlenen de `int` olduğundan sonuç tam sayıya kırpılır. Altı noktadan dördü içerideyse sonuç 2,667 yerine 2 olur. `4.0` ile başlamak bölmeyi `double`'a taşır.

12.8 Miller–Rabin asallık testi

12.8.1 Fermat testi ve Carmichael sayıları

Bir sayının asal olup olmadığını anlamanın en basit yolu, $\sqrt{n}$'ye kadar bütün sayılara bölmeyi denemektir. Bu yöntem $\sqrt{n}$ bölme yapar. Sayının basamak sayısına göre üstel bir süredir: 300 basamaklı bir sayı için 10^{150} bölme gerekir. Kriptografide ise tam olarak bu büyüklükte asal sayılara ihtiyaç vardır. **Asallık testi** (*primality test*), sayıyı çarpanlarına ayırmadan asallığını sınar.

Fikir sayılar kuramından gelir. **Fermat'nın küçük teoremine** göre n asal ve $1 < a < n$ ise

> $$a^{n-1} \equiv 1 \pmod{n}$$

olur. Bu eşitlik bir a için bozulursa n kesinlikle bileşiktir. Bu, çarpanlar bulunmadan bileşikliğin kanıtlanması demektir. Eşitlik sağlanıyorsa n "belki asal"dır. Sorun şudur: bazı bileşik sayılar, kendileriyle aralarında asal olan her taban için eşitliği sağlar. Bunlara **Carmichael sayıları** denir. En küçüğü $561 = 3 \cdot 11 \cdot 17$'dir. Örneğin $2^{560} \bmod 561 = 1$'dir. 561 ile aralarında asal olan bütün tabanlar, yani 1 ile 560 arasındaki 320 sayının hepsi, Fermat testini geçer. Fermat testi bu sayıları ancak ortak çarpanlı bir taban tutturarak yakalayabilir. Carmichael sayıları sonsuz tanedir.

12.8.2 Karekök özelliği ve Miller–Rabin dizisi

Miller–Rabin testi, Fermat testine ikinci bir denetim ekler. Bu denetim 1'in kareköklerine dayanır. n asalsa $x^2 \equiv 1 \pmod n$ denkliğinin yalnız iki çözümü vardır: $x \equiv 1$ ve $x \equiv -1 \equiv n-1$. Gerekçesi kısadır. $x^2 - 1 = (x-1)(x+1)$ çarpımı n'ye bölünüyorsa ve n asalsa, n çarpanlardan birini böler. Bileşik sayılarda ise 1'in başka kareköklerı de olabilir. 561'de örneğin $67^2 = 4489 = 8 \cdot 561 + 1$'dir; 67, 1'in "gizli" bir kareköküdür.

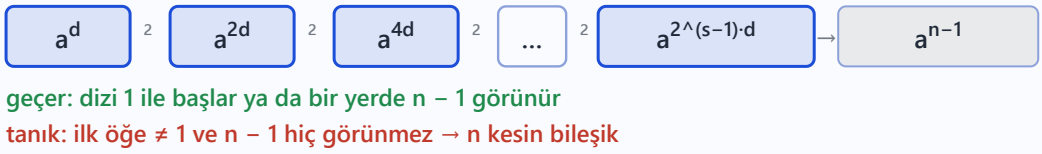
Test şöyle kurulur. n tek bir sayı olsun. $n-1$ çifttir ve $n - 1 = 2^s \cdot d$ biçiminde yazılır; burada d tektir. Rastgele bir a tabanı için şu dizi hesaplanır:

> $$a^d,\ a^{2d},\ a^{4d},\ \ldots,\ a^{2^{s-1}d} \pmod n$$

Her öğe bir öncekinin karesidir. Dizinin bir sonraki karesi $a^{2^s d} = a^{n-1}$'dir. n asalsa Fermat'ya göre bu son değer 1'dir. Diziye geriye doğru bakılırsa, 1'e ulaşan her karenin bir önceki öğesi 1'in kareköküdür; asal bir modülde bu öğe ya 1 ya da $n-1$ olmak zorundadır. Sonuç olarak n asalsa dizi ya 1 ile başlar ya da bir yerde $n-1$ değerini alır.

Tanım 12.4

Tek bir n sayısı ve $n - 1 = 2^s \cdot d$ (d tek) verilsin. $a^d \bmod n \neq 1$ ise ve $0 \le r < s$ için hiçbir $a^{2^r d}$ değeri $n-1$'e denk değilse, a, n'nin bileşik olduğunun **tanığıdır** (*witness*). n bileşik olduğu hâlde a tanık değilse, a'ya n için bir **güçlü yalancı** (*strong liar*) denir.



Şekil 12.13: Miller–Rabin dizisi: a^d'den başlayarak ardışık kareler. Dizi 1 ile başlarsa ya da bir yerde $n-1$ görünürse n turu geçer. İlk öğe 1 değilse ve $n-1$ hiç görünmezse a tanıktır ve n kesin bileşiktir.

Testin gücü bir sayı kuramı sonucundan gelir. Rabin, tek bileşik bir n için 1 ile $n-1$ arasındaki tabanların en çok dörtte birinin güçlü yalancı olduğunu göstermiştir. Burada bu teoremin kanıtı verilmez, ama sonucu küçük sayılarda kolayca sınanır. 3000'den küçük bütün tek bileşik sayılar için 2 ile $n-2$ arasındaki tabanlar Python ile tek tek denendi. En yüksek yalancı oranı $n = 1891 = 31 \cdot 61$ için $14/59 \approx 0{,}237$ çıktı; 1/4'ün altındadır. Taban rastgele seçilirse bileşik bir sayının bir turu geçme olasılığı en çok 1/4'tür. k bağımsız turda yanılma olasılığı en çok 4^{-k}'dır.

12.8.3 Kod

```
static boolean asalMi(long n, int k) {
  if (n < 4) return n == 2 || n == 3;
  if (n % 2 == 0) return false;
  long d = n - 1; int s = 0;     // n-1 = 2^s·d
  while (d % 2 == 0) { d /= 2; s++; }
  for (int tur = 0; tur < k; tur++) {
    long a = 2 + zar.nextLong(n - 3); // [2, n-2]
    long x = usMod(a, d, n);     // a^d mod n
    if (x == 1 || x == n - 1) continue;
    boolean tanik = true;
    for (int r = 1; r < s && tanik; r++) {
      x = x * x % n;
      if (x == n - 1) tanik = false;
    }
    if (tanik) return false;     // kesin bileşik
  }
  return true;                   // muhtemelen asal
}
```

İlk iki satır küçük durumları ayıklar. $n < 4$ ise yalnız 2 ve 3 asaldır. Çift sayılar 2 dışında bileşiktir; 2 zaten ilk satırda yakalanmıştır. Sonraki iki satır $n - 1$'i $2^s \cdot d$ biçimine getirir: d çift kaldıkça ikiye bölünür ve s sayılır. Ana döngü k tur yapar. Her turda `2 + zar.nextLong(n - 3)` ifadesi $[2, n - 2]$ aralığından düzgün bir taban seçer; `nextLong(n - 3)` 0 ile $n - 4$ arasında bir değer verir. (1 ve $n - 1$ tabanları her sayı için testi geçtiği için işe yaramaz.)

`x = usMod(a, d, n)` dizinin ilk öğesidir. x 1 ya da $n - 1$ ise bu tur geçilir ve `continue` ile sonraki tura gidilir. Değilse `tanik` önce `true` yapılır; iç döngü dizinin geri kalan $s - 1$ öğesini ardışık karelerle hesaplar. $n - 1$ görünürse `tanik = false` olur ve iç döngü durur. İç döngü bittiğinde `tanik` hâlâ `true` ise a bir tanıktır ve metot `false` döndürür. Bu yanıt kesindir. Bütün turlar geçilirse metot `true` döndürür: sayı "muhtemelen asal"dır.

Kod iki ayrıntı içerir. Birincisi, `zar.nextLong(long bound)` metodu Java 17 ile eklenmiştir; daha eski sürümlerde yoktur. İkincisi, `x * x % n` çarpımı `long` türünde yapılır. $x < n$ olduğundan çarpım n^2'den küçüktür. `long`'un en büyük değeri yaklaşık $9{,}22 \cdot 10^{18}$'dir; karekökü $3.037.000.499$'dur. Kod bu yüzden yalnız $n < 3 \cdot 10^9$ için güvenlidir. Daha büyük sayılar için `BigInteger.modPow` ya da doğrudan `BigInteger.isProbablePrime` kullanılır.

12.8.4 Hızlı modüler üs alma

$a^d \bmod n$ değerini a'yı $d - 1$ kez kendisiyle çarparak hesaplamak, büyük d'lerde imkânsızdır. **Kare alçarp** yöntemi üssün ikili gösteriminden yararlanır. Üs ikili tabanda yazılır; her adımda taban karesi alınarak bir sonraki ikinin kuvvetine çıkarılır. Üssün o biti 1 ise o kuvvet sonuca katılır.

```
static long usMod(long a, long d, long n) {
  long sonuc = 1;
  a %= n;
  while (d > 0) {
    if ((d & 1) == 1)          // bit 1 ise
      sonuc = sonuc * a % n;
    a = a * a % n;             // kare al
    d >>= 1;                   // sonraki bit
  }
  return sonuc;
}
```

Döngü değişmezi şudur: her turun başında $sonuc \cdot a^d$, aranan değere (mod n) eşittir. Bit 1 ise a sonuca çarpılır ve d'nin o biti tüketilir. Sonra a'nın karesi alınır ve d bir bit sağa kaydırılır, yani ikiye bölünür. $d = 0$ olduğunda değişmez *sonuc*'un aranan değer olduğunu söyler. Döngü d'nin bit sayısı kadar, yani $\lfloor \log_2 d \rfloor + 1$ kez döner. Her turda en çok iki modüler çarpma yapılır: süre $O(\log d)$ çarpmadır.

tur	d	en sağ bit	sonuc	a (kareden sonra)	açıklama
başlangıç	35	—	1	2	$a = 2^1$
1	35	1	$1 \cdot 2 = 2$	4	2^1 katıldı
2	17	1	$2 \cdot 4 = 8$	16	2^2 katıldı
3	8	0	8	256	yalnız kare
4	4	0	8	$65.536 \bmod 561 = 460$	yalnız kare
5	2	0	8	$211.600 \bmod 561 = 103$	yalnız kare
6	1	1	$8 \cdot 103 = 824 \bmod 561 = 263$	511	2^{32} katıldı

Tablo 12.17: `usMod(2, 35, 561)` izlemesi. $35 = 100011_2 = 32 + 2 + 1$.

Döngü $d = 0$ olunca biter ve 263 döner: $2^{35} \bmod 561 = 263$. Tabloda a sütunu sırasıyla $2^1, 2^2, 2^4, 2^8, 2^{16}, 2^{32}$ değerlerini (mod 561) taşır. Bit 1 olan turlarda 2^1, 2^2 ve 2^{32} sonuca katılır ve $2^1 \cdot 2^2 \cdot 2^{32} = 2^{35}$ elde edilir. Toplam 6 kare ve 3 çarpma yapıldı; düz çarpmayla 34 çarpma gerekirdi. Fark küçük üslerde önemsiz görünür. 300 basamaklı bir üste ise yaklaşık 1000 kare yeterken düz yöntem 10^{300} civarında çarpma ister. Son turdaki $a = 511$ değeri hesaplanır ama kullanılmaz.

12.8.5 561 sayısının izlemesi

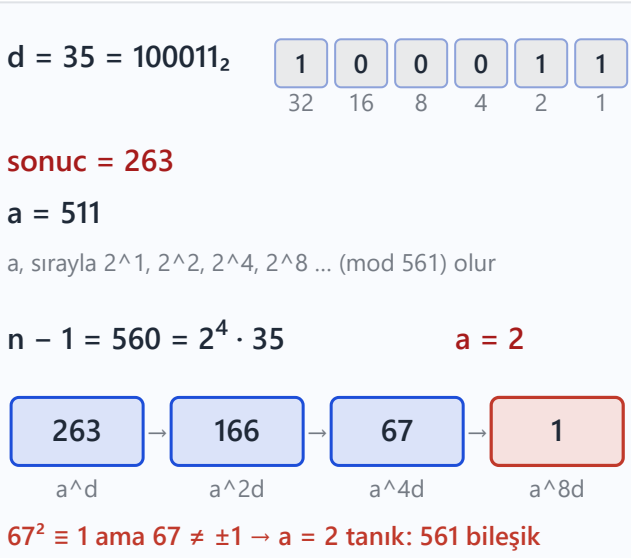
Örnek 12.11

`asalMi(561, 1)` çağrılsın ve rastgele taban $a = 2$ çıksın. 561 tektir ve 4'ten büyüktür; küçük durum denetimleri atlanır. Tablo 12.18 kalan adımları verir.

adım	işlem	değerler
1	$d = n - 1, s = 0$	d = 560, s = 0
2–5	d çift oldukça $d /= 2, s{+}{+}$	280/1, 140/2, 70/3, 35/4 → $560 = 2^4 \cdot 35$
6	tur 0: taban çekilir	a = 2
7	`x = usMod(2, 35, 561)`	x = 263; ne 1 ne 560 → `tanik = true`
8	r = 1: $x = 263^2 \bmod 561$	69.169 mod 561 = 166; 560 değil
9	r = 2: $x = 166^2 \bmod 561$	27.556 mod 561 = 67; 560 değil
10	r = 3: $x = 67^2 \bmod 561$	4.489 mod 561 = 1; 560 değil
11	iç döngü biter (r = 4 = s)	`tanik` hâlâ true → `return false`

Tablo 12.18: `asalMi(561, 1)`, taban $a = 2$.

Dizi 263, 166, 67, 1'dir ve 560'a hiç ulaşmaz. Metot `false` döndürür: 561 kesin bileşiktir. Dizinin son adımı özellikle öğreticidir. $67^2 \equiv 1$'dir ama 67 ne 1 ne de 560'tır. Asal bir modülde böyle bir karekök olamaz; bu yüzden 67 bileşikliğin somut kanıtıdır. Dizinin bir kare daha ilerisi $2^{560} \equiv 1$'dir. Fermat testi yalnız bu son değere baktığı için aldanır. Miller–Rabin ise 1'e nasıl ulaşıldığına bakar ve aldanmaz.



Şekil 12.14: Solda `usMod(2, 35, 561)` bitti: $35 = 100011_2$, sonuç 263. Sağda 561 için Miller–Rabin dizisi 263 → 166 → 67 → 1; $67^2 \equiv 1$ ama 67 ne 1 ne de −1'dir; dolayısıyla 2 bir tanıktır ve 561 bileşiktir.

12.8.6 Örnekler ve güçlü yalancılar

n	n − 1	a	$a^d, a^{2d}, \ldots \pmod n$	sonuç
61	$2^2 \cdot 15$	2	11, 60	geçti (asal olabilir)
221	$2^2 \cdot 55$	174	47, 220	geçti (asal olabilir)
221	$2^2 \cdot 55$	137	188, 205	bileşik (a tanık)

n	n − 1	a	a^d, a^{2d}, … (mod n)	sonuç
561	$2^4 \cdot 35$	2	263, 166, 67, 1	bileşik (a tanık)
25	$2^3 \cdot 3$	7	18, 24	geçti (asal olabilir)
25	$2^3 \cdot 3$	2	8, 14, 21	bileşik (a tanık)

Tablo 12.19: Miller–Rabin örnekleri. Her satır tek bir turdur.

61 asaldır. $2^{15} = 32.768 = 537 \cdot 61 + 11$, yani dizi 11 ile başlar. Karesi $121 = 61 + 60$, yani 60'tır ve bu $n - 1$'dir. Dizi $n - 1$'e ulaştığı için tur geçilir. Asal bir sayı her turu geçer; Miller–Rabin asal bir sayıya hiçbir zaman "bileşik" demez.

221 = 13 · 17 bileşiktir. $a = 174$ için dizi 47 ile başlar; $47^2 = 2209 = 9 \cdot 221 + 220$, yani 220'dir ve bu $n - 1$'dir. 174 bir güçlü yalancıdır: bileşik 221'i bu turda asal gibi gösterir. $a = 137$ için dizi 188 ve 205'tir; 220 görünmez ve 137 bir tanık olur. `asalMi(221, 2)` bu iki tabanla çağrılırsa ilk tur geçilir, ikinci tur `false` döndürür. Tek tur yetseydi ve yalnız 174 çekilseydi yanıt yanlış olarak "asal" olurdu. Birden çok tur bu yüzden gerekir. 221'in 2 ile 219 arasındaki 218 tabanından yalnız dördü (21, 47, 174, 200) güçlü yalancıdır. Tek turun yanılma olasılığı $4/218 \approx 0{,}018$'dir. Python'da sabit tohumla 100.000 rastgele taban denendi; oran 0,0185 çıktı.

$25 = 5^2$ için $n - 1 = 24 = 2^3 \cdot 3$'tür. $a = 7$'de $7^3 = 343 \equiv 18$ ve $18^2 = 324 \equiv 24 = n - 1$'dir; 7, 25 için güçlü yalancıdır. 25'in 22 tabanından yalnız ikisi (7 ve 18) yalancıdır. $a = 2$'de dizi 8, 14, 21'dir; 24 hiç görünmez ve sayı bileşik bulunur.

Bu örnekler Rabin sınırının çoğu sayıda çok kötümser olduğunu gösterir. 561'in 558 tabanından yalnız 8'i, 221'in 218 tabanından 4'ü güçlü yalancıdır. Sınıra en çok yaklaşan sayılar bile 1/4'ü geçmez. Yani gerçek hata olasılığı çoğu zaman 4^{-k}'dan da küçüktür; 4^{-k} en kötü durum için bir güvencedir.

12.8.7 Hata, maliyet ve taşma

Miller–Rabin tek yönlü hatalı bir Monte Carlo algoritmasıdır. "Bileşik" yanıtı kesindir, çünkü bir tanık bulunmuştur. "Asal" yanıtı yalnız muhtemelen doğrudur ve k turda yanılma olasılığı en çok 4^{-k}'dır. Asal bir sayı için yanıt her zaman doğrudur. Yanılma yalnız bileşik bir sayıya "asal" denmesi biçiminde olabilir.

Bir tur, bir modüler üs alma ve en çok $s - 1$ kare alma yapar. İkisi birlikte $O(\log n)$ modüler çarpmadır. b bitlik sayılar için okul yöntemiyle bir çarpma $O(b^2)$ bit işlemi sürer. $b = \log n$ olduğundan bir tur $O(\log^3 n)$, k tur $O(k \log^3 n)$ bit işlemidir. Bu, sayının basamak sayısına göre polinom bir süredir. Deneme bölmesinin üstel süresiyle karşılaştırıldığında aradaki fark çok büyüktür.

Kod Java ile denendi. `new Random(42)` tohumlu üreteçle 2 ile 200.000 arasındaki her sayı için `asalMi(n, 20)`, `BigInteger.isProbablePrime` ile karşılaştırıldı; yanlış yanıt çıkmadı. Aynı aralıkta tek turla ($k = 1$) ise 46 bileşik sayı yanlışlıkla "asal" bulundu. 1.000.000.007, $2^{31} - 1 = 2.147.483.647$ ve 2.999.999.929 sayıları 20 turla asal bulundu; `BigInteger` da aynı sonucu verir.

Sık yapılan hata

Modüler çarpımda taşma. `int` ile `x * x % n` yazılırsa x 46.341'e ulaştığında çarpım $2^{31} - 1$'i aşar: $46.341^2 = 2.147.488.281$. Yani $n > 46.341$ için sonuç sessizce yanlışlaşır. `long` ile sınır $n < 3{,}037 \cdot 10^9$'dur. Daha büyük sayılar için `Math.multiplyHigh` ile 128 bitlik çarpım ya da `BigInteger.modPow` kullanılır.

Sık yapılan hata

Monte Carlo yanıtını kesin saymak. Miller–Rabin'in "asal" yanıtı "muhtemelen asal" demektir. Tur sayısı k, hata sınırını belirler ve uygulamanın ihtiyacına göre seçilir. Kesin bir yanıt gerekiyorsa deterministik bir test (örneğin AKS ya da belirli sayı aralıkları için kanıtlanmış sabit taban kümeleri) kullanılır.

12.9 Karger minimum kesim algoritması

12.9.1 Minimum kesim problemi

Yönsüz ve bağlı bir çizge düşünülsün. Çizgenin düğümleri boş olmayan iki gruba ayrılsın. İki grup arasında kalan, yani bir ucu bir grupta öbür ucu öteki grupta olan kenarlar bir **kesimdir**. Bu kenarlar silinirse çizge iki parçaya kopar. **Minimum kesim** (*minimum cut*) problemi, en az kenarlı kesimi bulmaktır. Bu sayı, çizgenin bağlantısını koparmak için silinmesi gereken en az kenar sayısıdır; ağ güvenilirliğinin bir ölçüsüdür. Bilgisayar ağlarında, devre tasarımında ve görüntüleri bölgelere ayırmada kullanılır.

Tanım 12.5

$G = (V, E)$ yönsüz bir çoklu çizge olsun. V'nin boş olmayan ve birbirini tümleyen iki alt kümeye ayrılmasına $(S, V - S)$ **kesimi** denir. Kesimin büyüklüğü, bir ucu S'de öbür ucu $V - S$'de olan kenarların sayısıdır. Büyüklüğü en küçük olan kesime **minimum kesim** (global minimum kesim) denir.

Bu problem ağ akışındaki s–t kesiminden farklıdır. Orada iki özel düğüm verilir ve yalnız onları ayıran kesimler aranır. Burada ise hangi düğümlerin ayrılacağı serbesttir. Deterministik olarak çözmek mümkündür: bir düğüm sabitlenip öteki her düğümle bir s–t maksimum akış problemi çözülebilir. Ama bu, $n - 1$ akış hesabı demektir ve hem uzun hem karmaşık bir koddur. Karger'in 1993'te verdiği algoritma ise birkaç satırlık bir döngüden ibarettir.

12.9.2 Kenar büzme

Algoritmanın tek işlemi **kenar büzmedir**. Bir *(u, v)* kenarı büzülünce *u* ile *v* tek bir "süper düğümde" birleşir. Aralarındaki kenarlar artık bir düğümden kendisine giden öz-döngülere dönüşür ve silinir. *u*'dan ya da *v*'den başka düğümlere giden kenarlar süper düğüme bağlanır. Aynı iki süper düğüm arasında birden çok kenar oluşursa hepsi korunur; çizge bir çoklu çizgeye dönüşür. Bu paralel kenarlar önemlidir, çünkü her biri kesimde ayrı bir kenar olarak sayılır.

Algoritma şöyledir: çizgede iki süper düğüm kalana kadar, kalan kenarlar arasından düzgün rastgele bir kenar seçilir ve büzülür. Sonunda kalan iki süper düğüm, düğümlerin iki gruba ayrılışını verir; aralarında kalan kenar sayısı bir kesimdir. Algoritma her zaman tam *n − 2* büzme yapar. Sonucu ise bazen minimum değildir. Bu yüzden Monte Carlo algoritmasıdır.

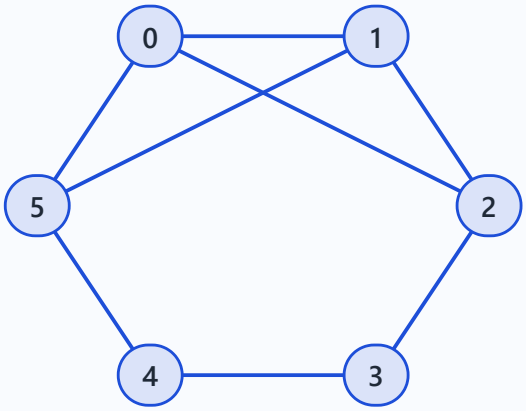
Örnek 12.12

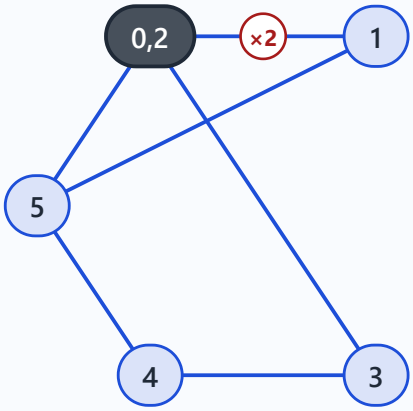
Altı düğümlü ve sekiz kenarlı bir çizge ele alınsın. Kenarlar (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (0, 5), (0, 2) ve (1, 5)'tir. Minimum kesim 2'dir. 3 ve 4 düğümleri çizgeye yalnız (2, 3) ve (4, 5) kenarlarıyla bağlıdır; {3, 4} kümesi bu iki kenarla ayrılır. Tek başına {3} ya da tek başına {4} de 2 kenarlı kesimdir. Bu çizgede üç minimum kesim vardır. Algoritma sırasıyla (0, 2), (1, 5), (3, 4) ve (0, 1) kenarlarını seçsin. Tablo 12.20 büzmeleri verir.

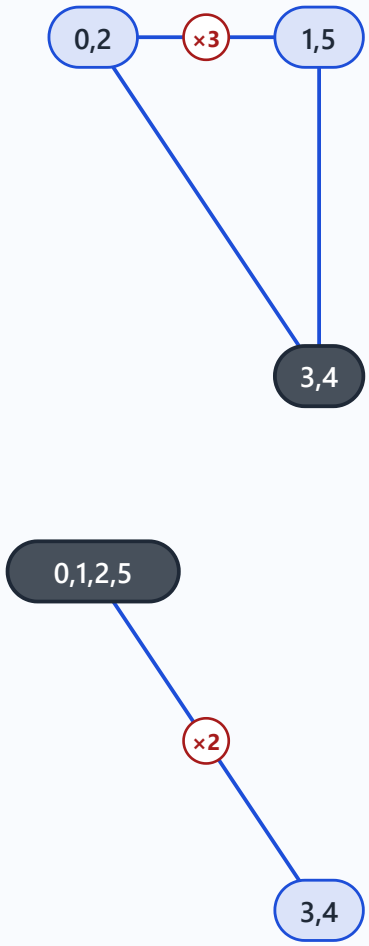
adım	büzülen kenar	süper düğümler	silinen öz-döngüler	kalan kenar
0	—	{0} {1} {2} {3} {4} {5}	—	8
1	(0, 2)	{0, 2} {1} {3} {4} {5}	(0, 2)	7
2	(1, 5)	{0, 2} {1, 5} {3} {4}	(1, 5)	6
3	(3, 4)	{0, 2} {1, 5} {3, 4}	(3, 4)	5
4	(0, 1)	{0, 1, 2, 5} {3, 4}	(0, 1), (1, 2), (0, 5)	2

Tablo 12.20: Başarılı bir Karger çalıştırması. Kalan kenarlar paralel kenarlar dahil sayılmıştır.

İlk büzmeden sonra {0, 2} süper düğümü ile 1 arasında iki paralel kenar oluşur: eski (0, 1) ve (1, 2). Üçüncü büzmeden sonra {0, 2} ile {1, 5} arasında üç paralel kenar vardır: (0, 1), (1, 2) ve (0, 5). Dördüncü adımda bunlardan biri, (0, 1), seçilir. Üç kenar birden öz-döngü olur ve silinir. İki süper düğüm kalır: {0, 1, 2, 5} ve {3, 4}. Aralarında (2, 3) ve (4, 5) kenarları vardır. Bulunan kesim 2'dir; bu çalıştırma başarılıdır.


0
1
5
2
4
3


0,2
×2
1
5
4
3



Şekil 12.15: Karger çalıştırması. Sol üstte başlangıç çizgesi (6 düğüm, 8 kenar). Sağ üstte (0, 2) büzüldü; {0, 2} ile 1 arasında iki paralel kenar (×2). Sol altta üç büzmeden sonra üç süper düğüm ve 5 kenar. Sağ altta iki süper düğüm kaldı; aralarında 2 kenar: minimum kesim bulundu. Koyu düğüm son büzmede oluşan süper düğümdür.

Başka bir rastgele seçim dizisi başka bir sonuç verir. Algoritma önce (1, 2), sonra (4, 5), (3, 4) ve (0, 1) kenarlarını seçsin. Bu kez süper düğümler {0, 1, 2} ve {3, 4, 5} olur. Aralarında (2, 3), (0, 5) ve (1, 5) kenarları kalır: kesim 3'tür ve minimum değildir. Hata, minimum kesimin bir kenarı olan (4, 5)'in büzülmesinden doğdu. Bir minimum kesim kenarı büzülünce o kesimin iki tarafı birleşir ve bu kesim artık bulunamaz.

12.9.3 Kod

Sunumda bu algoritmanın kodu verilmez. Aşağıdaki kısa sürüm, algoritmanın tamamını bir kenar listesi üzerinde gösterir. Süper düğümler bir `grup` dizisiyle izlenir: her düğüm bir "temsilci" düğüme bağlıdır ve `kok` metodu bu bağları izleyerek temsilciyi bulur. İki düğümün temsilcisi aynıysa aynı süper düğümdedirler.

```
static int kargerKesim(int n, int[][] kenar) {
  int[] grup = new int[n];            // grup[v]: v'nin bağlı olduğu düğüm
  for (int v = 0; v < n; v++) grup[v] = v;
  int kalan = n;                      // süper düğüm sayısı
  while (kalan > 2) {
    int[] e = kenar[zar.nextInt(kenar.length)];
    int g1 = kok(grup, e[0]), g2 = kok(grup, e[1]);
    if (g1 == g2) continue;           // öz-döngü: yeniden çek
    grup[g1] = g2;                    // kenarı büz
    kalan--;
  }
  int kesim = 0;
  for (int[] e : kenar)
    if (kok(grup, e[0]) != kok(grup, e[1])) kesim++;
  return kesim;
}
static int kok(int[] grup, int v) {
  while (grup[v] != v) v = grup[v];
  return v;
}
```

Döngü, kenar listesinden düzgün rastgele bir kenar çeker. Kenarın iki ucu zaten aynı süper düğümdeyse bu kenar bir öz-döngüdür; atlanır ve yeniden çekilir. Değilse iki grup birleştirilir ve süper düğüm sayısı bir azalır. Bu "yeniden çekme" yöntemi doğru dağılımı verir. Öz-döngü olmayan her kenar aynı olasılıkla seçilir ve iki süper düğüm arasındaki paralel kenarlar ayrı ayrı sayılır. Bu, algoritmanın tanımındaki seçimle aynıdır. İki süper düğüm kalınca kenar listesi bir kez taranır ve iki ucu farklı gruplarda olan kenarlar sayılır.

```
static int kargerTekrar(int n, int[][] kenar, int T) {
  int enIyi = Integer.MAX_VALUE;
  for (int t = 0; t < T; t++)
    enIyi = Math.min(enIyi, kargerKesim(n, kenar));
  return enIyi;
}
```

`kargerTekrar`, algoritmayı T kez bağımsız olarak çalıştırır ve bulunan en küçük kesimi döndürür. Karger'in hatası tek yönlüdür: bulunan kesim minimumdan hiçbir zaman küçük olamaz. Bu yüzden en küçük sonucu almak doğrudur. Kod okunaklı olsun diye yazılmıştır, hızlı olsun diye değil. Basit uygulamalarda tek çalıştırma $O(n^2)$ sürer.

Kod Java ile örnek çizgede denendi. Betikli bir üreteçle (0, 2), (1, 5), (3, 4), (0, 1) sırası verildiğinde 2; (1, 2), (4, 5), (3, 4), (0, 1) sırası verildiğinde 3 döndü. Sonra `zar = new Random(42)` ile 200.000 bağımsız çalıştırma yapıldı. 91.392 çalıştırma 2, 88.725 çalıştırma 3, 19.883 çalıştırma 4 kenarlı bir kesim buldu. Başarı oranı 0,457'dir.

12.9.4 Başarı olasılığı

Teorem 12.4 (Karger)

n düğümlü bağlı bir çoklu çizgede, Karger algoritmasının tek bir çalıştırması belirli bir minimum kesimi en az $2 / (n(n-1))$ olasılıkla bulur.

İspat. Belirli bir minimum kesim C olsun ve c kenarı olsun. Algoritma C'nin hiçbir kenarını büzmezse sonunda tam olarak C'yi bulur. Her düğümün derecesi en az c'dir; aksi hâlde o düğümü tek başına ayıran kesim c'den küçük olurdu. Büzmeler kesimleri korur, bu yüzden kalan çizgede de her süper düğümün derecesi en az c'dir. k süper düğüm kaldığında derecelerin toplamı en az kc, kenar sayısı ise en az $kc/2$'dir. O anda C'nin bir kenarının seçilme olasılığı en çok $c / (kc/2) = 2/k$'dir. C'nin hiç dokunulmadan kalma olasılığı adımların çarpımıdır:

$$(1 - 2/n)(1 - 2/(n-1)) \cdots (1 - 2/3) = (n-2)/n \cdot (n-3)/(n-1) \cdot \ldots \cdot 1/3 = 2 / (n(n-1))$$

Çarpımda payların ve paydaların çoğu sadeleşir; paydada yalnız $n(n-1)$, payda yalnız $2 \cdot 1$ kalır. □

$n = 6$ için alt sınır $2/30 = 1/15 \approx 0{,}067$'dir. Örnek çizgede gerçek başarı olasılığı bundan çok yüksektir. Python ile bütün büzme dizileri kesirli aritmetikle tek tek hesaplandı: herhangi bir minimum kesimin bulunma olasılığı tam $16/35 \approx 0{,}457$'dir. Benzetimdeki 0,457 ile uyumludur. Farkın iki nedeni vardır. Birincisi, çizgede üç minimum kesim vardır ve herhangi birinin bulunması başarı sayılır. İkincisi, kanıttaki "her düğümün derecesi en az c" sınırı bu çizgede gevşektir: dereceler 2 ile 3 arasındadır.

Teoremin ilginç bir yan sonucu vardır. Farklı minimum kesimlerin bulunma olayları birbirini dışlar ve her birinin olasılığı en az $2/(n(n-1))$'dir. Olasılıkların toplamı 1'i aşamayacağı için bir çizgede en çok $n(n-1)/2$ farklı minimum kesim olabilir. Bu sınır sıkıdır: n düğümlü bir döngü çizgesinde iki kenarın her seçimi bir minimum kesim verir ve tam $n(n-1)/2$ minimum kesim vardır. Altı düğümlü döngüde Karger her çalıştırmada bir minimum kesim bulur, ama belirli bir kesimi yalnız 1/15 olasılıkla bulur.

Örnek 12.13

Beş düğümlü bir çizgede alt sınır nedir? Kanıttaki çarpım $(1 - 2/5)(1 - 2/4)(1 - 2/3) = 3/5 \cdot 2/4 \cdot 1/3 = 1/10$'dur; formül de $2/(5 \cdot 4) = 1/10$ verir. Gerçek başarı çizgeye göre daha yüksek olabilir.

12.9.5 Tekrarla güçlendirme

Tek çalıştırmanın başarı olasılığı $p \geq 2/(n(n-1))$ küçüktür. Algoritma T kez bağımsız çalıştırılıp en küçük sonuç alınırsa, hepsinin başarısız olma olasılığı en çok $(1-p)^T$'dir. $1 - x \leq e^{-x}$ eşitsizliği kullanılırsa

> $(1 - 2/(n(n-1)))^T \le e^{-2T/(n(n-1))}$

bulunur. $T = n(n-1)/2 \cdot \ln n$ seçilirse sağ taraf $e^{-\ln n} = 1/n$ olur. Toplam süre $O(n^2 \log n)$ çalıştırma × $O(n^2) = O(n^4 \log n)$'dir. Bu basit bir uygulamanın süresidir. Karger ve Stein'ın geliştirdiği özyinelemeli sürüm, büzmenin ilk adımlarının güvenli, son adımlarının riskli olduğu gözleminden yararlanır ve süreyi $O(n^2 \log^3 n)$'ye indirir.

Örnek 12.14

Altı düğümlü bir çizgede başarı en az $1/15$'tir. Başarısızlık olasılığını $1/6$'nın altına indirmek için kaç tekrar yeter? Üstel sınırla $(14/15)^T \le e^{-T/15} \le 1/6$ koşulu $T \ge 15 \ln 6 \approx 26{,}9$ verir; 27 tekrar yeterlidir. Sınır kullanılmadan doğrudan hesaplanırsa $(14/15)^{26} \approx 0{,}1663$ zaten $1/6 \approx 0{,}1667$'nin altındadır; 26 tekrar da yeter, 25 yetmez. Üstel sınır hesabı kolaylaştırır, karşılığında bir tekrar fazla ister.

Bu sınırlar en kötü çizgeler içindir. Örnek çizgede gerçek başarı 16/35 olduğundan 27 tekrarın hepsinin başarısız olma olasılığı $(19/35)^{27} \approx 6{,}9 \cdot 10^{-8}$'dir. Java'da `kargerTekrar(6, G, 27)` 20.000 kez çağrıldı (tohum 42) ve her seferinde 2 döndü. Yalnız 3 tekrarla başarı oranı 0,836 çıktı; kesin değer $1 - (19/35)^3 \approx 0{,}840$'tır.

Sık yapılan hata

Kenarı düzgün seçmemek. Önce rastgele bir düğüm, sonra onun rastgele bir komşusu seçilirse kenarlar eşit olasılıklı olmaz. Derecesi küçük düğümlerin kenarları daha sık seçilir. Bu düğümler ise çoğu zaman tam minimum kesimin kenarlarını taşır. Kanıt, her kenarın eşit olasılıkla seçildiğini varsayar; paralel kenarlar da ayrı ayrı sayılmalıdır.

12.10 Diğer uygulamalar

Rastgelelik algoritma tasarımının birçok alanında kullanılır. Aşağıdaki örnekler bunlardan birkaçıdır.

Çizge boyama. Farklı boyama düzenleri rastgele kararlarla denenir. Amaç, komşu düğümlere aynı rengi vermeden az renkli iyi boyamalar bulmaktır. Rastgele bir başlangıçtan yapılan yerel iyileştirmeler, deterministik bir sıranın takıldığı kötü çözümlerden kurtulmayı sağlar.

Rastgele derinlik öncelikli arama. Komşular rastgele bir sırayla gezilir. Bağlantılılık ve kapsayan ağaç gibi sonuçlar değişmez, ama elde edilen ağaç her seferinde farklıdır. Labirent üretimi bu fikre dayanır: bir ızgaranın rastgele kapsayan ağacı, her hücreden her hücreye tek yolu olan bir labirenttir.

Eşleştirme. Rastgele artımlı algoritmalar kenarları rastgele bir sırayla ele alır ve en büyük eşleştirmeye yaklaşan çözümler üretir. Rastgele sıra, kötü bir girdi sırasının algoritmayı yanıltmasını önler.

Yaklaşık sayma. Count-Min Sketch gibi veri yapıları rastgele seçilmiş hash fonksiyonlarıyla çalışır. Çok büyük bir veri akışında her öğenin kaç kez geçtiğini, küçük bir bellekle ve denetimli bir hatayla tahmin eder.

Matris işlemleri. Rastgele matris yöntemleri, büyük bir matrisin düşük ranklı yaklaşımlarını hızla üretir. Boyut indirgeme ve veri sıkıştırma bu yaklaşımları kullanır.

Hesaplamalı geometri. Dışbükey örtü, en yakın nokta çifti ve Delaunay üçgenlemesi için rastgele artımlı algoritmalar vardır. Noktalar rastgele bir sırayla eklenir ve beklenen süre $O(n \log n)$ olur.

Rastgele zamanlama (iş atama), benzetimli tavlama ve genetik algoritmalar gibi metasezgisel yöntemler de aramada rastgelelikten yararlanır. Bu yöntemler büyük arama uzaylarında iyi çözümler bulur, ama optimuma ulaşacaklarını garanti etmez. Bu yönüyle Las Vegas ve Monte Carlo algoritmalarından ayrılır: onların doğruluğu ya da hata olasılığı kanıtlanmış bir sınırla verilir.

12.11 Sözde rastgele sayılar

Notta sürekli "rastgele bir sayı seçilir" dendi. Ama bilgisayar deterministik bir makinedir; gerçek rastgelelik üretmez. Programlar bunun yerine **sözde rastgele sayılar** (*pseudo-random number*) kullanır. Bunlar deterministik bir formülle üretilen, ama istatistiksel testlerde rastgele gibi davranan sayılardır. Formülün başlangıç değerine **tohum** (*seed*) denir.

En eski ve en basit yöntem **doğrusal eşlik üretecidir** (*linear congruential generator*, LCG):

> $X_{k+1} = (a \cdot X_k + c) \bmod m$

Örnek 12.15

$a = 5$, $c = 3$, $m = 16$ ve tohum $X_0 = 7$ olsun. Dizi 7, 6, 1, 8, 11, 10, 5, 12, 15, 14, 9, 0, 3, 2, 13, 4 diye ilerler ve 16 adım sonra yeniden 7'ye döner. Örneğin $5 \cdot 7 + 3 = 38$ ve $38 \bmod 16 = 6$'dır. Bu üreteç 0..15 arasındaki her değeri bir kez üretir; dizi ilk bakışta düzensiz görünür. Ama son bitlere bakıldığında zayıflık hemen görülür: sayılar tek, çift, tek, çift diye sırayla gelir. LCG'lerin düşük bitleri kısa dönemlidir.

`java.util.Random` bir LCG'dir. Parametreleri $a = 25.214.903.917$, $c = 11$ ve $m = 2^{48}$'dir. Sınıf, örnekteki zayıflık nedeniyle 48 bitlik durumun yalnız üst bitlerini çıktı olarak kullanır. Böyle bir üreteç algoritma deneyleri ve benzetimler için yeterlidir. Ama birkaç ardışık çıktısından iç durumu hesaplanabilir ve sonraki bütün sayılar öngörülebilir.

Aynı tohum her zaman aynı diziyi verir. Bu bir zayıflık değil, değerli bir özelliktir. `new Random(42)` ile başlayan bir program her çalıştırmada aynı "rastgele" kararları verir. Hata ayıklamada bir hatanın yeniden üretilmesi, deneylerin tekrarlanabilmesi ve testlerin her seferinde aynı sonucu vermesi bu sa-

yede sağlanır. Bu notun bütün benzetimleri de sabit tohumla yapılmıştır. π izlemesindeki sekiz nokta `new Random(2024)` ile her seferinde yeniden elde edilir.

Amaca göre farklı üreteçler kullanılır. Kriptografide, yani anahtar, parola ya da oturum belirteci üretirken öngörülemezlik şarttır; bunun için `java.security.SecureRandom` kullanılır. Çok iş parçacıklı programlarda tek bir `Random` nesnesini paylaşmak iş parçacıklarını birbirine bekletir; `ThreadLocalRandom.current()` her iş parçacığına kendi üretecini verir.

ifade	değer aralığı
`zar.nextInt(n)`	0 … n − 1
`alt + zar.nextInt(ust - alt + 1)`	alt … ust
`zar.nextDouble()`	[0, 1)
`zar.nextInt(6) + 1`	zar atışı: 1 … 6

Tablo 12.21: Sık kullanılan rastgele aralıklar (`zar` bir `Random` nesnesidir).

Tablodaki ikinci satır rastgele pivot seçiminde kullanılan kalıptır. Aralıkta *ust − alt + 1* değer vardır; `nextInt`'e bu sayı verilir ve sonuç *alt* kadar kaydırılır. `+ 1` unutulursa *ust* hiç seçilmez. Bu, Fisher–Yates'teki `nextInt(i)` hatasının aynısıdır.

Sık yapılan hata

Her çağrıda `new Random()`. Rastgele sayı gereken her yerde yeni bir üreteç oluşturmak gereksiz nesne üretir. Daha kötüsü, art arda oluşturulan üreteçlerin tohumları birbirine yakın ve ilişkili olabilir. Program boyunca tek bir `Random` nesnesi paylaşılır; notun kodlarındaki `static Random zar` bu yüzden sınıf düzeyindedir.

12.12 Sık yapılan hatalar

Rastgele algoritmalardaki hatalar çoğu zaman derleme hatası vermez ve tek bir çalıştırmada da görünmez. Program çalışır, makul görünen sonuçlar üretir, ama olasılık dağılımı ya da analiz yanlıştır. Aşağıdaki liste not boyunca karşılaşılan hataları toplar.

Sık yapılan hata

Saf karıştırma. Her *i* için `nextInt(n)` ile takas yanlıdır: n^n eşit olasılıklı yol *n!* permütasyona eşit dağılamaz. Doğrusu `nextInt(i + 1)`'dir. `nextInt(i)` ise Sattolo algoritmasını verir ve yalnız tek çevrimli permütasyonları üretir.

Sık yapılan hata

"Rastgele pivot en kötü durumu yok eder." En kötü durum hâlâ $O(n^2)$'dir. Değişen, hiçbir girdinin sürekli kötü olmaması ve kötü durumun olasılığının çok küçük olmasıdır: $n = 20$'de yaklaşık $2 \cdot 10^{-13}$.

Sık yapılan hata

Ortalama durum ile beklenen süreyi karıştırmak. Ortalama durum bir girdi dağılımı varsayar. Beklenen süre ise her sabit girdi için algoritmanın yazı-turaları üzerinden hesaplanır ve en kötü girdide de geçerlidir.

Sık yapılan hata

Monte Carlo yanıtını kesin saymak. Miller–Rabin'de "asal" yanıtı yalnız muhtemelen doğrudur; tur sayısı k, hata sınırını 4^{-k} olarak belirler. Karger'in tek çalıştırması çoğu zaman minimum olmayan bir kesim verir; birçok çalıştırmanın en küçüğü alınır.

Sık yapılan hata

Taşma. `int` ile `x * x % n`, $n > 46.341$ için taşabilir; `long` ile sınır yaklaşık $3 \cdot 10^9$'dur. Daha büyük modüllerde `Math.multiplyHigh` ya da `BigInteger.modPow` kullanılır.

Sık yapılan hata

Her çağrıda yeni üreteç ve düzgün olmayan seçim. Her çağrıda `new Random()` oluşturmak gereksizdir ve tohumlar ilişkili olabilir; tek bir `Random` paylaşılır. Karger'de kenar doğrudan, düzgün olarak seçilir; önce düğüm, sonra komşu seçmek düzgün değildir.

Sık yapılan hata

Aralık sınırında bir eksik. `alt + zar.nextInt(ust - alt)` yazılırsa *ust* hiç seçilmez. Rastgele pivotta bu, son öğenin hiçbir zaman pivot olamaması demektir; analizdeki "her öğe $1/n$ olasılıkla" varsayımı bozulur.

12.13 Özet

- Rastgele algoritma girdiye ek olarak rastgele bitler kullanır. Süre ve bazen çıktı bir rastgele değişkendir. Beklenen süre, her girdi için algoritmanın kendi seçimleri üzerinden alınır.
- Analizin temel araçları beklenen değerin doğrusallığı, gösterge değişkenleri, geometrik dağılım ($1/p$ deneme), bağımsız tekrar (q^k) ve Markov eşitsizliğidir.
- Las Vegas algoritması her zaman doğrudur, süresi rastgeledir. Monte Carlo algoritmasının süresi sınırlıdır, yanıtı küçük bir olasılıkla yanlıştır. Doğrulanabilir yanıtlı Monte Carlo, tekrarla Las Vegas'a; Las Vegas, Markov eşitsizliğiyle Monte Carlo'ya çevrilir.

Konu	Hatırlanacaklar
Fisher–Yates	`j = nextInt(i + 1)`, $\Theta(n)$, $n!$ sonucun her biri $1/n!$ olasılıklı; saf karıştırma yanlı
BogoSort	$p = 1/n!$, beklenen $n!$ karıştırma ve $\Theta(n \cdot n!)$ süre; en iyi $O(n)$, en kötü sınırsız
Rastgele QuickSort	$P(z_i, z_j \text{ karşılaştırılır}) = 2/(j - i + 1)$; beklenen $2(n+1)H_n - 4n \approx 1{,}39\, n \log_2 n$ karşılaştırma, her girdi için; en kötü $O(n^2)$
QuickSelect	tek tarafa iner; beklenen $O(n)$ (ortanca için $\approx 3{,}39\, n$), en kötü $O(n^2)$
π tahmini	$4 \cdot$ iç $/\, n$; standart sapma $1{,}642/\sqrt{n}$; bir basamak için 100 kat nokta
Miller–Rabin	$n - 1 = 2^s d$; dizi 1 ile başlamalı ya da $n - 1$ içermeli; hata $\le 4^{-k}$; "bileşik" kesin; $O(k \log^3 n)$
Karger	rastgele kenar büzme; başarı $\ge 2/(n(n-1))$; $n(n-1)/2 \cdot \ln n$ tekrarla başarısızlık $\le 1/n$

Tablo 12.22: Notta incelenen algoritmaların özeti.

- Bir algoritmanın hata olasılığı, bağımsız tekrarlarla üstel hızla küçültülür. Tek yönlü hatada tek bir kesin yanıt yeterlidir; Karger'de en küçük sonuç alınır.
- Bilgisayarın ürettiği sayılar sözde rastgeledir. Sabit tohum deneyleri tekrarlanabilir kılar; kriptografide `SecureRandom` gerekir.

12.14 Kavram listesi

rastgele algoritma *randomized algorithm* · beklenen değer *expected value* · beklenen çalışma süresi *expected running time* · ortalama durum *average case* · gösterge değişkeni *indicator random variable* · geometrik dağılım *geometric distribution* · Markov eşitsizliği *Markov's inequality* · Las Vegas algoritması *Las Vegas algorithm* · Monte Carlo algoritması *Monte Carlo algorithm* · tekrarla güçlendirme *amplification* · Fisher–Yates karıştırma *Fisher–Yates shuffle* · pivot *pivot* · bölümleme *partition* · rastgele seçme (QuickSelect) *quickselect* · asallık testi *primality test* · tanık, güçlü yalancı *witness, strong liar* · Carmichael sayısı *Carmichael number* · minimum kesim *minimum cut* · kenar büzme *edge contraction*

sözde rastgele sayı *pseudo-random number* | tohum *seed*

12.15 Kendini Dene

1. Fisher–Yates karıştırması [1, 2, 3, 4, 5, 6] dizisine uygulanıyor. *i = 5, 4, 3, 2, 1* için çekilen *j* değerleri sırasıyla 2, 4, 0, 1, 0'dır. Her adımdan sonra dizi nedir?

   Cevap

   i = 5, j = 2: [1, 2, 6, 4, 5, 3]. *i = 4, j = 4*: değişmez. *i = 3, j = 0*: [4, 2, 6, 1, 5, 3]. *i = 2, j = 1*: [4, 6, 2, 1, 5, 3]. *i = 1, j = 0*: [6, 4, 2, 1, 5, 3]. Beş takas yapıldı; biri öğeyi yerinde bıraktı.

2. Saf karıştırma beş öğeli bir diziye uygulanırsa kaç eşit olasılıklı yol vardır? Sonucun düzgün olamayacağı nereden anlaşılır? Fisher–Yates'te yol sayısı kaçtır?

   Cevap

   $5^5 = 3125$ yol ve $5! = 120$ permütasyon vardır. $3125 / 120 \approx 26{,}04$ tam sayı değildir; 3125 tek, 120 çifttir. Permütasyonlar eşit sayıda yolla üretilemez. Fisher–Yates'te $5 \cdot 4 \cdot 3 \cdot 2 = 120$ yol vardır ve her permütasyon tam bir kez elde edilir.

3. Sekiz farklı öğeli bir dizide BogoSort'un beklenen karıştırma sayısı kaçtır? Algoritmanın bu kadar karıştırmadan sonra hâlâ bitmemiş olma olasılığı yaklaşık kaçtır?

   Cevap

   $8! = 40.320$ karıştırma. Bitmemiş olma olasılığı $(1 - 1/40.320)^{40320} \approx e^{-1} \approx 0{,}368$'dir. Geometrik dağılımda beklenen değere kadar bitmeme olasılığı büyük *n*'lerde yaklaşık $1/e$'dir.

4. *a = [9, 4, 7, 1, 6, 3, 8]* için `bol(a, 0, 6)` çağrılıyor ve `zar.nextInt(7)` 4 döndürüyor. Bölümlemeden sonra dizi, dönüş değeri ve karşılaştırma sayısı nedir?

   Cevap

   Pivot `a[4] = 6`'dır ve sona taşınır: [9, 4, 7, 1, 8, 3, 6]. 4, 1 ve 3 sırayla sola alınır: [4, 9, 7, 1, 8, 3, 6], [4, 1, 7, 9, 8, 3, 6], [4, 1, 3, 9, 8, 7, 6]. Pivot $k = 3$'e yerleşir: [4, 1, 3, 6, 8, 7, 9]. Dönüş 3, karşılaştırma sayısı $ust - alt = 6$'dır.

5. Rastgele pivotlu hızlı sıralamada 10 öğeli bir dizide z_3 ile z_7 hangi olasılıkla karşılaştırılır? Beş

öğeli bir dizide beklenen karşılaştırma sayısı kaçtır?

Cevap

$\{z_3, \ldots, z_7\}$ kümesinde 5 öğe vardır; ilk pivot z_3 ya da z_7 olmalıdır: $2/5$. Olasılık dizinin boyutuna bağlı değildir. Beş öğe için $H_5 = 137/60$ ve $E[X] = 2 \cdot 6 \cdot 137/60 - 20 = 7{,}4$'tür. En kötü durum $5 \cdot 4/2 = 10$ karşılaştırmadır.

6. Monte Carlo yöntemiyle 10.000 noktadan 7.860'ı çeyrek dairenin içine düştü. π tahmini nedir? Bu nokta sayısında tahminin standart sapması yaklaşık kaçtır?

Cevap

$4 \cdot 7.860 / 10.000 = 3{,}144$. Standart sapma $1{,}642 / \sqrt{10.000} \approx 0{,}016$'dır. Gerçek hata $|3{,}144 - \pi| \approx 0{,}0024$'tür; tipik hatadan küçüktür.

7. $n = 91$ ve $a = 3$ için Fermat testi ne söyler? Miller–Rabin aynı tabanla ne söyler?

Cevap

$3^{90} \equiv 1 \pmod{91}$ olduğundan Fermat testi geçilir; 3, 91 için bir Fermat yalancısıdır. Miller–Rabin'de $90 = 2^1 \cdot 45$, yani $s = 1$, $d = 45$'tir. $3^{45} \equiv 27 \pmod{91}$; 27 ne 1 ne de 90'dır ve $s = 1$ olduğu için başka kare alınmaz. 3 bir tanıktır: 91 = 7 · 13 kesin bileşiktir.

8. `usMod(3, 13, 17)` çağrısında her turdan sonra `sonuc` ve `a` nedir? Dönüş değeri kaçtır?

Cevap

$13 = 1101_2$. Bit 1: sonuc = 3, a = 9. Bit 0: sonuc = 3, a = 81 mod 17 = 13. Bit 1: sonuc = 39 mod 17 = 5, a = 169 mod 17 = 16. Bit 1: sonuc = 80 mod 17 = 12, a = 256 mod 17 = 1. Dönüş 12'dir: $3^{13} \bmod 17 = 12$.

9. Sekiz düğümlü bir çizgede Karger algoritmasının tek çalıştırması için başarı alt sınırı nedir? Başarısızlık olasılığını $1/8$'in altına indirmek için üstel sınırla kaç tekrar yeter? Kesin hesap kaç tekrar verir?

Cevap

Alt sınır $2/(8 \cdot 7) = 1/28$'dir. $e^{-T/28} \le 1/8$ için $T \ge 28 \ln 8 \approx 58{,}2$, yani 59 tekrar. Doğrudan $(27/28)^T \le 1/8$ çözülürse $T \ge \ln 8 / \ln(28/27) \approx 57{,}2$, yani 58 tekrar yeter.

10. Beklenen süresi 100 ms olan bir Las Vegas algoritması 300 ms sonra durduruluyor. Yanıtsız kalma olasılığı için ne söylenebilir? Algoritma bu kuralla en çok beş kez bağımsız olarak yeniden başlatı-

lırsa hiç yanıt alınamama olasılığı en çok kaçtır?

Cevap

Markov eşitsizliğine göre $P(T \geq 300) \leq 100/300 = 1/3$'tür. Beş bağımsız denemenin hepsinin yanıtsız kalma olasılığı en çok $(1/3)^5 \approx 0{,}0041$'dir. Toplam süre en çok 1,5 saniyedir; bu yeni algoritma bir Monte Carlo algoritmasıdır.

12.16 İleri okuma

- R. Motwani ve P. Raghavan, *Randomized Algorithms* (1995): rastgele algoritmaların klasik başvuru kitabıdır; Las Vegas ve Monte Carlo türlerini, Karger algoritmasını ve asallık testlerini kanıtlarıyla işler.
- M. Mitzenmacher ve E. Upfal, *Probability and Computing* (2. baskı, 2017): gösterge değişkenleri, Markov ve Chernoff sınırları gibi olasılık araçlarını algoritma örnekleri üzerinden öğretir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 5, 7 ve 9: olasılıksal analiz, rastgele pivotlu hızlı sıralamanın ayrıntılı analizi ve beklenen doğrusal süreli seçme.
- D. E. Knuth, *The Art of Computer Programming*, Cilt 2 (3. baskı, 1997): sözde rastgele sayı üreteçlerini, doğrusal eşlik üreteçlerinin kuramını ve karıştırma algoritmasını ayrıntılı inceler.

12.17 Kaynaklar

1. R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
2. M. Mitzenmacher, E. Upfal, *Probability and Computing*, 2. baskı, Cambridge University Press, 2017.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 5, 7, 9, 31).
4. C. A. R. Hoare, “Quicksort”, *The Computer Journal* 5(1), 1962.
5. M. O. Rabin, “Probabilistic Algorithm for Testing Primality”, *Journal of Number Theory* 12(1), 1980.
6. D. R. Karger, “Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm”, SODA, 1993.
7. D. E. Knuth, *The Art of Computer Programming*, Cilt 2: Seminumerical Algorithms, 3. baskı, Addison-Wesley, 1997 (Bölüm 3.4.2: karıştırma).

KISIM V · BÖLÜM 13

Çevrim İçi Algoritmalar

Dersteki algoritmaların çoğu girdinin tamamını baştan görür. Sıralama algoritması bütün diziyi, Dijkstra algoritması bütün çizgeyi elinde tutar. Gerçek sistemlerde ise girdi çoğu zaman parça parça gelir. Önbellek, bir sonraki sayfa isteğini bilmeden hangi sayfayı atacağına karar verir. Bir borsa programı yarınki fiyatı bilmeden bugün alım ya da satım yapar. Bu tür kararlar bir kez verilince geri alınamaz. Çevrim içi algoritmalar bu koşullarda çalışan algoritmalardır. Not önce çevrim içi problem modelini ve rekabetçi analizi kurar: bir çevrim içi algoritma, geleceği bilen en iyi çözümle karşılaştırılarak değerlendirilir. Ardından üç klasik problem işlenir: kayak kiralama, sayfalama ve liste güncelleme. Her birinde hem algoritma hem de "daha iyisi olamaz" diyen alt sınır kanıtlanır. Notun ikinci yarısı veri akışlarına ayrılmıştır. Hareketli ortalama, çevrim içi ortanca, rezervuar örnekleme ve akışta örüntü arama, her öğeye bir kez bakan ve az bellek kullanan algoritmalardır. Not, artımlı öğrenme ve konunun genel bir karşılaştırmasıyla biter.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum13/src`

ÖĞRENME ÇIKTILARI

Model

Çevrim içi ve çevrim dışı problem modelini ayırt etmek; geri alınamaz karar kavramını örneklerle açıklamak.

Rekabetçi analiz

Rekabet oranını tanımlamak ve düşman modeliyle alt sınır kurmak. Kayak kiralama probleminde başa baş stratejisinin $2 - 1/B$ oranını ve bu oranın deterministik algoritmalar için en iyi değer olduğunu kanıtlamak.

Sayfalama ve listeler

FIFO, LRU ve OPT ile sayfa hatalarını adım adım hesaplamak; Belady anomalisini göstermek; LRU'nun k-rekabetçi, öne taşı kuralının 2-rekabetçi olduğunu yorumlamak.

Veri akışları
Hareketli ortalama, çevrim içi ortanca, rezervuar örnekleme ve kayan hash ile örüntü aramayı Java ile yazmak; öğe başına zamanı ve belleği analiz etmek.

13.1 Çevrim içi problem modeli

Bir algoritmanın girdisi iki biçimde gelebilir. Birinci biçimde girdinin tamamı algoritma çalışmaya başlamadan hazırdır. İkinci biçimde girdi zaman içinde, parça parça açığa çıkar. Algoritma her parçaya o anda bir cevap vermek zorundadır. Bu bölüm ikinci biçimi inceler. Önce modelin kendisi, sonra bu modelde "iyi algoritma" kavramının nasıl ölçüldüğü ele alınır.

13.1.1 Bilgi parça parça gelince

Gündelik kararların çoğu eksik bilgiyle verilir. Satranç oyuncusu rakibin bir sonraki hamlesini bilmeden hamlesini yapar. Hamle bir kez yapılınca geri alınamaz. Rakip hamlesini yaptıktan sonra oyuncu "keşke öbür taşı oynasaydım" diyebilir, ama tahtadaki durum değişmez. Bir sürücü de trafik sıkışıklığında hangi yola sapacağına o anki bilgiyle karar verir. Yolun ileride kapalı olduğunu ancak oraya varınca öğrenir.

Labirentte yürüyen bir yolcu bu durumun iyi bir örneğidir. Labirentin tamamı görünmez; yolcu yalnız önündeki koridoru görür. Her kavşakta o ana kadar gördükleriyle bir yön seçer. Kuşbakışı haritası olan biri aynı labirentte en kısa yolu hiç yanılmadan bulur. Yolcu ise bazı çıkmazlara girmek zorunda kalır. Çevrim içi algoritmanın sorusu şudur: haritası olmayan yolcu, haritası olana göre en kötü durumda ne kadar fazla yol yürür?

Bilgisayar sistemlerinde de aynı yapı sık görülür. Trafik ışıklarını yöneten bir program, kavşaktaki yoğunluğu anlık olarak ölçer ve yeşil ışığın süresini buna göre ayarlar. Program bir dakika sonra hangi yönden kaç araç geleceğini bilmez. Beklemek de bir seçenek değildir: ışığın şimdi bir renk göstermesi gerekir. **Çevrim içi algoritma** (*online algorithm*) bu koşullarda çalışan algoritmadır.

Tanım 13.1

Çevrim içi algoritma, girdisini $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ biçiminde bir istek dizisi olarak sırayla alır. σ_t geldiğinde algoritma bir karar verir. Bu karar yalnız $\sigma_1, \ldots, \sigma_t$ isteklerine dayanır ve sonradan değiştirilemez. **Çevrim dışı algoritma** (*offline algorithm*) ise dizinin tamamını baştan görür.

Tanımdaki iki koşul birlikte önemlidir. Birincisi, gelecek bilinmez: σ_t geldiğinde σ_{t+1} hakkında hiçbir bilgi yoktur. İkincisi, verilen karar geri alınamaz. Yalnız birinci koşul olsaydı algoritma bütün girdiyi bekleyip sonra karar verebilirdi. Yalnız ikinci koşul olsaydı geleceği bilen algoritma her seferinde doğru kararı verirdi. Zorluk iki koşulun bir arada olmasından doğar.

13.1.2 Çevrim içi ve çevrim dışı

İki model arasındaki fark, girdinin *içeriğinde* değil, algoritmanın girdiyi *ne zaman* gördüğündedir. Aynı istek dizisi iki modelde de işlenebilir. Çevrim dışı algoritma bütün diziye bakar, en iyi kararları hesaplar ve sonra uygular. Çevrim içi algoritma her isteğe geldiği anda cevap verir. Tablo 13.1 iki modeli beş açıdan karşılaştırır.

	Çevrim dışı	**Çevrim içi**
Girdi	Tamamı baştan bilinir	Parça parça gelir; σ_t gelince σ_{t+1} bilinmez
Karar	Bütün girdiye bakılarak verilir	σ_t geldiği anda verilir, geri alınamaz
Örnek	Sıralama, Dijkstra, sırt çantası (dinamik programlama)	Önbellek yönetimi, kayak kiralama, borsa
En iyi çözüm	OPT hesaplanabilir	OPT ancak geriye dönüp bakınca bilinir
Ölçüt	Zaman ve alan karmaşıklığı	Rekabet oranı: ALG / OPT

Tablo 13.1: Çevrim dışı ve çevrim içi problem modelleri.

Tablodaki son satır bölümün ana fikrini taşır. Çevrim dışı bir algoritma için "iyi" demek çoğunlukla "hızlı" demektir. Algoritma en iyi çözümü bulur; soru, bunu ne kadar sürede bulduğudur. Çevrim içi bir algoritma ise en iyi çözümü çoğu zaman bulamaz, çünkü en iyi çözüm geleceğe bağlıdır. Burada soru, verilen kararların toplam maliyetinin en iyi çözümden ne kadar uzak olduğudur. Bu uzaklığı ölçen sayıya **rekabet oranı** denir.

"Çevrim içi" sözcüğü burada internet bağlantısı anlamına gelmez. Sözcük girdinin zaman içinde açığa çıkmasını anlatır. Bir sıralama algoritması bile çevrim içi çalışabilir ya da çalışamaz. Bunu görmek için iki tanıdık sıralama algoritmasına bakmak yeterlidir.

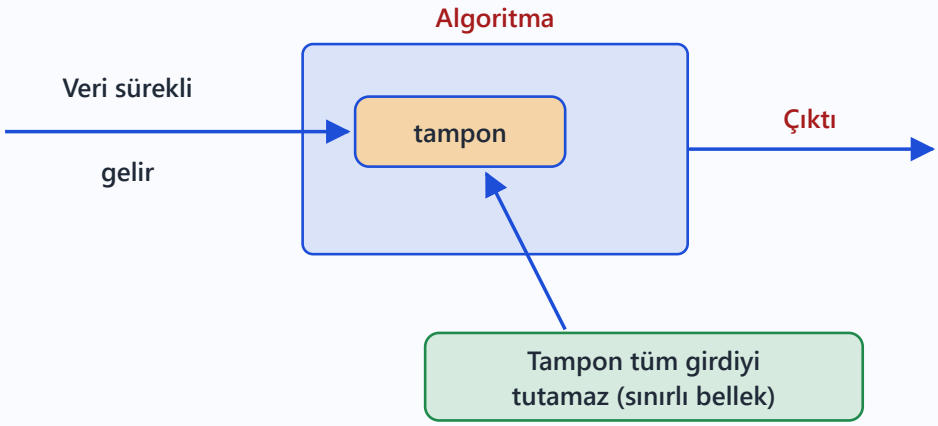
Örnek 13.1

Soru. Sayılar tek tek geliyor: 5, 2, 8, 1. Her sayı geldiğinde o ana kadarki sayıların sıralı hâli hazır olmalıdır. Eklemeli sıralama (*insertion sort*) ve seçimli sıralama (*selection sort*) bu koşulu sağlayabilir mi?

Çözüm. Eklemeli sıralama her yeni sayıyı sıralı kısma yerleştirir. 5 gelince dizi [5] olur. 2 gelince 5 bir sağa kayar ve dizi [2, 5] olur. 8 gelince karşılaştırma ilk adımda biter: [2, 5, 8]. 1 gelince üç öğe kayar: [1, 2, 5, 8]. Her adımda dizi sıralıdır ve algoritma geleceği bilmeye hiç ihtiyaç duymaz. Seçimli sıralama ise ilk turda bütün girdinin en küçüğünü seçer. 5 geldiğinde onu ilk konuma yazamaz, çünkü daha küçük bir sayının gelip gelmeyeceğini bilmez. Nitekim en küçük sayı (1) en son gelir. Eklemeli sıralama çevrim içi çalışabilir, seçimli sıralama çalışamaz. Birleştirmeli sıralama ve hızlı sıralama da girdiyi ikiye bölmek için girdinin tamamına ihtiyaç duyar.

13.1.3 Sınırlı bellek ve öğe başına iş

Çevrim içi modelin pratikteki en yaygın biçimi sürekli gelen veridir. Algılayıcılar saniyede binlerce ölçüm üretir. Bir ağ yönlendiricisinden her saniye milyonlarca paket geçer. Bu verinin tamamını saklamak çoğu zaman olanaksızdır. Şekil 13.1 bu durumu gösterir: veri algoritmaya sürekli gelir, algoritma küçük bir tamponla (*buffer*) çalışır ve hemen çıktı üretir.



Şekil 13.1: Çevrim içi algoritma modeli. Veri sürekli gelir; algoritma her parçayı geldiğinde işler ve çıktı üretir. Tampon girdinin tamamını tutamaz.

Tampon girdinin tamamını tutamıyorsa geçmiş ancak bir *özet* olarak saklanabilir. Özet; bir toplam, bir sayaç, iki heap ya da küçük bir örneklem olabilir. Doğru özeti seçmek, akış algoritması tasarımının özüdür. Bir özetin iki özelliği olmalıdır. Birincisi, yeni bir öğe geldiğinde özet hızla güncellenebilmelidir. İkincisi, sorulan soru özetten cevaplanabilmelidir. Örneğin “şu ana kadarki ortalama nedir?” sorusu için öğe sayısı ile toplam yeterlidir. Her yeni öğede toplam ve sayaç birer kez güncellenir, ortalama bir bölmeyle bulunur.

Bu yüzden akış algoritmalarında iki maliyet ölçülür: *öğe başına iş* ve *bellek*. Öğe başına işin çoğunlukla $O(1)$ ya da $O(\log n)$ olması istenir. Öğe başına $O(n)$ iş yapan bir algoritma, n öğe geldiğinde toplam $O(n^2)$ iş yapar. Saniyede bir milyon öğe gelen bir akışta bu kabul edilemez. Bellek de akışın uzunluğundan bağımsız ya da ona göre çok küçük olmalıdır.

13.1.4 Uygulama alanları

Çevrim içi problemler bilgisayar biliminin hemen her alanında karşımıza çıkar. Tablo 13.2 başlıca örnekleri toplar. Örneklerin ortak noktası üçtür: ortam dinamiktir, karar gerçek zamanlı verilir ve geleceğe dair bilgi eksiktir.

Alan	Çevrim içi karar
Borsa	Anlık fiyata göre alım ya da satım; gelecek fiyat bilinmez.
Önbellek	İşlemci, işletim sistemi ve web önbelleğinde hangi sayfanın bellekte kalacağı.

Alan	Çevrim içi karar
Yönlendirme	Bağlantıları sürekli değişen bir ağda paketin hangi yoldan gideceği.
Kaynak tahsisi	Bant genişliğinin ve işlemci zamanının dinamik paylaşımı.
Zamanlama	Gelen işlerin makinelere atanması (iş çizelgeleme).
Reklam ve öneri	Kullanıcı sayfaya girdiğinde hangi reklamın ya da ürünün gösterileceği.
Makine öğrenmesi	Modelin her yeni örnekle güncellenmesi (artımlı öğrenme).
Güvenlik	Dolandırıcılığın ve dağıtık hizmet engelleme (*DDoS*) saldırılarının akış üzerinde yakalanması.
Otonom araçlar	Algılayıcı verisi geldikçe gerçek zamanlı karar.

Tablo 13.2: Çevrim içi kararların verildiği başlıca alanlar.

Bu alanların bazılarında karar geri alınamaz, bazılarında ise geri almak pahalıdır. Önbellekten atılan bir sayfa geri yüklenebilir, ama yükleme bir maliyet doğurur. Gösterilen bir reklam geri alınamaz; reklam yeri başka bir reklama verilemez. Model hep aynıdır: her kararın bir maliyeti vardır ve toplam maliyet en aza indirilmek istenir.

13.1.5 Güçlü ve zayıf yanlar

Çevrim içi algoritmaların en büyük gücü beklememeleridir. Girdinin bitmesini beklemezler; dinamik ortamlarda çalışırlar. Karar anında verildiği için gecikme düşüktür. Az bellekle çok büyük, hatta sonsuz akışları işleyebilirler. Rekabet oranı sayesinde en kötü durum için de bir garanti verilebilir: “ne olursa olsun, maliyet en iyi çözümün iki katını geçmez” gibi.

Zayıf yanlar da aynı yerden gelir. İlk adımlarda verilen yanlış bir karar sonraki adımları etkiler. Satrançta tek bir kötü hamle oyunu kaybettirebilir. Çevrim içi algoritma en iyi küresel çözümü bulmayı garanti etmez. Başlangıçtaki girdiye duyarlıdır: aynı algoritma bir dizide çok iyi, başka bir dizide kötü sonuç verebilir. Tasarımı ve analizi de çevrim dışı algoritmalarınkinden daha zordur. Analiz, algoritmanın davranışını henüz görülmemiş bütün olası geleceklere karşı değerlendirmek zorundadır.

13.1.6 Tasarım yaklaşımları

Çevrim içi algoritmalar birkaç ortak fikir üzerine kurulur. Bu fikirler birbirini dışlamaz; bir algoritma birkaçını birlikte kullanabilir.

- **Açgözlü yaklaşım** (*greedy*): gelen isteği o an en iyi görünen biçimde karşılar ve küresel etkiyi düşünmez. Bir istek geldiğinde onu en yakın boş sunucuya atamak bu türdendir.
- **Özetleme**: geçmişi tümüyle saklamak yerine küçük bir özet tutar. Hareketli ortalama bir toplam ve bir kuyruk, çevrim içi ortanca iki heap, rezervuar örnekleme sabit boyutlu bir örneklem saklar.
- **Uyarlanır yaklaşım** (*adaptive*): geçmişteki isteklerden öğrenir ve davranışını zamanla değiştirir. LRU önbelleği yakın zamanda kullanılan sayfaları tutar. Öne taşı kuralı sık istenen öğeleri listenin başına getirir. Artımlı öğrenme modeli her yeni örnekle günceller.

- **Deterministik yaklaşım**: aynı girdiye her zaman aynı kararı verir. Bu öngörülebilirlik bir zayıflık da yaratır. Algoritmayı bilen kötü niyetli bir girdi üreticisi, yani düşman, onu kolayca tuzağa düşürebilir.
- **Rastgele yaklaşım** (*randomized*): kararlarına rastlantı katar. Düşman algoritmanın ne yapacağını önceden bilemez. Rezervuar örnekleme ve rastgele işaretleme algoritması bu türdendir.

Deterministik ile rastgele yaklaşım arasındaki fark, rekabetçi analizde sayılarla görülür. Kayak kiralamada en iyi deterministik oran 2'ye yakındır, rastgele strateji ise yaklaşık 1,58'e iner. Sayfalamada deterministik oran k'dir, rastgele işaretleme algoritmasınınki ise $2 \ln k$ civarındadır.

13.2 Rekabetçi analiz

Çevrim dışı algoritmalar için alışılmış ölçüt zaman karmaşıklığıdır. Çevrim içi algoritmalar için bu ölçüt yetmez. Önbellek politikası her istekte $O(1)$ iş yapabilir ama çok sayıda gereksiz sayfa hatası üretebilir. Asıl ölçülmesi gereken, kararların kalitesidir. Kaliteyi ölçmek için bir karşılaştırma noktası gerekir. Bu nokta, geleceği bilen en iyi algoritmadır.

13.2.1 Rekabet oranı

Bir istek dizisi σ verilsin. Çevrim içi algoritmanın bu dizideki toplam maliyetine $ALG(\sigma)$ denir. Dizinin tamamını baştan bilen en iyi çevrim dışı algoritmanın maliyetine de $OPT(\sigma)$ denir. OPT bir algoritmadan çok bir ölçüttür: her dizi için ulaşılabilecek en küçük maliyeti verir. Çevrim içi algoritma ne yaparsa yapsın, maliyeti OPT'den küçük olamaz. Soru, ondan ne kadar büyük olabileceğidir.

Tanım 13.2

Bir maliyet en küçükleme probleminde ALG çevrim içi algoritması, her istek dizisi σ için

$$ALG(\sigma) \le c \cdot OPT(\sigma) + \alpha$$

koşulunu sağlıyorsa ***c*-rekabetçidir** (*c-competitive*). Burada α, σ'dan bağımsız bir sabittir. $\alpha = 0$ ise algoritma *kesin* c-rekabetçidir. Bu koşulu sağlayan en küçük c değerine algoritmanın **rekabet oranı** (*competitive ratio*) denir. $\alpha = 0$ iken oran, bütün diziler üzerinden $ALG(\sigma) / OPT(\sigma)$ değerinin en küçük üst sınırıdır.

Tanımın üç yanı ayrıca vurgulanmalıdır. Birincisi, koşul *her* dizi için sağlanmalıdır. Rekabetçi analiz bir en kötü durum analizidir; ortalama durum ya da olasılık varsayımı içermez. “Gerçek dizilerde genelde iyi çalışıyor” demek bir rekabet oranı vermez. İkincisi, oran ne kadar 1'e yakınsa gelecek bilgisinin eksikliği o kadar az zarar verir. Oranı 1 olan bir çevrim içi algoritma geleceği bilen algoritma

kadar iyidir. Üçüncüsü, toplanan α sabiti başlangıç etkilerini dışarıda bırakır. Örneğin önbellek başlangıçta boşsa ilk birkaç hata her algoritmada olur. Bu hatalar uzun dizilerde önemsizdir ve α onları emer.

Tanım maliyet problemleri içindir: amaç maliyeti en aza indirmektir ve oran her zaman $c \geq 1$'dir. Bazı problemlerde ise bir kazanç en büyüklenir. Örneğin reklam yerleştirmede toplam gelir artırılmak istenir. Kazanç problemlerinde oran ters çevrilerek *OPT / ALG* olarak tanımlanır; böylece oran yine 1'den büyük ya da 1'e eşit olur.

Örnek 13.2

Soru. Bir çevrim içi algoritmanın dört istek dizisindeki maliyetleri 12, 30, 8 ve 50; aynı dizilerde OPT'nin maliyetleri 6, 20, 8 ve 25'tir. Yalnız bu dört dizi olsaydı algoritmanın rekabet oranı ne olurdu? Bu sayılar gerçek oran hakkında ne söyler?

Çözüm. Oranlar sırasıyla $12/6 = 2$, $30/20 = 1{,}5$, $8/8 = 1$ ve $50/25 = 2$'dir. Rekabet oranı en kötü durumu ölçtüğü için ortalamaya değil en büyük değere bakılır: 2. Ortalama oran 1,625 olsa da bu sayı bir garanti vermez. Gerçek rekabet oranı bütün olası diziler üzerinden tanımlanır. Bu dört dizi oranın en az 2 olduğunu gösterir. Oranın 2'yi aşmadığını göstermek içinse her dizi için geçerli bir kanıt gerekir.

13.2.2 Düşman modeli

Bir algoritmanın rekabet oranının *en az* c olduğunu göstermek için tek bir kötü dizi bulmak yeter. Hiçbir algoritmanın c'den iyi olamayacağını göstermek ise daha zordur. Bunun için her algoritmaya karşı ayrı bir kötü dizi kurulmalıdır. Bu kurguyu düzenli hâle getiren araç **düşman** (*adversary*) modelidir. Düşman, algoritmaya karşı oynayan hayalî bir rakiptir. İstek dizisini o üretir ve aynı diziyi OPT ile de çözer. Amacı $ALG(\sigma) / OPT(\sigma)$ oranını olabildiğince büyütmektir.

Düşmanın ne bildiği, analiz edilen algoritmanın türüne göre değişir.

- **Deterministik algoritmaya karşı düşman** algoritmanın kodunu bilir. Algoritma deterministik olduğu için düşman onun her adımda ne yapacağını önceden hesaplayabilir. Her adımda algoritmanın en çok zarar göreceği isteği üretir.
- **Habersiz düşman** (*oblivious adversary*) rastgele algoritmaya karşı kullanılır. Algoritmanın kodunu bilir ama yazı-tura sonuçlarını göremez. Bu yüzden diziyi baştan seçer. Algoritmanın *beklenen* maliyeti ölçülür.
- **Uyarlanır düşman** (*adaptive adversary*) algoritmanın rastgele kararlarını görür ve sonraki isteği buna göre seçer. Bu düşmana karşı rastgeleliğin faydası büyük ölçüde kaybolur.

Alt sınır kanıtları hep aynı kalıbı izler. “Hiçbir deterministik algoritma c'den iyi olamaz” demek için keyfî bir ALG ele alınır. Bu ALG'ye karşı $ALG(\sigma) / OPT(\sigma) \geq c$ yapan bir σ dizisi kurulur. ALG keyfî seçildiği için sonuç bütün deterministik algoritmalar için geçerlidir. Kalıbın gücü, düşmanın ALG'yi tanı-

masından gelir. Deterministik bir algoritma bir sonraki adımda ne yapacağını gizleyemez; düşman da tam o adımın açığını kullanır. Kayak kiralamada düşman, algoritma kayağı satın aldığı gün sezonu bitirir. Sayfalamada ise algoritmanın önbelleğinde bulunmayan sayfayı ister.

Sık yapılan hata

Rekabet oranını ortalama durumla karıştırmak. "LRU gerçek programlarda OPT'ye çok yakın sonuç veriyor, o hâlde oranı 1'e yakındır" çıkarımı yanlıştır. Rekabet oranı en kötü diziye göre tanımlanır. LRU'nun oranı k'dir, çünkü onu her istekte hataya düşüren bir dizi vardır. Gerçek dizilerdeki iyi başarım ayrı bir gözlemdir; yerellikten kaynaklanır ve bir garanti değildir.

13.3 Kayak kiralama

Kayak kiralama problemi, çevrim içi algoritmaların en yalın örneğidir. Tek bir karar vardır: ne zaman satın alınacağı. Buna karşın problem, rekabetçi analizin bütün parçalarını içerir: geleceği bilen OPT, basit bir strateji, stratejinin üst sınırı ve hiçbir stratejinin aşamayacağı alt sınır.

13.3.1 Problem

Bir kayakçı sezon boyunca her sabah kayak merkezine gider. Kayak takımı günlüğü 1 TL'ye kiralanabilir ya da B TL'ye satın alınabilir. Satın alınan takım sezonun geri kalanında bedavadır. Kayakçı sezonun kaç gün süreceğini bilmez. Her sabah yalnız "bugün de kayacak mıyım?" sorusunun cevabını öğrenir. Hava bir gün bozarsa sezon o gün biter. Toplam kayak günü sayısına d denir.

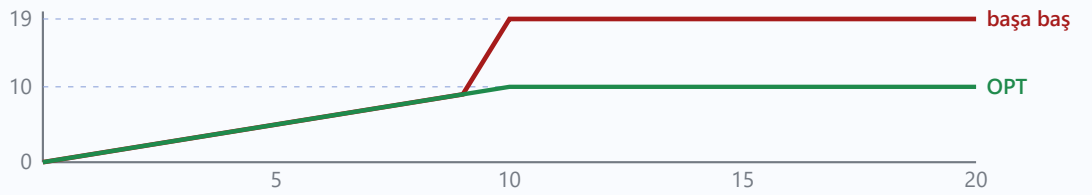
Tanım 13.3

Kayak kiralama problemi (*ski rental problem*): kiralama günlüğü 1, satın alma B birimdir. Kayak yapılan her gün başında algoritma iki seçenekten birini seçer: o gün için kiralamak ya da satın almak. Satın aldıktan sonra başka maliyet yoktur. Gün sayısı d önceden bilinmez. Amaç toplam maliyeti en aza indirmektir.

Geleceği bilen OPT için problem kolaydır. d biliniyorsa iki seçenek karşılaştırılır. Hep kiralamak d, ilk gün satın almak B tutar. Arada bir gün satın almak hiçbir zaman daha iyi değildir: önce birkaç gün kira ödeyip sonra satın almak, ilk gün satın almaktan pahalıdır. O hâlde

$$OPT(d) = \min(d, B)$$

olur. $d < B$ ise OPT hep kiralar, $d \geq B$ ise ilk gün satın alır. Çevrim içi algoritma ise d'yi bilmez. Hangi günü seçerse seçsin, düşman sezonu ona en kötü gelecek biçimde bitirebilir. Şekil 13.2, $B = 10$ için OPT'nin maliyetini ve aşağıda incelenen başa baş stratejisinin maliyetini gösterir.



Şekil 13.2: *B = 10* için maliyetler; yatay eksen gerçek gün sayısı *d*. OPT *min(d, 10)*'dur. Başa baş stratejisi *d < 10* iken OPT ile aynıdır, sonra 19'da sabitlenir. En kötü oran *d = B*'de görülür: *19/10 = 2 − 1/B*.

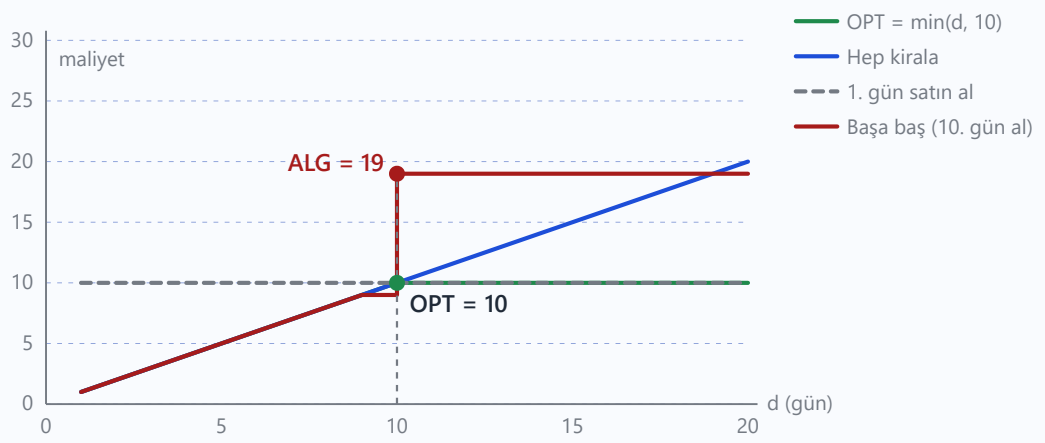
Problemin yapısı yalnız kayakla sınırlı değildir. Bir bulut sunucusu saatlik kiralanabilir ya da yıllık olarak satın alınabilir. Bir TCP bağlantısı açık tutulabilir ya da kapatılıp gerektiğinde yeniden açılabilir. Açık tutmak her saniye küçük bir maliyet doğurur, yeniden açmak ise tek seferlik büyük bir maliyettir. Bir iş parçacığı kilidin açılmasını dönerek bekleyebilir ya da uykuya geçebilir. Dönerek beklemek kira gibidir, uykuya geçip uyanmak satın alma gibidir. Hepsinde soru aynıdır: küçük ve sürekli bir maliyet ne zamana kadar ödenmeli, büyük ve tek seferlik maliyete ne zaman geçilmeli?

13.3.2 Üç stratejinin maliyeti

Deterministik bir strateji tek bir sayıyla tanımlanır: satın alma günü *k*. Strateji *k − 1* gün kiralar, *k*. gün hâlâ kayılıyorsa satın alır. Sezon *k*. günden önce biterse hiç satın almaz. Bu ailenin üç doğal üyesi vardır.

- **Hep kirala** (*k = ∞*): maliyet her zaman *d*'dir. *d* büyüdükçe OPT *B*'de sabit kalır, maliyet ise sınırsız artar. Oran *d / B* olur ve *d → ∞* iken sonsuza gider. Bu strateji rekabetçi değildir.
- **İlk gün satın al** (*k = 1*): maliyet her zaman *B*'dir. Ama sezon tek gün sürerse OPT 1'dir. Oran *B* olur. *B* büyükse bu çok kötüdür.
- **Başa baş** (*break-even*, *k = B*): *B − 1* gün kiralar, *B*. gün satın alır. Kiraya ödenen toplam, satın alma bedeline yaklaştığı anda satın alır.

Şekil 13.3 bu üç stratejinin maliyetini OPT ile birlikte, *B = 10* için gün sayısının fonksiyonu olarak çizer. Tablo 13.3 aynı değerleri birkaç *d* için sayılarla verir.



Şekil 13.3: *B = 10* için stratejilerin maliyeti. Hep kirala doğrusu sınırsız yükselir. İlk gün satın al 10'da sabittir. Başa baş stratejisi 9. güne kadar OPT ile çakışır, 10. günden itibaren 19'da kalır. En kötü durum *d = 10*'dur: ALG = 19, OPT = 10.

d	OPT	Hep kirala	İlk gün satın al	Başa baş
1	1	1 (1)	10 (10)	1 (1)
5	5	5 (1)	10 (2)	5 (1)
9	9	9 (1)	10 (1,11)	9 (1)
10	10	10 (1)	10 (1)	19 (1,9)
15	10	15 (1,5)	10 (1)	19 (1,9)
50	10	50 (5)	10 (1)	19 (1,9)
1000	10	1000 (100)	10 (1)	19 (1,9)

Tablo 13.3: *B = 10* için üç stratejinin maliyeti ve OPT'ye oranı (parantez içinde).

Tablo, stratejilerin zayıf yanlarını açıkça gösterir. Hep kirala kısa sezonlarda kusursuzdur, uzun sezonlarda felakettir. İlk gün satın al uzun sezonlarda kusursuzdur, kısa sezonlarda *B* kat kötüdür. Başa baş stratejisi hiçbir zaman kusursuz olmaz ama hiçbir zaman da 1,9 kattan kötü olmaz. Rekabetçi analiz tam bu tür bir dengeyi arar: en kötü durumu en iyi yapan stratejiyi.

13.3.3 Başa baş stratejisi: kod ve izleme

Başa baş stratejisinin Java kodu aşağıdadır. Metot, sezonun gerçek uzunluğu *d*'yi bir parametre olarak alır. Bu yalnız benzetim içindir: algoritma *d*'yi bilmez. Döngü her gün başında yalnız o gün kayılıp kayılmadığını "görür"; *d*'nin değerine göre bir karar vermez.

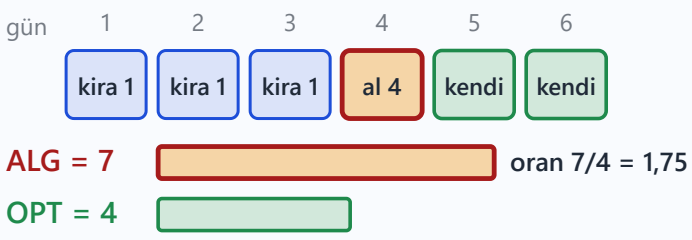
```
int basaBas(int B, int d) {  // d bilinmez
  int maliyet = 0;
  for (int gun = 1; gun <= d; gun++) {
    if (gun < B) {
      maliyet += 1;            // kirala
    } else {
      maliyet += B;            // satın al
      break;                   // ücret biter
    }
  }
  return maliyet;
}
```

Kod satır satır şöyle çalışır. `maliyet` sıfırdan başlar. Döngü 1. günden *d*. güne kadar döner; her tur, kayılan bir günü temsil eder. `gun < B` ise o gün kiralanır ve maliyet 1 artar. `gun` ilk kez *B*'ye eşit olduğunda `else` dalı çalışır: maliyete *B* eklenir ve `break` döngüden çıkar. Kayak artık kayakçınındır; kalan günler bedavadır. Sezon *B*. günden önce biterse döngü kendiliğinden sona erer ve hiç satın alma yapılmaz.

Metodun zamanı *O(min(d, B))*'dir: döngü en çok *B* tur döner. Bellek *O(1)*'dir. Ama çevrim içi algoritmalarda asıl ilgilenilen bu süre değil, döndürülen `maliyet`'in OPT'ye oranıdır. Tablo 13.4, *B = 4* ve *d = 6* için kodun izlemesini verir. Şekil 13.4 aynı çalışmanın son durumunu gösterir.

`gun`	`gun < B`	**Karar**	`maliyet`
1	true	kiralanır	1
2	true	kiralanır	2
3	true	kiralanır	3
4	false	satın alınır, `break`	3 + 4 = 7
5	—	döngü bitti; kayak bedava	7
6	—	döngü bitti; kayak bedava	7

Tablo 13.4: `basaBas(4, 6)` çağrısının izlemesi. Sezon gerçekte 6 gün sürüyor; algoritma bunu bilmiyor.



Şekil 13.4: *B = 4, d = 6*: üç gün kira, 4. gün satın alma, 5. ve 6. günler bedava. ALG = 7, OPT = *min(6, 4) = 4*, oran *7/4 = 1,75 = 2 − 1/B*.

Sonuçta ALG = 7'dir. OPT sezonun 6 gün süreceğini bildiği için ilk gün satın alır ve 4 öder. Oran *7/4 = 1,75*'tir; bu değer *2 − 1/4*'e eşittir. Sezon 4. günde bitseydi de ALG 7, OPT 4 olurdu. Sezon 3. günde bitseydi ALG 3 olurdu; OPT de 3 öderdi. Kodun genel bir sürümü satın alma gününü ayrı bir parametre olarak alır:

```
int kayakMaliyet(int B, int k, int d) {
  int maliyet = 0;
  for (int gun = 1; gun <= d; gun++) {
    if (gun == k) { maliyet += B; break; }  // satın al
    maliyet += 1;  // kirala
  }
  return maliyet;
}
```

Bu metot *k*. gün satın alan stratejinin maliyetini hesaplar. `kayakMaliyet(B, B, d)`, her *B* ve *d* için `basaBas(B, d)` ile aynı değeri verir. İki metot *1 ≤ B ≤ 15* ve *1 ≤ d ≤ 40* için çalıştırılarak karşılaştırıldı; bütün değerler eşit çıktı.

Sık yapılan hata

`break`'i unutmak. `else` dalında `break` yoksa döngü sürer ve *B*. günden sonra her gün yeniden *B* eklenir. Kayakçı her sabah yeni bir takım satın almış olur. *B = 4*, *d = 6* için maliyet *3 + 4 + 4 + 4 = 15* çıkar; *d* büyüdükçe oran sınırsız artar. Koşulu `gun <= B` yazmak da bir kayma hatasıdır. Bu durumda strateji *B* gün kiralar ve *B + 1*. gün satın alır. *d = B + 1* için maliyet *2B*, OPT *B* olur ve oran *2 − 1/B* yerine 2'ye çıkar.

13.3.4 Üst sınır: başa baş stratejisinin oranı

İzlemedeki oran rastlantı değildir. Başa baş stratejisinin oranı hiçbir *d* için *2 − 1/B*'yi aşmaz.

Teorem 13.1 (Başa baş stratejisi)

B − 1 gün kiralayıp *B*. gün satın alan strateji kesin *(2 − 1/B)*-rekabetçidir. Yani her *d ≥ 1* için *ALG(d) ≤ (2 − 1/B) · OPT(d)*'dir ve eşitlik *d ≥ B* iken sağlanır.

İspat. İki durum vardır. *d < B* ise strateji hiç satın almaz; *d* gün kiralar ve *d* öder. OPT de *min(d, B) = d* öder. Oran 1'dir. *d ≥ B* ise strateji *B − 1* gün kira öder ve *B*. gün satın alır: *ALG = (B − 1) + B = 2B − 1*. OPT ilk gün satın alır ve *B* öder. Oran *(2B − 1)/B = 2 − 1/B*'dir. İki durumda da oran *2 − 1/B*'yi aşmaz. □

Kanıt, başa baş fikrinin neden işe yaradığını da açıklar. Satın alma anına kadar algoritma OPT'nin ödeyeceğinden fazla ödemez. Satın aldığı anda ise toplamda en çok “bir kez kira, bir kez satın alma” kadar öder; bunların her biri en fazla OPT kadardır. Bu yüzden toplam OPT'nin iki katını geçemez. Aynı “iki

katı geçmez" fikri birçok çevrim içi problemde karşımıza çıkar.

13.3.5 Alt sınır: daha iyisi neden yok?

Başa baş stratejisi $2 - 1/B$ oranını sağlar. Daha akıllıca bir deterministik strateji bu oranı düşürebilir mi? Düşman modeli bunun olanaksız olduğunu gösterir.

> **Teorem 13.2 (Kayak kiralama alt sınırı)**
>
> Kayak kiralama problemi için hiçbir deterministik çevrim içi algoritmanın rekabet oranı $2 - 1/B$'den küçük olamaz.

İspat. Her deterministik strateji bir satın alma günü k ile tanımlanır. Hiç satın almayan strateji $k = \infty$'dur; onun oranı sınırsızdır. Düşman stratejiyi bilir ve sezonu tam satın alma gününde bitirir: $d = k$. Bu durumda algoritma $k - 1$ gün kira ve bir satın alma öder: $ALG = (k - 1) + B$. OPT ise $\min(k, B)$ öder. İki durum vardır.

$k < B$ ise oran $(k - 1 + B)/k = 1 + (B - 1)/k$'dir. $k \le B - 1$ olduğu için $(B - 1)/k \ge 1$, yani oran en az 2'dir. Bu da $2 - 1/B$'den büyüktür.

$k \ge B$ ise oran $(k - 1 + B)/B$'dir. Pay k ile artar; en küçük değeri $k = B$'de alınır: $(2B - 1)/B = 2 - 1/B$.

Her k için düşmanın seçtiği dizide oran en az $2 - 1/B$'dir. □

Tablo 13.5, $B = 10$ için farklı satın alma günlerine karşı düşmanın elde ettiği oranı verir. Erken satın almak da geç satın almak da kötüdür; en kötü oran tam $k = B$'de en küçük olur.

k	Kira	ALG	OPT	Oran
1	0	10	1	10
2	1	11	2	5,5
5	4	14	5	2,8
9	8	18	9	2
10	9	19	10	1,9
11	10	20	10	2
15	14	24	10	2,4
20	19	29	10	2,9

Tablo 13.5: $B = 10$; düşman sezonu satın alma gününde bitirir ($d = k$).

Düşmanın $d = k$ seçmesi gerçekten en kötü seçimdir. $d < k$ olursa algoritma hiç satın almaz ve yalnız kira öder; oran en çok 1 olur. $d > k$ olursa algoritmanın maliyeti değişmez ama OPT'nin maliyeti artabilir ya da aynı kalır; oran büyümez. Bu gözlem bir benzetimle de doğrulandı. $B = 10$ için $k = 3, 5, 10,$

14 stratejileri *d = 1, ..., 60* için çalıştırıldı. Her birinde en kötü oran *d = k*'de çıktı: sırasıyla 4, 2,8, 1,9 ve 2,3.

Örnek 13.3

Soru. Kayak kiralamada *B = 20*'dir. Başa baş stratejisinin rekabet oranı nedir? 5. gün satın alan stratejinin en kötü oranı nedir?

Çözüm. Başa baş stratejisinin oranı *2 − 1/20 = 1,95*'tir. Düşman sezonu 20. gün bitirir: ALG = *19 + 20 = 39*, OPT = 20, oran *39/20 = 1,95*. 5. gün satın alan stratejiye karşı düşman *d = 5* seçer. Algoritma 4 gün kira ve bir satın alma öder: *4 + 20 = 24*. OPT sezonun 5 gün süreceğini bildiği için hep kiralar ve 5 öder. Oran *24/5 = 4,8*'dir. Erken satın alan strateji başa baş stratejisinden yaklaşık 2,5 kat kötüdür.

13.3.6 Rastgele strateji

Alt sınır kanıtının gücü, düşmanın satın alma gününü bilmesinden gelir. Algoritma satın alma gününü rastgele seçerse düşman bu bilgiyi kaybeder. Habersiz düşman diziyi, yani *d*'yi baştan seçmek zorundadır. Algoritmanın beklenen maliyeti ölçülür.

Örnek 13.4

Soru. *B = 2* olsun. Deterministik başa baş stratejisinin oranı *2 − 1/2 = 1,5*'tir. Algoritma *p* olasılıkla 1. gün, *1 − p* olasılıkla 2. gün satın alırsa beklenen oran ne olur? En iyi *p* nedir?

Çözüm. Düşmanın iki anlamlı seçeneği vardır. *d = 1* seçerse 1. gün satın alan strateji 2, 2. gün satın alan strateji 1 öder; OPT 1'dir. Beklenen oran *2p + (1 − p) = 1 + p* olur. *d ≥ 2* seçerse 1. gün satın alan 2, ötekisi *1 + 2 = 3* öder; OPT 2'dir. Beklenen oran *(2p + 3(1 − p))/2 = (3 − p)/2* olur. Düşman büyük olanı seçer. Algoritma iki değeri eşitleyerek en kötü durumu küçültür: *1 + p = (3 − p)/2*, buradan *p = 1/3*. Beklenen oran *4/3 ≈ 1,33*'tür. Rastgelelik oranı 1,5'ten 1,33'e indirdi. *p = 0* deterministik başa baş stratejisidir ve en kötü oranı 1,5'tir; *p = 1/2* ise 1,5 verir.

Aynı fikir her *B* için genelleşir. Karlin, Manasse, McGeoch ve Owicki (1994) en iyi rastgele stratejiyi buldu. Bu stratejide *i*. gün satın alma olasılığı $((B-1)/B)^{B-i}$ ile orantılıdır. Yani geç günlere daha büyük olasılık verilir. Elde edilen beklenen oran

$$1 / (1 - (1 - 1/B)^{B})$$

olur. *B = 2* için bu formül *1/(1 − 1/4) = 4/3* verir ve örnekteki değerle uyuşur. *B* büyüdükçe $(1 - 1/B)^{B}$ değeri *1/e*'ye yaklaşır. Oran da *e/(e − 1) ≈ 1,58*'e gider. Tablo 13.6 deterministik ve rastgele oranları karşılaştırır.

B	Deterministik $2 - 1/B$	Rastgele (beklenen)
2	1,5	1,333
3	1,667	1,421
4	1,75	1,463
10	1,9	1,535
100	1,99	1,577
$B \to \infty$	2	$e/(e-1) \approx 1{,}582$

Tablo 13.6: Kayak kiralamada en iyi deterministik ve en iyi rastgele stratejinin oranı.

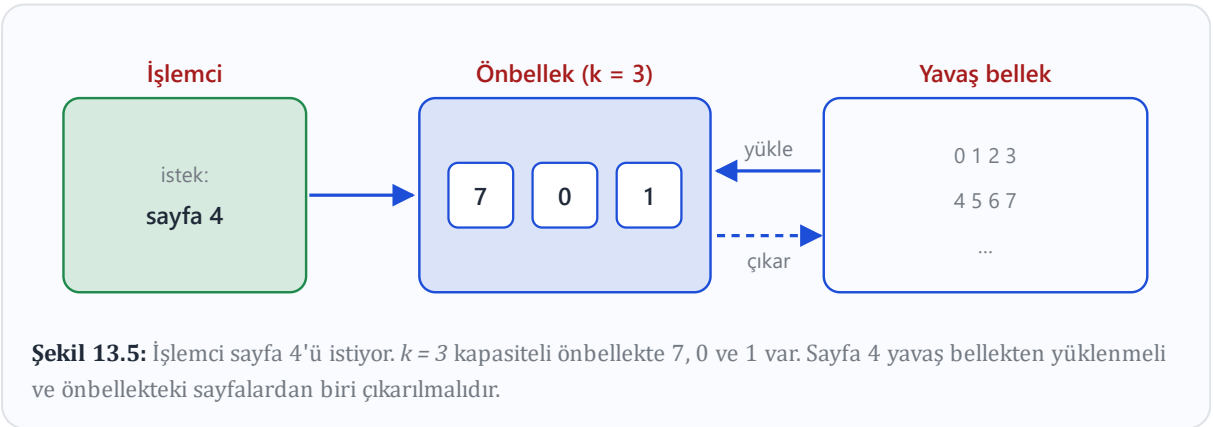
$B = 10$ için olasılıklar 1. gün 0,059'dan 10. gün 0,154'e kadar artar. Bu dağılımla her d için beklenen oran hesaplandı; en kötü değer 1,535 çıktı. Rastgelelik, düşmanın en güçlü silahını, yani algoritmanın ne zaman satın alacağını bilmesini elinden alır. Ama bunun da bir sınırı vardır. Karlin ve arkadaşları, bu oranların rastgele stratejiler için de en iyi değerler olduğunu gösterdi. Büyük B için hiçbir rastgele strateji $e/(e-1)$'den belirgin biçimde iyi olamaz.

13.4 Sayfalama

Sayfalama, çevrim içi algoritmaların en çok çalışılmış problemidir. Bilgisayarların bellek düzeninde her yerde karşımıza çıkar: işlemci önbelleğinde, işletim sisteminin sanal belleğinde, veritabanı tamponunda ve web tarayıcısının önbelleğinde. Hepsinde küçük ve hızlı bir bellek, büyük ve yavaş bir belleğin bir kısmını tutar. Hangi kısmın tutulacağına, gelecek istekler bilinmeden karar verilir.

13.4.1 Problem

İki düzeyli bir bellek düşünelim. Hızlı bellek, yani **önbellek** (*cache*), aynı anda k sayfa tutabilir. Yavaş bellek sınırsızdır ve bütün sayfaları içerir. İşlemci sırayla sayfa ister. İstenen sayfa önbellekteyse bu bir **isabettir** (*hit*) ve maliyeti 0'dır. Sayfa önbellekte değilse bir **sayfa hatası** (*page fault*) oluşur. Sayfa yavaş bellekten yüklenir ve maliyet 1 olur. Önbellek doluysa yeni sayfaya yer açmak için bir sayfa çıkarılmalıdır. Şekil 13.5 bu anı gösterir.



Şekil 13.5: İşlemci sayfa 4'ü istiyor. $k = 3$ kapasiteli önbellekte 7, 0 ve 1 var. Sayfa 4 yavaş bellekten yüklenmeli ve önbellekteki sayfalardan biri çıkarılmalıdır.

Tanım 13.4

Sayfalama problemi (*paging*): k sayfalık bir önbellek ve bir sayfa istekleri dizisi σ verilir. Her istek ya isabettir (maliyet 0) ya da sayfa hatasıdır (maliyet 1). Hata durumunda önbellek doluysa algoritma çıkarılacak sayfayı seçer. Amaç toplam sayfa hatası sayısını en aza indirmektir. Çıkarılacak sayfayı seçen kurala **çıkarma politikası** (*eviction policy*) denir.

Bu modelde yalnız hatalar sayılır. Gerçek bir sistemde isabetin de küçük bir maliyeti vardır, ama yavaş bellekten yükleme bundan yüzlerce, hatta milyonlarca kat pahalıdır. Diskten bir sayfa okumak milisaniyeler, bellekten okumak nanosaniyeler sürer. Bu yüzden hata sayısı, başarımı belirleyen asıl etkendir. Ayrıca modelde sayfa yalnız hata olduğunda yüklenir; önceden yükleme yapılmaz. Bu kısıt genelliği bozmaz: önceden yükleme yapan her algoritma, yüklemeyi gerçekten gerektiği ana erteleyen bir algoritmaya çevrilebilir ve hata sayısı artmaz.

Problem tipik bir çevrim içi problemdir. Hangi sayfanın çıkarılması gerektiği gelecekteki isteklere bağlıdır. Bir sonraki istekte hangi sayfanın isteneceği bilinseydi doğru seçim kolay olurdu. Ama algoritma yalnız geçmişi görür.

13.4.2 Sayfa çıkarma politikaları

Çıkarma politikaları geçmişten farklı bilgiler kullanır. Tablo 13.7 başlıcalarını, rekabet oranlarıyla birlikte verir.

Politika	Hangi sayfayı çıkarır?	Çevrim içi mi?	Rekabet oranı
FIFO	Önbelleğe en önce yüklenen sayfayı (kuyruk)	evet	k
LRU	En uzun süredir kullanılmayan sayfayı (*least recently used*)	evet	k
FWF	Hata olunca ve önbellek doluysa hepsini boşaltır (*flush when full*)	evet	k
LFU	En az sıklıkla kullanılan sayfayı (*least frequently used*)	evet	sınırsız
LIFO	En son yüklenen sayfayı	evet	sınırsız
OPT (MIN)	Bir sonraki kullanımı en uzak gelecekte olan sayfayı (Belady, 1966)	hayır	1

Tablo 13.7: Sayfa çıkarma politikaları (k: önbellek boyutu).

FIFO (*first in, first out*) önbelleği bir kuyruk gibi yönetir. Kuyruğun başındaki, yani en önce yüklenen sayfa çıkar. Bir sayfanın ne kadar sık kullanıldığına bakmaz. **LRU** her sayfanın son kullanım zamanını izler ve en eskisini çıkarır. Yakın geçmişte kullanılan sayfanın yakında yine kullanılacağını varsayar. **FWF** kaba bir politikadır. Önbellek doluyken hata olursa bütün sayfaları atar ve baştan başlar. Kaba görünse de rekabet oranı FIFO ve LRU ile aynıdır: k.

LFU ve **LIFO** ise rekabetçi değildir. LIFO'nun neden kötü olduğu küçük bir dizide görülür. $k = 3$ için 1 2 3 4 3 4 3 4 … dizisini düşünelim. İlk üç istek önbelleği doldurur. 4 gelince LIFO en son yüklenen 3'ü çıkarır. Ardından 3 istenir; bu kez en son yüklenen 4 çıkar. Dizi böyle sürdükçe her istek bir hatadır. OPT ise 4 geldiğinde bir daha hiç istenmeyecek olan 1'i çıkarır ve sonra hiç hata yapmaz. Dizi uzadık-

ça LIFO'nun hata sayısı artar, OPT'ninki 4'te kalır. Oran sınırsız büyür. LFU da benzer biçimde tuzağa düşer. 1 ve 2 sayfaları *m* kez istenip sonra 3 ve 4 dönüşümlü istenirse LFU sayaçları büyük olan 1 ve 2'yi korur. 3 ile 4 ise sayaçları *m*'ye yaklaşana kadar birbirini çıkarır. *m* büyüdükçe oran da büyür.

OPT, Belady'nin 1966'da tanımladığı politikadır; MIN ya da "en uzak ileri uzaklık" (*longest forward distance*) adlarıyla da anılır. Hata olduğunda önbellekteki her sayfanın bir sonraki kullanımına bakar ve en uzak gelecekte kullanılacak olanı çıkarır. Hiç kullanılmayacak sayfa varsa önce o çıkarılır. OPT geleceği bilmeyi gerektirir; bu yüzden gerçek bir sistemde uygulanamaz. Ama karşılaştırma ölçütü olarak vazgeçilmezdir. Her dizi için en az sayfa hatasını verdiği kanıtlanmıştır. Rekabet oranındaki OPT tam olarak budur.

13.4.3 FIFO adım adım

Üç politikayı aynı dizide karşılaştırmak için istek dizisi 7 0 1 2 0 3 0 4 2 3 0 3 2 ve $k = 3$ kullanılır. Bu dizide 0 sayfası sık, öteki sayfalar seyrek istenir. Şekil 13.6 FIFO'nun çerçeve içeriklerini her istekten sonra gösterir; Ç1, Ç2 ve Ç3 önbelleğin üç çerçevesidir. Yeni yüklenen sayfa çıkan sayfanın çerçevesine yazılır. Bu yüzden çerçevedeki konumlar yükleme sırasını göstermez. Yükleme sırası Tablo 13.8'deki kuyruk sütunundadır.



Şekil 13.6: FIFO ile sayfalama, istek dizisi 7 0 1 2 0 3 0 4 2 3 0 3 2, $k = 3$. H: sayfa hatası, ✓: isabet. Toplam 10 hata.

#	İstek	Sonuç	Çıkan	Kuyruk	Hata
1	7	H	—	[7]	1
2	0	H	—	[7, 0]	2
3	1	H	—	[7, 0, 1]	3
4	2	H	7	[0, 1, 2]	4
5	0	✓	—	[0, 1, 2]	4
6	3	H	0	[1, 2, 3]	5
7	0	H	1	[2, 3, 0]	6
8	4	H	2	[3, 0, 4]	7
9	2	H	3	[0, 4, 2]	8

#	İstek	Sonuç	Çıkan	Kuyruk	Hata
10	3	H	0	[4, 2, 3]	9
11	0	H	4	[2, 3, 0]	10
12	3	✓	—	[2, 3, 0]	10
13	2	✓	—	[2, 3, 0]	10

Tablo 13.8: FIFO izlemesi. Kuyruk en eskiden en yeniye yazılmıştır; isabet kuyruğu değiştirmez.

İzlemenin kritik anı 6. istektir. 0 sayfası az önce, 5. istekte kullanılmıştı. FIFO bunu dikkate almaz. 0, kuyruğun başında olduğu için çıkarılır. Hemen ardından 7. istekte 0 yeniden istenir ve gereksiz bir hata oluşur. 11. istekte de aynı durum yaşanır: 0, 10. istekte çıkarılmış ve bir istek sonra geri istenmiştir. FIFO toplam 10 hata yapar.

13.4.4 LRU adım adım

LRU her isabette sayfanın son kullanım zamanını günceller. Bu küçük fark, sık kullanılan sayfaların önbellekte kalmasını sağlar. Şekil 13.7 aynı dizinin LRU ile işlenişini, Tablo 13.9 ise her adımdaki kullanım sırasını gösterir.



Şekil 13.7: LRU ile sayfalama, aynı dizi ve $k = 3$. Toplam 9 hata.

#	İstek	Sonuç	Çıkan	Kullanım sırası	Hata
1	7	H	—	[7]	1
2	0	H	—	[7, 0]	2
3	1	H	—	[7, 0, 1]	3
4	2	H	7	[0, 1, 2]	4
5	0	✓	—	[1, 2, 0]	4
6	3	H	1	[2, 0, 3]	5
7	0	✓	—	[2, 3, 0]	5
8	4	H	2	[3, 0, 4]	6
9	2	H	3	[0, 4, 2]	7

#	İstek	Sonuç	Çıkan	Kullanım sırası	Hata
10	3	H	0	[4, 2, 3]	8
11	0	H	4	[2, 3, 0]	9
12	3	✓	—	[2, 0, 3]	9
13	2	✓	—	[0, 3, 2]	9

Tablo 13.9: LRU izlemesi. Kullanım sırası en eskiden en yeniye yazılmıştır; isabet sayfayı en sona taşır.

Fark 5. istekte başlar. 0 isabet aldığında LRU onu sıranın sonuna taşır. 6. istekte çıkarılan sayfa artık 0 değil, en uzun süredir kullanılmayan 1'dir. 7. istekte 0 yine isabet alır. FIFO'nun bu iki istekte yaptığı gereksiz hata LRU'da olmaz. LRU'nun da çaresiz kaldığı anlar vardır. 10. istekte 0 en eski kullanılan sayfadır ve çıkarılır; 11. istekte geri istenir. LRU toplam 9 hata yapar.

13.4.5 OPT adım adım

OPT her hatada geleceğe bakar. Önbellekteki her sayfa için bir sonraki isteğin sırası bulunur; en uzak olan çıkarılır. Şekil 13.8 OPT'nin çerçevelerini, Tablo 13.10 ise her çıkarmada hesaplanan sonraki kullanım sıralarını verir.



Şekil 13.8: OPT ile sayfalama, aynı dizi ve $k = 3$. Toplam 7 hata.

#	İstek	Önbellek	Sonraki kullanım	Çıkan	Hata
4	2	{7, 0, 1}	7: yok, 0: 5., 1: yok	7	4
6	3	{2, 0, 1}	2: 9., 0: 7., 1: yok	1	5
8	4	{2, 0, 3}	2: 9., 0: 11., 3: 10.	0	6
11	0	{2, 4, 3}	2: 13., 4: yok, 3: 12.	4	7

Tablo 13.10: OPT'nin çıkarma kararları. Sonraki kullanım sütunu her sayfanın bir sonraki istendiği sırayı verir; “yok”, sayfanın bir daha istenmediğini gösterir.

İlk üç istek zorunlu hatalardır. 4. istekte önbellekte 7, 0 ve 1 vardır. 7 ve 1 bir daha hiç istenmez; ikisi de çıkarılabilir. Şekil 7'yi çıkarır. 1'i çıkarmak da aynı sayıda hata verir: 1, 6. istekte zaten çıkarılacaktır. 8. istekteki karar en öğretici olanıdır. 0 sayfası dizinin en sık istenen sayfasıdır, ama bir sonraki kullanımı en uzaktadır: 11. istek. OPT onu çıkarır. 9. ve 10. isteklerdeki 2 ve 3 isabet alır. LRU aynı anda 2'yi çıkarmıştı ve iki hata yapmıştı. OPT toplam 7 hata yapar.

> **Sık yapılan hata**
>
> OPT'yi "en az kullanılacak sayfayı çıkar" diye anlamak. OPT sayfanın gelecekte kaç kez kullanılacağına değil, bir sonraki kullanımının ne zaman olduğuna bakar. Yukarıdaki dizide 0 sayfası gelecekte bir kez daha istenir, 4 hiç istenmez. Buna karşın 8. istekte 0 çıkarılır, çünkü önbellekteki adaylar 2, 0 ve 3'tür; 4 o anda önbellekte değildir. Adaylar arasında bir sonraki kullanımı en uzak olan 0'dır.

13.4.6 Üç politikanın karşılaştırması

Tablo 13.11 üç politikanın sonuçlarını bir arada verir.

Politika	Sayfa hatası	İsabet	ALG / OPT	Gerekli bilgi
FIFO	10	3	1,43	Yükleme sırası (kuyruk)
LRU	9	4	1,29	Son kullanım zamanı
OPT	7	6	1	Gelecekteki bütün istekler

Tablo 13.11: 7 0 1 2 0 3 0 4 2 3 0 3 2 dizisinde, $k = 3$ için üç politika.

İlk $k = 3$ hata hiçbir politikada önlenemez. Önbellek başlangıçta boştur ve ilk üç farklı sayfa ne yapılırsa yapılsın yüklenmelidir. Bu hatalara **zorunlu hata** ya da **soğuk hata** (*compulsory miss, cold miss*) denir. OPT bile sıfır hata yapamaz. Rekabet oranı tanımındaki α sabiti tam bu tür başlangıç etkilerini emmek için vardır.

LRU'nun FIFO'dan iyi olmasının nedeni **yerelliktir** (*locality of reference*). Programlar belleğe rastgele erişmez. Bir döngü aynı değişkenlere tekrar tekrar erişir; bir dizi ardışık adreslerden okunur. Yakın geçmişte kullanılan sayfa yakında yine kullanılır. LRU bu düzenliliği kullanır, FIFO kullanmaz. Örnek dizide 0 sayfası sık istendiği için FIFO onu iki kez gereksiz yere çıkardı.

Gerçek sistemlerde LRU'nun kendisi de pahalıdır. Her bellek erişiminde zaman damgasını güncellemek donanım için ağır bir yüktür. Bu yüzden işletim sistemleri LRU'ya yakın, ucuz yaklaşımlar kullanır. En bilineni **saat algoritmasıdır** (*clock*), diğer adıyla ikinci şans algoritması. Her sayfanın bir "kullanıldı" biti vardır; donanım sayfaya erişildiğinde biti 1 yapar. Sayfalar dairesel bir listede durur ve bir ibre bu listede döner. Çıkarma gerektiğinde ibre biti 1 olan sayfanın bitini 0 yapıp geçer, biti 0 olan ilk sayfayı çıkarır. Yakın zamanda kullanılmış sayfalar böylece bir tur daha şans kazanır.

13.4.7 LRU'nun Java kodu ve izlemesi

LRU'nun en doğrudan uygulaması bir bağlı listeyle yapılır. Liste başı en eski, sonu en yeni kullanılan sayfadır. Metot istek dizisini ve önbellek boyutunu alır, toplam sayfa hatası sayısını döndürür.

```
int lruHata(int[] istekler, int k) {
  var onb = new LinkedList<Integer>();
  int hata = 0;
  for (int sayfa : istekler) {
    if (onb.remove((Integer) sayfa)) {
      onb.addLast(sayfa); // en yeni yap
    } else {
      hata++;
      if (onb.size() == k)
        onb.removeFirst(); // en eskiyi at
      onb.addLast(sayfa);
    }
  }
  return hata;
}
```

Kodun çekirdeği `onb.remove((Integer) sayfa)` çağrısıdır. Bu çağrı iki iş birden yapar. Sayfa listedeyse onu listeden çıkarır ve `true` döndürür; değilse `false` döndürür. `true` durumunda sayfa isabet almıştır. `addLast` onu listenin sonuna, yani "en yeni" konuma geri ekler. `false` durumunda sayfa hatası vardır. Hata sayacı artar. Önbellek doluysa `removeFirst` listenin başındaki, en eski kullanılan sayfayı atar. Son olarak yeni sayfa sona eklenir.

Tablo 13.12, kodu 1 2 3 1 4 2 dizisi ve $k = 3$ ile izler.

sayfa	remove	İşlem	onb	hata
1	false	yer var; sona eklenir	[1]	1
2	false	yer var; sona eklenir	[1, 2]	2
3	false	yer var; sona eklenir	[1, 2, 3]	3
1	true	isabet; sona eklenir	[2, 3, 1]	3
4	false	dolu; baştaki 2 atılır	[3, 1, 4]	4
2	false	dolu; baştaki 3 atılır	[1, 4, 2]	5

Tablo 13.12: `lruHata(new int[]{1, 2, 3, 1, 4, 2}, 3)` çağrısının izlemesi. `onb` en eskiden en yeniye yazılmıştır.



Şekil 13.9: İzlemenin sonu: liste [1, 4, 2]. Altı istekte 5 sayfa hatası ve 1 isabet.

Metot 5 döndürür (Şekil 13.9). 4. istekteki isabet önemlidir: 1, listenin başından sonuna taşınır. Bu yüzden 5. istekte 1 değil 2 atılır. FIFO aynı noktada 1'i atardı.

Kodun maliyeti `remove` çağrısından gelir. `LinkedList.remove(Object)` listeyi baştan tarar ve en çok k öğeye bakar. Bu yüzden her istek $O(k)$ sürer; n istekli bir dizi $O(n \cdot k)$ zaman alır. Bellek $O(k)$'dir. Küçük k için bu yeterlidir. Binlerce sayfalık önbelleklerde ise istek başına $O(1)$ gerekir.

Sık yapılan hata

`(Integer)` dönüşümünü yazmamak. `onb.remove(sayfa)` yazılırsa `sayfa` bir `int` olduğu için Java `remove(int index)` sürümünü seçer. Çağrı, *değeri* `sayfa` olan öğeyi değil, *konumu* `sayfa` olan öğeyi siler. Sayfa 1 istendiğinde listenin ikinci öğesi silinir. Sayfa 7 istendiğinde ise `IndexOutOfBoundsException` fırlar. `(Integer) sayfa` dönüşümü, `remove(Object)` sürümünün çağrılmasını sağlar.

13.4.8 O(1) LRU önbelleği

İstek başına $O(1)$ için iki yapı birlikte kullanılır. Bir hash tablosu, sayfanın önbellekte olup olmadığını ve listedeki yerini $O(1)$'de bulur. Çift bağlı bir liste, sayfaları kullanım sırasıyla tutar. Düğümün yeri bilindiği için onu listeden çıkarıp sona taşımak $O(1)$'dir. Java'nın `LinkedHashMap` sınıfı tam olarak bu birleşimdir.

```
class LruOnbellek<A, D>
    extends LinkedHashMap<A, D> {
  private final int k;

  LruOnbellek(int k) {
    super(16, 0.75f, true); // erişim sırası
    this.k = k;
  }

  @Override
  protected boolean removeEldestEntry(
      Map.Entry<A, D> enEski) {
    return size() > k;      // k aşılınca at
  }
}
```

Yapıcıdaki `super(16, 0.75f, true)` çağrısının üçüncü parametresi `accessOrder`'dır. `true` verildiğinde her `get` ve `put`, erişilen öğeyi listenin sonuna taşır. Böylece listenin başı her zaman en eski kullanılan öğe olur. İlk iki parametre başlangıç kapasitesi ve yük faktörüdür; burada varsayılan değerleri verilmiştir. `removeEldestEntry` metodu her ekleme sonrasında çağrılır. `true` döndürürse sınıf, listenin başındaki öğeyi kendiliğinden siler. Burada boyut k'yi aştığında `true` döner. Bul, taşı ve sil işlemlerinin hepsi $O(1)$'dir; bellek $O(k)$'dir.

Örnek 13.5

Soru. `new LruOnbellek<Integer, String>(3)` nesnesine sırasıyla `put(1)`, `put(2)`, `put(3)`, `get(1)`, `put(4)` ve `put(2)` uygulanıyor (değerler önemsiz). Her adımdan sonra `keySet()` ne yazdırır?

Çözüm. Üç eklemeden sonra sıra [1, 2, 3]'tür. `get(1)` 1'i sona taşır: [2, 3, 1]. `put(4)` sonrasında boyut 4 olur, `removeEldestEntry` `true` döner ve baştaki 2 silinir: [3, 1, 4]. `put(2)` sonrasında baştaki 3 silinir: [1, 4, 2]. Bu dizi, önceki izlemedeki 1 2 3 1 4 2 dizisinin aynısıdır ve son durum Tablo 13.12 ile örtüşür. Kod çalıştırılarak doğrulandı.

Aynı sınıf `accessOrder` parametresi `false` verilerek kurulursa FIFO önbelleği olur. Bu durumda liste ekleme sırasını tutar ve erişim sırayı değiştirmez. Tek bir parametre iki politikayı birbirinden ayırır: isabet sırayı güncelliyor mu, güncellemiyor mu?

13.4.9 FIFO'nun Java kodu ve Belady dizisi

FIFO'nun Java kodu bir kuyruk ve bir kümeyle yazılır. Kuyruk yükleme sırasını tutar. Küme ise "bu sayfa önbellekte mi?" sorusunu $O(1)$'de cevaplar.

```
int fifoHata(int[] istekler, int k) {
  Queue<Integer> kuyruk = new LinkedList<>();
  Set<Integer> kume = new HashSet<>();
  int hata = 0;
  for (int p : istekler) {
    if (kume.contains(p)) continue;   // isabet
    hata++;
    if (kume.size() == k)
      kume.remove(kuyruk.poll()); // en eski
    kuyruk.offer(p); kume.add(p);
  }
  return hata;
}
```

Döngü her sayfa için önce kümeye bakar. Sayfa kümedeyse isabettir ve `continue` sonraki isteğe geçer. FIFO'da isabet hiçbir şeyi değiştirmez. Hata durumunda sayaç artar. Önbellek doluysa `kuyruk.poll()` kuyruğun başındaki en eski sayfayı alır ve `kume.remove` onu kümeden de siler. Son olarak yeni sayfa hem kuyruğun sonuna hem kümeye eklenir. Her istek *O(1)*'dir; bellek *O(k)*'dir.

Kod bu kez 1 2 3 4 1 2 5 1 2 3 4 5 dizisi ve *k = 3* ile izlenir (Tablo 13.13). Bu dizi, Belady anomalisinin klasik örneğidir.

`p`	**Sonuç**	**Çıkan**	`kuyruk`	`hata`
1	H (zorunlu)	—	[1]	1
2	H (zorunlu)	—	[1, 2]	2
3	H (zorunlu)	—	[1, 2, 3]	3
4	H	1	[2, 3, 4]	4
1	H	2	[3, 4, 1]	5
2	H	3	[4, 1, 2]	6
5	H	4	[1, 2, 5]	7
1	✓	—	[1, 2, 5]	7
2	✓	—	[1, 2, 5]	7
3	H	1	[2, 5, 3]	8
4	H	2	[5, 3, 4]	9
5	✓	—	[5, 3, 4]	9

Tablo 13.13: `fifoHata` ile 1 2 3 4 1 2 5 1 2 3 4 5, *k = 3*. Kuyruk en eskiden en yeniye yazılmıştır.

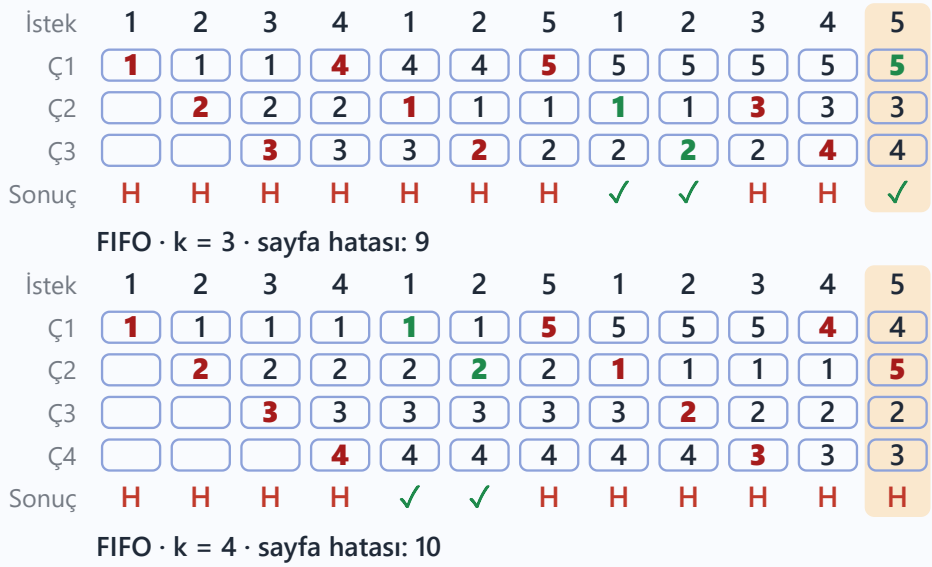


Şekil 13.10: Belady dizisinde FIFO'nun son durumu: kuyruk [5, 3, 4], 9 hata.

Metot 9 döndürür (Şekil 13.10). Aynı kod *k = 4* ile çalıştırıldığında 10 döndürür. Önbellek büyüdü, hata sayısı arttı.

13.4.10 Belady anomalisi

Önbelleği büyütmek hata sayısını azaltmalıdır. Bu beklenti o kadar doğal görünür ki 1960'ların sonuna kadar sorgulanmadı. Belady ve arkadaşları 1969'da FIFO'nun bu beklentiyi bozabildiğini gösterdi. Şekil 13.11 aynı diziyi FIFO ile önce 3, sonra 4 çerçeveyle işler.



Şekil 13.11: Belady anomalisi. Dizi 1 2 3 4 1 2 5 1 2 3 4 5. Üstte FIFO, 3 çerçeve: 9 hata. Altta FIFO, 4 çerçeve: 10 hata.

Tanım 13.5

Belady anomalisi (*Belady's anomaly*): bir sayfa çıkarma politikasında çerçeve sayısı arttığı hâlde aynı istek dizisinde sayfa hatası sayısının artmasıdır.

Anomalinin nedeni 4 çerçeveli çalışmada görülür. 4 çerçeveyle ilk dört istek önbelleği doldurur ve 1 ile 2 isabet alır. 5 geldiğinde FIFO kuyruğun başındaki 1'i çıkarır. Ama 1 hemen ardından istenir. 1 yüklenirken 2 çıkar, 2 yüklenirken 3 çıkar; zincir böyle sürer. Her sayfa, tam istenmeden hemen önce çıkarılmış olur. 3 çerçeveli çalışmada ise yükleme sırası farklıdır. 5 yüklendiğinde önbellekte 1 ve 2 kalır ve ardından gelen iki istek isabet alır.

LRU ve OPT'de bu anomali hiç görülmez. Aynı dizide LRU 3 çerçeveyle 10, 4 çerçeveyle 8 hata yapar. OPT 7 ve 6 hata yapar. İki politikanın ortak özelliği, birer **yığın algoritması** (*stack algorithm*) olmalarıdır. Yığın algoritmasında her an, k çerçeveli önbellekteki sayfalar $k + 1$ çerçeveli önbellekteki sayfaların alt kümesidir. LRU için bunun nedeni açıktır. k çerçeveli LRU, en son kullanılan k farklı sayfayı tutar. $k + 1$ çerçeveli LRU ise en son kullanılan $k + 1$ farklı sayfayı tutar. Birincisi ikincisinin içindedir. Küçük önbellekte isabet alan her istek büyük önbellekte de isabet alır. Bu yüzden çerçeve artınca hata sayısı artamaz.

FIFO bir yığın algoritması değildir. 4 çerçeveli çalışmada 7. istekten sonra önbellek {5, 2, 3, 4}, 3 çerçeveli çalışmada {5, 1, 2}'dir. 1 küçük önbellektedir ama büyük önbellekte yoktur. 8. istek 1'dir; küçük önbellek isabet alır, büyük önbellek hata yapar.

Örnek 13.6

Soru. 1 2 3 4 1 2 5 1 2 3 4 5 dizisinde FIFO, LRU ve OPT'nin hata sayıları $k = 1, \ldots, 5$ için nedir? Anomali hangi k değerinde görülür?

Çözüm. Üç politika da Java ile çalıştırıldı. $k = 1$'de üçü de 12 hata yapar: dizide art arda aynı sayfa yoktur. $k = 2$'de FIFO ve LRU 12, OPT 9 hata yapar. $k = 3$'te FIFO 9, LRU 10, OPT 7 hata yapar. $k = 4$'te FIFO 10, LRU 8, OPT 6 hata yapar. $k = 5$'te beş farklı sayfanın hepsi sığar ve yalnız 5 zorunlu hata kalır. LRU ve OPT'nin hata sayısı k arttıkça hiç artmaz. FIFO ise 3'ten 4'e geçerken 9'dan 10'a çıkar. İlginç bir ayrıntı da vardır: $k = 3$'te FIFO, LRU'dan daha iyidir. Bir dizideki tek bir sonuç, politikalar arasındaki genel sıralamayı belirlemez.

13.4.11 Deterministik alt sınır: k

Örnek dizide LRU, OPT'den yalnız 1,29 kat kötüydü. Bu oran her dizide geçerli midir? Hayır. Düşman, herhangi bir deterministik politikayı k kat kötü duruma düşüren bir dizi kurabilir.

Teorem 13.3 (Sleator ve Tarjan, 1985)

k sayfalık önbellekle çalışan hiçbir deterministik sayfalama algoritmasının rekabet oranı k'den küçük olamaz.

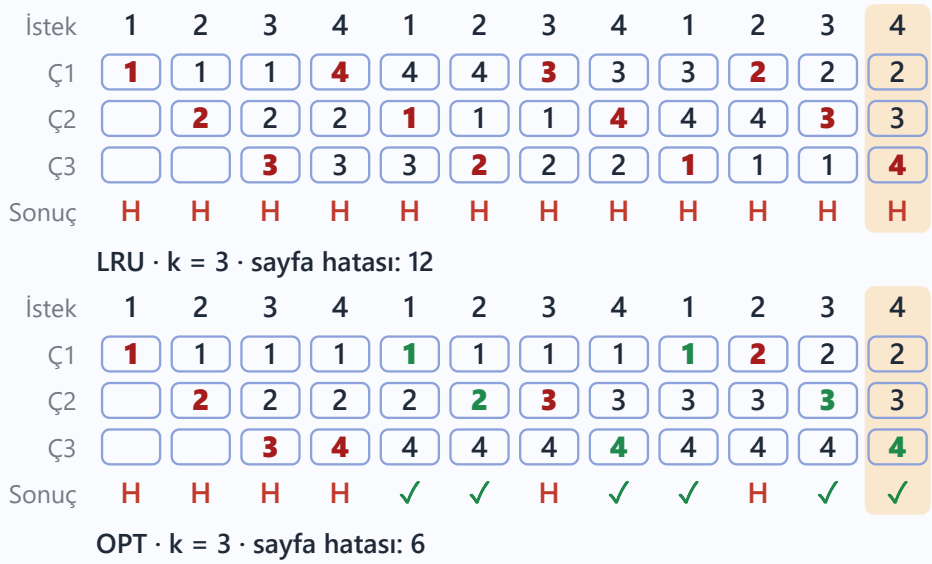
İspat. Düşman yalnız $k + 1$ farklı sayfa kullanır. Önbellek k sayfa tuttuğu için bu sayfalardan en az biri her an önbellek dışındadır. Düşman algoritmanın kodunu bildiği için hangi sayfanın dışarıda olduğunu her adımda hesaplayabilir. Her adımda tam o sayfayı ister. Böylece algoritma *her* istekte hata

yapar: n istekte n hata.

OPT aynı diziyi işlesin. OPT bir hata yaptığında bir sonraki kullanımı en uzak olan sayfayı çıkarır. Önbellekte kalan öteki $k - 1$ sayfanın her biri, çıkarılan sayfadan önce istenecektir. Önbellek dışında yalnız çıkarılan sayfa vardır. Dolayısıyla çıkarılan sayfa istenene kadar gelen bütün istekler isabettir. Bu sürede öteki $k - 1$ sayfanın hepsi istenir. Yani OPT'nin iki hatası arasında en az $k - 1$ isabet vardır. OPT en çok her k istekte bir hata yapar; toplam hata sayısı en çok yaklaşık n / k'dir.

Oran $ALG / OPT \geq n / (n/k) = k$ olur. n büyüdükçe toplanan sabit önemsizleşir; daha küçük bir c ile $ALG \leq c \cdot OPT + \alpha$ sağlanamaz. □

Şekil 13.12 bu kanıtın $k = 3$ ve LRU için somut bir örneğini verir. 4 sayfa ile kurulan 1 2 3 4 1 2 3 4 1 2 3 4 dizisi, LRU'nun her seferinde tam az önce çıkardığı sayfayı ister.



Şekil 13.12: Düşman dizisi 1 2 3 4 1 2 3 4 1 2 3 4, $k = 3$. Üstte LRU: her istek hata, toplam 12. Altta OPT: toplam 6 hata.

LRU 12 istekte 12 hata yapar. OPT ise 6 hata yapar. İlk üç zorunlu hata dışarıda tutulursa oran 9'a 3'tür, yani tam $k = 3$. OPT'nin davranışı kanıttaki gibidir. 4. istekte önbellek {1, 2, 3}'tür ve en uzak gelecekte istenecek olan 3 çıkarılır. Ardından 1 ve 2 isabet alır. 7. istekte 3 gelince, önbellekteki 1, 2 ve 4 arasından en uzak gelecekte istenecek olan 2 çıkarılır. Her hatadan sonra iki isabet gelir.

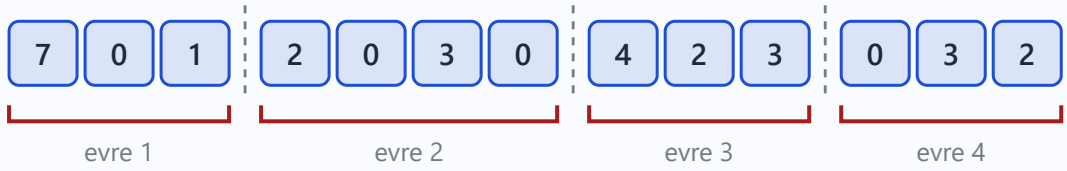
Aynı dizide FIFO da 12 hata yapar. Bu dizide FIFO ile LRU aynı davranır, çünkü hiç isabet yoktur. Dizi uzadıkça oran k'ye yaklaşır. Örneğin 5 sayfa ve $k = 4$ ile 1 2 3 4 5 dizisi üç kez tekrarlanırsa LRU 15, OPT 7 hata yapar.

13.4.12 Üst sınır: LRU k-rekabetçidir

Alt sınır hiçbir deterministik algoritmanın k'den iyi olamayacağını söyler. LRU ve FIFO bu sınıra ulaşır. Kanıt, istek dizisini evrelere bölmeye dayanır.

Tanım 13.6

İstek dizisi soldan sağa **evrelere** (*phases*) bölünür. Her evre, en çok k farklı sayfa içeren en uzun parçadır. Bir evrede k farklı sayfa istendikten sonra gelen ilk yeni sayfa, yani $(k + 1)$. farklı sayfa, yeni evreyi başlatır.



Şekil 13.13: 7 0 1 2 0 3 0 4 2 3 0 3 2 dizisinin $k = 3$ için evreleri: 7 0 1 | 2 0 3 0 | 4 2 3 | 0 3 2.

Teorem 13.4 (LRU ve FIFO k-rekabetçidir)

Her istek dizisi σ için $LRU(\sigma) \le k \cdot OPT(\sigma) + k$'dir. Aynı eşitsizlik FIFO için de geçerlidir.

İspat. Dizi m evreye bölünsün. Kanıt iki adımdan oluşur.

LRU bir evrede en çok k hata yapar. Evrede istenen bir sayfa p olsun. p istendiği anda en yeni kullanılan sayfadır. Evre bitene kadar en çok k farklı sayfa istenir. Bu yüzden p'den daha yakın zamanda kullanılmış en çok $k - 1$ sayfa olabilir. Önbellek dolu ve p içerideyken ondan daha eski kullanılmış en az bir sayfa hep vardır. LRU bu yüzden p'yi evre bitene kadar çıkarmaz. Evrede her farklı sayfa en çok bir kez hata verir. Evrede en çok k farklı sayfa olduğu için en çok k hata vardır.

OPT ardışık iki evre başına en az 1 hata yapar. i. evrenin ikinci isteğinden $(i + 1)$. evrenin ilk isteğine kadar olan parçaya bakalım. i. evrenin ilk isteği q olsun. Parçanın başında q önbellektedir, çünkü az önce istenmiştir. Parçada q'dan farklı en az k sayfa istenir: i. evrenin geri kalan $k - 1$ sayfası ve yeni evreyi başlatan sayfa. Önbellekte q dışında yalnız $k - 1$ yer vardır. Bu k sayfadan en az biri parça içinde hata verir. Parçalar birbirinden ayrık olduğu için m evre boyunca OPT en az $m - 1$ hata yapar.

İki adım birleştirilirse $LRU \le k \cdot m$ ve $OPT \ge m - 1$ olur. Buradan $LRU \le k \cdot (OPT + 1) = k \cdot OPT + k$ çıkar. FIFO için ilk adım şöyle gösterilir. Evrede hata veren bir sayfa kuyruğun sonuna girer. Kuyruğun başına gelip çıkarılması için arkasından k yeni sayfanın daha yüklenmesi gerekir. Evrede en çok k farklı sayfa olduğu için bu evre bitmeden olmaz. İkinci adım OPT ile ilgilidir ve aynen geçerlidir. □

Kanıt örnek dizide de denetlenebilir. Şekil 13.13'teki dizide 4 evre vardır. LRU'nun 9 hatası $k \cdot m = 3 \cdot 4 = 12$'yi aşmaz. OPT'nin hata sayısı en az $m - 1 = 3$ olmalıdır; 7 hata bu sınırı sağlar. Görüldüğü gibi teori yalnız en kötü durumu sınırlar. Gerçek dizilerde LRU, yerellik sayesinde OPT'ye bu sınırın söylediğinden çok daha yakındır.

13.4.13 Rastgele sayfalama: işaretleme

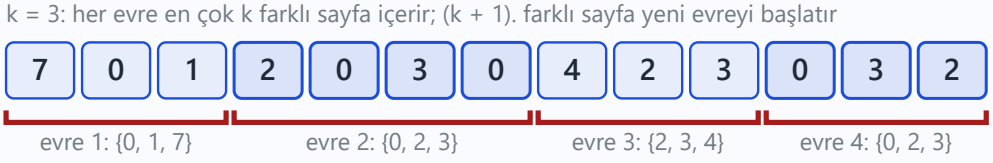
Deterministik alt sınır kanıtının anahtarı, düşmanın algoritmanın önbelleğini bilmesidir. Algoritma çıkaracağı sayfayı rastgele seçerse habersiz düşman hangi sayfanın dışarıda olduğunu bilemez. Bu fikir **işaretleme algoritmasına** (*marking algorithm*) götürür.

Tanım 13.7

İşaretleme algoritması önbellekteki her sayfa için bir işaret biti tutar ve şöyle çalışır:

1. Evre başında bütün sayfaların işareti kaldırılır.
2. İstenen sayfa işaretlenir; istek isabet de olsa hata da olsa.
3. Hata olursa ve önbellek doluysa işaretsiz sayfalardan biri rastgele seçilip çıkarılır.
4. Hata anında önbellekteki bütün sayfalar işaretliyse yeni evre başlar ve 1. adıma dönülür.

İşaretli bir sayfa, bu evrede istenmiş bir sayfadır. Algoritma bu evrede istenmiş sayfaları korur ve yalnız işaretsizler arasından seçer. Evre kavramı, LRU kanıtındaki evrelerle aynıdır. Şekil 13.14 örnek dizinin evrelerini ve her evrede istenen sayfa kümelerini gösterir.



Şekil 13.14: Örnek dizinin evreleri ve her evrede istenen sayfalar. Her evre en çok $k = 3$ farklı sayfa içerir.

Fiat, Karp, Luby, McGeoch, Sleator ve Young (1991), rastgele işaretleme algoritmasının habersiz düşmana karşı $2H_k$-rekabetçi olduğunu gösterdi. Burada H_k, k. harmonik sayıdır:

$$H_k = 1 + 1/2 + 1/3 + \ldots + 1/k \approx \ln k$$

Aynı çalışma, hiçbir rastgele algoritmanın H_k'den iyi olamayacağını da gösterir. Tablo 13.14 deterministik ve rastgele oranları karşılaştırır. Fark k büyüdükçe açılır: deterministik oran doğrusal, rastgele oran logaritmik büyür.

k	Deterministik (LRU, FIFO)	H_k (rastgele alt sınır)	$2H_k$ (işaretleme)
2	2	1,5	3
3	3	1,83	3,67
8	8	2,72	5,44
16	16	3,38	6,76
100	100	5,19	10,37

Tablo 13.14: Sayfalamada deterministik ve rastgele rekabet oranları.

Küçük k için $2H_k$ deterministik orandan büyük olabilir. $k = 2$'de $2H_2 = 3$, deterministik oran ise 2'dir. Ama $k = 8$'de işaretleme 5,44 ile deterministik orandan iyidir. Gerçek önbellekler binlerce sayfa tuttuğu için fark büyüktür. LRU da bir işaretleme algoritmasıdır. LRU'nun çıkardığı en eski kullanılan sayfa, her zaman işaretsiz bir sayfadır. LRU işaretsizler arasından rastgele değil, en eskisini seçer. Rastgelelik, düşmanın "algoritmanın önbelleğinde olmayan sayfayı iste" stratejisini bozar; düşman hangi sayfanın çıktığını bilemez.

13.4.14 İşaretleme algoritmasının kod izlemesi

İşaretleme algoritmasının Java kodu iki küme kullanır: `onb` önbellekteki sayfaları, `isr` işaretli sayfaları tutar. `zar` bir `Random` nesnesidir.

```
int isaretleme(int[] istekler, int k) {
  Set<Integer> onb = new HashSet<>();
  Set<Integer> isr = new HashSet<>();   // işaretli
  int hata = 0;
  for (int p : istekler) {
    if (!onb.contains(p)) {
      hata++;
      if (onb.size() == k) {
        if (isr.size() == k) isr.clear(); // yeni evre
        List<Integer> aday = new ArrayList<>(onb);
        aday.removeAll(isr);          // işaretsizler
        int i = zar.nextInt(aday.size());
        onb.remove(aday.get(i));
      }
      onb.add(p);
    }
    isr.add(p);
  }
  return hata;
}
```

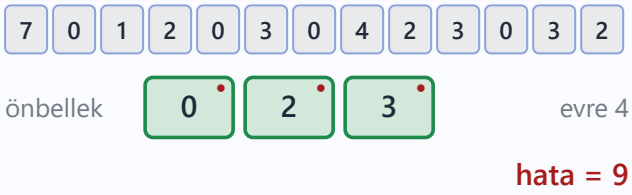
Sayfa önbellekte yoksa hata sayacı artar. Önbellek doluysa önce evre denetlenir: işaretli sayfa sayısı k'ye eşitse önbellekteki her sayfa işaretlidir. Yeni evre başlar ve `isr.clear()` bütün işaretleri siler. Ardından `aday` listesi kurulur: önbellekteki sayfalardan işaretli olanlar çıkarılır.

`zar.nextInt(aday.size())` bu listeden rastgele bir indis seçer ve o sayfa önbellekten çıkarılır. Yeni sayfa önbelleğe eklenir. Son satırdaki `isr.add(p)` isabet ya da hata fark etmeksizin istenen sayfayı işaretler.

Algoritma rastgele olduğu için bir izleme ancak belli çekilişlerle verilebilir. Tablo 13.15, örnek dizide bir çalışmayı adım adım verir. Kod, çekilişleri bu tablodaki gibi döndüren sabit bir rastgele sayı üreteciyle çalıştırıldı ve 9 döndürdü.

`p`	Sonuç	Yeni evre?	Adaylar	Çıkan	Önbellek	`hata`
7	H	—	—	—	7•	1
0	H	—	—	—	0• 7•	2
1	H	—	—	—	0• 1• 7•	3
2	H	evet	[0, 1, 7]	7	0 1 2•	4
0	✓	—	—	—	0• 1 2•	4
3	H	hayır	[1]	1	0• 2• 3•	5
0	✓	—	—	—	0• 2• 3•	5
4	H	evet	[0, 2, 3]	3	0 2 4•	6
2	✓	—	—	—	0 2• 4•	6
3	H	hayır	[0]	0	2• 3• 4•	7
0	H	evet	[2, 3, 4]	2	0• 3 4	8
3	✓	—	—	—	0• 3• 4	8
2	H	hayır	[4]	4	0• 2• 3•	9

Tablo 13.15: `isaretleme` ile 7 0 1 2 0 3 0 4 2 3 0 3 2, *k* = *3*. "Adaylar" işaretsiz sayfalardır; "•" işaretli sayfayı gösterir.



Şekil 13.15: İzlemenin sonu: önbellek {0, 2, 3}, üçü de işaretli; 4. evre, 9 hata.

Tablonun bazı satırlarında seçim zorunludur. Aday listesi tek öğeliyse rastgelelik bir şey değiştirmez: 6., 10. ve 13. satırlardaki gibi. Gerçek seçimler yalnız yeni evrenin ilk hatasında yapılır. Bu çekilişlerle algoritma 9 hata yaptı (Şekil 13.15). Aynı dizide LRU 9, FIFO 10, OPT 7 hata yapmıştı. Başka çekilişlerle sonuç değişir. Kod 200.000 farklı tohumla çalıştırıldı. Hata sayısı 7 ile 12 arasında değişti; ortalaması yaklaşık 9,0 çıktı.

Sık yapılan hata

İşaretleri yalnız hata durumunda koymak. `isr.add(p)` satırı `if` bloğunun içine taşınırsa isabet alan sayfalar işaretlenmez. Bu evrede yeni kullanılmış bir sayfa işaretsiz kalır ve rastgele seçimle çıkarılabilir. Algoritma, yakın zamanda kullanılmış sayfaları koruma özelliğini kaybeder. Yukarıdaki izlemede 5. satırdaki isabetten sonra 0 işaretsiz kalır. 6. satırda aday listesi [0, 1] olur ve 0 çıkarılabilir; 0 hemen ardından yeniden istenir.

13.5 Liste güncelleme

Sayfalamada maliyet ikiliydi: isabet ya da hata. Liste güncelleme probleminde ise maliyet, öğenin listedeki konumuna bağlıdır. Problem eski ama hâlâ kullanılan bir fikre dayanır: sık istenen öğeler listenin başına yakın tutulursa aramalar ucuzlar. Hangi öğelerin sık isteneceği önceden bilinmediği için problem çevrim içidir.

13.5.1 Problem ve üç kural

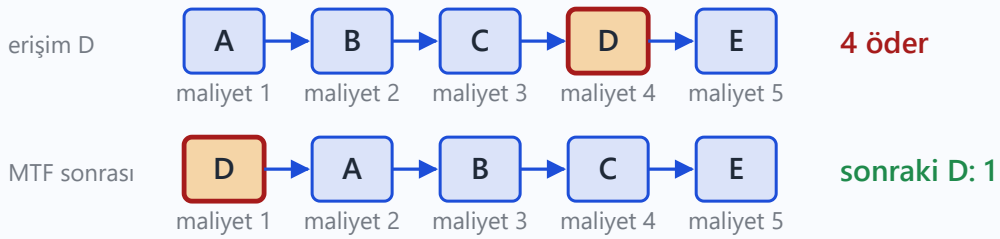
n öğeli bir bağlı liste verilsin. Bir öğeye erişmek için liste baştan taranır. i. konumdaki öğeye erişmenin maliyeti i'dir. Erişilen öğe, erişimden sonra ücretsiz olarak listenin önüne doğru taşınabilir. İki komşu öğenin başka bir biçimde yer değiştirmesi ise 1 birim tutar; bu tür takaslara *ücretli takas* denir.

Tanım 13.8

Liste güncelleme problemi (*list update problem*): n öğeli bir liste ve bir erişim istekleri dizisi verilir. i. konumdaki öğeye erişim i birim tutar. Erişilen öğe ücretsiz olarak öne taşınabilir; öteki takaslar birer birim tutar. Algoritma her erişimden sonra listeyi nasıl düzenleyeceğine geleceği bilmeden karar verir. Amaç toplam maliyeti en aza indirmektir.

Üç doğal kural vardır.

- **Öne taşı** (*move-to-front, MTF*): erişilen öğe listenin başına alınır.
- **Transpoz** (*transpose*), diğer adıyla yer değiştir: erişilen öğe bir önündeki öğeyle yer değiştirir.
- **Sıklık sayacı** (*frequency count*): her öğenin erişim sayısı tutulur ve liste bu sayılara göre azalan sırada tutulur.



Şekil 13.16: A B C D E listesinde D'ye erişim 4 birim tutar. Öne taşı kuralıyla D başa alınır ve liste D A B C E olur. Bir sonraki D erişimi yalnız 1 birim tutar.

Bu kurallara uyan listelere **kendini düzenleyen listeler** (*self-organizing lists*) denir. Kullanım alanları geniştir. Zincirleme hash tablosunda her kovanın zinciri kısa bir bağlı listedir; öne taşı kuralı sık aranan anahtarları zincirin başına getirir. Veri sıkıştırmada *öne taşı dönüşümü* kullanılır: her simge, o anki listedeki konumuyla kodlanır ve sonra öne taşınır. bzip2 sıkıştırıcısı bu dönüşümü kullanır. Ardışık tekrarlar küçük sayılara, çoğunlukla 0'a dönüşür ve sonraki aşamada iyi sıkışır. Örneğin “banana” sözcüğü a, b, n alfabesiyle 1, 1, 2, 1, 1, 1 olarak kodlanır.

Öne taşı kuralının Java kodu bağlı listeyle kısa yazılır:

```
int mtfMaliyet(LinkedList<Character> liste, char[] istekler) {
  int toplam = 0;
  for (char x : istekler) {
    int konum = liste.indexOf(x) + 1;  // 1 tabanlı
    toplam += konum;
    liste.remove((Character) x);
    liste.addFirst(x);  // öne taşı
  }
  return toplam;
}
```

`indexOf` öğenin 0 tabanlı konumunu bulur; 1 eklenerek maliyet elde edilir. `remove((Character) x)` öğeyi listeden çıkarır, `addFirst` başa ekler. `(Character)` dönüşümü burada da `remove(int index)` yerine `remove(Object)` sürümünün çağrılmasını sağlar. `indexOf` ve `remove` listeyi taradığı için her istek *O(n)* sürer. Bu süre, problemin kendi maliyet modeliyle uyumludur: erişim zaten konum kadar tutar.

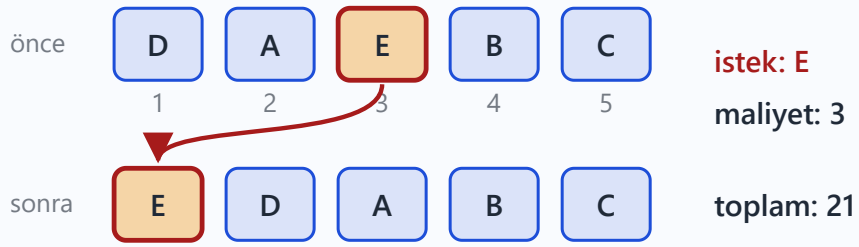
13.5.2 Öne taşı adım adım

Başlangıç listesi A B C D E, istekler E D E A D E olsun. Tablo 13.16 öne taşı kuralının her adımını verir.

İstek	Konum (maliyet)	Toplam	Sonraki liste
E	5	5	E A B C D
D	5	10	D E A B C

İstek	Konum (maliyet)	Toplam	Sonraki liste
E	2	12	E D A B C
A	3	15	A E D B C
D	3	18	D A E B C
E	3	21	E D A B C

Tablo 13.16: Öne taşı ile A B C D E listesinde E D E A D E istekleri.



Şekil 13.17: Son istek: E 3. konumda, maliyet 3. Liste E D A B C olur; toplam 21.

Aynı dizide öteki kurallar daha çok öder (Tablo 13.17). Hiç taşımayan liste her E için 5, her D için 4 öder. Transpoz ise bu dizide çok kötüdür. E ile D listenin sonunda birbirinin yerini alıp durur: E öne geçince D sona düşer, D öne geçince E sona düşer. İkisi de hep 5. konumdan istenir.

Strateji	E	D	E	A	D	E	Toplam
Öne taşı	5	5	2	3	3	3	21
Taşıma yok	5	4	5	1	4	5	24
Transpoz	5	5	5	1	5	5	26

Tablo 13.17: Aynı istek dizisinde üç stratejinin istek başına maliyeti.

Geleceği bilen bir algoritma daha da az öder. Bütün listeler ve ücretli takaslar denenerek bu küçük örnekte en iyi maliyet hesaplandı: 17. Bu maliyete şöyle ulaşılır: başta 7 ücretli takasla liste E D A B C yapılır. Sonra erişimler *1 + 2 + 1 + 3 + 2 + 1 = 10* tutar.

Örnek 13.7

Soru. A B C D listesinde öne taşı kuralıyla D C D A isteklerinin toplam maliyeti nedir?

Çözüm. D 4. konumdadır: maliyet 4, liste D A B C. C 4. konumdadır: maliyet 4, liste C D A B. D 2. konumdadır: maliyet 2, liste D C A B. A 3. konumdadır: maliyet 3, liste A D C B. Toplam *4 + 4 + 2 + 3 = 13*'tür. Hiç taşımayan liste *4 + 3 + 4 + 1 = 12* öderdi. Öne taşı kuralı her dizide taşımamaktan iyi değildir; bu dizide az farkla kötüdür.

13.5.3 Öne taşı kuralı 2-rekabetçidir

Sleator ve Tarjan 1985'te öne taşı kuralının rekabet oranının 2 olduğunu gösterdi. Aynı makale sayfalamadaki k sonucunu da içerir. İkisi birlikte rekabetçi analizin başlangıç noktası sayılır.

Teorem 13.5 (Sleator ve Tarjan, 1985)

Aynı listeyle başlayan her algoritma A ve m istekli her dizi σ için $MTF(\sigma) \le 2 \cdot A(\sigma) - m$'dir. A'nın maliyetine ücretli takasları da katılır. Özel olarak $MTF(\sigma) \le 2 \cdot OPT(\sigma) - m$'dir.

Kanıt **amortize analize** (*amortized analysis*) dayanır. Bir işlemin gerçek maliyeti dalgalı olabilir. Pahalı işlemler, ucuz işlemlerin biriktirdiği "krediyle" ödenir. Kredi bir **potansiyel fonksiyonu** Φ ile ölçülür. Bir işlemin amortize maliyeti, gerçek maliyeti ile potansiyeldeki değişimin toplamıdır: *gerçek* $+ \Delta\Phi$. Potansiyel hiç negatif olmaz ve başta 0'sa, amortize maliyetlerin toplamı gerçek maliyetlerin toplamından küçük olamaz.

İspat. Potansiyel, MTF listesi ile A listesi arasındaki **ters çift** (*inversion*) sayısıdır. Ters çift, iki listede farklı sırada duran öğe çiftidir. Başta iki liste aynıdır ve $\Phi = 0$'dır.

İstenen öğe x olsun. MTF listesinde x'ten önce gelen öğeleri iki gruba ayıralım. Bunlardan A listesinde de x'ten önce gelenlerin sayısı a, A listesinde x'ten sonra gelenlerin sayısı b olsun. MTF'nin gerçek maliyeti $a + b + 1$'dir. A'da x'in konumu i ise $i \ge a + 1$'dir.

MTF, x'i başa taşır. b öğe ile x arasındaki ters çiftler ortadan kalkar. a öğe ile x arasında yeni ters çiftler oluşur. Bu yüzden $\Delta\Phi \le a - b$'dir. Amortize maliyet $(a + b + 1) + (a - b) = 2a + 1 \le 2i - 1$ olur.

A'nın kendi hareketleri de potansiyeli değiştirir. A, x'i ücretsiz olarak öne taşırsa x MTF listesinde zaten baştadır; her adım bir ters çifti yok eder. Ücretli bir takas ise potansiyeli en çok 1 artırır ve A'ya 1 birime mal olur. Bütün istekler toplanırsa $MTF(\sigma) \le \Sigma(2i - 1) + P = 2C - m + P$ olur. Burada C A'nın erişim maliyeti, P ücretli takas sayısıdır. $2C + P \le 2(C + P) = 2 \cdot A(\sigma)$ olduğundan sonuç çıkar. □

Kanıttaki eşitsizlik örnek dizide adım adım izlenebilir. Tablo 13.18, A olarak hiç taşımayan A B C D E listesini alır. Her adımda amortize maliyet $2i - 1$'i aşmaz. Toplamda $21 \le 2 \cdot 24 - 6 = 42$ olur. Gerçek OPT ile de sınır sağlanır: $21 \le 2 \cdot 17 - 6 = 28$.

İstek	MTF maliyeti	Sabit listede i	Φ önce → sonra	Amortize	$2i - 1$
E	5	5	0 → 4	9	9
D	5	4	4 → 6	7	7
E	2	5	6 → 7	3	9
A	3	1	7 → 5	1	1
D	3	4	5 → 5	3	7
E	3	5	5 → 7	5	9

Tablo 13.18: Potansiyel fonksiyonu ile öne taşı; karşılaştırılan liste sabit A B C D E. Φ, iki liste arasındaki ters çift sayısıdır.

4. satır kanıtın gücünü gösterir. A'ya erişim MTF'ye 3 birime mal olur, sabit listeye ise yalnız 1 birime. Gerçek maliyetle bakılırsa MTF bu adımda 3 kat kötüdür. Ama A'nın başa taşınması iki ters çifti yok eder ve potansiyel 2 azalır. Amortize maliyet 1'e iner. Önceki adımlarda biriken kredi, bu pahalı adımı öder.

13.5.4 Transpoz ve sıklık sayacı neden rekabetçi değil?

Transpoz kuralı öğeyi yalnız bir adım taşıdığı için temkinli görünür. Ama düşman bu temkini kullanır. Listenin son iki öğesi sırayla istenir: $x_n, x_{n-1}, x_n, x_{n-1}, \ldots$ Her erişim iki öğenin yerini değiştirir, ama ikisi de hep n. ve $(n-1)$. konumda kalır. İstenen öğe her seferinde n. konumdadır. İstek başına maliyet n'dir. MTF ise ilk iki erişimden sonra iki öğeyi listenin başında tutar ve istek başına 2 öder. $n = 5$ ve 20 istekle bu dizi çalıştırıldı: transpoz 100, MTF 46, taşımayan liste 90 ödedi. n büyüdükçe oran $n/2$'ye gider; sınırsızdır.

Sıklık sayacı da tuzağa düşer. Önce x_1 öğesi n kez, x_2 öğesi $n-1$ kez istenir ve bu böyle sürer. Sonra aynı öğeler ters sırada, daha çok sayıda istenir. Sayaçlar eski sırayı uzun süre korur. Yeni sık öğeler sayaçlar yetişene kadar listenin sonunda kalır. A B C D listesinde A dört, B üç, C iki, D bir kez istenip ardından D beş, C altı, B yedi, A dokuz kez istendi. Sıklık sayacı 81, MTF 49 ödedi. Uzun listelerde maliyet $\Theta(n)$ kat kötüleşebilir.

Sık yapılan hata

"2-rekabetçi" sözünü "her dizide iki kattan az öder, o hâlde hep iyidir" diye okumak. Teorem MTF'yi geleceği bilen algoritmayla karşılaştırır. MTF bazı dizilerde hiç taşımayan listeden de kötüdür. Örneğin A B C D listesinde C A C D B A isteklerine MTF 19, taşımayan liste 14 öder. Garanti yalnız şudur: hiçbir dizide en iyi algoritmanın iki katını geçmez.

13.6 Veri akışları

Önceki problemlerde çevrim içi algoritmanın sorunu yanlış karar vermekti. Veri akışlarında sorun başkadır: girdi o kadar büyüktür ki saklanamaz. Algoritma her öğeye bir kez bakar ve küçük bir özet tutar. Her an bir soru sorulabilir; cevap yalnız özetten verilmelidir.

13.6.1 Veri akışı modeli

Tanım 13.9

Veri akışı (*data stream*) modelinde öğeler $x_1, x_2, \ldots$ tek tek gelir. Akış çok uzun ya da sonsuz olabilir. Her öğeye bir kez bakılır (tek geçiş) ve geçmiş tümüyle saklanamaz. Her an "şu ana kadarki ortalama, ortanca ya da örneklem nedir?" gibi bir sorunun cevabı hazır olmalıdır.

Tablo 13.19, bu kesimde işlenen problemleri, her biri için saklanan özeti ve maliyetleri verir.

Problem	Saklanan özet	Öğe başına	Bellek
Hareketli ortalama (son k)	Kuyruk ve toplam	$O(1)$	$O(k)$
Çevrim içi ortanca	Max-heap ve min-heap	$O(\log n)$	$O(n)$
Rezervuar örnekleme	k öğelik örneklem	$O(1)$	$O(k)$
Akışta örüntü arama	Son m karakter ve kayan hash	$O(1)$	$O(m)$
Çalışan ortalama	n ve μ	$O(1)$	$O(1)$

Tablo 13.19: Akış problemleri ve özetleri.

Çevrim içi ortanca tablonun istisnasıdır: belleği akışla birlikte büyür. Kesin ortanca için bu kaçınılmazdır. Akıştaki her öğe, ileride ortanca olma adayıdır. Çok büyük akışlarda bu yüzden yaklaşık ortanca veren özetler kullanılır.

13.6.2 Hareketli ortalama

Hareketli ortalama (*moving average*), akıştaki son k öğenin ortalamasıdır. Gürültülü algılayıcı verisini yumuşatmak için kullanılır. Borsada 50 günlük ortalama da bir hareketli ortalamadır. Saf yöntem her yeni öğede son k öğeyi baştan toplar: $O(k)$. Daha iyi yöntem **kayan penceredir** (*sliding window*). Toplam bir değişkende tutulur. Yeni öğe girerken toplama eklenir, en eski öğe pencereden çıkarken toplamdan çıkarılır:

$$toplam \leftarrow toplam - x_{t-k} + x_t, \quad ortalama = toplam \,/\, \min(t, k)$$

```
class HareketliOrtalama {
  private final int k;
  private final Queue<Integer> pencere
      = new LinkedList<>();
  private int toplam = 0;

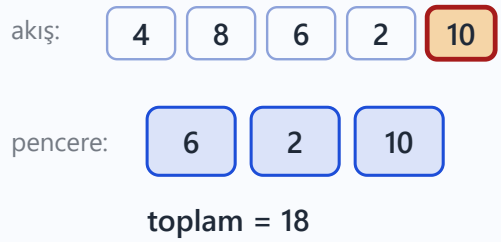
  HareketliOrtalama(int k) { this.k = k; }

  double sonraki(int deger) {
    if (pencere.size() == k)
      toplam -= pencere.poll();
    pencere.offer(deger);
    toplam += deger;
    return toplam / (double) pencere.size();
  }
}
```

Pencere bir kuyruktur. `sonraki` metodu önce pencerenin dolu olup olmadığına bakar. Doluysa `poll` en eski öğeyi kuyruktan alır ve toplamdan çıkarır. Sonra yeni öğe kuyruğa eklenir ve toplama katılır. Ortalama, toplamın pencere boyuna bölümüdür. Pencere henüz dolmamışsa bölen k değil o anki öğe sayısıdır. `(double)` dönüşümü tam sayı bölmesini önler. Her çağrı *O(1)*'dir; bellek *O(k)*'dir. Tablo 13.20, $k = 3$ ile 4, 8, 6, 2, 10 akışını izler.

`deger`	Çıkan	`pencere`	`toplam`	Dönen ortalama
4	—	[4]	4	*4 / 1 = 4*
8	—	[4, 8]	12	*12 / 2 = 6*
6	—	[4, 8, 6]	18	*18 / 3 = 6*
2	4	[8, 6, 2]	*18 − 4 + 2 = 16*	*16 / 3 ≈ 5,33*
10	8	[6, 2, 10]	*16 − 8 + 10 = 18*	*18 / 3 = 6*

Tablo 13.20: `new HareketliOrtalama(3)` ile 4, 8, 6, 2, 10 akışı.



Şekil 13.18: Son adım: pencere [6, 2, 10], toplam 18, ortalama 6.

Sık yapılan hata

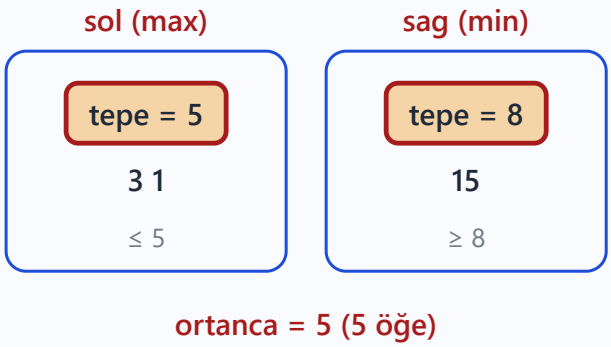
Çıkarma ile eklemenin sırasını karıştırmak. Önce eklenip sonra `pencere.size() > k` denetlenirse kod yine doğru çalışır. Ama önce eklenip sonra `size() == k` denetlenirse pencere hiç k'ye ulaşmadan küçültülür. Bu durumda yalnız son $k - 1$ öğenin ortalaması alınır. Bir başka hata, `toplam`'ı `double` tutmaktır. Uzun akışlarda ekleme ve çıkarma yuvarlama hatası biriktirir; tam sayı verilerde toplam tam sayı tutulmalıdır.

13.6.3 Çevrim içi ortanca

Her yeni öğeden sonra o ana kadarki öğelerin **ortancası** (*median*) isteniyor. Her seferinde sıralamak *O(n log n)*, sıralı bir diziye eklemek *O(n)* sürer. İki heap ile ekleme *O(log n)*'e, ortanca okuma *O(1)*'e iner. Bu yönteme **çevrim içi ortanca** (*running median*) denir.

Fikir şudur. Öğeler iki yarıya bölünür. Küçük yarı bir max-heap'te tutulur; buna `sol` denir. Büyük yarı bir min-heap'te tutulur; buna `sag` denir. Ortanca, iki yarının sınırındadır. Bu sınırdaki öğeler de tam heap tepeleridir. İki değişmez korunur:

- `sol`'daki her öğe, `sag`'daki her öğeden küçük ya da ona eşittir.
- Boyutlar ya eşittir ya da `sol` bir fazladır: *|sol| = |sag|* ya da *|sol| = |sag| + 1*.



Şekil 13.19: 5 öğe: `sol` (max-heap) 5, 3, 1 tutar; `sag` (min-heap) 8, 15 tutar. Öğe sayısı tek olduğu için ortanca `sol`'un tepesidir: 5.

Öğe sayısı tekse ortanca `sol`'un tepesidir. Çiftse iki tepenin ortalamasıdır. Java'da `PriorityQueue` varsayılan olarak min-heap'tir; max-heap ters karşılaştırıcıyla kurulur:

```
PriorityQueue<Integer> sol = new PriorityQueue<>((a, b) -> b - a); // max-heap
PriorityQueue<Integer> sag = new PriorityQueue<>();                // min-heap
```

```
void ekle(int x) {
  if (sol.isEmpty() || x <= sol.peek())
    sol.offer(x);          // küçük yarı
  else
    sag.offer(x);          // büyük yarı
  if (sol.size() > sag.size() + 1)
    sag.offer(sol.poll());
  else if (sag.size() > sol.size())
    sol.offer(sag.poll());
}
double ortanca() {
  if (sol.size() == sag.size())
    return (sol.peek() + sag.peek()) / 2.0;
  return sol.peek();
}
```

`ekle` iki aşamada çalışır. Önce öğe doğru yarıya konur: `sol` boşsa ya da öğe `sol`'un tepesinden büyük değilse küçük yarıya, değilse büyük yarıya gider. Bu adım birinci değişmezi korur. Sonra denge düzeltilir. `sol` iki fazla olduysa tepesi `sag`'a taşınır. `sag` büyük olduysa tepesi `sol`'a taşınır.

Taşınan öğe hep bir yarının sınırdaki öğesidir. Bu yüzden taşıma birinci değişmezi bozmaz. Her ekleme en çok üç heap işlemi yapar: *O(log n)*. `ortanca` yalnız tepelere bakar: *O(1)*. Tablo 13.21, 5, 15, 1, 3, 8, 7 akışını izler.

`x`	Girdiği yarı	Denge	`sol`	`sag`	Ortanca
5	`sol` (boş)	tamam	5	—	5
15	`sag` (15 > 5)	tamam	5	15	*(5 + 15)/2 = 10*
1	`sol` (1 ≤ 5)	tamam	5, 1	15	5
3	`sol` (3 ≤ 5)	3 > 1 + 1: 5 → `sag`	3, 1	5, 15	*(3 + 5)/2 = 4*
8	`sag` (8 > 3)	3 > 2: 5 → `sol`	5, 3, 1	8, 15	5
7	`sag` (7 > 5)	tamam	5, 3, 1	7, 8, 15	*(5 + 7)/2 = 6*

Tablo 13.21: Çevrim içi ortanca ile 5, 15, 1, 3, 8, 7. Heap içerikleri tepe önce yazılmıştır.



Şekil 13.20: İzlemenin sonu: `sol` 5, 3, 1; `sag` 7, 8, 15; ortanca 6.

Dördüncü ve beşinci adımlar dengenin neden gerektiğini gösterir. 3 eklendikten sonra `sol` 3, `sag` 1 öğe tutar. Ortanca artık `sol`'un tepesi değildir. 5'in `sag`'a taşınmasıyla iki yarı yine eşitlenir. Beşinci adımda bu kez `sag` büyük kalır ve 5 geri döner. Aynı öğe iki kez taşınmıştır; bu olağandır. Kod çalıştırıldığında ortancalar sırasıyla 5,0, 10,0, 5,0, 4,0, 5,0 ve 6,0 çıktı.

Örnek 13.8

Soru. Akış 2, 9, 4, 7 için çevrim içi ortancalar nedir?

Çözüm. 2 `sol`'a girer: ortanca 2. 9 > 2 olduğu için `sag`'a girer: ortanca *(2 + 9)/2 = 5,5*. 4 > 2 olduğu için `sag`'a girer. `sag` artık büyüktür ve tepesi 4, `sol`'a taşınır. `sol` 4, 2; `sag` 9 tutar; ortanca 4. 7 > 4 olduğu için `sag`'a girer. Boyutlar 2 ve 2'dir; ortanca *(4 + 7)/2 = 5,5*. Ortancalar 2, 5,5, 4, 5,5'tir. Sıralı listelerle denetlenebilir: [2, 9], [2, 4, 9] ve [2, 4, 7, 9].

Sık yapılan hata

Dengeyi unutmak. Öğeyi doğru yarıya koymak tek başına yetmez. Akış 5, 1, 3 ise üç öğe de `sol`'a girer. Denge yapılmazsa `ortanca()` `sol`'un tepesi olan 5'i döndürür; doğrusu 3'tür. Boyut farkı 1'i aşınca tepe öbür heap'e taşınmalıdır. İkinci bir hata max-heap'i `(a, b) -> b - a` ile kurmaktır. Çok büyük ve çok küçük sayılarda bu çıkarma taşar. Güvenli seçenek `Comparator.reverseOrder()`'dır.

13.6.4 Rezervuar örnekleme

Uzunluğu bilinmeyen bir akıştan k öğelik rastgele bir örneklem tutulmak isteniyor. Her öğe örneklemde eşit olasılıkla bulunmalıdır. Akışın uzunluğu n bilinseydi iş kolay olurdu: $1, \ldots, n$ arasından k indis seçilirdi. Ama n bilinmez ve öğeler saklanamaz. **Rezervuar örnekleme** (*reservoir sampling*) bu problemi $O(k)$ bellekle çözer.

Tanım 13.10

Algoritma R (Vitter, 1985): ilk k öğe rezervuara konur. Ardından 0 tabanlı her $i \geq k$ öğesi için $0 \ldots i$ aralığında düzgün rastgele bir j seçilir. $j < k$ ise rezervuardaki j. öğenin yerine x_i yazılır; değilse x_i atlanır.

Algoritmaya göre i. öğe $k / (i + 1)$ olasılıkla rezervuara girer. Girerse rastgele bir yerdeki öğeyi çıkarır. Geç gelen öğelerin girme olasılığı küçüktür. Ama erken gelen öğelerin de çıkarılma riski uzun süre devam eder. İki etki tam olarak dengelenir.

Teorem 13.6 (Algoritma R düzgündür)

$n \geq k$ öğelik bir akış bittiğinde her öğe rezervuarda k / n olasılıkla bulunur.

İspat. $i \geq k$ olan bir x_i öğesine bakalım. Öğe $k/(i + 1)$ olasılıkla girer. Sonraki her $t > i$ adımında x_t öğesi $k/(t + 1)$ olasılıkla girer ve girerse k yerden birini eşit olasılıkla seçer. x_i'nin bu adımda çıkarılma olasılığı $(k/(t + 1)) \cdot (1/k) = 1/(t + 1)$'dir. Kalma olasılığı $t/(t + 1)$'dir. Adımlar bağımsız olduğu için olasılıklar çarpılır:

$$k/(i + 1) \cdot (i + 1)/(i + 2) \cdot (i + 2)/(i + 3) \cdot \ldots \cdot (n - 1)/n = k / n$$

Çarpımda her pay bir önceki paydayı sadeleştirir; geriye k/n kalır. $i < k$ olan ilk k öğe için girme olasılığı 1'dir. Kalma çarpımı $k/(k + 1) \cdot \ldots \cdot (n - 1)/n = k/n$ olur. Her öğe için sonuç aynıdır. □



Şekil 13.21: *k = 2, n = 5*, 0 tabanlı 2. öğe: girme olasılığı 2/3, sonraki iki adımda kalma 3/4 ve 4/5. Çarpım *2/5 = k/n*.

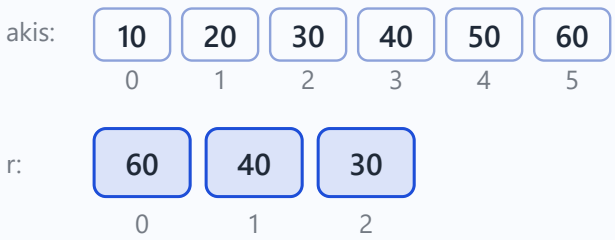
```
List<Integer> rezervuar(int[] akis, int k) {
  List<Integer> r = new ArrayList<>();
  for (int i = 0; i < k; i++) r.add(akis[i]);
  for (int i = k; i < akis.length; i++) {
    int j = rastgele.nextInt(i + 1); // 0..i
    if (j < k)
      r.set(j, akis[i]);
  }
  return r;
}
```

Kod bir listeyle çalışır. İlk döngü ilk *k* öğeyi rezervuara koyar. İkinci döngü kalan her öğe için `rastgele.nextInt(i + 1)` çağırır. Bu çağrı *0 … i* aralığında, uçlar dahil düzgün bir tam sayı döndürür. `j < k` ise `r.set` o yerdeki öğenin üzerine yazar. Öğe başına *O(1)* iş, toplam *O(k)* bellek kullanılır. Gerçek bir akışta dizi yerine bir yineleyiciyle çalışılır; algoritma değişmez.

Tablo 13.22 kodu 10, 20, 30, 40, 50, 60 akışı ve *k = 3* ile izler. Rastgele sayılar bu izleme için sabitlenmiştir: 1, 4 ve 0.

`i`	Değer	`j`	İşlem	`r`	Girme olasılığı
0–2	10, 20, 30	—	doğrudan eklenir	[10, 20, 30]	1
3	40	1	`r[1] = 40` (20 çıkar)	[10, 40, 30]	3/4
4	50	4	*4 ≥ 3*: girmez	[10, 40, 30]	3/5
5	60	0	`r[0] = 60` (10 çıkar)	[60, 40, 30]	3/6

Tablo 13.22: `rezervuar` ile 10, 20, 30, 40, 50, 60 ve *k = 3*.



Şekil 13.22: Akış bitti: örneklem [60, 40, 30].

Başka `j` değerleriyle başka bir örneklem çıkar. Ama teoreme göre her öğe *3/6 = 1/2* olasılıkla örneklemdedir. Bu iddia bir benzetimle de denetlendi. 1, ..., 10 akışı ve *k = 3* ile kod 300.000 kez çalıştırıldı. Her öğenin örneklemde bulunma sıklığı 0,299 ile 0,301 arasında çıktı; beklenen değer *3/10 = 0,3*'tür.

Örnek 13.9

Soru. *k = 2*'dir. Akışın 0 tabanlı 9. öğesi rezervuara hangi olasılıkla girer? Akış 10 öğeyle biterse bu öğe sonda hangi olasılıkla rezervuardadır?

Çözüm. Girme olasılığı *k/(i + 1) = 2/10 = 1/5*'tir. Akış 10 öğeyle bittiği için 9. öğe son öğedir ve ondan sonra çıkarılma riski yoktur. Sonda rezervuarda olma olasılığı da *1/5*'tir. Bu değer teoremdeki *k/n = 2/10* ile aynıdır.

Sık yapılan hata

Aralığı yanlış seçmek. `nextInt(i)` yazılırsa `j` *0 ... i − 1* aralığından gelir ve öğe *k/i* olasılıkla girer. Örneklem düzgün olmaz. Özellikle *i = k*'de olasılık 1'e çıkar: *k*. öğe her zaman girer. Hatalı kod 1, ..., 10 akışı ve *k = 3* ile 300.000 kez çalıştırıldı. İlk üç öğenin sıklığı 0,22'ye düştü, öteki yedi öğeninki 0,33'e çıktı.

13.6.5 Akışta örüntü arama: kayan hash

Bir ağ izleme programı, geçen paketlerde belli bir imzayı arıyor olsun. Karakterler akış hâlinde gelir. Örüntünün uzunluğu *m* ise yalnız son *m* karaktere bakmak yeterlidir. Her yeni karakterde bu pencereyi örüntüyle karşılaştırmak *O(m)* sürer. **Kayan hash** (*rolling hash*) karşılaştırmayı çoğu zaman *O(1)*'e indirir.

Pencerenin hash değeri, karakterleri *B* tabanında bir sayının basamakları gibi kullanır. Yeni karakter girdiğinde değer baştan hesaplanmaz. Çıkan karakterin katkısı çıkarılır, kalan değer bir basamak kaydırılır ve giren karakter eklenir:

$$h_{yeni} = (h_{eski} - c_{çıkan} \cdot B^{m-1}) \cdot B + c_{giren}$$

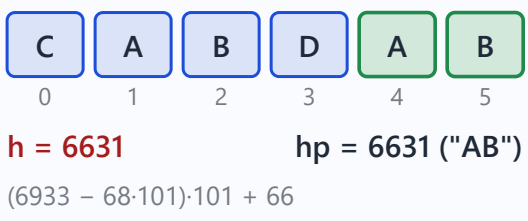
```
void ara(String akis, String oruntu) {
  int m = oruntu.length(), B = 101, h = 0, hp = 0;
  int gucu = 1;                          // B^(m-1)
  for (int i = 0; i < m - 1; i++) gucu *= B;
  for (int i = 0; i < m; i++)
    hp = hp * B + oruntu.charAt(i);
  for (int t = 0; t < akis.length(); t++) {
    char c = akis.charAt(t);
    if (t >= m) h -= akis.charAt(t - m) * gucu;
    h = h * B + c;          // giren eklenir
    if (t >= m - 1 && h == hp
        && akis.startsWith(oruntu, t - m + 1))
      System.out.println("eşleşme: " + (t - m + 1));
  }
}
```

Kod önce `gucu` değişkenine B^{m-1} değerini, `hp` 'ye de örüntünün hash değerini hesaplar. Ana döngü her karakter için çalışır. `t >= m` ise pencereden bir karakter çıkar ve katkısı `akis.charAt(t - m) * gucu` olarak çıkarılır. Sonra `h` bir basamak kaydırılır ve giren karakter eklenir. Pencere dolduktan sonra `h == hp` ise karakterler de `startsWith` ile karşılaştırılır. İki farklı dizgi aynı hash değerini verebilir; buna çakışma denir. Bu yüzden hash eşitliği tek başına yetmez. Hash'ler farklıysa karakterlere hiç bakılmaz. Çakışma seyrekse öğe başına iş *O(1)*'dir; bellek pencere için *O(m)*'dir.

Tablo 13.23, "AB" örüntüsünü "CABDAB" akışında arar. *B = 101*, *m = 2* ve `gucu` = 101'dir. Karakter kodları A = 65, B = 66, C = 67, D = 68'dir. Örüntünün hash değeri *65 · 101 + 66 = 6631*'dir.

`t`	Çıkan	Giren	Pencere	`h`	Sonuç
0	—	C	"C"	67	pencere dolmadı
1	—	A	"CA"	*67 · 101 + 65 = 6832*	≠ 6631
2	C	B	"AB"	*(6832 − 6767) · 101 + 66 = 6631*	eşleşme: 1
3	A	D	"BD"	6734	≠ 6631
4	B	A	"DA"	6933	≠ 6631
5	D	B	"AB"	*(6933 − 6868) · 101 + 66 = 6631*	eşleşme: 4

Tablo 13.23: `ara("CABDAB", "AB")` çağrısında kayan hash.



Şekil 13.23: Son adım: çıkan D, giren B; *h = 6631 = hp*. İkinci eşleşme 4. konumdadır.

Kod çalıştırıldığında "eşleşme: 1" ve "eşleşme: 4" yazdırdı. Örüntü uzadıkça `int` taşar. Bu durum doğruluğu bozmaz. Java'da `int` işlemleri 2^{32} modunda yapılır ve çıkarma, çarpma, toplama bu modda tutarlıdır. `h` ve `hp` aynı modda hesaplandığı için eşit pencereler eşit hash verir. Taşma yalnız çakışmaları artırabilir; karakter karşılaştırması bunları ayıklar. Uygulamada çoğunlukla büyük bir asal sayı modunda çalışılır.

13.7 Artımlı öğrenme

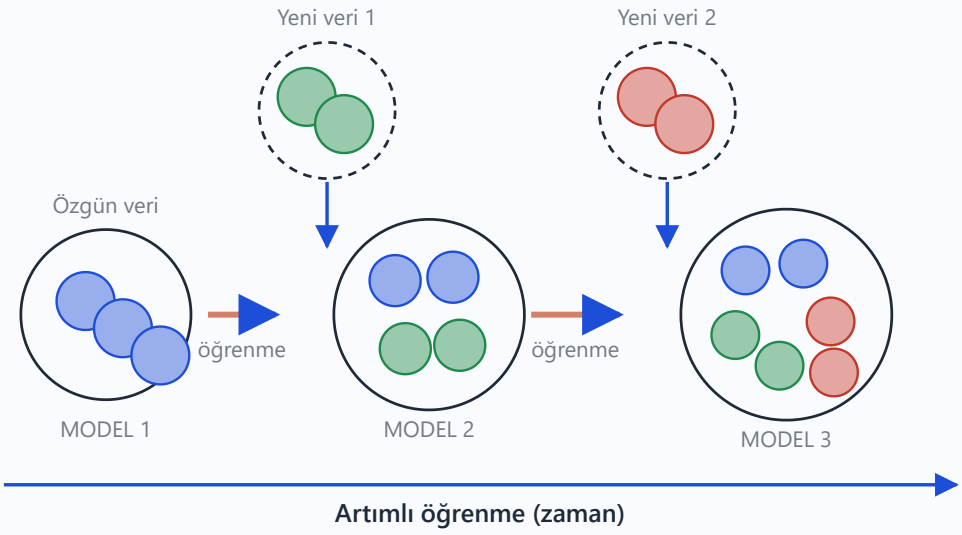
Makine öğrenmesinde model çoğunlukla bütün veriyle bir kez eğitilir. Veri sürekli geliyorsa bu yaklaşım pahalıdır. Her yeni veri parçasında modeli baştan eğitmek, o ana kadar gelen bütün veriyi yeniden işlemek demektir. Çevrim içi bakış burada da işe yarar: model yalnız yeni veriyle güncellenir.

13.7.1 Modeli yeni veriyle güncellemek

Tanım 13.11

Artımlı öğrenme (*incremental learning*), modelin her yeni veriyle, geçmiş veriyi yeniden işlemeden güncellenmesidir. Güncellenmiş model hem eski hem yeni verideki bilgiyi temsil eder.

Şekil 13.24 süreci gösterir. Başlangıçta model özgün veri kümesiyle eğitilir ve MODEL 1 elde edilir. Yeni veri 1 geldiğinde model yalnız bu veriyle güncellenir ve MODEL 2 olur. MODEL 2 iki veri kümesinin bilgisini taşır, ama özgün veri yeniden işlenmemiştir. Yeni veri 2 ile aynı adım tekrarlanır. Model her an günceldir ve bir güncellemenin maliyeti yalnız yeni verinin boyutu kadardır.



Şekil 13.24: Artımlı öğrenme. Model her yeni veri kümesiyle güncellenir; MODEL 3 üç veri kümesinin bilgisini taşır.

Akış algoritmalarıyla benzerlik açıktır. Model, geçmiş verinin bir özetidir; tıpkı hareketli ortalamadaki toplam ya da çevrim içi ortancadaki iki heap gibi. Fark, özetin daha karmaşık olmasıdır: bir sınıflandırıcının ağırlıkları, bir karar ağacının dalları ya da bir sinir ağının parametreleri.

13.7.2 Çalışan ortalama ve çevrim içi gradyan inişi

Artımlı güncellemenin en basit örneği **çalışan ortalamadır**. Akıştaki bütün öğelerin ortalaması, geçmişi saklamadan güncellenebilir:

$$\mu_n = \mu_{n-1} + (x_n - \mu_{n-1}) / n$$

Formül, yeni öğenin eski ortalamadan farkının $1/n$'si kadar düzeltme yapar. Akış 4, 8, 6 için $\mu_1 = 4$, $\mu_2 = 4 + (8 - 4)/2 = 6$, $\mu_3 = 6 + (6 - 6)/3 = 6$ olur. Bu değer $(4 + 8 + 6)/3 = 6$ ile aynıdır. Saklanan yalnız n ve μ'dür; bellek $O(1)$'dir. Formül toplamı ayrıca tutmadığı için uzun akışlarda büyük sayılarla da sorun çıkarmaz.

Aynı fikir model eğitimine **çevrim içi gradyan inişi** (*online gradient descent*) olarak taşınır. Model bir ağırlık vektörü w ile tanımlanır. Her örnek (x_t, y_t) geldiğinde ağırlık, kaybın gradyanının tersi yönünde küçük bir adım atar:

$$w \leftarrow w - \eta \cdot \nabla \ell(w; x_t, y_t)$$

Burada ℓ kayıp fonksiyonu, η ise öğrenme oranıdır. Her örnek bir kez kullanılır ve atılır.

Örnek 13.10

Soru. Model $\hat{y} = w \cdot x$, kayıp $\ell = (w \cdot x - y)^2 / 2$ olsun. Bu kaybın w'ye göre türevi $(w \cdot x - y) \cdot x$'tir. Başlangıçta $w = 0$, öğrenme oranı $\eta = 0{,}1$'dir. Örnekler (1, 2), (2, 4), (3, 6) sırasıyla geliyor. Her örnekten sonra w nedir?

Çözüm. Birinci örnekte tahmin 0, hata $0 - 2 = -2$, gradyan $-2 \cdot 1 = -2$'dir. $w = 0 - 0{,}1 \cdot (-2) = 0{,}2$ olur. İkinci örnekte tahmin $0{,}2 \cdot 2 = 0{,}4$, hata $-3{,}6$, gradyan $-3{,}6 \cdot 2 = -7{,}2$'dir. $w = 0{,}2 + 0{,}72 = 0{,}92$ olur. Üçüncü örnekte tahmin $2{,}76$, hata $-3{,}24$, gradyan $-9{,}72$'dir. $w = 0{,}92 + 0{,}972 = 1{,}892$ olur. Verideki gerçek ilişki $y = 2x$'tir ve w üç örnekte 2'ye yaklaşmıştır. Dördüncü bir örnek olarak (1, 2) yeniden gelseydi w 1,9028 olurdu.

13.7.3 Zorluklar

Artımlı öğrenme ucuz ve güncel bir model verir, ama kendine özgü sorunları vardır.

- **Felaket boyutunda unutma** (*catastrophic forgetting*): model yeni veriye uyum sağlarken eski bilgiyi kaybeder. Bir sinir ağı yeni bir görevi öğrenirken eski görevdeki başarısı birden düşebilir.

- **Kavram kayması** (*concept drift*): verinin dağılımı zamanla değişir. Geçen yılın alışveriş alışkanlıkları bu yılı iyi temsil etmeyebilir. Eski veri bu durumda yanıltıcı olur.
- **Sıra etkisi**: aynı veri farklı sırayla gelirse model farklı biçimlenir. Çevrim dışı eğitimde veri karıştırılabilir; akışta sıra veriyle birlikte gelir.
- **Gürültü**: tek bir aykırı örnek modeli bozabilir. Küçük bir öğrenme oranı ve kayan pencere bu etkiyi azaltır.

Unutma ile kavram kayması birbirine ters yönde çeker. Kavram kayması varsa eski verinin bir ölçüde unutulması istenir. Kayma yoksa unutma bir kayıptır. Hareketli ortalamadaki pencere boyu ya da öğrenme oranı bu dengeyi ayarlayan düğmelerdir.

13.8 Problemler ve sonuçlar

Tablo 13.24 bölümdeki problemleri ve sonuçlarını bir arada verir. İlk beş satırdaki sayılar rekabet oranıdır. Alttaki satırlar ise öğe başına zamanı verir.

Problem	Çevrim içi algoritma	Sonuç	Not
Kayak kiralama	$B - 1$ gün kiralanır, sonra satın alınır	$2 - 1/B$	deterministikte en iyi; rastgele $e/(e-1)$
Sayfalama	LRU, FIFO, FWF	k	deterministikte en iyi
Sayfalama	LFU, LIFO	sınırsız	rekabetçi değil
Sayfalama (rastgele)	İşaretleme	$2H_k$	alt sınır H_k
Liste güncelleme	Öne taşı	2	transpoz ve sıklık sayacı rekabetçi değil
Hareketli ortalama	Kuyruk ve toplam	$O(1)$	bellek $O(k)$
Çevrim içi ortanca	İki heap	$O(\log n)$	ortanca okuma $O(1)$
Rezervuar örnekleme	Algoritma R	$O(1)$	her öğe k/n olasılıkla

Tablo 13.24: Çevrim içi problemler, algoritmalar ve sonuçlar.

Tablodaki üç rekabetçi problemin kanıtları aynı iskeleti izler. Üst sınır için algoritmanın maliyeti OPT'nin bir katıyla sınırlanır. Kayak kiralamada bu bir durum ayrımıyla, sayfalamada evrelerle, liste güncellemede potansiyel fonksiyonuyla yapılır. Alt sınır için ise düşman, deterministik algoritmanın bir sonraki kararını bilerek onun en çok zarar göreceği isteği üretir. Rastgelelik üç problemde de oranı düşürür, çünkü düşmanın bu bilgisini elinden alır.

13.9 Sık yapılan hatalar

- **“Çevrim içi = internet” sanmak.** Çevrim içi algoritma, girdinin zamanla açığa çıktığı modeldir. Ağ bağlantısıyla ilgisi yoktur.
- **Oranı ortalama durumla karıştırmak.** Rekabet oranı en kötü girdiye göredir. “Genelde iyi çalışır” bir rekabet oranı değildir.

- **FIFO ile LRU'yu karıştırmak.** FIFO'da isabet sırayı değiştirmez; LRU'da isabet sayfayı en yeni yapar. İki politikanın tek farkı budur.
- **Zorunlu hataları unutmak.** İlk k farklı sayfa her politikada hata verir. OPT bile sıfır hata yapamaz.
- **OPT'yi "en az kullanılacak sayfa" sanmak.** OPT, bir sonraki kullanımı en uzak olan sayfayı çıkarır; kullanım sayısına bakmaz.
- **Büyük önbelleğin hep daha iyi olduğunu sanmak.** FIFO'da çerçeve artınca hata artabilir: Belady anomalisi. LRU ve OPT'de bu olmaz.
- **Rezervuarda yanlış aralık.** j, $0 \ldots i$ aralığından, i dahil seçilmelidir. `nextInt(i)` yazmak olasılıkları bozar.
- **Ortancada dengeyi unutmak.** Öğeyi doğru heap'e koymak yetmez. Boyut farkı 1'i aşınca tepe öbür heap'e taşınmalıdır.
- **Hash eşitliğini eşleşme saymak.** Kayan hash'te eşit değerler çakışmadan da gelebilir. Karakterler ayrıca karşılaştırılmalıdır.

13.10 Özet

- **Model:** girdi bir istek dizisi olarak parça parça gelir. Karar istek geldiği anda verilir ve geri alınamaz. Çevrim dışı algoritma ise girdinin tamamını baştan görür.
- **Rekabet oranı:** her σ için $ALG(\sigma) \le c \cdot OPT(\sigma) + \alpha$ sağlanıyorsa algoritma c-rekabetçidir. Analiz en kötü durumu ölçer. Alt sınırlar düşman modeliyle kurulur.
- **Kayak kiralama:** $OPT = min(d, B)$. Başa baş stratejisi $B - 1$ gün kiralar ve B. gün satın alır; oranı $2 - 1/B$'dir. Hiçbir deterministik strateji daha iyi değildir; rastgele strateji $e/(e - 1) \approx 1{,}58$'e iner.
- **Sayfalama:** FIFO en önce yükleneni, LRU en uzun süredir kullanılmayanı, OPT bir sonraki kullanımı en uzak olanı çıkarır. Örnek dizide 10, 9 ve 7 hata. LRU ve FIFO k-rekabetçidir ve deterministikte daha iyisi yoktur. Rastgele işaretleme $2H_k$'ye iner.
- **Belady anomalisi:** FIFO'da çerçeve artınca hata artabilir (9 → 10). LRU ve OPT yığın algoritmasıdır; anomali göstermez.
- **Uygulama:** LRU bağlı listeyle istek başına $O(k)$, `LinkedHashMap` ile $O(1)$'dir. FIFO bir kuyruk ve bir kümeyle $O(1)$'dir.
- **Liste güncelleme:** maliyet konumdur. Öne taşı kuralı $MTF \le 2 \cdot OPT - m$ sağlar; kanıt ters çift sayısını potansiyel alır. Transpoz ve sıklık sayacı rekabetçi değildir.
- **Veri akışları:** tek geçiş ve az bellek. Hareketli ortalama kayan pencereyle $O(1)$, çevrim içi ortanca iki heap ile $O(\log n)$, rezervuar örnekleme $O(1)$ ve $O(k)$ bellekle çalışır. Rezervuarda her öğe k/n olasılıkla bulunur. Kayan hash, pencereyi $O(1)$'de günceller.
- **Artımlı öğrenme:** model yeni veriyle güncellenir. Çalışan ortalama ve çevrim içi gradyan inişi temel örneklerdir. Unutma ve kavram kayması başlıca zorluklardır.

13.11 Kavram listesi

çevrim içi algoritma *online algorithm* · çevrim dışı algoritma *offline algorithm* · rekabetçi analiz *competitive analysis* · rekabet oranı *competitive ratio* · düşman *adversary* · kayak kiralama problemi *ski rental problem* · rastgele algoritma *randomized algorithm* · sayfalama *paging* · önbellek *cache* · sayfa hatası *page fault* · çıkarma politikası *eviction policy* · Belady anomalisi *Belady's anomaly* · yığın algoritması *stack algorithm* · işaretleme algoritması *marking algorithm* · liste güncelleme problemi *list update problem* · öne taşı *move-to-front (MTF)* · transpoz *transpose* · amortize analiz *amortized analysis* · ters çift *inversion* · veri akışı *data stream* · çevrim içi ortanca *running median* · rezervuar örnekleme *reservoir sampling* · kayan hash *rolling hash* · artımlı öğrenme *incremental learning*

13.12 Kendini Dene

1. Kayak kiralamada $B = 8$'dir. 6. gün satın alan stratejinin en kötü durum oranı nedir? Bu oran başa baş stratejisininkinden ne kadar farklıdır?

   Cevap

   Düşman sezonu 6. gün bitirir. ALG = $5 + 8 = 13$, OPT = $\min(6, 8) = 6$; oran $13/6 \approx 2{,}17$. Başa baş stratejisinin oranı $2 - 1/8 = 15/8 = 1{,}875$'tir. Erken satın alan strateji daha kötüdür.

2. 1 2 1 3 1 4 1 2 dizisi $k = 2$ çerçeveyle işleniyor. FIFO ve LRU kaçar sayfa hatası yapar?

   Cevap

   FIFO 6 hata yapar: 3 gelince en önce yüklenen 1 çıkar ve 1 hemen geri istenir. 1 gelince 2, 4 gelince 3, son 2 gelince 1 çıkar. LRU 5 hata yapar. 1 her isabette en yeni olduğu için hiç çıkmaz; 3, 4 ve son 2 gelince sırasıyla 2, 3 ve 4 çıkar. Bu dizide OPT de 5 hata yapar.

3. 1 2 3 1 4 2 dizisi $k = 3$ ile OPT politikasıyla işleniyor. Kaç hata olur? 4 geldiğinde hangi sayfalar çıkarılabilir?

   Cevap

   4 hata olur: üç zorunlu hata ve 4'ün hatası. 4 geldiğinde önbellek {1, 2, 3}'tür. 1 ve 3 bir daha istenmez, 2 son istekte istenir. 1 ya da 3 çıkarılabilir; ikisi de aynı sonucu verir. Son istekteki 2 isabet alır. Aynı dizide LRU 5 hata yapmıştı.

4. A B C listesinde öne taşı kuralıyla C C B A isteklerinin toplam maliyeti nedir? Hiç taşımayan liste ne öder?

   Cevap

   C 3. konumda: 3, liste C A B. C 1. konumda: 1. B 3. konumda: 3, liste B C A. A 3. konumda: 3, liste A B C. Toplam *3 + 1 + 3 + 3 = 10*. Taşımayan liste *3 + 3 + 2 + 1 = 9* öder. Bu kısa dizide öne taşı kuralı biraz daha pahalıdır; 2-rekabetçi olmak her dizide en iyi olmak demek değildir.

5. `new HareketliOrtalama(2)` nesnesine sırasıyla 3, 5, 9, 1 veriliyor. `sonraki` hangi değerleri döndürür?

   Cevap

   3,0; *(3 + 5)/2 = 4,0*; 3 çıkar ve *(5 + 9)/2 = 7,0*; 5 çıkar ve *(9 + 1)/2 = 5,0*.

6. Çevrim içi ortanca yapısına 10, 2, 7, 4 sırasıyla ekleniyor. Her eklemeden sonra ortanca nedir? 7 eklendiğinde heap'ler arasında taşıma olur mu?

   Cevap

   Ortancalar 10, 6, 7, 5,5'tir. 2 eklenince `sol`'a girer, `sol` iki öğe tutar ve tepe 10 `sag`'a taşınır: ortanca *(2 + 10)/2 = 6*. 7 eklenince *7 > 2* olduğu için `sag`'a girer. `sag` iki, `sol` bir öğe tutar. `sag`'ın tepesi 7 `sol`'a taşınır; ortanca 7'dir. 4 `sol`'a girer; `sol` üç öğe tutar ve tepesi 7 `sag`'a taşınır. Boyutlar 2 ve 2 olur: *(4 + 7)/2 = 5,5*.

7. *k = 1* ile rezervuar örneklemede 0 tabanlı 4. öğe hangi olasılıkla rezervuara girer? Akış 8 öğeyle biterse bu öğe sonda hangi olasılıkla seçilmiş olur?

   Cevap

   Girme olasılığı *1/5*'tir. Sonraki üç adımda kalma olasılıkları *5/6*, *6/7* ve *7/8*'dir. Çarpım *1/5 · 5/6 · 6/7 · 7/8 = 1/8 = k/n* olur.

8. *k = 6* sayfalık bir önbellekte deterministik LRU ile rastgele işaretleme algoritmasının rekabet oranları nedir? Hangisi daha iyi bir garanti verir?

   Cevap

   LRU'nun oranı 6'dır. $H_6 = 1 + 1/2 + \ldots + 1/6 = 49/20 = 2{,}45$; işaretleme algoritmasının oranı $2H_6 = 4{,}9$'dur. İşaretleme bu durumda deterministik sınırın altındadır. Rastgele hiçbir algoritma 2,45'ten iyi olamaz.

9. `ara("ABA", "BA")` çağrısında *B = 101* ile `hp` nedir? `t = 2` anında `h` hangi değeri alır ve ne yaz-

dırılır?

Cevap

`hp` = *66 · 101 + 65 = 6731*. `t = 1`'de pencere "AB" ve `h` = 6631'dir. `t = 2`'de A çıkar: *(6631 − 65 · 101) · 101 + 65 = 66 · 101 + 65 = 6731*. Hash eşittir ve karakterler de tutar; "eşleşme: 1" yazdırılır.

13.13 İleri okuma

- A. Borodin ve R. El-Yaniv, *Online Computation and Competitive Analysis*: alanın temel kitabıdır. Sayfalama, liste güncelleme, düşman modelleri ve rastgele algoritmaları ayrıntılı kanıtlarla işler.
- T. H. Cormen ve arkadaşları, *Introduction to Algorithms*, 4. baskı, 27. bölüm: çevrim içi algoritmalara giriş yapar; asansör bekleme (kayak kiralamanın bir türü), liste güncelleme ve önbellek problemlerini ele alır.
- S. Albers, "Online algorithms: a survey" (2003): rekabetçi analizin başlıca sonuçlarını kısa ve okunaklı bir özetle verir.
- J. S. Vitter, "Random Sampling with a Reservoir" (1985): Algoritma R'yi ve onu hızlandıran atlamalı sürümleri tanıtır.

13.14 Kaynaklar

1. A. Borodin ve R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.
2. D. D. Sleator ve R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules", *Communications of the ACM* 28(2), 1985.
3. L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer", *IBM Systems Journal* 5(2), 1966.
4. L. A. Belady, R. A. Nelson ve G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine", *Communications of the ACM* 12(6), 1969.
5. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator ve N. E. Young, "Competitive Paging Algorithms", *Journal of Algorithms* 12(4), 1991.
6. A. R. Karlin, M. S. Manasse, L. A. McGeoch ve S. Owicki, "Competitive Randomized Algorithms for Nonuniform Problems", *Algorithmica* 11(6), 1994.
7. J. S. Vitter, "Random Sampling with a Reservoir", *ACM Transactions on Mathematical Software* 11(1), 1985.
8. S. Albers, "Online Algorithms: A Survey", *Mathematical Programming* 97(1–2), 2003.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (27. bölüm: Online Algorithms).

KISIM V · BÖLÜM 14

Paralel Algoritmalar

Bugünkü bir dizüstü bilgisayarda bile birden çok işlemci çekirdeği bulunur. Sıralı bir algoritma bu çekirdeklerden yalnız birini kullanır; ötekiler boşta bekler. Paralel algoritma, işi parçalara ayırıp parçaları aynı anda farklı işlemcilerde çözer ve sonuçları birleştirir. Bu not önce paralelliğin neden gerektiğini ve hangi bedelleri getirdiğini anlatır. Ardından paralel bilgisayarların sınıflandırmasını, PRAM modelini ve başarım ölçütlerini verir: hızlanma, verimlilik, maliyet, Amdahl yasası, iş ve derinlik. Temel teknikler küçük örnekler üzerinde adım adım çalıştırılır: ağaçla toplama, önek toplamı, boru hattı ve boru hatlı ağaç. Son kesimde aynı fikirler Java'da kurulur. Fork/Join ile birleştirme sıralaması, MapReduce ile kelime sayma ve satır görevleriyle matris çarpımı yazılır. Paylaşılan durumun doğurduğu yarış durumu ölçülerek gösterilir.

Bu bölümün kodu: `github.com/sercankulcu/algorithms-java/Bolum14/src`

ÖĞRENME ÇIKTILARI

Kavramlar

Paralel algoritmanın böl, ata, çöz ve birleştir yapısını; getirdiği kazanımları ve zorlukları (iletişim, senkronizasyon, yük dengeleme) açıklamak.

Modeller ve ölçütler

Flynn sınıflarını, bellek mimarilerini ve PRAM modelini tanımak; hızlanma, verimlilik, maliyet, iş ve derinliği hesaplamak; Amdahl yasasını ve Brent teoremini uygulamak.

Temel teknikler

PRAM ağaç toplamını, paralel önek toplamını, boru hattını ve boru hatlı ağacı adım adım yürütmek; tur ve adım sayılarını bulmak.

Java'da paralellik

Fork/Join ile paralel birleştirme sıralamasını, MapReduce'u ve veri paralelliği örneklerini okumak; yarış durumunu tanımak ve önlemek.

14.1 Paralel algoritma fikri

Bir sınıfın 1000 sınav kâğıdını tek bir öğretmen okusun. Kâğıt başına yaklaşık 36 saniye harcanırsa iş 10 saat sürer. Aynı kâğıtlar 10 öğretmen arasında paylaştırılırsa her öğretmen 100 kâğıt okur ve iş yaklaşık 1 saatte biter. Ama bu hesap eksiktir. Kâğıtları öğretmenlere dağıtmak zaman alır. Herkes bitirdikten sonra notları tek bir listede toplamak da zaman alır. Bir öğretmen geç kalırsa öteki dokuzu onu bekler. Paralel algoritmaların bütün konusu bu küçük örnekte gizlidir. İş bölünür, parçalar aynı anda yapılır ve sonuçlar birleştirilir. Bölmenin ve birleştirmenin bedeli ise kazancın bir kısmını geri alır.

14.1.1 Neden paralellik?

Bazı işler tek bir işlemcide çok uzun sürer. Milyarlarca satırlık bir veri kümesini çözümlemek, bir hava durumu modelini günlerce ileri doğru çalıştırmak, binlerce görüntü üzerinde bir sinir ağını eğitmek bu türdendir. Bu işlerin süresini kısaltmanın iki yolu vardır: tek işlemciyi hızlandırmak ya da işi birden çok işlemciye bölmek.

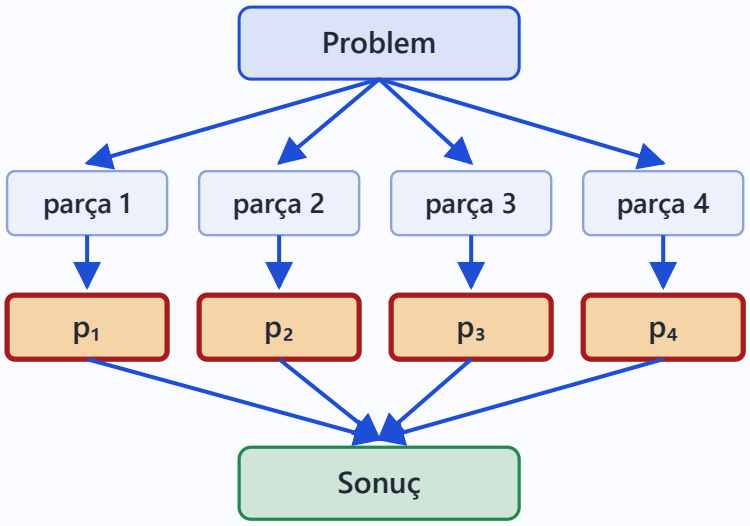
Uzun yıllar boyunca birinci yol yeterli oldu. İşlemcilerin saat hızı her birkaç yılda bir katlanıyordu ve aynı program hiçbir değişiklik yapılmadan daha hızlı çalışıyordu. 2000'lerin ortasından bu yana bu artış neredeyse durdu. Saat hızını yükseltmek ısınmayı ve güç tüketimini kabul edilemez ölçüde artırıyor. Üreticiler bunun yerine tek bir yongaya birden çok çekirdek yerleştirmeye başladı. Bugün başarım artışı büyük ölçüde çekirdek sayısından gelir. Bir program bu artıştan ancak işini çekirdeklere bölebiliyorsa yararlanır.

> **Tanım 14.1 (Paralel algoritma)**
>
> **Paralel algoritma** (*parallel algorithm*), bir problemi aynı anda çalışan birden çok işlemci (ya da çekirdek) kullanarak çözen algoritmadır. Her işlemci işin bir bölümünü yapar; işlemciler gerektiğinde veri alışverişi yapar ve kendi aralarında eşgüdüm sağlar.

Tanımdaki “aynı anda” ifadesi önemlidir. Tek çekirdekli bir bilgisayar da birden çok programı “aynı anda” çalıştırıyor gibi görünür. Gerçekte işletim sistemi programlar arasında hızla geçiş yapar ve herhangi bir anda yalnız biri çalışır. Paralel algoritmada ise birden çok işlemci gerçekten aynı saat diliminde iş yapar. Bu notta işlemci, çekirdek ve iş parçacığı sözcükleri bağlama göre aynı anlamda, yani “bağımsız çalışabilen bir hesap birimi” anlamında kullanılır.

14.1.2 Böl, ata, çöz, birleştir

Paralel bir algoritmanın iskeleti çoğunlukla dört adımdan oluşur. Problem önce birbirinden olabildiğince bağımsız parçalara **bölünür**. Parçalar farklı işlemcilere **atanır**. İşlemciler parçalarını aynı anda **çözer**. Son olarak kısmi sonuçlar **birleştirilerek** asıl problemin sonucu elde edilir. Şekil 14.1 bu akışı dört işlemci için gösterir.



Şekil 14.1: Paralel algoritmanın genel yapısı. Problem dört parçaya bölünür, parçalar dört işlemcide (p_1 … p_4) aynı anda çözülür ve sonuçlar birleştirilir.

Bu yapı, böl ve fethet yöntemini andırır. Böl ve fethette de problem alt problemlere ayrılır ve sonuçlar birleştirilir. Fark, alt problemlerin çözülme biçimindedir. Sıralı böl ve fethette alt problemler birbiri ardına çözülür. Paralel algoritmada ise bağımsız alt problemler aynı anda çözülür. Bu yüzden böl ve fethet algoritmaları paralelleştirmeye en elverişli algoritmalar arasındadır.

Bir grup insanın ortak bir işi yürütmesi iyi bir benzetmedir. Bir ev taşınırken bir kişi kitapları, bir kişi mutfağı, bir kişi giysileri toplar. Herkes kendi işini bitirince kutular aynı kamyona yüklenir. İşin hızlanması için iki koşul gerekir. Birincisi, parçalar gerçekten bağımsız olmalıdır: mutfağı toplayan kişi kitapları toplayanı beklememelidir. İkincisi, parçalar kabaca eşit büyüklükte olmalıdır: kitaplar iki saat, giysiler on dakika sürüyorsa giysileri toplayan kişi çoğu zaman boş kalır.

Paralel hesaplama, **yüksek başarımlı hesaplamanın** (*high-performance computing*, HPC) temelidir. Büyük veri çözümlemesi, bilimsel benzetim ve makine öğrenmesi gibi alanlar bugün büyük ölçüde paralel algoritmalara dayanır.

14.1.3 Kazanımlar ve zorluklar

Paralelliğin üç temel kazanımı vardır:

- **Hızlanma.** Aynı iş daha kısa sürede biter. Bir hava tahmini modeli yarın sabaha değil bu akşama hazır olur.
- **Ölçeklenebilirlik.** Tek bir makinenin belleğine ya da diskine sığmayan büyük problemler, birçok makineye dağıtılarak çözülebilir.
- **Donanımın verimli kullanımı.** Bilgisayardaki bütün çekirdekler ve grafik işlemcisi (GPU) iş yapar; boşta bekleyen donanım azalır.

Bu kazanımların bedeli de vardır:

- **Bölme ve birleştirme ek çaba ister.** Problemi parçalara ayırmak ve kısmi sonuçları birleştirmek sıralı algoritmada olmayan ek işlemlerdir.
- **İletişim ve senkronizasyon ek yük getirir.** İşlemciler veri alışverişi yapmak ve birbirini beklemek zorundadır. Bu süre boyunca yararlı iş yapılmaz.
- **Yük dengeleme gerekir.** İş, işlemcilere eşit dağıtılmalıdır. En uzun süren parça, bütün işin süresini belirler.
- **Hata ayıklama zordur.** Paralel programlarda **yarış durumu** ve **kilitlenme** (*deadlock*) gibi hatalar ortaya çıkar. Bu hatalar her çalıştırmada görülmeyebilir.
- **Bazı problemler yapısı gereği sıralıdır.** Bir adım bir önceki adımın sonucuna muhtaçsa o adımlar aynı anda yapılamaz.

Son madde paralelliğin kuramsal sınırını çizer. $x_k = f(x_{k-1})$ biçimindeki bir yinelemeyi düşünelim; f herhangi bir fonksiyon olsun. x_5'i hesaplamak için x_4 bilinmelidir, x_4 için de x_3. Bu bir **veri bağımlılığı** zinciridir. İşlemci sayısı ne olursa olsun zincirin halkaları birer birer hesaplanır. Böyle bir hesap paralelleştirilemez.

Her zincir ise bu kadar katı değildir. $y_i = y_{i-1} + x_i$ yinelemesi de ilk bakışta aynı yapıdadır. Ama toplama birleşmeli bir işlem olduğu için parantezler yeniden düzenlenebilir ve hesap ağaç biçiminde paralel yapılabilir. Bu fikir, önek toplamı kesiminde ayrıntılı işlenir. Bir zincirin paralelleştirilip paralelleştirilemeyeceği, zincirdeki işlemin cebirsel özelliklerine bağlıdır.

> **Sık yapılan hata**
>
> Her döngünün paralelleştirilebileceğini sanmak. `for (i = 1; i < n; i++) x[i] = x[i - 1] * x[i - 1] + 1;` döngüsünde her yineleme bir öncekinin yazdığı değeri okur. Döngünün yinelemeleri farklı iş parçacıklarına dağıtılırsa bir iş parçacığı henüz hesaplanmamış bir değeri okur ve sonuç yanlış çıkar. Paralelleştirmeden önce yinelemeler arasında veri bağımlılığı olup olmadığına bakılmalıdır.

14.1.4 Uygulama alanları

Paralelliğin işe yaradığı problemlerde ortak bir özellik vardır: işin büyük bir bölümü, birbirinden bağımsız küçük parçalara ayrılabilir. Dört tipik örnek şunlardır:

- **Matris çarpımı.** Bilimsel hesaplamanın temel işlemidir. $C = A \cdot B$ çarpımında sonuç matrisinin her satırı, hatta her hücresi ötekilerden bağımsız hesaplanabilir. Bu notun sonunda satır başına bir görevle çalışan bir Java çözümü verilir.
- **Görüntü işleme.** Kenar bulma, bulanıklaştırma ve nesne tanıma gibi işlemlerde her piksel ya da her piksel bloğu için aynı hesap yapılır. Bu, veri paralelliğinin en açık örneğidir. Grafik işlemcileri tam da bu tür iş için binlerce küçük çekirdek barındırır.

- **Hesaplamalı akışkanlar dinamiği.** Uçak, gemi ya da araba tasarımında hava ve su akışının benzetimi yapılır. Akış alanı bir ızgaraya bölünür ve her bölge bir işlemciye verilir. İşlemciler her zaman adımında yalnız komşu bölgelerle sınır değerlerini paylaşır.
- **Genetik algoritmalar.** Bir popülasyondaki bireylerin uygunluk değerleri birbirinden bağımsızdır ve aynı anda hesaplanabilir. Çizelgeleme ve rota bulma gibi problemlerde bu değerlendirme işin en pahalı kısmıdır.

14.2 Paralel hesaplama modelleri

Paralel bir algoritmayı tasarlamak ve çözümlemek için önce hangi tür makinede çalışacağı belirlenmelidir. Sıralı algoritmalarda bu soru pek sorulmaz. Her bilgisayar aşağı yukarı aynı biçimde çalışır: bir işlemci komutları sırayla yürütür ve belleğe erişir. Paralel makineler ise birbirinden çok farklıdır. İşlemcilerin aynı komutu mu yoksa farklı komutları mı çalıştırdığı, belleği paylaşıp paylaşmadığı ve birbirleriyle nasıl haberleştiği değişir. Bu kesim paralel makineleri sınıflandıran iki bakış açısını ve kuramsal çözümlemede kullanılan PRAM modelini anlatır.

14.2.1 Flynn sınıflandırması

Michael Flynn 1966'da bilgisayarları iki soruya göre sınıflandırdı. Birinci soru, aynı anda kaç komut akışının yürütüldüğüdür. İkinci soru, aynı anda kaç veri akışının işlendiğidir. Her sorunun yanıtı “tek” ya da “çok” olabildiği için dört sınıf ortaya çıkar. Tablo 14.1 bu sınıfları özetler. Sınıf adları İngilizce baş harflerden oluşur: *S* tek (*single*), *M* çok (*multiple*), *I* komut (*instruction*), *D* veri (*data*) anlamındadır.

Sınıf	Açılım	Anlamı	Örnek
SISD	*Single Instruction, Single Data*	tek komut akışı, tek veri	klasik tek çekirdekli işlemci
SIMD	*Single Instruction, Multiple Data*	aynı komut, birçok veri üzerinde aynı anda	vektör komutları (AVX), GPU
MISD	*Multiple Instruction, Single Data*	farklı komutlar aynı veri üzerinde	nadir; hata toleranslı sistemler
MIMD	*Multiple Instruction, Multiple Data*	her işlemci kendi programını kendi verisinde	çok çekirdekli CPU, kümeler

Tablo 14.1: Flynn sınıflandırması.

SISD, sıralı algoritmaların varsaydığı klasik makinedir. Her saat adımında tek bir komut tek bir veri üzerinde yürütülür.

SIMD makinede tek bir denetim birimi vardır ama bu birim aynı komutu birçok veri üzerinde aynı anda uygular. Modern işlemcilerin vektör komutları bu türdendir. Örneğin AVX komut kümesindeki bir toplama komutu, sekiz tane 32 bitlik kayan noktalı sayıyı tek adımda başka sekiz sayıyla toplar. Grafik işlemcileri de aynı fikri çok daha büyük ölçekte uygular. SIMD, bir dizinin her öğesine aynı işlemin uygulandığı işlerde çok etkilidir. Öğeye göre farklı yollar izleyen dallanmalı kodda ise verimi düşer.

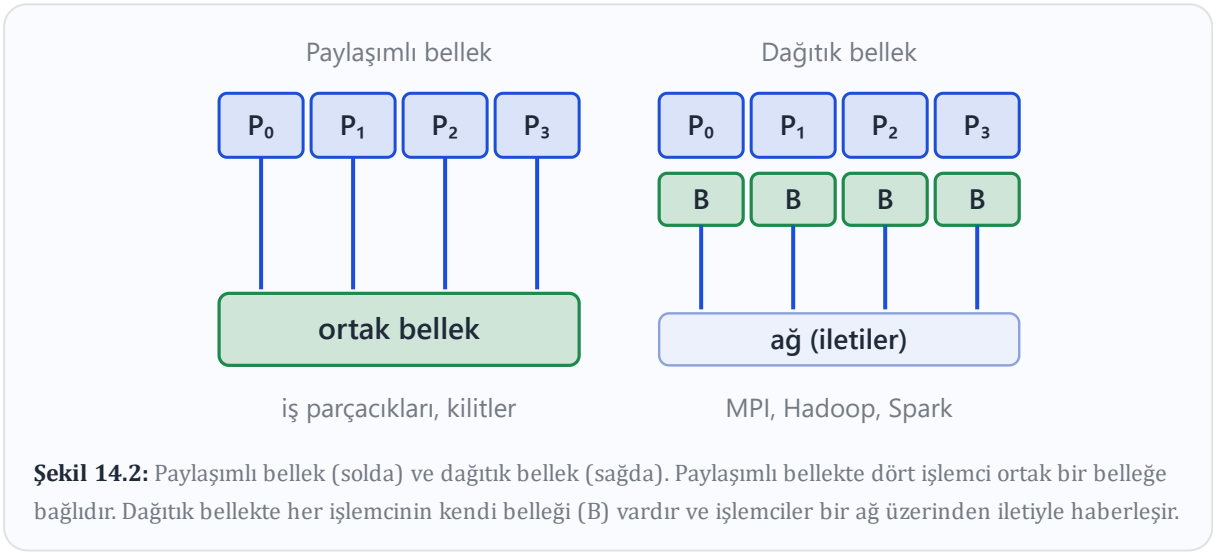
MISD uygulamada nadirdir. Aynı veri farklı komutlarla işlenir. Uçak denetim sistemleri gibi hata toleranslı sistemlerde aynı girdi birkaç ayrı birimde bağımsız olarak hesaplanır ve sonuçlar karşılaştırılır. Bu düzen MISD sınıfına örnek gösterilir.

MIMD en genel sınıftır. Her işlemci kendi programını kendi verisi üzerinde, ötekilerden bağımsız çalıştırır. Çok çekirdekli kişisel bilgisayarlar ve birçok bilgisayarın ağla bağlandığı kümeler MIMD makinelerdir.

Flynn sınıflandırması iki paralellik biçimini de ayırt etmeye yarar. **Veri paralelliğinde** aynı işlem verinin farklı parçalarına aynı anda uygulanır; bu SIMD tarzıdır. **Görev paralelliğinde** ise farklı görevler aynı anda yürütülür; bu MIMD tarzıdır. Bir web sunucusunun aynı anda farklı istemcilere farklı sayfalar hazırlaması görev paralelliğidir. Bir görüntünün bütün piksellerinin parlaklığının aynı oranda artırılması veri paralelliğidir.

14.2.2 Bellek mimarileri

MIMD makineler belleği düzenleme biçimlerine göre ikiye ayrılır. Şekil 14.2 iki düzeni yan yana gösterir.



Şekil 14.2: Paylaşımlı bellek (solda) ve dağıtık bellek (sağda). Paylaşımlı bellekte dört işlemci ortak bir belleğe bağlıdır. Dağıtık bellekte her işlemcinin kendi belleği (B) vardır ve işlemciler bir ağ üzerinden iletiyle haberleşir.

Paylaşımlı bellek (*shared memory*) düzeninde bütün işlemciler aynı adres uzayını görür. Bir işlemcinin belleğe yazdığı değeri öteki işlemci doğrudan okuyabilir. İletişim bu yüzden ucuzdur: veri taşımak yerine ortak bir değişken kullanmak yeter. Bedeli, eşzamanlı erişimin denetlenmesidir. İki işlemci aynı değişkeni aynı anda değiştirmeye çalışırsa sonuç bozulabilir; bu yüzden erişimler bir senkronizasyon düzeneğiyle sıraya konmalıdır. Java'nın iş parçacıkları ve `ForkJoinPool` sınıfı, C ve Fortran için OpenMP bu düzene yönelik araçlardır. Tek bir bilgisayarın çekirdekleri paylaşımlı bellekle çalışır.

Dağıtık bellek (*distributed memory*) düzeninde her düğümün kendi belleği vardır. Bir düğüm başka bir düğümün belleğine doğrudan erişemez. Veri gerekiyorsa sahibi onu bir **ileti** (*message passing*) olarak ağ üzerinden gönderir. Bu düzen çok iyi ölçeklenir: binlerce düğüm aynı problemi çözebilir. Ama iletişim pahalıdır. Ağ üzerinden bir ileti göndermek, yerel bellekten okumaktan yüzlerce kat

uzun sürebilir. Bu düzen için yazılan programlar ileti sayısını en aza indirmeye çalışır. Bilimsel hesaplamada yaygın kullanılan MPI (*Message Passing Interface*) kütüphanesi ve Hadoop MapReduce dağıtık bellek araçlarıdır.

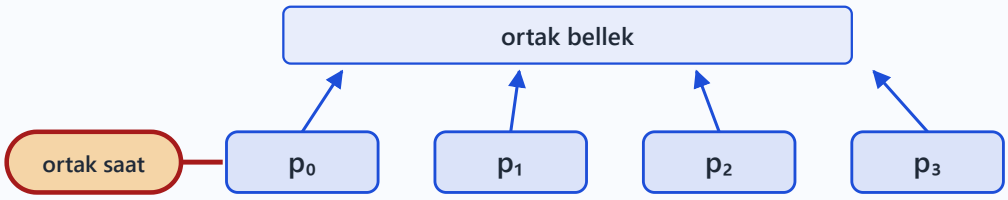
Bu notta Java örnekleri paylaşımlı bellekte çalışır. Boru hattı ve boru hatlı ağaç örnekleri ise işlemcilerin yalnız komşularıyla ileti alışverişi yaptığı dağıtık bir düzeni varsayar. MapReduce de aslen dağıtık bellek için tasarlanmıştır; buradaki Java örneği aynı fikri tek makinede gösterir.

14.2.3 PRAM modeli

Sıralı algoritmaları çözümlerken gerçek bir bilgisayar yerine basitleştirilmiş bir model kullanılır: RAM (*Random Access Machine*) modeli. Bu modelde her temel işlem ve her bellek erişimi bir birim zaman sürer. Önbellek, sanal bellek ve boru hattı gibi ayrıntılar yok sayılır. Model gerçeği tam yansıtmaz ama algoritmaları karşılaştırmak için yeterince doğrudur. Paralel algoritmalar için de benzer bir model gerekir.

Tanım 14.2 (PRAM modeli)

PRAM (*Parallel Random Access Machine*), RAM modelinin paralel karşılığıdır. p işlemci, ortak bir bellek ve ortak bir saatten oluşur. İşlemciler adımlarını eş zamanlı atar (*lockstep*). Her adımda her işlemci bellekten okur, yerel bir hesap yapar ve belleğe yazar. Bir adım $O(1)$ zaman sürer. İletişim ve senkronizasyonun maliyeti yok sayılır.



Şekil 14.3: PRAM modeli. Dört işlemci (p_0 … p_3) ortak belleğe bağlıdır ve ortak saatle eş zamanlı adım atar.

Modelin en güçlü varsayımı, iletişimin bedava olmasıdır. Gerçek bir makinede iki işlemcinin bir değeri paylaşması zaman alır; PRAM'da bir işlemcinin yazdığı değer bir sonraki adımda bütün işlemcilerce okunabilir. İşlemcilerin ortak saatle çalışması da senkronizasyonu bedava kılar. Bir tur bittiğinde bütün işlemcilerin o turu bitirdiği kesindir. Bu varsayımlar gerçekçi değildir, ama bir amaca hizmet eder. Model, iletişim ve donanım ayrıntılarını kenara bırakarak algoritmanın **içsel paralelliğini** ölçer. Bir algoritma PRAM'da bile paralel çalışamıyorsa gerçek bir makinede de paralel çalışamaz. PRAM'da hızlı çalışan bir algoritma ise gerçek makinede iletişim maliyeti ölçüsünde yavaşlar.

Ortak bellekte bir sorun daha vardır: birden çok işlemci aynı adımda aynı bellek hücresine erişirse ne olur? PRAM'ın türleri bu soruya verilen yanıta göre adlandırılır. Tablo 14.2 üç yaygın türü gösterir.

Tür	Okuma	Yazma
EREW	dışlayıcı	dışlayıcı
CREW	eşzamanlı	dışlayıcı
CRCW	eşzamanlı	eşzamanlı

Tablo 14.2: PRAM türleri. E = *Exclusive* (dışlayıcı), C = *Concurrent* (eşzamanlı), R = okuma, W = yazma.

Dışlayıcı erişim, aynı hücreye bir adımda yalnız tek işlemcinin erişebileceği anlamına gelir. **Eşzamanlı** erişimde birden çok işlemci aynı hücreye aynı adımda erişebilir. EREW en kısıtlı, CRCW en güçlü türdür. Eşzamanlı okuma zararsızdır: herkes aynı değeri okur. Eşzamanlı yazma ise bir kural gerektirir. Üç işlemci aynı hücreye 5, 7 ve 9 yazmaya çalışırsa hücrede hangi değer kalır? Yaygın kurallar şunlardır:

- **Ortak değer kuralı:** Yazma ancak bütün işlemciler aynı değeri yazıyorsa geçerlidir.
- **Rastgele kural:** İşlemcilerden biri rastgele seçilir ve onun değeri yazılır.
- **Öncelik kuralı:** En küçük numaralı (en yüksek öncelikli) işlemcinin değeri yazılır.

Bir algoritma EREW'de çalışıyorsa CREW ve CRCW'de de çalışır, çünkü güçlü model zayıf modelin her işlemine izin verir. Tersi her zaman doğru değildir. Bu yüzden bir algoritmanın hangi modelde çalıştığı belirtilirken olabildiğince zayıf model seçilir. Bu notta incelenen ağaç toplamı ve önek toplamı algoritmaları EREW ile yetinir.

Örnek 14.1

Bir PRAM algoritmasında n işlemcinin her biri, ortak bellekteki tek bir `esik` değerini okuyup kendi öğesiyle karşılaştırmak istiyor. Bu işlem hangi modelde tek adımda yapılır?

Bütün işlemciler aynı adımda aynı hücreyi okur; bu eşzamanlı okumadır. Yazma yoktur. CREW yeterlidir ve işlem bir adımda biter. EREW'de ise aynı hücreyi bir adımda yalnız bir işlemci okuyabilir. Değer önce kopyalanarak çoğaltılmalıdır. İlk adımda bir işlemci kopyalar ve 2 kopya olur; ikinci adımda 2 işlemci kopyalar ve 4 kopya olur. Kopya sayısı her adımda iki katına çıkar. n kopyaya $\lceil \log_2 n \rceil$ adımda ulaşılır. Bu örnek, model seçiminin süreyi değiştirebileceğini gösterir.

14.3 Başarım ölçütleri

Paralel bir algoritmanın iyi olup olmadığı tek bir sayıyla söylenemez. Algoritma hızlı olabilir ama bunun için çok fazla işlemci harcıyor olabilir. Çok işlemciyle hızlı olabilir ama işlemci sayısı arttıkça kazancı duraksayabilir. Bu kesim, paralel algoritmaları karşılaştırmak için kullanılan ölçütleri tanıtır.

14.3.1 Hızlanma, verimlilik, maliyet

Ölçütlerin hepsi iki süreye dayanır. T_1, problemi tek işlemcide çözen *en iyi sıralı algoritmanın* süresidir. T_p ise paralel algoritmanın p işlemcideki süresidir. Tablo 14.3 bu iki süreden türetilen üç ölçütü verir.

Ölçüt	Tanım
T_1	en iyi sıralı algoritmanın süresi
T_p	p işlemcideki paralel süre
Hızlanma (*speedup*)	$S = T_1 / T_p$ (ideal: $S = p$)
Verimlilik (*efficiency*)	$E = S / p$ ($0 < E \le 1$)
Maliyet (*cost*)	$C = p \cdot T_p$

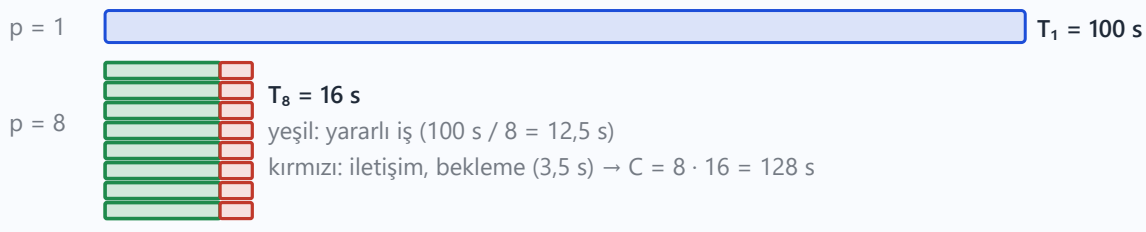
Tablo 14.3: Paralel başarım ölçütleri.

Tanım 14.3 (Hızlanma, verimlilik, maliyet)

Hızlanma, sıralı sürenin paralel süreye oranıdır: $S = T_1 / T_p$. **Verimlilik**, işlemci başına düşen hızlanmadır: $E = S / p$. **Maliyet**, bütün işlemcilerin harcadığı toplam süredir: $C = p \cdot T_p$. Maliyeti en iyi sıralı süreyle aynı mertebede olan, yani $C = \Theta(T_1)$ sağlayan paralel algoritmaya **maliyet-optimal** denir.

Hızlanma, işlemci eklemenin işi kaç kat kısalttığını söyler. p işlemciyle en iyi durumda p kat hızlanma beklenir; buna **doğrusal hızlanma** denir. Verimlilik, işlemcilerin zamanının ne kadarının yararlı işe gittiğini gösterir. $E = 1$ ise hiçbir işlemci boşa vakit harcamamıştır. $E = 0{,}5$ ise işlemci zamanının yarısı bekleme, iletişim ya da gereksiz işle geçmiştir.

Maliyet, paralel algoritmanın toplam olarak ne kadar "işlemci-saniye" tükettiğini ölçer. Sıralı algoritma T_1 işlemci-saniye harcar. Paralel algoritma ondan daha az harcayamaz; çünkü öyle olsaydı paralel algoritmanın adımları tek işlemcide sırayla yürütülerek T_1'den kısa süren bir sıralı algoritma elde edilirdi. Bu, T_1'in en iyi sıralı süre olmasıyla çelişir. Demek ki her zaman $C \ge T_1$, dolayısıyla $S \le p$ ve $E \le 1$'dir. Maliyet-optimal bir algoritma, sıralı algoritmanın yaptığı işi en fazla sabit bir çarpanla artırarak işlemcilere dağıtır.



Şekil 14.4: Tek işlemcide 100 s süren iş sekiz işlemcide 16 s sürüyor. Her işlemcinin 12,5 s'si yararlı iş (yeşil), 3,5 s'si iletişim ve bekleme (kırmızı). Maliyet $8 \cdot 16 = 128$ s.

Örnek 14.2

Bir iş en iyi sıralı algoritmayla $T_1 = 100$ s sürüyor. Paralel algoritma $p = 8$ işlemcide $T_8 = 16$ s sürüyor. Üç ölçüt şöyle hesaplanır:

1. Hızlanma: $S = 100 / 16 = 6{,}25$. İş 6,25 kat kısaldı; ideal değer 8 olurdu.
2. Verimlilik: $E = 6{,}25 / 8 \approx 0{,}78$. İşlemci zamanının yaklaşık %78'i yararlı işe gitti.
3. Maliyet: $C = 8 \cdot 16 = 128$ s. Sıralı algoritma 100 s harcıyordu; 28 işlemci-saniye ek yüke gitti.

Şekil 14.4 bu ek yükün kaynağını gösterir. 100 s'lik iş sekize bölününce her işlemciye 12,5 s düşer. Her işlemci 16 s çalıştığına göre 3,5 s'sini iletişime, beklemeye ya da birleştirmeye harcamıştır. Sekiz işlemcinin toplam kaybı $8 \cdot 3{,}5 = 28$ s'dir; maliyetle sıralı süre arasındaki fark da tam bu kadardır.

Hızlanmanın p'nin altında kalmasının dört yaygın nedeni vardır. Birincisi **iletişimdir**: işlemciler veri alışverişi yaparken hesap yapmaz. İkincisi **beklemedir**: bir işlemci başka bir işlemcinin sonucunu beklerken boştadır. Üçüncüsü **yük dengesizliğidir**: işi erken bitiren işlemciler en yavaşını bekler. Dördüncüsü **sıralı kısımlardır**: programın paralelleştirilemeyen bölümü, işlemci sayısından bağımsız olarak hep aynı süreyi alır. Son nedenin etkisi Amdahl yasasıyla hesaplanır.

Sık yapılan hata

Hızlanmayı paralel algoritmanın tek işlemcideki süresine göre hesaplamak. Paralel algoritma çoğu zaman sıralı algoritmadan fazla iş yapar. Örneğin Hillis–Steele önek toplamı tek işlemcide çalıştırılırsa sıralı taramadan yavaştır. Hızlanma bu yavaş süreye bölünerek hesaplanırsa olduğundan iyi görünür. T_1, her zaman en iyi sıralı algoritmanın süresidir.

14.3.2 Amdahl yasası ve Gustafson yasası

Gerçek programların neredeyse hepsinde paralelleştirilemeyen bir bölüm vardır. Girdiyi diskten okumak, ayarları yüklemek, kısmi sonuçları tek bir çıktıda birleştirmek bu türden işlerdir. Gene Amdahl 1967'de bu sıralı bölümün hızlanmaya getirdiği sınırı basit bir hesapla gösterdi.

Programın tek işlemcide süresi T_1 olsun. Bu sürenin f kesri paralelleştirilebilir, $1 - f$ kesri sıralıdır. p işlemcide sıralı kısım yine $(1 - f)\,T_1$ sürer. Paralel kısım ise en iyi durumda p'ye tam bölünür ve $f\,T_1 / p$ sürer. Toplam paralel süre

$$T_p = (1 - f)\,T_1 + f\,T_1 / p$$

olur. Hızlanma T_1 / T_p olduğundan T_1 sadeleşir ve aşağıdaki sonuç elde edilir.

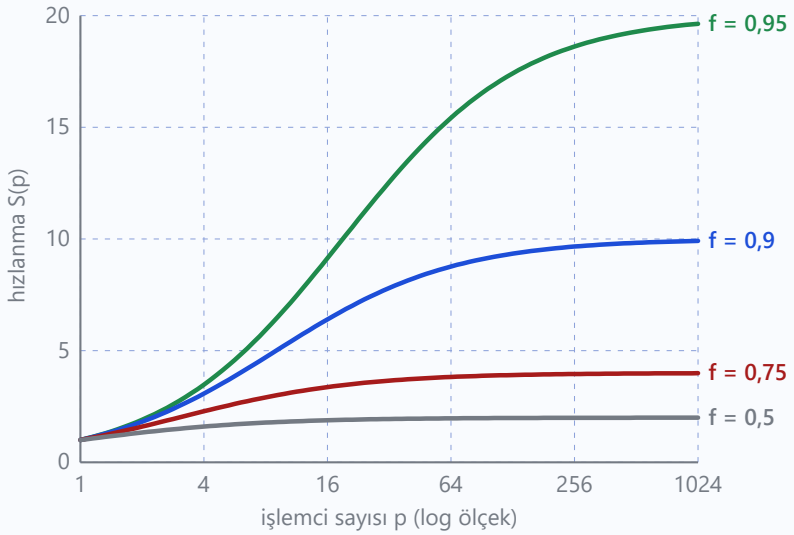
Teorem 14.1 (Amdahl yasası)

Programın f kesri paralelleştirilebiliyor ve $1 - f$ kesri sıralıysa p işlemcide elde edilebilecek hızlanma en fazla

$$S(p) = 1 / ((1 - f) + f / p)$$

olur. $p \to \infty$ iken $S(p) \to 1 / (1 - f)$'dir.

Yasanın asıl mesajı limittedir. İşlemci sayısı ne kadar artarsa artsın f / p terimi sıfıra yaklaşır, ama $1 - f$ terimi yerinde kalır. Sıralı kısım hızlanmayı sınırlar. Programın %10'u sıralıysa ($f = 0{,}9$) sonsuz sayıda işlemciyle bile hızlanma 10'u geçemez. %5'i sıralıysa sınır 20'dir.



Şekil 14.5: Amdahl yasasına göre hızlanmanın işlemci sayısıyla değişimi (yatay eksen logaritmik). $f = 0{,}95$, $0{,}9$, $0{,}75$ ve $0{,}5$ eğrileri sırasıyla 20, 10, 4 ve 2 değerlerinde doyar.

Örnek 14.3

Programın %90'ı paralelleştirilebiliyor: $f = 0{,}9$. Sekiz işlemcide hızlanma nedir?

1. Sıralı kısım: $1 - f = 0{,}1$.
2. Paralel kısmın payı: $f / p = 0{,}9 / 8 = 0{,}1125$.
3. Hızlanma: $S = 1 / (0{,}1 + 0{,}1125) = 1 / 0{,}2125 \approx 4{,}71$.
4. Üst sınır: $1 / 0{,}1 = 10$.

Sekiz işlemci sekiz kat değil, yaklaşık 4,7 kat hızlanma sağlar. Verimlilik *4,71 / 8 ≈ 0,59*'dur. Programın yalnız %10'unun sıralı olması, sekiz işlemcide sürenin yaklaşık yarısını sıralı kısma bırakır: *0,1 / 0,2125 ≈ 0,47*. Paralel kısım hızlandıkça sıralı kısmın toplam süredeki payı büyür.

Şekil 14.5'teki eğriler bu etkiyi gösterir. $f = 0{,}9$ eğrisi 4 işlemcide yaklaşık 3,1'e, 16 işlemcide 6,4'e, 64 işlemcide 8,8'e ulaşır. 1024 işlemcide ise yalnız 9,9'dur. 64'ten 1024'e çıkmak işlemci sayısını 16 katına çıkarır ama hızlanmayı yalnız %13 artırır. Bir programı çok işlemciye taşımadan önce sıralı kısmı küçültmek bu yüzden daha kârlıdır. Sıralı kısmı %10'dan %1'e indirmek, 1000 işlemcide hızlanmayı yaklaşık 10'dan 91'e çıkarır.

Amdahl yasası bir varsayıma dayanır: problemin boyutu sabittir ve yalnız işlemci sayısı artar. Uygulamada çoğu zaman böyle olmaz. Daha çok işlemci bulan bir araştırmacı aynı problemi daha hızlı çözmek yerine daha büyük bir problem çözer. Hava tahmini daha ince bir ızgarayla yapılır, benzetim daha çok parçacıkla çalıştırılır. Bu durumda sıralı kısım çoğunlukla aynı kalır ama paralel kısım işlemci sayısıyla birlikte büyür.

John Gustafson 1988'de bu bakış açısını formüle döktü. Paralel çalıştırmanın süresi 1 birim olsun. Bunun $1 - f$ kesri sıralı, f kesri paraleldir. Aynı iş tek işlemcide yapılsaydı sıralı kısım yine $1 - f$ sürerdi. Paralel kısım ise p işlemcinin yaptığı işi tek başına yapmak zorunda kalır ve $f \cdot p$ sürer. Ölçeklenmiş hızlanma

$$S = (1 - f) + f \cdot p$$

olur. Buna **Gustafson yasası** denir. $f = 0{,}9$ ve $p = 8$ için $S = 0{,}1 + 7{,}2 = 7{,}3$ bulunur. Aynı değerlerle Amdahl yasası 4,71 vermişti. İki yasa çelişmez; farklı soruları yanıtlar. Amdahl yasası “aynı problemi kaç kat hızlı çözerim?” sorusunu, Gustafson yasası “aynı sürede kaç kat büyük problem çözerim?” sorusunu yanıtlar. Gustafson yasasına göre büyük problemlerde paralellik daha kârlıdır, çünkü sıralı kısmın toplam işteki payı küçülür.

Sık yapılan hata

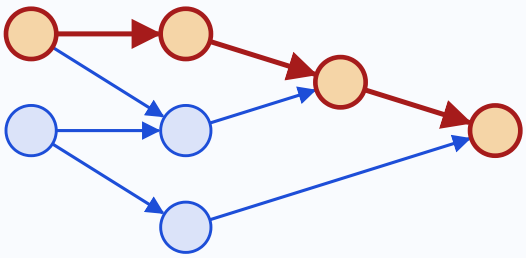
Amdahl hesabında sıralı kısmı unutmak. $f = 0{,}9$ ve $p = 8$ için “paralel kısım 8 kat hızlanır, öyleyse program da yaklaşık 8 kat hızlanır” denemez. Doğru hesapta sıralı %10 olduğu gibi kalır ve hızlanma 4,71'dir. Programın %10'u sıralıysa en iyi durumda bile hızlanma 10 ile sınırlıdır.

14.3.3 İş, derinlik ve Brent teoremi

Hızlanma ve verimlilik belirli bir işlemci sayısı için hesaplanır. Bir algoritmanın paralelliğini işlemci sayısından bağımsız olarak anlatmak için iki büyüklük kullanılır: iş ve derinlik. İkisi de algoritmanın işlemlerini bir **bağımlılık çizgesi** olarak görmeye dayanır. Çizgenin her düğümü bir temel işlemdir. $u \to v$ kenarı, v işleminin u'nun sonucunu kullandığını, yani u bitmeden başlayamayacağını gösterir. Döngü olamaz; çünkü bir işlem kendi sonucunu bekleyemez.

Tanım 14.4 (İş ve derinlik)

İş (*work*) W, algoritmanın yaptığı bütün işlemlerin sayısıdır; bağımlılık çizgesindeki düğüm sayısına eşittir ve tek işlemcideki süreyi verir: $W = T_1$. **Derinlik** (*depth* ya da *span*) D, çizgedeki en uzun bağımlılık zincirinin uzunluğudur; sınırsız işlemcideki süreyi verir: $D = T_\infty$. W / D oranına algoritmanın **paralelliği** denir.



Şekil 14.6: Yedi işlemli bir bağımlılık çizgesi. En uzun zincir dört düğümdür (kırmızı yol). $W = 7$, $D = 4$, paralellik $W / D = 1{,}75$.

Şekil 14.6'daki çizgede yedi işlem vardır; tek işlemci bunları yedi adımda bitirir. Sınırsız işlemci olsa bile kırmızı yol üzerindeki dört işlem birbirini bekler ve dört adımdan önce bitemez. Derinlik, işlemci sayısıyla aşılamayan bir sınırdır. Paralellik $7 / 4 = 1{,}75$'tir: ortalama olarak her adımda 1,75 işlem aynı anda yapılabilir. İkiden fazla işlemci bu çizgeye neredeyse hiç yarar sağlamaz.

Paralellik, **yararlı biçimde kullanılabilecek en fazla işlemci sayısı** olarak yorumlanır. İşlemci sayısı paralelliği aşarsa bazı işlemciler çoğu adımda boş kalır. İki temel alt sınır şöyledir:

- $T_p \geq W / p$: p işlemci bir adımda en çok p işlem yapar; W işlem en az W / p adım ister.
- $T_p \geq D$: en uzun zincirdeki işlemler birbirini bekler; işlemci sayısı bunu değiştirmez.

Bir üst sınır da vardır ve paralel algoritmaların çözümlemesinde çok kullanılır.

Teorem 14.2 (Brent teoremi)

İşi W, derinliği D olan bir hesap p işlemcide $T_p \leq W / p + D$ adımda yürütülebilir.

İspat. İşlemleri düzeylere ayıralım. Düzey i'de, en uzun öncül zincirinin uzunluğu $i - 1$ olan işlemler bulunsun. Aynı düzeydeki işlemler birbirini beklemez; çünkü aralarında kenar olsaydı biri ötekinden daha derin olurdu. Düzey sayısı D'dir. Düzey i'de W_i işlem varsa p işlemci bu düzeyi $\lceil W_i / p \rceil \leq W_i / p + 1$ adımda bitirir. Düzeyler sırayla yürütülür. Bütün düzeylerin toplamı $\Sigma\,(W_i / p + 1) = W / p + D$'dir. ■

Alt ve üst sınırlar birlikte okunduğunda $max(W / p, D) \le T_p \le W / p + D$ elde edilir. Üst sınır alt sınırın en çok iki katıdır. Bu yüzden W ve D bilinince herhangi bir işlemci sayısı için süre iki kat yakınlıkla kestirilebilir. p paralellikten çok küçükse W / p terimi baskındır ve hızlanma neredeyse doğrusaldır. p paralellikten büyükse D terimi baskındır ve yeni işlemciler süreyi kısaltmaz.

Bir paralel algoritmanın işi, aynı problemin en iyi sıralı algoritmasının süresiyle aynı mertebedeyse algoritmaya **iş-verimli** (*work-efficient*) denir. İş-verimli olmayan bir algoritma işlemci az olduğunda sıralı algoritmadan bile yavaş kalabilir. Önek toplamı kesimi bunun somut bir örneğini verir.

Örnek 14.4

$n = 1024$ sayı dengeli bir ikili ağaçla toplanıyor. İlk turda sayılar ikişer ikişer toplanır, ikinci turda bu toplamlar ikişer ikişer toplanır ve böyle sürer. İş, derinlik ve 32 işlemcideki süre nedir?

1. Her toplama iki değeri bire indirir. 1024 değeri bire indirmek için $W = n - 1 = 1023$ toplama gerekir.
2. Her turda değer sayısı yarıya iner: 1024, 512, …, 1. Tur sayısı $D = log_2 1024 = 10$'dur.
3. Paralellik: $1023 / 10 \approx 102$. Yaklaşık yüz işlemciye kadar ek işlemci yarar sağlar.
4. Brent: $T_{32} \le 1023 / 32 + 10 \approx 31{,}97 + 10 \approx 42$ adım.
5. Alt sınır: $T_{32} \ge max(1023 / 32, 10) \approx 32$ adım.

32 işlemciyle süre 32 ile 42 adım arasındadır. Sıralı algoritma 1023 adım sürer; hızlanma 24 ile 32 arasında kalır.

14.4 PRAM ile toplama

Paralel algoritmaların en basit ve en öğretici örneği n sayının toplamıdır. Problem çok basittir ama paralel çözüm, bu notun geri kalanında tekrar tekrar karşılaşılacak bir fikri içerir: birleşmeli bir işlemle kurulan zincir, dengeli bir ağaca çevrilebilir.

14.4.1 Problem ve birleşmeli işlem

Girdi n sayıdır; çıktı bu sayıların toplamıdır. Sıralı algoritma bir toplam değişkenini sıfırdan başlatır ve her sayıyı ona ekler. $n - 1$ toplama yapar ve $O(n)$ sürer. Bu sıralı döngünün bağımlılık çizgesi bir zincirdir: her toplama, bir önceki toplamanın sonucunu bekler. Zincirin uzunluğu $n - 1$'dir; derinlik işe eşittir ve paralellik yoktur.

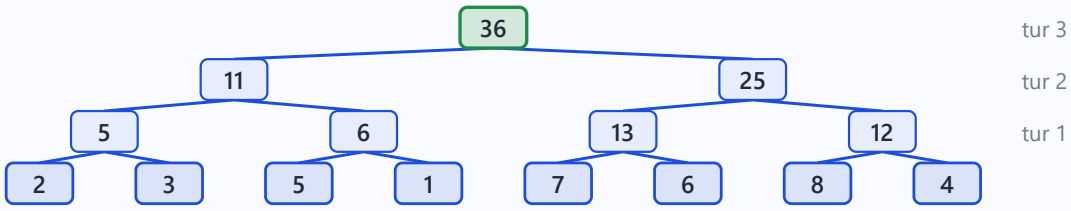
Zinciri kıran şey toplamanın **birleşmeli** (*associative*) olmasıdır: $(a + b) + c = a + (b + c)$. Birleşmeli bir işlemde parantezler istenildiği gibi yerleştirilebilir. Sekiz sayının toplamı sıralı döngüde

> *((((((2 + 3) + 5) + 1) + 7) + 6) + 8) + 4*

biçiminde hesaplanır. Aynı toplam şöyle de yazılabilir:

> *((2 + 3) + (5 + 1)) + ((7 + 6) + (8 + 4))*

İkinci yazılış dengeli bir ikili ağaçtır. En içteki dört toplama birbirinden bağımsızdır ve aynı anda yapılabilir. Onların sonuçlarını kullanan iki toplama da birbirinden bağımsızdır. En dıştaki toplama son olarak yapılır. Ağacın her düzeyindeki toplamlar aynı turda paralel yapılır. Şekil 14.7 bu ağacı gösterir.



Şekil 14.7: Dengeli toplama ağacı. 2 3 5 1 7 6 8 4 dizisi üç turda toplanır: 5 6 13 12 → 11 25 → 36.

Aynı fikir her birleşmeli işlem için çalışır: en büyük, en küçük, çarpım, mantıksal VE (AND), mantıksal VEYA (OR). Bir diziyi birleşmeli bir işlemle tek bir değere indiren bu genel işleme **indirgeme** (*reduction*) denir. Toplama ağacında toplama yerine `Math.max` kullanılırsa aynı turlarda en büyük öğe bulunur.

Sık yapılan hata

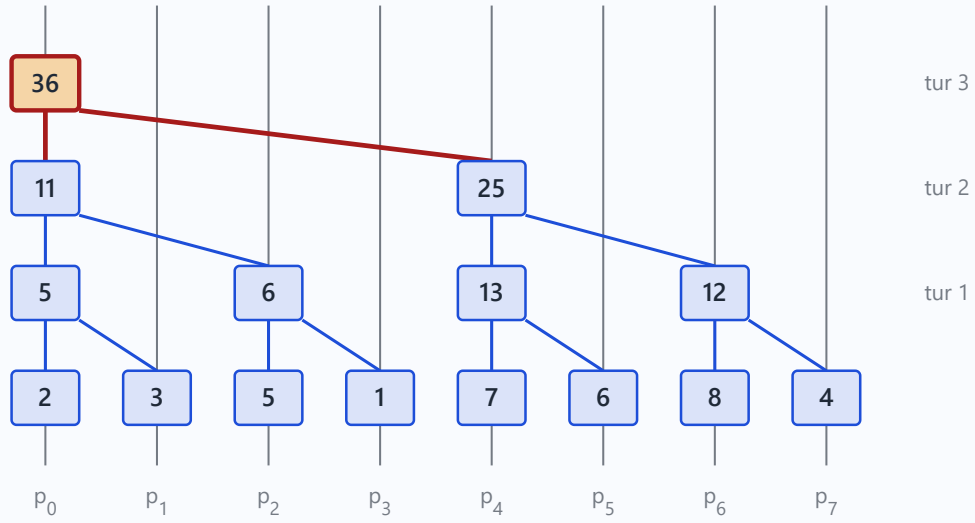
Birleşmeli olmayan bir işlemi ağaçla indirgemek. Çıkarma birleşmeli değildir: *(8 − 4) − 2 = 2* ama *8 − (4 − 2) = 6*. Ağaç sırası, sıralı sıradan farklı parantezleme kullandığı için sonuç değişir. Gerçel sayılarda toplama birleşmeli olsa da kayan noktalı toplama bilgisayarda yuvarlama yüzünden tam birleşmeli değildir. Paralel toplam sıralı toplamdan son basamaklarda farklı çıkabilir.

14.4.2 Ağaç toplamı adım adım

Örnek dizi 2, 3, 5, 1, 7, 6, 8, 4'tür; toplamı 36'dır. Sekiz işlemci p_0 … p_7 birer sayı tutar. Algoritma turlar hâlinde ilerler:

1. **Başlangıç.** Sekiz sayı sekiz işlemciye birer birer dağıtılır.
2. **Tur 1.** Dört işlemci aynı anda komşu çiftleri toplar: *2 + 3 = 5*, *5 + 1 = 6*, *7 + 6 = 13*, *8 + 4 = 12*. Sonuçlar çiftin solundaki işlemcide (p_0, p_2, p_4, p_6) kalır.
3. **Tur 2.** İki işlemci, aralarındaki uzaklık 2 olan kısmi toplamları birleştirir: p_0'da *5 + 6 = 11*, p_4'te *13 + 12 = 25*.

4. **Tur 3.** p_0, $11 + 25 = 36$ sonucunu bulur.



Şekil 14.8: PRAM toplamanın son durumu. Alt sırada sekiz işlemcinin başlangıç değerleri, üstteki sıralarda tur 1, 2 ve 3'ün kısmi toplamları vardır. Sonuç 36, p_0'da üç turda bulunur.

Sekiz sayı $\log_2 8 = 3$ turda toplandı. Sıralı algoritma 7 adım sürerdi. Toplama sayısı ise değişmedi: $4 + 2 + 1 = 7$. Paralel algoritma fazladan iş yapmadan yalnız sırayı değiştirdi. Bu yüzden ağaç toplamı iş-verimlidir.

Turlardan dikkat çeken bir şey daha görülür. Tur 1'de dört, tur 2'de iki, tur 3'te bir işlemci çalışır. Öteki işlemciler boşta bekler. Sekiz işlemcinin hiçbir turda hepsi çalışmaz; hatta ilk tur için bile dört işlemci yeterlidir. Bu gözlem, maliyet çözümlemesinde önemli olacaktır.

14.4.3 Kod izleme

Ağaç toplamı Java'da aşağıdaki gibi yazılır. Kod sıralı bir programdır; ama iç döngünün yinelemeleri aynı turda birbirinden bağımsızdır ve PRAM'da paralel yürütülür. Dış döngünün her yinelemesi bir turdur.

```
int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
int n = a.length;
for (int adim = 1; adim < n; adim *= 2) {
  // aynı turdaki tüm i'ler paralel
  for (int i = 0; i + adim < n; i += 2 * adim)
    a[i] = a[i] + a[i + adim];
}  // tur sonu: senkronizasyon
// toplam a[0]'da
```

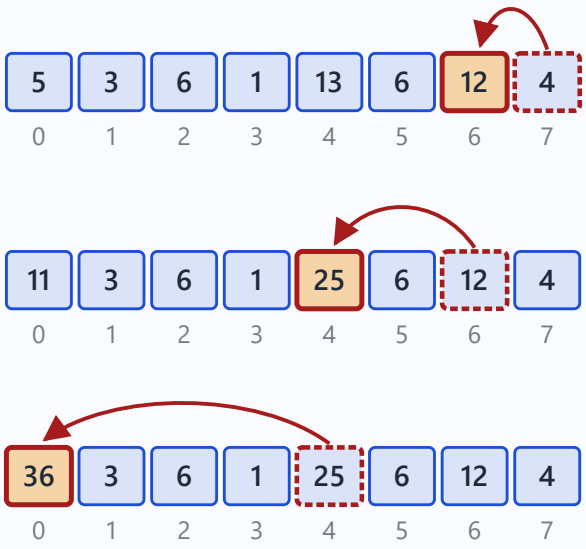
Kodun parçaları şöyle çalışır:

- `adim`, bir turda toplanan iki hücrenin arasındaki uzaklıktır. 1'den başlar ve her turda iki katına çıkar: 1, 2, 4. `adim` en az `n` olunca döngü biter.
- İç döngüde `i`, `2 * adim`'in katlarını gezer. Turda iş yapan hücreler bunlardır. Tur 1'de 0, 2, 4, 6; tur 2'de 0, 4; tur 3'te yalnız 0.
- `a[i] = a[i] + a[i + adim]`, sağdaki kısmi toplamı soldakine ekler. Her turdan sonra `a[i]`, `i`'den başlayan `2 * adim` uzunluğundaki bloğun toplamını tutar.
- `i + adim < n` koşulu, eşi olmayan hücrenin dizi dışını okumasını engeller. *n* ikinin kuvveti değilse bazı hücreler bir tur boyunca eşsiz kalır ve bir sonraki tura beklemeden geçer.
- Tur sonundaki yorum, senkronizasyonu hatırlatır. Bir sonraki tur, önceki turun bütün yazmaları bittikten sonra başlamalıdır.

Aynı turdaki yazmalar ayrı hücrelere yapılır. Okunan hücreler de turdan tura farklıdır: tur 1'de `a[1]`, `a[3]`, `a[5]`, `a[7]` okunur ve bunlara yazılmaz. Hiçbir hücreye aynı turda iki işlemci erişmez. Bu yüzden EREW PRAM yeterlidir. Tablo 14.4 kodun izlemesidir; değerler programın çalıştırılmasıyla doğrulanmıştır.

Tur	`adim`	Paralel toplamalar	Tur sonunda `a`
—	—	başlangıç	[2, 3, 5, 1, 7, 6, 8, 4]
1	1	`a[0]` = 2 + 3 = 5; `a[2]` = 5 + 1 = 6; `a[4]` = 7 + 6 = 13; `a[6]` = 8 + 4 = 12	[5, 3, 6, 1, 13, 6, 12, 4]
2	2	`a[0]` = 5 + 6 = 11; `a[4]` = 13 + 12 = 25	[11, 3, 6, 1, 25, 6, 12, 4]
3	4	`a[0]` = 11 + 25 = 36	[36, 3, 6, 1, 25, 6, 12, 4]
—	8	`adim` = 8, döngü biter	toplam `a[0]` = 36

Tablo 14.4: `a = {2, 3, 5, 1, 7, 6, 8, 4}` için ağaç toplamı izlemesi.



Şekil 14.9: Dizinin üç turun sonundaki durumu (çizim sırasıyla tur 1, 2, 3). Turda yazılan hücreler vurguludur. Son durumda toplam `a[0]`'dadır; öteki hücrelerde ara toplamlar kalır.

Son dizide `a[4] = 25` ve `a[2] = 6` gibi ara toplamlar durur. Algoritma diziyi bozar; yalnız `a[0]` anlamlıdır. Asıl dizi gerekiyorsa algoritma bir kopya üzerinde çalıştırılmalıdır.

Örnek 14.5

n ikinin kuvveti olmayınca kod nasıl çalışır? `a = {3, 1, 4, 1, 5}` (*n = 5*) için izleme şöyledir:

1. `adim` = 1: `i` = 0 için `a[0]` = 3 + 1 = 4; `i` = 2 için `a[2]` = 4 + 1 = 5. `i` = 4'te *4 + 1 < 5* sağlanmaz; `a[4]` = 5 eşsiz kalır. Dizi [4, 1, 5, 1, 5].
2. `adim` = 2: `i` = 0 için `a[0]` = 4 + 5 = 9. `i` = 4'te *4 + 2 < 5* sağlanmaz. Dizi [9, 1, 5, 1, 5].
3. `adim` = 4: `i` = 0 için *0 + 4 < 5* sağlanır; `a[0]` = 9 + 5 = 14.
4. `adim` = 8: döngü biter. `a[0]` = 14 = 3 + 1 + 4 + 1 + 5.

Tur sayısı $\lceil \log_2 5 \rceil = 3$, toplama sayısı $n - 1 = 4$'tür. Eşsiz kalan `a[4]`, iki tur bekledikten sonra son turda toplama katılır.

14.4.4 Analiz ve maliyet-optimal sürüm

Ağaç toplamının süresi tur sayısıdır: $\lceil \log_2 n \rceil$. İşi $n - 1$ toplamadır; sıralı algoritmayla aynıdır. İşlemci sayısı ilk turdaki toplama sayısı kadar, yani $n / 2$ olmalıdır. Çözümlemede kolaylık için çoğu zaman n işlemci yazılır; mertebe değişmez. Maliyet ise iyi değildir: $(n / 2) \cdot \log_2 n = \Theta(n \log n)$. En iyi sıralı süre $\Theta(n)$ olduğuna göre algoritma maliyet-optimal değildir.

Maliyetin yüksek olmasının nedeni turlarda görülmüştü. İşlemcilerin yarısı tur 1'den sonra, dörtte üçü tur 2'den sonra boştur. Boş bekleyen işlemciler de maliyete sayılır; çünkü maliyet işlemci sayısını sürenin tamamıyla çarpar.

Çözüm, işlemci sayısını azaltıp her işlemciye daha çok iş vermektir. $p = n / \log_2 n$ işlemci kullanılsın. Algoritma iki evreden oluşur:

1. **Yerel toplam.** Dizi p bloğa bölünür; her blokta $\log_2 n$ sayı vardır. Her işlemci kendi bloğunu sıralı olarak toplar. Bu evre $O(\log n)$ sürer.
2. **Ağaçla birleştirme.** $p = n / \log_2 n$ kısmi toplam, ağaç toplamıyla birleştirilir. Bu evre $\lceil \log_2 p \rceil \le \log_2 n$ tur, yani $O(\log n)$ sürer.

Toplam süre yine $O(\log n)$'dir. Maliyet ise $(n / \log n) \cdot O(\log n) = \Theta(n)$ olur. Algoritma artık maliyet-optimaldir. Tablo 14.5 iki sürümü karşılaştırır.

Ölçüt	n işlemci ($n / 2$ yeterli)	$p = n / \log_2 n$ işlemci
Süre T_p	$\log_2 n$	$O(\log n)$
İş W	$n - 1$	$n - 1$
Maliyet $p \cdot T_p$	$\Theta(n \log n)$	$\Theta(n)$

Ölçüt	n işlemci ($n / 2$ yeterli)	$p = n / \log_2 n$ işlemci
Maliyet-optimal mi?	hayır	evet

Tablo 14.5: PRAM ağaç toplamının iki sürümü.

Bu teknik, Brent teoreminin somut bir uygulamasıdır. Ağaç toplamında $W = n - 1$ ve $D = \log_2 n$'dir. $p = n / \log_2 n$ için Brent üst sınırı $W / p + D \approx \log_2 n + \log_2 n = 2 \log_2 n$ verir. Süre yalnız iki katına çıkar, işlemci sayısı ise $\log_2 n$ kat azalır. Örneğin $n = 1024$ için işlemci sayısı 1024'ten yaklaşık 102'ye iner.

Algoritmanın doğru çalışması için her turun sonunda bir **bariyer** (*barrier*) gerekir. Bariyer, bütün işlemciler ona ulaşmadan hiçbirinin geçemediği bir eşgüdüm noktasıdır. PRAM ortak saatle çalıştığı için bariyeri bedava sağlar. Gerçek bir makinede ise her bariyerin bir bedeli vardır; bu yüzden tur sayısının az olması gerçek makinede de önemlidir.

Sık yapılan hata

Turlar arasındaki senkronizasyonu unutmak. Gerçek iş parçacıklarıyla yazılmış bir ağaç toplamında tur 2'yi yapan iş parçacığı, tur 1'in bütün yazmaları bitmeden `a[2]` 'yi okursa eski değeri (5) okur. `a[0]` = 5 + 6 yerine 5 + 5 = 10 hesaplanır ve sonuç yanlış çıkar. Hata zamanlamaya bağlı olduğu için her çalıştırmada görülmeyebilir.

14.5 Paralel önek toplamı

Ağaç toplamı yalnız tek bir sonuç üretir: dizinin toplamını. Birçok problemde ise her konum için ayrı bir toplam gerekir. Bir banka hesabının her işlemden sonraki bakiyesi, bir histogramın birikimli değerleri, saymalı sıralamada her öğenin yerleşeceği konum bu türdendir. Bu kesim, bütün bu değerleri aynı anda hesaplayan iki paralel algoritmayı anlatır.

14.5.1 Problem

Tanım 14.5 (Önek toplamı)

$x_0, x_1, \ldots, x_{n-1}$ dizisinin **kapsayıcı önek toplamı** (*inclusive prefix sum* ya da *scan*), her i için $y_i = x_0 + x_1 + \ldots + x_i$ değerlerinden oluşan dizidir. **Dışlayıcı önek toplamında** i. konuma x_i katılmaz: $y_0 = 0$ ve $y_i = x_0 + \ldots + x_{i-1}$.

2 3 5 1 7 6 8 4 dizisinin kapsayıcı önek toplamı 2 5 10 11 18 24 32 36'dır. Dışlayıcı önek toplamı ise bir konum sağa kaymış hâlidir: 0 2 5 10 11 18 24 32. Biri ötekinden kolayca elde edilir. Kapsayıcıdan dışlayıcıya geçmek için her konumdan kendi öğesi çıkarılır; ters yönde her konuma kendi öğesi eklenir.

Sıralı algoritma tek bir döngüdür: $y_i = y_{i-1} + x_i$. $n - 1$ toplama yapar ve $O(n)$ sürer. Ama tamamen zincirlemedir: her adım bir öncekinin sonucunu bekler. İlk bakışta paralel bir çözüm olanaksız görünür, çünkü y_7, y_6'ya; y_6 de y_5'e bağlıdır.

Ağaç toplamı bu zinciri kırmanın yolunu göstermişti. Ağaç toplamında da her ara toplam bir bloğun toplamıdır. Fark şudur: ağaç toplamı yalnız son hücrede, yani bütün dizinin toplamında durur. Önek toplamında ise her hücrenin kendi önekini öğrenmesi gerekir. Ağaçtaki ara toplamlar bu bilginin parçalarıdır; geriye onları doğru biçimde dağıtmak kalır.

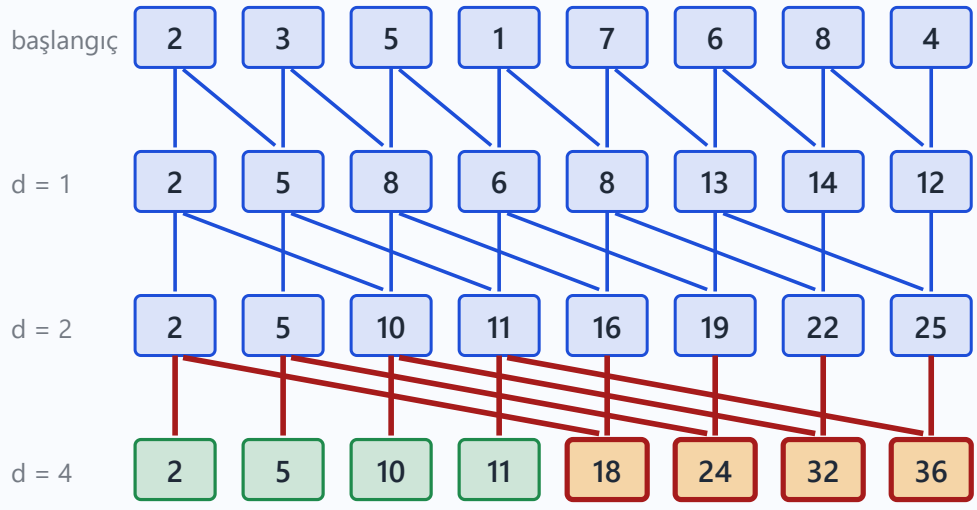
14.5.2 Hillis–Steele algoritması

W. Daniel Hillis ve Guy Steele'in 1986'da yayımladığı algoritma çok basit bir kurala dayanır. $d = 1, 2, 4, \ldots$ değerleri için her turda, $i \geq d$ olan her konum kendi değerine d konum soldaki değeri ekler:

> $x[i] \leftarrow x[i] + x[i - d]$ (her $i \geq d$ için, aynı turda paralel)

Turdaki bütün okumalar, o turun yazmalarından önceki değerleri görmelidir. PRAM'da bütün işlemciler önce okur, sonra yazar; bu koşul kendiliğinden sağlanır. Sıralı bir benzetimde ise her turda eski dizinin bir kopyası tutulur ve okumalar kopyadan yapılır.

Algoritmanın neden doğru olduğu bir değişmezle görülür. d turu bittiğinde her $x[i]$, i'de biten ve en çok $2d$ öğe içeren pencerenin toplamını tutar. Başta ($d = 1$ turundan önce) her hücre yalnız kendisini, yani 1 öğelik pencereyi tutar. d turunda $x[i]$'ye $x[i - d]$ eklenir. Turdan önce $x[i]$, $i - d + 1 \ldots i$ aralığını; $x[i - d]$ ise $i - 2d + 1 \ldots i - d$ aralığını tutar. İki aralık bitişiktir ve toplamları $i - 2d + 1 \ldots i$ aralığını verir. Pencere dizinin başına dayanınca kesilir. $\lceil \log_2 n \rceil$ turdan sonra pencere boyu n'yi geçer ve her hücre kendi önekinin tamamını tutar.



Şekil 14.10: Hillis–Steele önek toplamı. Üstte girdi, altta $d = 1, 2, 4$ turlarının sonuçları vardır. Her çizgi, bir hücrenin o turda okuduğu soldaki hücreyi gösterir. Son satır kapsayıcı önek toplamıdır: 2 5 10 11 18 24 32 36.

Tablo 14.6 örnek dizi için her turun sonucunu verir. Değerler bir Java programıyla, kopyalı benzetimle doğrulanmıştır.

Tur	İşlem	Toplama	Tur sonunda x	Pencere
başlangıç	—	—	2 3 5 1 7 6 8 4	1
$d = 1$	$i \ge 1$: $x[i] \mathrel{+}= x[i-1]$	7	2 5 8 6 8 13 14 12	2
$d = 2$	$i \ge 2$: $x[i] \mathrel{+}= x[i-2]$	6	2 5 10 11 16 19 22 25	4
$d = 4$	$i \ge 4$: $x[i] \mathrel{+}= x[i-4]$	4	2 5 10 11 18 24 32 36	8

Tablo 14.6: 2 3 5 1 7 6 8 4 için Hillis–Steele turları.

Tablonun bir satırını açıkça kontrol edelim. $d = 2$ turunun sonunda $x[5] = 19$'dur. Bu değer, 5'te biten 4 öğelik pencerenin toplamıdır: $x_2 + x_3 + x_4 + x_5 = 5 + 1 + 7 + 6 = 19$. Hesap şöyle yapılmıştır: turdan önce $x[5] = 13$ (6 + 7) ve $x[3] = 6$ (1 + 5); $13 + 6 = 19$.

Algoritma $\log_2 8 = 3$ turda bitti. Derinlik $D = \lceil \log_2 n \rceil$'dir. Ama toplama sayısı $7 + 6 + 4 = 17$'dir; sıralı algoritma 7 toplama yapar. Genel olarak d turunda $n - d$ toplama yapılır. n ikinin kuvvetiyse toplam iş

$$W = \sum_{d = 1, 2, 4, \ldots, n/2} (n - d) = n \log_2 n - (n - 1)$$

olur. $n = 8$ için $24 - 7 = 17$. İş $\Theta(n \log n)$'dir ve sıralı algoritmanın $\Theta(n)$ işinden fazladır. Hillis–Steele **işverimli değildir**. İşlemci sayısı çoksa bu önemsizdir; çünkü fazladan işlemler aynı turda bedavaya yapılır. İşlemci azsa fazladan iş doğrudan süreye eklenir. Örneğin tek işlemcide Hillis–Steele $n \log n$ sürer, sıralı tarama ise n sürer.

Sık yapılan hata

Sıralı benzetimde tur başındaki kopyayı unutmak. Hesap aynı dizi üzerinde ve i artan sırayla yapılırsa $x[i - d]$ bu turda zaten güncellenmiş olabilir. 2 3 5 1 dizisinde $d = 1$ turu kopyasız yapılırsa $x[1] = 5$, ardından $x[2] = 5 + 5 = 10$, $x[3] = 1 + 10 = 11$ olur. Tek turda tam önek toplamı çıkar; sonraki tur bu değerlere yeniden ekler ve sonuç bozulur: $d = 2$ turundan sonra 2 5 12 16 elde edilir. Doğrusu 2 5 10 11'dir. Gerçek paralel kodda bu hata, turlar arası bariyerin unutulmasına karşılık gelir.

14.5.3 Blelloch algoritması

Guy Blelloch'un 1990'da yaygınlaştırdığı algoritma önek toplamını $O(n)$ işle ve $O(\log n)$ derinlikle hesaplar. Diziyi örtük bir ikili ağaç gibi kullanır ve iki evrede çalışır. **Yukarı süpürme** (*up-sweep*) evresi, ağaç toplamının aynısıdır: yapraklardan köke doğru kısmi toplamlar kurulur. **Aşağı süpürme** (*down-sweep*) evresi kökten yapraklara doğru iner ve her düğüme “solundaki bütün öğelerin toplamını” taşır.

```
// yukarı süpürme; x = 2 3 5 1 7 6 8 4
for (int d = 1; d < n; d *= 2)
  for (int i = 2*d - 1; i < n; i += 2*d)
    x[i] += x[i - d];
x[n - 1] = 0;
// aşağı süpürme (iç döngüler paralel)
for (int d = n / 2; d >= 1; d /= 2)
  for (int i = 2*d - 1; i < n; i += 2*d) {
    int t = x[i - d];
    x[i - d] = x[i];
    x[i] += t;
  }
// x: dışlayıcı önek toplamları
```

Kodun parçaları şöyle çalışır:

- **Yukarı süpürme.** *d = 1, 2, 4* için *i = 2d − 1, 4d − 1, …* hücrelerinde `x[i] += x[i - d]` yapılır. Ağaç toplamından tek farkı, toplamların bloğun soluna değil sağına, yani son hücresine yazılmasıdır. Evre sonunda `x[n - 1]` bütün dizinin toplamını tutar.
- **Kök sıfırlanır.** `x[n - 1] = 0`. Kökün solunda hiçbir öğe yoktur; solundakilerin toplamı 0'dır.
- **Aşağı süpürme.** *d = n/2, …, 2, 1* için her *(i − d, i)* çiftinde üç satır çalışır. `t`, sol çocuğun eski değerini, yani sol alt ağacın toplamını saklar. Sol çocuk ebeveynin değerini alır: sol alt ağacın solundaki öğeler, ebeveynin solundakilerle aynıdır. Sağ çocuk ebeveynin değerine sol alt ağacın toplamını ekler: sağ alt ağacın solunda, ebeveynin solundakilere ek olarak sol alt ağaç da vardır.
- Bir turdaki çiftler birbirinden ayrık hücrelere dokunur; iç döngü paralel çalışır ve EREW yeterlidir.

Tablo 14.7 örnek dizi için iki evrenin bütün turlarını verir. Değerler kodun çalıştırılmasıyla doğrulanmıştır.

Evre	*d*	Paralel işlemler	Tur sonunda *x*
girdi	—	—	2 3 5 1 7 6 8 4
yukarı	1	*x[1] += x[0], x[3] += x[2], x[5] += x[4], x[7] += x[6]*	2 5 5 6 7 13 8 12
yukarı	2	*x[3] += x[1], x[7] += x[5]*	2 5 5 11 7 13 8 25
yukarı	4	*x[7] += x[3]*	2 5 5 11 7 13 8 36
kök	—	*x[7] = 0*	2 5 5 11 7 13 8 0
aşağı	4	çift (3, 7): *x[3] = 0, x[7] = 0 + 11*	2 5 5 0 7 13 8 11
aşağı	2	çift (1, 3): *x[1] = 0, x[3] = 0 + 5*; çift (5, 7): *x[5] = 11, x[7] = 11 + 13*	2 0 5 5 7 11 8 24
aşağı	1	çiftler (0, 1), (2, 3), (4, 5), (6, 7)	0 2 5 10 11 18 24 32

Tablo 14.7: 2 3 5 1 7 6 8 4 için Blelloch önek toplamı izlemesi.

Son turu tek tek açalım. Çift (0, 1): `t = x[0] = 2`; `x[0] = x[1] = 0`; `x[1] = 0 + 2 = 2`. Çift (2, 3): `t = 5`; `x[2] = 5`; `x[3] = 5 + 5 = 10`. Çift (4, 5): `t = 7`; `x[4] = 11`; `x[5] = 11 + 7 = 18`. Çift (6, 7): `t = 8`; `x[6] = 24`; `x[7] = 24 + 8 = 32`. Sonuç 0 2 5 10 11 18 24 32'dir; bu, dışlayıcı önek

toplamıdır. Kapsayıcı sonuç gerekiyorsa her hücreye girdideki kendi değeri eklenir ya da dizi bir konum sola kaydırılıp sona toplam (36) yazılır.



Şekil 14.11: Blelloch algoritmasında yukarı süpürmenin sonu (birinci çizim; kök *x[7] = 36*) ve aşağı süpürmenin sonu (ikinci çizim; dışlayıcı önek toplamı 0 2 5 10 11 18 24 32).

Algoritmanın maliyeti şöyle sayılır. Yukarı süpürme ağaç toplamıyla aynıdır: $n - 1$ toplama, $\log_2 n$ tur. Aşağı süpürmede her iç düğüm için bir işlem yapılır: yine $n - 1$ işlem, $\log_2 n$ tur. Toplam iş $2(n - 1)$'dir; örnekte 14 işlem. Derinlik $2 \log_2 n$'dir; örnekte 6 tur. İş $O(n)$ olduğu için Blelloch algoritması iş-verimlidir. Hillis–Steele'e göre iki kat tur harcar ama toplam işi $\log n$ kat azdır.

Örnek 14.6

Aşağı süpürmenin sonunda *x[5] = 18* çıktı. Bu değer nereden gelir?

1. Hücre 5, örtük ağaçta (4, 5) çiftinin sağ çocuğudur. Bu çiftin ebeveyni (5, 7) çiftindeki sol hücre 5'tir; onun ebeveyni de (3, 7) çiftindeki sağ hücre 7'dir.
2. Aşağı süpürmenin ilk turunda ($d = 4$) *x[7] = 11* olur: sağ yarının solundaki öğelerin toplamı *2 + 3 + 5 + 1 = 11*.
3. $d = 2$ turunda (5, 7) çiftinin sol çocuğu ebeveynin değerini alır: *x[5] = 11*. Hücreler 4 ve 5'in solunda yine ilk dört öğe vardır.
4. $d = 1$ turunda (4, 5) çiftinin sağ çocuğu *x[5] = 11 + t* olur. *t*, sol çocuğun yukarı süpürmeden kalan değeridir: *x[4] = 7*. Sonuç *11 + 7 = 18*.

18, gerçekten de hücre 5'in solundaki beş öğenin toplamıdır: *2 + 3 + 5 + 1 + 7 = 18*.

14.5.4 İki algoritmanın karşılaştırması

Algoritma	İş W	Derinlik D	Sonuç
Sıralı tarama	$n - 1$ (7)	$n - 1$ (7)	kapsayıcı

Algoritma	İş W	Derinlik D	Sonuç
Hillis–Steele	$n \log_2 n - (n-1)$ (17)	$\log_2 n$ (3)	kapsayıcı
Blelloch	$2(n-1)$ (14)	$2 \log_2 n$ (6)	dışlayıcı

Tablo 14.8: Önek toplamı algoritmaları ($n = 8$ değerleri parantez içinde).

Hangi algoritmanın seçileceği işlemci sayısına bağlıdır. İşlemci sayısı n düzeyindeyse, örneğin bir GPU'da bir blok içinde, Hillis–Steele'in az turu avantajdır. İşlemci sayısı n'den çok küçükse Blelloch'un az işi belirleyicidir. Brent teoremine göre p işlemcide Hillis–Steele $O(n \log n / p + \log n)$, Blelloch $O(n / p + \log n)$ sürer.

Önek toplamı paralel algoritmaların en çok kullanılan yapı taşlarından biridir. Paralel saymalı sıralamada her anahtarın çıktıdaki konumu önek toplamıyla bulunur. Bir diziden belirli bir koşulu sağlayan öğeleri seçip sıkıştırmak (*filter*) için her öğeye koşulu sağlıyorsa 1, sağlamıyorsa 0 yazılır. Bu 0/1 dizisinin dışlayıcı önek toplamı, seçilen her öğenin yeni dizideki konumunu verir. Java'da `Arrays.parallelPrefix(dizi, Integer::sum)` metodu bir diziyi kapsayıcı önek toplamıyla yerinde değiştirir. Örnek dizi için 2 5 10 11 18 24 32 36 sonucunu verir.

Sık yapılan hata

Yalnız derinliğe bakmak. Hillis–Steele $O(\log n)$ turda biter ve bu ilk bakışta Blelloch'tan iyi görünür. Ama $O(n \log n)$ iş yapar. Dört çekirdekli bir makinede bir milyon öğelik bir dizi için Hillis–Steele yaklaşık 20 milyon toplama yapar; çekirdek başına 5 milyon eder. Sıralı tarama ise tek çekirdekte 1 milyon toplama yapar. Az işlemcide iş-verimli olmayan algoritma sıralı algoritmadan yavaş kalabilir.

14.6 Boru hattı

Şimdiye kadarki algoritmalar işlemcilerin ortak belleği paylaştığını varsaydı. Bu kesimde işlemciler bir zincir biçiminde bağlıdır ve yalnız komşularıyla ileti alışverişi yapar. Böyle bir düzende paralelliğin önemli bir kaynağı boru hattıdır.

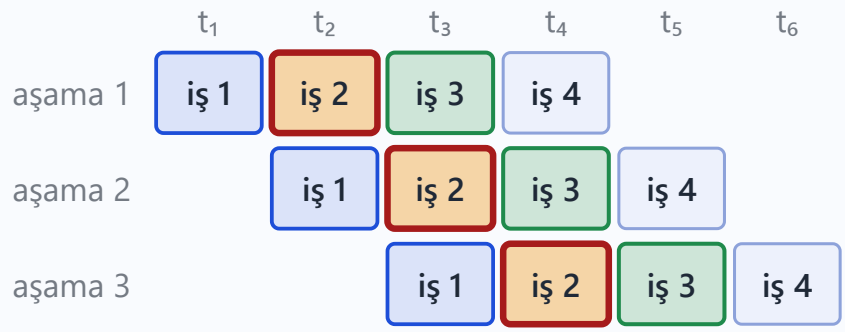
14.6.1 Boru hattı fikri

Tanım 14.6 (Boru hattı)

Boru hattı (*pipeline*), bir işin ardışık aşamalara bölündüğü ve her aşamanın ayrı bir işlemcide yürütüldüğü düzendir. Bir iş bir aşamadan ötekine geçerken önceki aşama bir sonraki işe başlar; böylece farklı işler farklı aşamalarda üst üste işlenir.

Fabrikadaki montaj hattı iyi bir benzetmedir. Bir otomobil gövdesi kaynak bölümünden boya bölümüne geçtiğinde kaynak bölümü bir sonraki gövdeye başlar. Tek bir otomobilin hattan çıkma süresi kısalmaz; her gövde bütün aşamalardan geçmek zorundadır. Ama hattan birim zamanda çıkan otomobil sayısı, yani **verim** (*throughput*), aşama sayısı kadar artar.

k aşamalı bir boru hattında m iş yapılsın; her aşama bir zaman adımı sürsün. Sıralı yürütmede her iş k adım sürer ve işler birbiri ardına yapılır: $k \cdot m$ adım. Boru hattında ilk iş k adımda biter. Sonraki her iş, öncekinden bir adım sonra biter. Toplam süre $k + m - 1$ adımdır.



Şekil 14.12: Üç aşamalı boru hattında dört iş. Her zaman adımında (t_1 … t_6) her iş bir aşama ilerler. İşler $3 \cdot 4 = 12$ yerine $3 + 4 - 1 = 6$ adımda biter.

Şekil 14.12 üç aşama ve dört iş için hattı gösterir. Başta hat dolarken (t_1 ve t_2) ve sonda boşalırken (t_5 ve t_6) bazı aşamalar boştur. Hat yalnız t_3 ve t_4 anlarında doludur; o anlarda üç aşamanın üçü de çalışır. İş sayısı aşama sayısına göre çok büyükse dolma ve boşalma süresi önemsizleşir ve süre m'ye yaklaşır.

14.6.2 Zincirde satır toplamları

Boru hattının paralel algoritmalardaki kullanımını somut bir problem gösterir. Beş işlemci p_0 … p_4 doğrusal bir zincir oluşturur. Her işlemci yalnız sağ komşusuna ileti gönderebilir. Her p_j, 4 değerlik bir sütun tutar; satırlar 0'dan 3'e numaralıdır. İşlemcilerin sütunları yan yana konunca 4 satırlık, 5 sütunluk bir tablo oluşur.

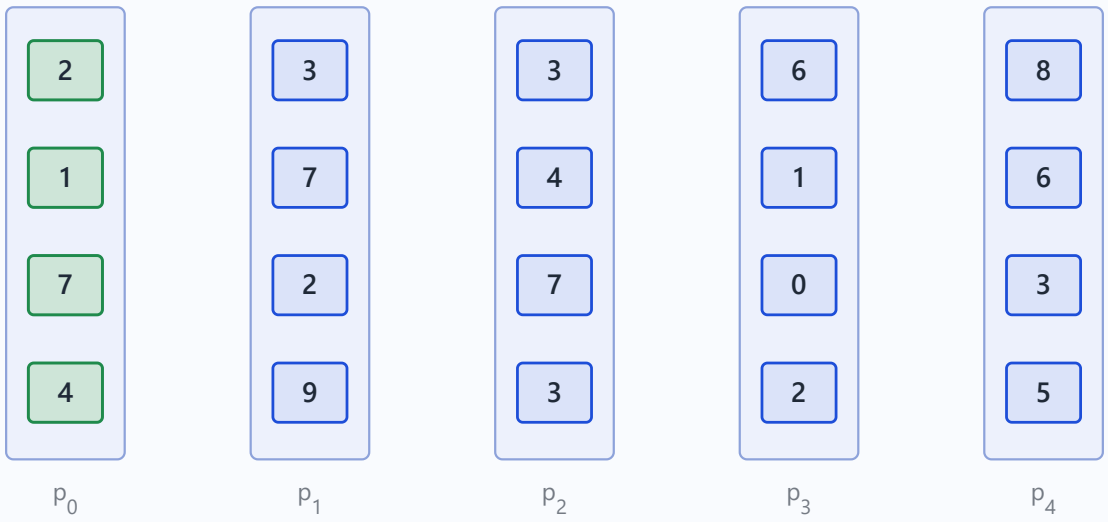
Amaç şudur: her satır r için p_j'deki değer, p_0 … p_j'nin o satırdaki değerlerinin toplamı olsun. Bu, her satırda ayrı bir önek toplamıdır. Sonunda p_4 her satırın toplamını tutar. Bir adımda her işlemci sağ komşusuna bir değer gönderebilir; alıcı onu kendi değerine ekler. p_0'ın değerleri zaten sonuçtur, çünkü solunda kimse yoktur.

Saf yöntem satırları birer birer işler. Önce satır 0 zincir boyunca taşınır: $p_0 \to p_1$, sonra $p_1 \to p_2$ ve böyle sürer. Bu, dört bağlantıdan geçtiği için 4 adım alır. Sonra sırayla satır 1, satır 2 ve satır 3 taşınır. Toplam $4 \cdot 4 = 16$ adım sürer. Her adımda dört bağlantıdan yalnız biri kullanılır.

Boru hattı yöntemi satırları zincirde arka arkaya akıtır. p_j, satır r'yi $t = r + j$ anında gönderir. Satır 0 yola çıktıktan bir adım sonra satır 1 yola çıkar; satır 0 o sırada bir bağlantı ileridedir. Satırlar dalga gibi ilerler ve bağlantıların çoğu aynı anda çalışır.

14.6.3 Adım adım çalıştırma

Örnekte işlemcilerin sütunları şöyledir: p_0 = (2, 1, 7, 4), p_1 = (3, 7, 2, 9), p_2 = (3, 4, 7, 3), p_3 = (6, 1, 0, 2), p_4 = (8, 6, 3, 5). Her adımda hangi bağlantının hangi satırı taşıdığı Tablo 14.9'da verilmiştir.

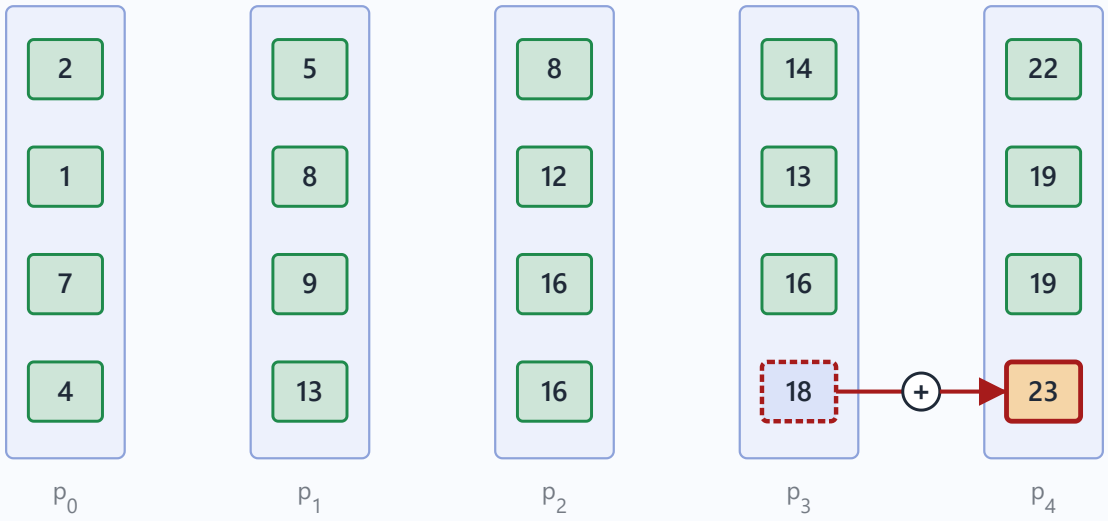


Şekil 14.13: Boru hattının başlangıcı. Beş işlemcinin her biri 4 satırlık bir sütun tutar. p_0'ın değerleri zaten sonuçtur.

Adım	$p_0 \to p_1$	$p_1 \to p_2$	$p_2 \to p_3$	$p_3 \to p_4$
1	satır 0: 2 + 3 = 5	—	—	—
2	satır 1: 1 + 7 = 8	satır 0: 5 + 3 = 8	—	—
3	satır 2: 7 + 2 = 9	satır 1: 8 + 4 = 12	satır 0: 8 + 6 = 14	—
4	satır 3: 4 + 9 = 13	satır 2: 9 + 7 = 16	satır 1: 12 + 1 = 13	satır 0: 14 + 8 = 22
5	—	satır 3: 13 + 3 = 16	satır 2: 16 + 0 = 16	satır 1: 13 + 6 = 19
6	—	—	satır 3: 16 + 2 = 18	satır 2: 16 + 3 = 19
7	—	—	—	satır 3: 18 + 5 = 23

Tablo 14.9: Boru hattının yedi adımı. Her hücre, o adımda o bağlantının taşıdığı satırı ve yapılan toplamayı verir.

Tablo çapraz bir desen gösterir. Satır 0 birinci adımda yola çıkar ve dördüncü adımda p_4'e ulaşır. Satır 3 dördüncü adımda yola çıkar ve yedinci adımda ulaşır. Dördüncü adımda hat doludur: dört bağlantının dördü de çalışır. Öncesinde hat dolar, sonrasında boşalır.



Şekil 14.14: Yedinci adım: p_3 satır 3'ü gönderir (18 + 5 = 23). Hat boşalmıştır; p_4 satır toplamlarını tutar: 22, 19, 19, 23.

Sonucu bir satır için doğrulayalım. Satır 2'nin değerleri soldan sağa 7, 2, 7, 0, 3'tür. Toplamları *7 + 2 + 7 + 0 + 3 = 19*'dur. Boru hattında bu satır p_0'dan 3. adımda çıkar (9), 4. adımda p_2'ye (16), 5. adımda p_3'e (16) ve 6. adımda p_4'e (19) ulaşır. Ara değerler de doğrudur: p_2'deki 16, satırın ilk üç değerinin toplamıdır.

14.6.4 Kod izleme ve analiz

Boru hattının zamanlaması kısa bir döngüyle yazılır. `x[j][r]`, p_j'nin satır r'deki değeridir. Dış döngünün her yinelemesi bir zaman adımıdır; iç döngünün yinelemeleri aynı adımda paralel çalışır.

```
for (int t = 0; t < m + p - 2; t++)
  // tüm j'ler aynı anda
  for (int j = 0; j < p - 1; j++) {
    int r = t - j;
    if (r >= 0 && r < m)
      x[j + 1][r] += x[j][r];
  }
```

- `t`, sıfırdan başlayan zaman adımıdır. Döngü $m + p - 2$ kez döner; örnekte $4 + 5 - 2 = 7$.
- `j`, $p_j \to p_{j+1}$ bağlantısını gösterir; $p - 1$ bağlantı vardır.
- `r = t - j`, bu bağlantının `t` anında taşıdığı satırdır. p_j satır r'yi $t = r + j$ anında gönderir.
- `if (r >= 0 && r < m)` koşulu, satırın henüz yola çıkmadığı ($r < 0$, hat dolarken) ya da bütün satırların geçtiği ($r \geq m$, hat boşalırken) durumları eler.
- `x[j + 1][r] += x[j][r]`, gönderilen değeri alıcının değerine ekler.

Zamanlamanın doğru olduğunu iki gözlem gösterir. Birincisi, p_j satır r'yi gönderdiğinde değeri hazırdır. Bu değeri p_{j-1}, bir adım önce, $t - 1 = r + j - 1$ anında güncellemiştir. İkincisi, aynı adımdaki işlemler çakışmaz. `t` anında j bağlantısı `x[j][r]`'yi okur. $j - 1$ bağlantısı ise aynı sütunda `x[j][r + 1]`'e yazar. Okunan ve yazılan hücreler farklıdır.

Tablo 14.10 kodun her adımda yaptığı toplamaları ve p_4'ün sütununu verir. Değerler kodun çalıştırılmasıyla doğrulanmıştır; Tablo 14.9 ile aynıdır, yalnız zaman sıfırdan sayılır.

`t`	Çalışan `j` (`r = t - j`)	Toplama sayısı	p_4 sütunu
0	0 (r = 0)	1	8, 6, 3, 5
1	0 (1), 1 (0)	2	8, 6, 3, 5
2	0 (2), 1 (1), 2 (0)	3	8, 6, 3, 5
3	0 (3), 1 (2), 2 (1), 3 (0)	4	22, 6, 3, 5
4	1 (3), 2 (2), 3 (1)	3	22, 19, 3, 5
5	2 (3), 3 (2)	2	22, 19, 19, 5
6	3 (3)	1	22, 19, 19, 23

Tablo 14.10: Boru hattı kodunun izlemesi ($p = 5$, $m = 4$).

Adım sayısının genel formülü şöyle bulunur. Son satır $r = m - 1$'dir ve son bağlantı $j = p - 2$'dir. Son satır son bağlantıdan $t = (m - 1) + (p - 2)$ anında geçer. Zaman sıfırdan sayıldığı için adım sayısı

$$m + p - 2$$

olur. Örnekte $4 + 5 - 2 = 7$'dir. Saf yöntem her satırı $p - 1$ bağlantıdan ayrı ayrı geçirir: $m(p - 1) = 16$ adım. Bu sonuç boru hattı fikrindeki $k + m - 1$ formülüyle aynıdır: burada aşama sayısı $k = p - 1$ bağlantıdır.

Verimlilik de hesaplanabilir. Yararlı iş, yapılan toplamaların sayısıdır: her satır her bağlantıdan bir kez geçer, toplam $m(p - 1)$ toplama. Kullanılan kaynak, $p - 1$ bağlantının $m + p - 2$ adım boyunca açık kalmasıdır. Verimlilik

$$E = m(p - 1) / ((p - 1)(m + p - 2)) = m / (m + p - 2)$$

olur. Örnekte $4 / 7 \approx 0{,}57$, yani yaklaşık %57'dir. Satır sayısı işlemci sayısına göre çok büyükse ($m \gg p$) verimlilik 1'e yaklaşır. Örneğin $p = 5$ ve $m = 1000$ için $1000 / 1003 \approx 0{,}997$'dir. Boru hattı, çok sayıda bağımsız veri öğesi aynı aşamalardan geçecekse etkilidir.

Sık yapılan hata

Boru hattının adım sayısını m sanmak. Hat başta boştur ve dolması zaman alır. Son satır yola ancak $m - 1$ adım sonra çıkar ve zinciri geçmesi için $p - 1$ adım daha gerekir. m satır, p işlemci için doğru sayı $m + p - 2$'dir. $p = 6$, $m = 50$ için bu fark 4 adımdır; $p = 100$, $m = 10$ için ise 98

adımdır ve sonucu tamamen değiştirir.

14.7 Boru hatlı ağaç

Boru hattı fikri yalnız zincirlere değil ağaçlara da uygulanır. Bu kesimde işlemciler bir ağaç biçiminde bağlıdır ve her işlemci bir vektör tutar. Amaç, vektörlerin bileşen bileşen toplamını kökte toplamaktır.

14.7.1 Ağaçta indirgeme

İşlemciler köklü bir ağaç oluştursun. Her düğüm değerini ebeveynine gönderebilir; ebeveyn gelen değeri kendi değerine ekler. Yapraklar değerlerini gönderir, iç düğümler çocuklarından gelenleri ekleyip yukarı iletir ve sonunda toplam kökte birikir. Bu, ağaç toplamının ileti alışverişiyle yapılan biçimidir. Ağacın derinliği h ise tek bir değerin indirgenmesi h aşama sürer.

Örnekte yedi işlemci vardır. Kök p_0'ın çocukları p_1 ve p_5'tir. p_1'in çocukları p_2 ve p_4, p_2'nin çocuğu p_3, p_5'in çocuğu p_6'dır. En uzun yol $p_3 \to p_2 \to p_1 \to p_0$'dır; derinlik $h = 3$'tür.

Şimdi her işlemcide tek bir sayı değil, k bileşenli bir vektör olsun. Örnekte $k = 3$'tür ve bileşenler yeşil, mavi ve kırmızı renklerle gösterilir. Her bileşen ayrı ayrı toplanacaktır. Bir ileti tek bir bileşen taşır ve kuralı şudur: **bir düğüm bir turda en fazla bir ileti alabilir**; aynı turda bir ileti de gönderebilir.

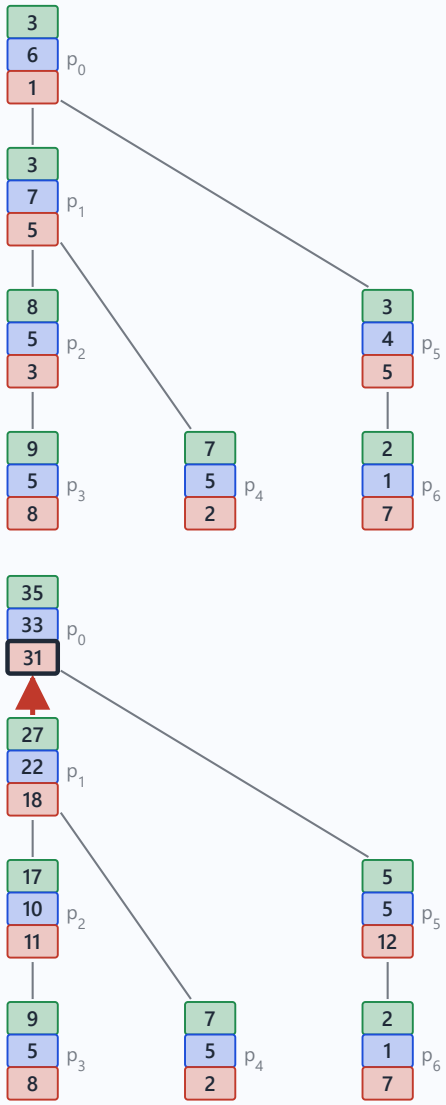
Saf yöntem bileşenleri sırayla indirger. Önce yeşil bileşen köke çıkarılır, sonra mavi, sonra kırmızı. Her bileşen $h = 3$ turda köke çıkar; saf yöntem toplam $k \cdot h = 9$ tur ister. **Boru hatlı ağaç** (*pipelined tree*) yönteminde bir bileşen yukarı çıkarken bir sonraki bileşen yapraklardan yola çıkar.

14.7.2 Adım adım çalıştırma

Her bileşen köke üç aşamada çıkar:

1. **Aşama 1:** $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$.
2. **Aşama 2:** $p_2 \to p_1$, $p_5 \to p_0$.
3. **Aşama 3:** $p_1 \to p_0$.

p_1 iki çocuğundan ileti alır: aşama 1'de p_4'ten, aşama 2'de p_2'den. p_0 da aşama 2'de p_5'ten, aşama 3'te p_1'den alır. Aşamalar bu yüzden ayrı turlarda yapılmak zorundadır. Bir bileşenin aşama 1'i ile bir sonraki bileşenin aşama 1'i arasında da en az iki tur olmalıdır. Aksi hâlde p_1 aynı turda hem p_4'ten yeni bileşeni hem p_2'den eski bileşeni almak zorunda kalır. Bu nedenle yeni bileşen iki turda bir yola çıkar. Tablo 14.11 turları verir.



Şekil 14.15: Boru hatlı ağaç. Birinci çizimde başlangıç: yedi işlemcinin her biri üç bileşenli bir vektör tutar (yeşil, mavi, kırmızı). İkinci çizimde yedinci turun sonu: kökte (35, 33, 31). Kenarın rengi o turda taşınan bileşeni, kalın çerçeve o turda güncellenen değeri gösterir.

Tur	Aşama 1	Aşama 2	Aşama 3	Toplamalar
1	yeşil	—	—	$p_3 \to p_2$: 9 + 8 = 17; $p_4 \to p_1$: 7 + 3 = 10; $p_6 \to p_5$: 2 + 3 = 5
2	—	yeşil	—	$p_2 \to p_1$: 17 + 10 = 27; $p_5 \to p_0$: 5 + 3 = 8
3	mavi	—	yeşil	yeşil $p_1 \to p_0$: 27 + 8 = 35; mavi 5 + 5 = 10, 5 + 7 = 12, 1 + 4 = 5
4	—	mavi	—	$p_2 \to p_1$: 10 + 12 = 22; $p_5 \to p_0$: 5 + 6 = 11
5	kırmızı	—	mavi	mavi $p_1 \to p_0$: 22 + 11 = 33; kırmızı 8 + 3 = 11, 2 + 5 = 7, 7 + 5 = 12
6	—	kırmızı	—	$p_2 \to p_1$: 11 + 7 = 18; $p_5 \to p_0$: 12 + 1 = 13
7	—	—	kırmızı	$p_1 \to p_0$: 18 + 13 = 31

Tablo 14.11: Boru hatlı ağacın yedi turu. Değerler kodlanmış bir benzetimle doğrulanmıştır.

Tur 3, boru hattının işe yaradığı ilk turdur. Yeşil bileşen kökte tamamlanırken mavi bileşen yapraklardan yola çıkar. O turda p_0 yalnız p_1'den, p_1 yalnız p_4'ten, p_2 yalnız p_3'ten, p_5 yalnız p_6'dan ileti alır. Kural çiğnenmez. p_1 aynı turda hem alır (mavi, p_4'ten) hem gönderir (yeşil, p_0'a).

Sonuç yeşil 35, mavi 33, kırmızı 31'dir. Doğrulama için yeşil bileşenlerin hepsi toplanır: $3 + 3 + 8 + 9 + 7 + 3 + 2 = 35$. Burada sıra p_0, p_1, ..., p_6'dır.

14.7.3 Analiz

Her bileşen köke $h = 3$ turda çıkar. Yeni bileşen iki turda bir başlar. Son bileşen $(k - 1) \cdot 2$ tur sonra başlar ve h turda biter. Toplam tur sayısı

$$h + (k - 1) \cdot 2 = 3 + 2 \cdot 2 = 7$$

olur. Saf yöntem $k \cdot h = 9$ tur ister. Kazanç küçük görünür, ama k büyüdükçe artar. Boru hatlı ağaç $2k + 1$, yani yaklaşık $2k$ tur sürer; saf yöntem $3k$ tur sürer. Örneğin $k = 10$ için 21 ve 30 tur.

Yedi turun bu ağaç için en iyi sonuç olduğu basit bir sayımla görülür. Kökün iki çocuğu vardır ve üç bileşenin her biri için her çocuktan bir ileti almalıdır: toplam 6 ileti. Kök bir turda bir ileti alabildiğine göre en az 6 tur ileti alır. İlk ileti en erken 2. turda gelebilir; p_5'in göndereceği değer önce p_6'nın katkısını almalıdır, p_1'in değeri ise daha da derin bir alt ağaca bağlıdır. 2. turdan başlayan altı tur, en erken 7. turda biter.

Genel olarak tur sayısı yaklaşık $h + (k - 1) \cdot c$'dir; burada c, bir düğümün en fazla çocuk sayısıdır. Çok çocuklu bir düğüm bir bileşeni almak için c tur harcar ve hattın hızını o belirler. Ağaç bir zincirse ($c = 1$), formül $h + k - 1$'e iner; bu, boru hattının $k + m - 1$ formülünün aynısıdır.

> **Sık yapılan hata**
>
> Bileşenlerin her turda bir yola çıkabileceğini sanmak. O zaman tur sayısı $h + k - 1 = 5$ bulunur. Ama ikinci bileşen 2. turda yola çıkarsa p_1 o turda hem p_4'ten mavi bileşeni hem p_2'den yeşil bileşeni almak zorunda kalır. Bu, "bir turda bir ileti" kuralını çiğner. İki çocuklu düğümler hattın hızını yarıya düşürür.

14.8 Paralel algoritma türleri ve Java

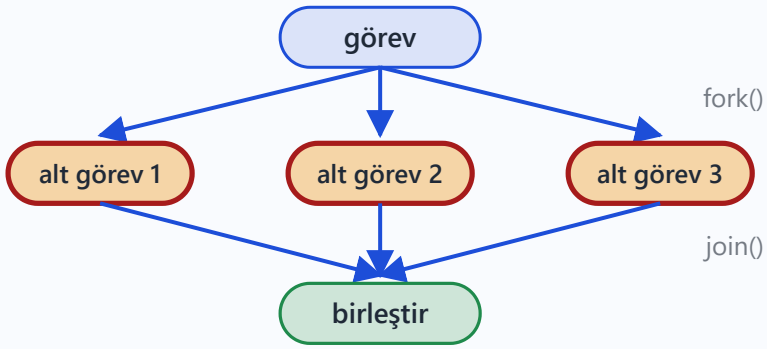
Önceki kesimlerdeki algoritmalar PRAM ya da ileti alışverişi yapan işlemciler gibi soyut modellerde çalıştı. Bu kesim aynı fikirleri gerçek bir dilde, Java'da kurar. Java programları paylaşımlı bellekli bir makinede çalışır. İş parçacıkları ortak nesnelere erişebilir. İş parçacıklarını doğrudan yönetmek yerine çoğunlukla görev tabanlı araçlar kullanılır.

14.8.1 Üç tür paralellik

Uygulamada karşılaşılan paralel algoritmaların çoğu üç kalıptan birine uyar:

- **Böl ve fethet.** Problem bağımsız alt problemlere bölünür. Alt problemler farklı çekirdeklerde aynı anda çözülür ve sonuçlar birleştirilir. Paralel birleştirme sıralaması, paralel hızlı sıralama ve blok blok matris çarpımı bu kalıptadır.
- **MapReduce.** Büyük bir veri kümesi birçok düğümde işlenir. *Map* adımı her parçadan ara anahtar–değer çiftleri üretir; *reduce* adımı aynı anahtarın değerlerini birleştirir. Hadoop ve Spark bu kalıbın yaygın uygulamalarıdır. Kelime sayma, arama motorları için dizin kurma ve günlük kaydı çözümlemesi tipik kullanım alanlarıdır.
- **Veri paralelliği.** Aynı hesaplama verinin farklı parçalarına aynı anda uygulanır. SIMD komutları ve GPU'lar bu kalıp için tasarlanmıştır. Bilimsel benzetim, görüntü işleme ve sayısal hesaplama bu kalıbı kullanır.

Böl ve fethet kalıbının Java'daki karşılığı **Fork/Join** yapısıdır. Şekil 14.16 yapının fikrini gösterir. Bir görev alt görevlere **çatallanır** (*fork*). Alt görevler paralel çalışır. Ana görev alt görevlerin bitmesini **bekler** (*join*) ve sonuçlarını birleştirir.



Şekil 14.16: Fork/Join yapısı. Bir görev üç alt göreve çatallanır (fork), alt görevler paralel çalışır, sonuçlar join ile birleştirilir.

14.8.2 Fork/Join ile paralel birleştirme sıralaması

Birleştirme sıralaması böl ve fethetin en tanıdık örneğidir. Dizi ikiye bölünür, iki yarı ayrı ayrı sıralanır, sonra sıralı iki yarı birleştirilir. İki yarının sıralanması birbirinden bağımsızdır; paralel yapılmaları için yalnız ayrı görevlere verilmeleri gerekir.

```
class SiralamaGorevi extends RecursiveTask<int[]> {
  private final int[] dizi;
  SiralamaGorevi(int[] dizi) { this.dizi = dizi; }

  protected int[] compute() {
    if (dizi.length <= 1) return dizi;
    int orta = dizi.length / 2;
    var sol = new SiralamaGorevi(
        Arrays.copyOfRange(dizi, 0, orta));
    var sag = new SiralamaGorevi(
        Arrays.copyOfRange(dizi, orta, dizi.length));
    sol.fork();               // paralel başlat
    sag.fork();
    return birlestir(sol.join(), sag.join());
  }
}
// kullanım:
int[] s = new ForkJoinPool().invoke(
    new SiralamaGorevi(dizi));
```

Kodun öğeleri şunlardır:

- `RecursiveTask<int[]>`, sonuç döndüren bir Fork/Join görevidir. Görevin işi `compute()` metodunda yazılır. Sonuç döndürmeyen görevler için `RecursiveAction` kullanılır.
- Taban durumu: bir öğeli dizi zaten sıralıdır ve olduğu gibi döner.
- Dizi `orta` noktasından ikiye ayrılır. `Arrays.copyOfRange` her yarının bir kopyasını çıkarır; böylece iki görev aynı diziyi paylaşmaz.
- `fork()`, alt görevi havuza verir ve hemen döner. Görev başka bir iş parçacığında çalışabilir; çağıran görev beklemeden ilerler. Bu işlem **eşzamansızdır** (*asynchronous*).
- `join()`, alt görevin bitmesini bekler ve sonucunu döndürür.
- `birlestir`, iki sıralı diziyi tek bir sıralı diziye çeviren bilinen sıralı birleştirmedir; *O(n)* sürer.
- `new ForkJoinPool().invoke(...)`, kök görevi bir havuzda çalıştırır ve sonucunu bekler.

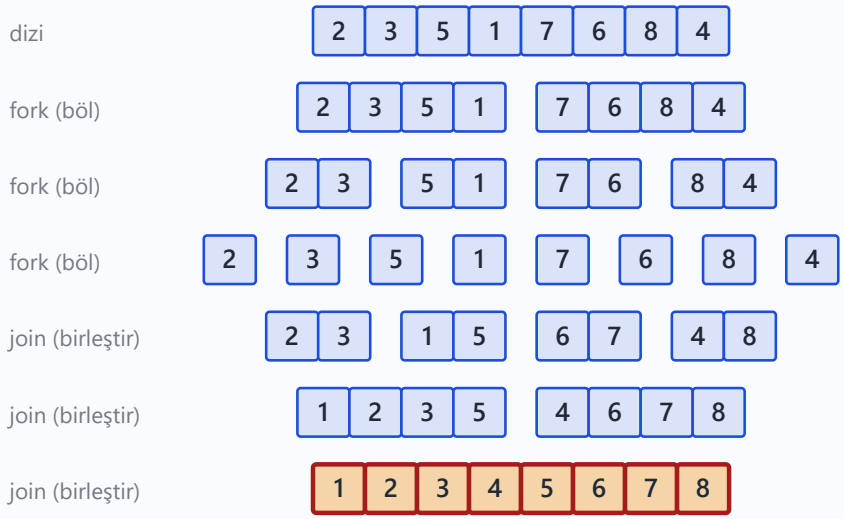
`ForkJoinPool`, bir iş parçacığı havuzudur. Havuzdaki her iş parçacığının kendi görev kuyruğu vardır. Bir iş parçacığı `fork()` ettiği görevi kendi kuyruğuna koyar. Kuyruğu boşalan iş parçacığı boş beklemez; başka bir iş parçacığının kuyruğunun öbür ucundan bir görev alır. Bu düzene **iş çalma** (*work stealing*) denir. İş çalma, yük dengelemeyi kendiliğinden sağlar: hangi görevin ne kadar süreceği önceden bilinmese de boşta kalan çekirdek iş bulur.

Tablo 14.12 örnek dizi 2 3 5 1 7 6 8 4 için görevlerin düzey düzey yaptıklarını gösterir. Yukarıdan aşağı bölme, aşağıdan yukarı birleştirme yapılır.

Düzey	İşlem	Görevler	Paralel görev sayısı
0	kök	[2 3 5 1 7 6 8 4]	1
1	fork	[2 3 5 1] [7 6 8 4]	2

Düzey	İşlem	Görevler	Paralel görev sayısı
2	fork	[2 3] [5 1] [7 6] [8 4]	4
3	taban	8 tek öğeli dizi	8
4	join + birleştir	[2 3] [1 5] [6 7] [4 8]	4
5	join + birleştir	[1 2 3 5] [4 6 7 8]	2
6	join + birleştir	[1 2 3 4 5 6 7 8]	1

Tablo 14.12: Paralel birleştirme sıralamasında görev düzeyleri.



Şekil 14.17: 2 3 5 1 7 6 8 4 dizisinin paralel birleştirme sıralaması. Üç düzey bölme (fork), tek öğeli taban durumları ve üç düzey birleştirme (join). Kök görev son birleştirmeyi tek başına yapar.

Program 12 mantıksal çekirdekli bir makinede JDK 21 ile 11 kez çalıştırıldı ve her seferinde [1, 2, 3, 4, 5, 6, 7, 8] sonucunu verdi. Birleştirmelerin hangi iş parçacığında ve hangi sırayla yapıldığı ise çalıştırmaya göre değişti. On çalıştırmalık bir dizide ilk biten birleştirme beş kez [8, 4], ikişer kez [2, 3] ve [7, 6], bir kez [5, 1] oldu. Dört taban birleştirmesini dört ayrı iş parçacığı yürüttü. Paralel programın sonucu belirlenimlidir, ama yürütme sırası belirlenimli değildir. Bunun güvencesi, görevlerin ortak bir veriyi değiştirmemesidir: her görev kendi kopyası üzerinde çalışır ve sonucunu döndürür.

Sık yapılan hata

Çok ince taneli görevler üretmek. Kod her alt dizi için, bir öğeli dizilere kadar yeni bir görev oluşturur. Bir milyon öğelik dizide bu yaklaşık iki milyon görev demektir. Görev oluşturmak, kuyruğa koymak ve beklemek, iki sayıyı karşılaştırmaktan çok daha pahalıdır. Uygulamada bir eşik seçilir. Örneğin 1000 öğenin altındaki parçalar `Arrays.sort` ile sıralı sıralanır.

Kodda küçük bir iyileştirme daha yapılabilir. `sol.fork(); sag.fork();` yerine `sol.fork(); int[] y = sag.compute(); int[] x = sol.join();` yazılsın. Çağıran görev sağ yarıyı kendisi hesaplar; yalnız sol yarı havuza verilir. Böylece çağıran iş parçacığı `join()`'de boş beklemez ve havuza verilen görev sayısı yarıya iner.

14.8.3 Paralel sıralamanın analizi

Paralel birleştirme sıralamasının işi sıralı sürümle aynıdır: her düzeyde toplam n öğe birleştirilir ve $\log_2 n$ düzey vardır. $W = \Theta(n \log n)$. Derinlik ise yinelemeyle bulunur. İki yarı paralel sıralandığı için derinliğe yalnız birinin derinliği katılır. Birleştirme sıralıdır ve $\Theta(n)$ derinlik ekler:

> $$D(n) = D(n / 2) + \Theta(n) = \Theta(n)$$

Yineleme açılınca $n + n/2 + n/4 + \ldots \le 2n$ elde edilir. Paralellik $W / D = \Theta(n \log n) / \Theta(n) = \Theta(\log n)$'dir.

Sürüm	İş W	Derinlik D	Paralellik W / D
Sıralı birleştirme sıralaması	$\Theta(n \log n)$	$\Theta(n \log n)$	1
Paralel özyineleme + sıralı birleştirme	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(\log n)$
Paralel özyineleme + paralel birleştirme	$\Theta(n \log n)$	$\Theta(\log^3 n)$	$\Theta(n / \log^2 n)$

Tablo 14.13: Birleştirme sıralamasının üç sürümü.

Paralellik yalnız $\Theta(\log n)$'dir ve bu oldukça azdır. $n = 10^6$ için $\log_2 n \approx 20$'dir; bu kodla 20 civarından fazla çekirdek pek yarar sağlamaz. Darboğaz, kök görevin yaptığı son birleştirmedir. O birleştirme bütün diziyi tek başına gezer ve bu sırada öteki çekirdekler boştur.

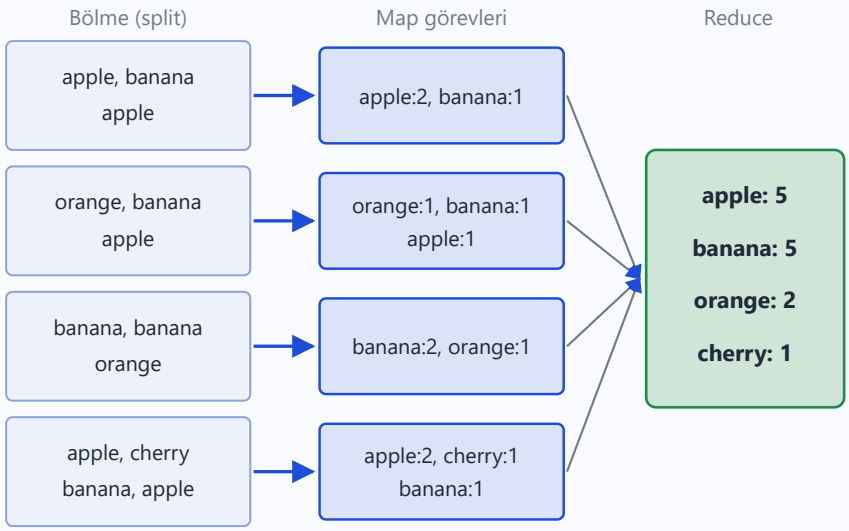
Darboğazı kaldırmak için birleştirmenin kendisi de paralelleştirilir. Fikir şöyledir: büyük yarının ortanca öğesi seçilir, öteki yarıda bu öğenin yeri ikili aramayla bulunur. Böylece iki yarı ikişer parçaya ayrılır ve parçalar birbirinden bağımsız olarak, yine paralel birleştirilir. Bu sürümün derinliği $\Theta(\log^3 n)$'e iner ve paralellik $\Theta(n / \log^2 n)$ olur. Ayrıntısı Cormen ve arkadaşlarının kitabında verilir. Java'da hazır bir çözüm de vardır: `Arrays.parallelSort(dizi)`, büyük dizileri Fork/Join ile paralel sıralar.

14.8.4 MapReduce

MapReduce, Google'da 2004'te büyük veri kümelerini binlerce makinede işlemek için geliştirilen bir programlama modelidir. Programcı yalnız iki fonksiyon yazar; verinin dağıtılması, hata durumunda işin yeniden yapılması ve sonuçların toplanması çerçeve tarafına bırakılır. Model üç adımdan oluşur:

1. **Map.** Girdi parçalara bölünür. Her parça bağımsız işlenir ve ara anahtar–değer çiftleri üretir. Kelime saymada çift "kelime → adet" biçimindedir.
2. **Karıştırma** (*shuffle*). Aynı anahtara sahip çiftler aynı indirgeyiciye (*reducer*) gönderilir.
3. **Reduce.** Her anahtarın değerleri birleştirilir; kelime saymada toplanır.

Örnekte 13 kelime vardır: apple, banana, apple, orange, banana, apple, banana, banana, orange, apple, cherry, banana, apple. Dört iş parçacığı kullanılır. Parça boyu *⌊13 / 4⌋ = 3*'tür; ilk üç parça üçer kelime alır, son parça kalan dört kelimeyi alır. Şekil 14.18 akışı gösterir.



Şekil 14.18: MapReduce ile kelime sayma. 13 kelime dört parçaya bölünür (soldaki sütun), her map görevi kendi parçasını sayar (ortadaki sütun), reduce sonuçları birleştirir: apple 5, banana 5, orange 2, cherry 1.

Java'da map görevleri Fork/Join görevleri olarak yazılır. Her görev bir kelime listesini (`parca`) alır ve kendi sayım haritasını döndürür.

```
// Map: bir parçadaki kelimeleri say
protected Map<String, Integer> compute() {
  Map<String, Integer> sonuc = new HashMap<>();
  for (String kelime : parca)
    sonuc.merge(kelime, 1, Integer::sum);
  return sonuc;
}
// Ana akış
for (MapGorevi g : mapGorevleri) g.fork();
List<Map<String, Integer>> ara = new ArrayList<>();
for (MapGorevi g : mapGorevleri) ara.add(g.join());
// Reduce: ara sonuçları birleştir
Map<String, Integer> toplam = new HashMap<>();
for (Map<String, Integer> m : ara)
  m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```

Kodun parçaları şöyle çalışır:

- `compute()` , yeni bir `HashMap` oluşturur. Bu harita yalnız bu göreve aittir.

- `sonuc.merge(kelime, 1, Integer::sum)`, kelime haritada yoksa 1 değeriyle ekler. Varsa eski değerle 1'i toplar. Tek satırda "yoksa ekle, varsa artır" işlemini yapar.
- Ana akışta önce bütün map görevleri `fork()` edilir; görevler havuzda paralel çalışmaya başlar. Ardından her görevin sonucu `join()` ile alınır ve `ara` listesine eklenir. Bütün görevler fork edildikten sonra join edilir. Her görev fork edilir edilmez join edilseydi görevler sırayla çalışırdı.
- Reduce adımı ara haritaları sırayla gezer. Her (anahtar, değer) çifti `toplam.merge(k, v, Integer::sum)` ile toplam haritasına eklenir.

Tablo 14.14 programın çalıştırılmasından elde edilen ara sonuçları ve reduce adımlarını verir. Haritaların yazdırılma sırası `HashMap`'in iç düzenine bağlıdır. Bu sıra anahtarların hash değerlerine göre belirlenir; aynı anahtarlarla her çalıştırmada aynı çıkar ama alfabetik değildir.

Adım	Girdi	Sonuç
map 0	apple, banana, apple	banana 1, apple 2
map 1	orange, banana, apple	banana 1, orange 1, apple 1
map 2	banana, banana, orange	orange 1, banana 2
map 3	apple, cherry, banana, apple	banana 1, cherry 1, apple 2
reduce ara[0]	boş harita + ara[0]	banana 1, apple 2
reduce ara[1]	+ ara[1]	orange 1, banana 2, apple 3
reduce ara[2]	+ ara[2]	orange 2, banana 4, apple 3
reduce ara[3]	+ ara[3]	orange 2, banana 5, cherry 1, apple 5

Tablo 14.14: Kelime sayma programının izlemesi.

Bu tasarımın en önemli özelliği **paylaşılan durumun olmamasıdır**. Her map görevinin kendi `HashMap`'i vardır. Görevler birbirinin verisine dokunmaz; bu yüzden kilit gerekmez ve yarış durumu oluşamaz. Birleştirme, bütün görevler bittikten sonra tek bir yerde yapılır. Aynı fikir ağaç toplamında da vardı: her işlemci kendi kısmi toplamını üretir, kısmi sonuçlar sonra birleştirilir.

Örnekte reduce tek bir görevdir ve bütün anahtarları o işler. Gerçek bir MapReduce sisteminde anahtarlar birçok indirgeyiciye bölüştürülür. Bölüştürme için çoğunlukla *hash(k) mod r* kullanılır; *r* indirgeyici sayısıdır. Aynı anahtar her zaman aynı indirgeyiciye gider. Böylece reduce adımı da paralel çalışır.

Örnekteki parça boyları dengesizdir: son görev dört kelime, ötekiler üçer kelime alır. Küçük bir örnekte bu önemsizdir. Ama milyonlarca kelimede bir görevin ötekilerden çok daha fazla iş alması, öteki görevlerin onu beklemesine yol açar. Gerçek sistemler girdiyi görev sayısından çok daha fazla parçaya bölerek yükü dengeler.

14.8.5 Veri paralelliği: matris çarpımı

$n \times n$ boyutlu iki matrisin çarpımı $C = A \cdot B$'de her hücre $C[i][j] = \Sigma_k A[i][k] \cdot B[k][j]$ ile hesaplanır. Sonuç matrisinin her satırı, A'nın yalnız bir satırına ve B'nin tamamına bağlıdır. Satırlar birbirinden bağımsızdır; her satır ayrı bir görev olarak hesaplanabilir.

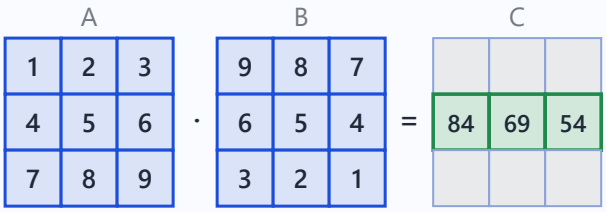
```
// her satır ayrı bir görev
class SatirGorevi extends RecursiveAction {
  protected void compute() {
    for (int j = 0; j < n; j++) {
      C[satir][j] = 0;
      for (int k = 0; k < n; k++)
        C[satir][j] += A[satir][k] * B[k][j];
    }
  }
}
try (var havuz = new ForkJoinPool()) {
  for (int i = 0; i < n; i++)
    havuz.submit(new SatirGorevi(i));
}   // close(): tüm görevleri bekler
```

- `SatirGorevi`, sonuç döndürmeyen bir görevdir (`RecursiveAction`). Hesapladığı değerleri doğrudan `C` matrisine yazar. `satir` alanı görevin hangi satırı hesaplayacağını tutar; yapıcısı kısalık için gösterilmemiştir.
- İç içe iki döngü bir satırın n hücresini hesaplar. Her hücre n çarpma ve toplama ister.
- `try (var havuz = new ForkJoinPool())` bloğu her satır için bir görevi havuza verir (`submit`). Java 19'dan beri `ForkJoinPool` kapatılabilir bir kaynaktır. Blok bitince çağrılan `close()`, havuzdaki bütün görevlerin bitmesini bekler.

Örnekte A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] ve B = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]'dir. Program çalıştırıldığında C = [[30, 24, 18], [84, 69, 54], [138, 114, 90]] sonucunu verdi. Tablo 14.15, satır 1 görevinin yaptığı her işlemi gösterir.

`j`	`k`	**İşlem**	`C[1][j]`
0	—	başlangıç	0
0	0	+ 4 · 9	36
0	1	+ 5 · 6	66
0	2	+ 6 · 3	84
1	0, 1, 2	4 · 8 + 5 · 5 + 6 · 2 (32, 57, 69)	69
2	0, 1, 2	4 · 7 + 5 · 4 + 6 · 1 (28, 48, 54)	54

Tablo 14.15: Satır 1 görevinin izlemesi. A'nın 1. satırı (4, 5, 6), B'nin sütunlarıyla çarpılır.



Satır 0 ve 2 başka görevlerde, aynı anda hesaplanır.

Şekil 14.19: Satır 1 görevinin sonu: $C[1] = (84, 69, 54)$. Öteki satırlar başka görevlerde hesaplanır.

Algoritmanın işi ve derinliği şöyledir. n görevin her biri $\Theta(n^2)$ işlem yapar; toplam iş $W = \Theta(n^3)$'tür ve sıralı algoritmayla aynıdır. Görevler paralel çalıştığı için derinlik tek bir görevin süresidir: $D = \Theta(n^2)$. Paralellik $W / D = \Theta(n)$'dir. Satır sayısından fazla çekirdek bu tasarıma yarar sağlamaz. Daha fazla paralellik gerekiyorsa her hücre ayrı görev yapılabilir ya da matris bloklara bölünebilir.

Bu kodda yarış durumu yoktur, çünkü her görev yalnız kendi satırına yazar. A ve B bütün görevlerce okunur ama hiçbir görev onlara yazmaz. Eşzamanlı okuma zararsızdır. C'nin her satırına ise tek bir görev yazar. Görevlerin yazdığı bölgeler ayrık olduğu sürece paylaşılan bir matris güvenle kullanılabilir.

14.9 Paylaşılan durum ve yarış

Şimdiye kadarki Java örneklerinde görevler ya kendi verilerinde çalıştı ya da ayrık bölgelere yazdı. Bu bir rastlantı değil, bilinçli bir tasarım kararıydı. Birden çok iş parçacığı aynı veriyi değiştirince programın davranışı öngörülemez hâle gelir. Bu kesim sorunun kaynağını gösterir ve çözüm yollarını anlatır.

14.9.1 Yarış durumu

İki iş parçacığı paylaşılan bir `sayac` değişkenini artırsın. Java'da `sayac++` tek bir ifade gibi görünür. Ama işlemci düzeyinde üç adımdır: bellekten **oku**, yerel kopyayı **artır**, belleğe **yaz**. İki iş parçacığının adımları iç içe geçerse bir artış kaybolabilir. Tablo 14.16 böyle bir iç içe geçişi gösterir.

Zaman	İş parçacığı 1	İş parçacığı 2	`sayac`
1	oku (0)		0
2		oku (0)	0
3	yaz 0 + 1		1
4		yaz 0 + 1	1

Tablo 14.16: Kayıp güncelleme. Başta `sayac` = 0; iki iş parçacığı birer kez artırır.

İki iş parçacığı da 0 okudu ve 1 yazdı. Beklenen sonuç 2'dir, ama sayaç 1'dir. İkinci yazma birincinin sonucunu ezdi. Bu hataya **kayıp güncelleme** (*lost update*) denir.

Tanım 14.7 (Yarış durumu)

Yarış durumu (*race condition*), bir programın sonucunun iş parçacıklarının zamanlamasına, yani işletim sisteminin onları hangi sırayla çalıştırdığına bağlı olmasıdır. En az iki iş parçacığı aynı veriye eşgüdümsüz erişir ve en az biri yazar.

Yarış durumunu gerçek bir programda görmek için aşağıdaki kod yazıldı. İki iş parçacığının her biri paylaşılan `sayac`'ı bir milyon kez artırır.

```
public class SayacYarisi {
  static int sayac = 0;                       // paylaşılan durum

  public static void main(String[] args) throws InterruptedException {
    Runnable is = () -> {
      for (int i = 0; i < 1_000_000; i++)
        sayac++;                              // oku → artır → yaz
    };
    Thread t1 = new Thread(is), t2 = new Thread(is);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println(sayac);                // beklenen 2000000
  }
}
```

`Runnable`, bir iş parçacığının yürüteceği işi tanımlar; burada bir lambda ifadesidir. `start()` iş parçacığını başlatır ve hemen döner; iki iş parçacığı bu noktadan sonra aynı anda çalışır. `join()`, ana iş parçacığının o iş parçacığı bitene kadar beklemesini sağlar. Böylece `println`, iki döngü de bittikten sonra çalışır.

Program JDK 21 ile 12 mantıksal çekirdekli bir Windows makinesinde, her seferinde yeni bir JVM başlatılarak 30 kez çalıştırıldı. **Sonuç çalıştırmaya göre değişti.** 30 çalıştırmanın hiçbirinde beklenen 2.000.000 çıkmadı. En küçük değer 1.049.868, en büyük değer 1.920.460 oldu; bazı çalıştırmalarda artışların yaklaşık yarısı kayboldu. Bu sayılar başka bir makinede, başka bir JDK sürümünde ya da aynı makinede başka bir anda farklı çıkar. Değişmeyen tek şey, sonucun 2.000.000'u geçememesidir: kayıp güncelleme bir artışı siler ama hiçbir zaman fazladan artış eklemez.

Aynı deney iş parçacığı başına 1000 artışla da yapıldı. Bu kez 20 çalıştırmanın hepsinde doğru sonuç (2000) çıktı. Neden şudur: 1000 artış o kadar kısa sürer ki birinci iş parçacığı çoğu zaman ikincisi başlamadan biter. İki iş parçacığı hiç çakışmaz. Ama bu, programın doğru olduğu anlamına gelmez. Makine meşgulse ya da zamanlama değişirse aynı program yanlış sonuç verebilir. Yarış durumlarının ayıklanmasını zorlaştıran da budur: hata sınamada görünmeyebilir ve ancak gerçek kullanımda, rastlantısal olarak ortaya çıkar. İş parçacığı başına 100.000 artışla yapılan 20 çalıştırmada ise sonuçlar 109.546 ile 161.741 arasında değişti; beklenen değer 200.000'dir.

Örnek 14.7

İki iş parçacığı, paylaşılan `sayac`'ı (başta 0) eşgüdümsüz olarak ikişer kez artırıyor. Her artış "oku, yaz" biçimindedir: okunan değerin bir fazlası yazılır. Sonuç en az kaç olabilir?

1. İP1 okur: 0.
2. İP2 ilk artışını tam yapar: 0 okur, 1 yazar. `sayac` = 1.
3. İP1 ilk artışını tamamlar: okuduğu 0'ın bir fazlasını, yani 1 yazar. İP2'nin artışı silinir. `sayac` = 1.
4. İP2 ikinci artışı için okur: 1.
5. İP1 ikinci artışını tam yapar: 1 okur, 2 yazar.
6. İP2 ikinci artışını tamamlar: okuduğu 1'in bir fazlası, 2 yazılır.

Sonuç 2'dir; dört artıştan ikisi kaybolmuştur. Bütün olası iç içe geçişleri deneyen bir program, ikişer artışta sonucun 2, 3 ya da 4 olabileceğini gösterir. Benzer bir senaryoyla, her iş parçacığı en az iki kez artırdığı sürece artış sayısı ne olursa olsun sonuç 2'ye kadar düşebilir.

14.9.2 Yarışı önlemek

Yarış durumunu önlemenin üç temel yolu vardır.

Kilit. `synchronized (kilit) { sayac++; }` bloğuna aynı anda yalnız bir iş parçacığı girebilir. Öteki iş parçacığı kilit serbest kalana kadar bekler. Oku, artır ve yaz adımları böylece bölünmez bir bütün olur. Kilitli sürüm 20 çalıştırmanın hepsinde 2.000.000 verdi.

Atomik işlem. `java.util.concurrent.atomic` paketindeki sınıflar, işlemcinin özel komutlarıyla bölünmez işlemler sunar. `AtomicInteger.incrementAndGet()` oku, artır ve yaz adımlarını tek bir atomik adımda yapar.

```
import java.util.concurrent.atomic.AtomicInteger;

public class SayacAtomik {
  static final AtomicInteger sayac = new AtomicInteger();

  public static void main(String[] args) throws InterruptedException {
    Runnable is = () -> {
      for (int i = 0; i < 1_000_000; i++)
        sayac.incrementAndGet();              // tek atomik adım
    };
    Thread t1 = new Thread(is), t2 = new Thread(is);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println(sayac.get());          // her zaman 2000000
  }
}
```

Atomik sürüm 10 ayrı çalıştırmanın hepsinde 2.000.000 yazdı. Eşzamanlı yapılar da aynı ilkeyle çalışır. `ConcurrentHashMap`, birçok iş parçacığının aynı haritaya güvenle yazmasına izin verir.

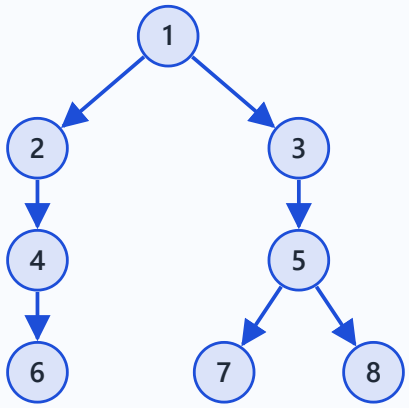
Paylaşılan durumdan kaçınmak. En iyi çözüm, paylaşılan değişkeni hiç kullanmamaktır. Her görev kendi kısmi sonucunu üretir; sonuçlar görevler bittikten sonra birleştirilir. Ağaç toplamı, MapReduce ve satır görevleri bu ilkeyle çalışır. Sayaç örneğinde her iş parçacığı kendi yerel sayacını artırır ve `join()`'den sonra iki yerel sayaç toplanır. Bu çözüm hem doğrudur hem en hızlısıdır, çünkü iş parçacıkları hiç beklemez.

Kilitler bedelsiz değildir. Kilitli bir bölgede aynı anda yalnız bir iş parçacığı çalışır; bölge sıralı hâle gelir. Amdahl yasasına göre bu sıralı kısım hızlanmayı sınırlar. Ayrıca birden çok kilit kullanılınca **kilitlenme** riski doğar. İP1 A kilidini tutup B'yi, İP2 B kilidini tutup A'yı beklerse ikisi de sonsuza kadar bekler.

> **Sık yapılan hata**
>
> Paylaşılan sayacı `volatile int` yapmanın yarışı çözdüğünü sanmak. `volatile`, bir iş parçacığının yazdığı değerin ötekilerce hemen görülmesini sağlar. Ama `sayac++`'nın oku, artır, yaz adımlarını bölünmez yapmaz. İki iş parçacığı yine aynı değeri okuyup aynı değeri yazabilir. Atomiklik için `AtomicInteger` ya da kilit gerekir.

14.9.3 Eşzamanlı çizge gezinme

Yarış durumu yalnız sayaçlarda değil, daha karmaşık veri yapılarında da ortaya çıkar. Derinlik öncelikli gezinmenin paralel bir sürümünü düşünelim. Her ziyaret edilmemiş komşu için yeni bir gezinme görevi `fork()` edilir ve alt ağaçlar paralel gezilir. Şekil 14.20'deki çizgede 1'den başlayan gezinme 2 ve 3 için iki görev açar; bu görevler de kendi komşuları için yeni görevler açar.



Şekil 14.20: Örnek çizge: 1→2, 1→3, 2→4, 3→5, 4→6, 5→7, 5→8.

Görevler, hangi düğümlerin ziyaret edildiğini tutan ortak bir kümeyi paylaşır. Bu kümede iki ayrı sorun vardır. Birincisi, küme sıradan bir `HashSet` ise eşzamanlı yazmalarda iç yapısı bozulabilir. İkincisi, küme güvenli olsa bile "içinde var mı?" sorusu ile "ekle" işlemi iki ayrı adımdır. İki görev aynı düğümü aynı anda sorabilir, ikisi de "yok" yanıtı alabilir ve ikisi de düğümü ziyaret eder. Doğru kod şudur:

```
Set<Dugum> ziyaret = ConcurrentHashMap.newKeySet();
if (!ziyaret.add(komsu)) continue; // atomik kontrol+ekle
```

`ConcurrentHashMap.newKeySet()`, eşzamanlı erişime dayanıklı bir küme döndürür. `add` metodu öğe eklendiyse `true`, öğe zaten varsa `false` döndürür. Kontrol ile ekleme tek bir atomik işlemdir. İki görev aynı düğümü eklemeye çalışırsa yalnız biri `true` alır ve düğümü yalnız o ziyaret eder.

Bu farkı görmek için bir deney yapıldı. Sekiz iş parçacığının her biri 0'dan 99.999'a kadar aynı 100.000 düğümü sırayla "ziyaret" etmeye çalıştı. Her iş parçacığı bir düğümü yalnız "ilk ziyaret eden" olduğuna karar verirse saydı. Doğru bir küme 100.000 ziyaret vermelidir. Her yöntem 20 kez çalıştırıldı ve **sonuçlar çalıştırmaya göre değişti**:

Küme	Kontrol + ekleme	Fazladan ziyaret	Not
`HashSet`	`contains`, sonra `add`	44.821 – 147.433	`size()` 104.205 – 137.960; küme bozuldu
`newKeySet()`	`contains`, sonra `add`	53.122 – 212.552	küme sağlam, ama düğümler birden çok ziyaret edildi
`newKeySet()`	`if (!add(k)) continue`	0 (20 çalıştırmanın hepsinde)	doğru

Tablo 14.17: Sekiz iş parçacığı, 100.000 düğüm; her yöntem 20 çalıştırma (JDK 21, 12 mantıksal çekirdek).

`HashSet` sürümünde kümede yalnız 100.000 farklı sayı olabileceği hâlde `size()` bu sayıyı aştı. Bu, kümenin iç sayacının da yarışa girdiğini gösterir. İkinci satır, güvenli bir kümenin tek başına yetmediğini gösterir: iki adımlı kontrol yine yarışa açıktır. Yalnız üçüncü satırdaki atomik kontrol ve ekleme

her çalıştırmada doğru sonucu verdi.

Çizgelerde başka bir paralel gezinme biçimi de kullanılır: **düzey eşzamanlı genişlik öncelikli arama**. Her düzeyin bütün düğümleri paralel genişletilir ve düzey sonunda bir bariyer konur. Sonraki düzey, önceki düzeyin bütün düğümleri işlendikten sonra başlar. Düzey sayısı çizgenin çapı kadar olduğundan derinlik *O(çap)*'tır. Çapı küçük, düğüm sayısı büyük çizgelerde, örneğin sosyal ağlarda, bu yöntem çok paralellik sunar.

14.9.4 Java'da paralellik araçları

Java, farklı düzeylerde paralellik araçları sunar. Tablo 14.18 hangisinin ne zaman kullanılacağını özetler.

Araç	Ne zaman?
`Thread`, `Runnable`	en alt düzey; elle yönetim, nadiren doğrudan kullanılır
`ExecutorService`	bağımsız görevleri sabit bir iş parçacığı havuzunda çalıştırmak
`ForkJoinPool`, `RecursiveTask`	böl ve fethet; iş çalma ile özyinelemeli görevler
`list.parallelStream()`	veri paralel map / filter / reduce (ortak `ForkJoinPool` kullanır)
`Arrays.parallelSort`	büyük dizileri paralel sıralamak
`Arrays.parallelPrefix`	paralel önek toplamı (tarama)
`AtomicInteger`, `ConcurrentHashMap`	paylaşılan sayaç ya da küme gerektiğinde güvenli erişim

Tablo 14.18: Java'nın paralellik araçları.

Tablonun üst satırlarındaki araçlar daha çok denetim verir ama daha çok kod ister. Alt satırlardaki araçlar işi tek bir metot çağrısına indirir. Örneğin kelime sayma, akış (*stream*) arayüzüyle tek bir ifadeye sığar:

```
Map<String, Long> adet = kelimeler.parallelStream()
    .collect(Collectors.groupingByConcurrent(k -> k, Collectors.counting()));
```

`parallelStream()`, listeyi parçalara bölüp ortak `ForkJoinPool` üzerinde işler. `groupingByConcurrent`, kelimeleri eşzamanlı bir haritada gruplar ve her grubun öğelerini sayar. Bu ifade 13 kelimelik örnekte banana 5, orange 2, apple 5, cherry 1 sonucunu verdi. MapReduce kodunun bütün işi, yani parçalama, map, karıştırma ve reduce, tek ifadede yapılmış olur. Ortak havuzun paralellik düzeyi varsayılan olarak çekirdek sayısının bir eksiğidir; deneyin yapıldığı makinede 11'di.

14.10 Sık yapılan hatalar

- ***p* işlemcinin *p* kat hız getireceğini sanmak.** Sıralı kısım, iletişim ve bekleme yüzünden hızlanma *p*'nin altında kalır. Programın %10'u sıralıysa Amdahl yasasına göre hızlanma 10'u geçemez.

- **Hızlanmayı yanlış tabana göre hesaplamak.** T_1, paralel algoritmanın tek işlemcideki süresi değil, en iyi sıralı algoritmanın süresidir.
- **Paylaşılan `HashSet` ya da sayaç kullanmak.** Eşzamanlı yazma yarış durumuna yol açar. `ConcurrentHashMap`, `AtomicInteger` ya da yerel sonuç ile sonradan birleştirme kullanılmalıdır. Güvenli bir kümede bile "kontrol et, sonra ekle" iki adımdır; tek adım olan `add`'in dönüş değerine bakılmalıdır.
- **Çok ince taneli görevler.** Her öğe için `fork()` yapmak, görev oluşturma yükünü kazancın üstüne çıkarır. Bir eşiğin altındaki parçalar sıralı işlenmelidir.
- **Turlar arası senkronizasyonu unutmak.** PRAM toplamada tur $t + 1$, tur t'nin bütün yazmaları bitmeden başlarsa eski değerler okunur. Hillis–Steele'in sıralı benzetiminde aynı hata tur başı kopyanın unutulmasıyla ortaya çıkar.
- **Yalnız derinliğe bakmak.** Hillis–Steele $O(\log n)$ turda biter ama $O(n \log n)$ iş yapar; az işlemcide sıralı taramadan yavaş kalabilir. Algoritma seçilirken iş ve derinlik birlikte değerlendirilir.
- **Boru hattının adım sayısını yanlış saymak.** Hattın dolma süresi unutulur. m satır ve p işlemci için adım sayısı m değil, $m + p - 2$'dir.
- **Birleşmeli olmayan işlemi ağaçla indirgemek.** Çıkarma ya da bölme ağaçla indirgenirse sonuç değişir. Kayan noktalı toplamada da paralel sonuç son basamaklarda sıralı sonuçtan farklı olabilir.
- **Bir kez doğru çalışan paralel programı doğru sanmak.** Yarış durumu her çalıştırmada görülmez. İş parçacığı başına 1000 artışlı sayaç deneyi 20 çalıştırmanın hepsinde doğru sonucu verdi; aynı kod bir milyon artışla 30 çalıştırmanın hiçbirinde doğru sonuç vermedi.

14.11 Özet

- Paralel algoritma problemi parçalara böler, parçaları işlemcilere atar, aynı anda çözer ve sonuçları birleştirir. Bedeli iletişim, senkronizasyon ve yük dengelemedir. Veri bağımlılığı zinciri paralelleştirilemez; ama zincirdeki işlem birleşmeliyse zincir ağaca çevrilebilir.
- Flynn sınıflandırması makineleri komut ve veri akışlarına göre SISD, SIMD, MISD ve MIMD diye ayırır. Bellek paylaşımlı ya da dağıtık olabilir. PRAM, ortak bellekli ve ortak saatli kuramsal modeldir; EREW, CREW ve CRCW türleri aynı hücreye eşzamanlı erişime göre ayrılır.
- Hızlanma $S = T_1 / T_p$, verimlilik $E = S / p$, maliyet $C = p \cdot T_p$'dir. Maliyet-optimal algoritmada $C = \Theta(T_1)$'dir. Amdahl yasası $S \le 1 / (1 - f)$ sınırını verir. Gustafson yasası problem büyüdükçe ölçeklenmiş hızlanmanın $(1 - f) + f \cdot p$ olduğunu söyler.
- İş W toplam işlem sayısı, derinlik D en uzun bağımlılık zinciridir. $\max(W / p, D) \le T_p \le W / p + D$ (Brent). Paralellik W / D, yararlı kullanılabilecek en fazla işlemci sayısıdır.
- PRAM ağaç toplamı $\log_2 n$ turda, $n - 1$ toplamayla çalışır ve EREW ile yetinir. $n / \log n$ işlemciyle maliyet-optimal olur.
- Önek toplamında Hillis–Steele $\log_2 n$ tur ve $n \log_2 n - (n - 1)$ iş harcar; iş-verimli değildir. Blelloch yukarı ve aşağı süpürmeyle $2(n - 1)$ iş ve $2 \log_2 n$ turda dışlayıcı önek toplamını bulur.

- Boru hattında p işlemci ve m satır için adım sayısı $m + p - 2$'dir (saf yöntem $m(p - 1)$), verimlilik $m / (m + p - 2)$'dir. Boru hatlı ağaçta tur sayısı $h + (k - 1) \cdot c$'dir; örnekte 7 tur (saf 9).
- Java'da böl ve fethet `ForkJoinPool` ile yazılır: `fork()` alt görevi havuza verir, `join()` sonucunu bekler; iş çalma yükü dengeler. Sıralı birleştirmeli paralel birleştirme sıralamasının derinliği $\Theta(n)$, paralelliği $\Theta(\log n)$'dir.
- MapReduce map, karıştırma ve reduce adımlarından oluşur. Her map görevi kendi haritasını kullandığı için kilit gerekmez. Satır görevleriyle matris çarpımında da görevler ayrık bölgelere yazar.
- Yarış durumunda sonuç zamanlamaya bağlıdır ve çalıştırmaya göre değişir. Çözümler kilit, atomik işlem, eşzamanlı yapılar ve en iyisi paylaşılan durumdan kaçınmaktır.

14.12 Kavram listesi

paralel algoritma *parallel algorithm* | PRAM modeli *PRAM model* | hızlanma *speedup* | verimlilik *efficiency* | Amdahl yasası *Amdahl's law* | iş / derinlik *work / span* | önek toplamı *prefix sum (scan)* | boru hattı *pipelining* | ileti *message passing* | iş çalma *work stealing* | yarış durumu *race condition*

maliyet-optimal *cost-optimal* iş-verimli *work-efficient* Brent teoremi *Brent's theorem* Gustafson yasası *Gustafson's law* indirgeme *reduction* bariyer *barrier* kayıp güncelleme *lost update* kilitlenme *deadlock*

14.13 Kendini Dene

1. $n = 32$ sayı PRAM ağaç toplamıyla toplanıyor. Kaç tur sürer, ilk turda kaç toplama yapılır, toplam kaç toplama yapılır? 16 işlemciyle maliyet nedir; algoritma maliyet-optimal midir?

   Cevap

   $\log_2 32 = 5$ tur. İlk turda 16 toplama yapılır; toplam $16 + 8 + 4 + 2 + 1 = 31 = n - 1$. 16 işlemciyle maliyet $16 \cdot 5 = 80$'dir; sıralı algoritma 31 adım sürer. Oran $\log_2 n / 2$ gibi büyür ve n arttıkça sınırsız artar; algoritma maliyet-optimal değildir. $n / \log_2 n \approx 6$ işlemciyle önce bloklar sıralı toplanırsa maliyet $\Theta(n)$'e iner.

2. Programın %95'i paralelleştirilebiliyor. 16 işlemcide Amdahl hızlanması ve verimliliği nedir? İşlemci sayısı sonsuza giderken sınır kaçtır? Problem boyutu işlemci sayısıyla büyütülürse Gustafson yasası ne verir?

   Cevap

   $S = 1 / (0{,}05 + 0{,}95 / 16) = 1 / (0{,}05 + 0{,}059375) \approx 9{,}14$. Verimlilik $9{,}14 / 16 \approx 0{,}57$. Sınır $1 / 0{,}05 = 20$. Gustafson: $0{,}05 + 0{,}95 \cdot 16 = 15{,}25$.

3. Bir iş en iyi sıralı algoritmayla 200 s sürüyor. 10 işlemcili paralel sürüm 25 s sürüyor. Hızlanma, verimlilik ve maliyet nedir? Kaç işlemci-saniye ek yüke gitmiştir?

   Cevap

   $S = 200 / 25 = 8$, $E = 8 / 10 = 0{,}8$, $C = 10 \cdot 25 = 250$ s. Ek yük $250 - 200 = 50$ işlemci-saniyedir; işlemci başına 5 s.

4. [1, 2, 3, 4] dizisine Blelloch algoritması uygulanıyor. Yukarı süpürmenin, kökün sıfırlanmasının ve aşağı süpürmenin her turundan sonra dizi nedir? Kaç işlem ve kaç tur yapılır?

   Cevap

   Yukarı $d = 1$: [1, 3, 3, 7]; $d = 2$: [1, 3, 3, 10]. Kök: [1, 3, 3, 0]. Aşağı $d = 2$, çift (1, 3): $t = 3$, `x[1]` = 0, `x[3]` = 0 + 3 → [1, 0, 3, 3]. Aşağı $d = 1$, çift (0, 1): `x[0]` = 0, `x[1]` = 0 + 1; çift (2, 3): `x[2]` = 3, `x[3]` = 3 + 3 → [0, 1, 3, 6]. Bu dışlayıcı önek toplamıdır. $2(n - 1) = 6$ işlem, $2 \log_2 4 = 4$ tur.

5. Üç işlemcili bir zincirde p_0 = (1, 2, 3, 4, 5), p_1 = (1, 1, 1, 1, 1), p_2 = (2, 2, 2, 2, 2) sütunları var. Boru hattı kaç adım sürer, saf yöntem kaç adım sürer, verimlilik nedir? Sonunda p_2 ne tutar ve satır 4 p_2'ye hangi adımda ulaşır?

   Cevap

   $m + p - 2 = 5 + 3 - 2 = 6$ adım; saf yöntem $m(p - 1) = 10$ adım. Verimlilik $5 / 6 \approx 0{,}83$. p_1 (2, 3, 4, 5, 6) olur, p_2 (4, 5, 6, 7, 8) olur. Satır 4'ü p_1, $t = 4 + 1 = 5$ anında gönderir; bu altıncı ve son adımdır.

6. Dört işlemci $p_3 \to p_2 \to p_1 \to p_0$ biçiminde bir zincir (tek çocuklu ağaç) oluşturuyor ve her biri 5 bileşenli bir vektör tutuyor. Kök p_0. Boru hatlı indirgeme kaç tur sürer, saf yöntem kaç tur sürer?

   Cevap

   $h = 3$, $k = 5$, her düğümün en çok bir çocuğu var ($c = 1$). Yeni bileşen her turda yola çıkabilir; çünkü her düğüm bir turda yalnız bir ileti alır. Tur sayısı $h + (k - 1) \cdot 1 = 7$; saf yöntem $k \cdot h = 15$ tur.

7. $n = 2^{20}$ sayı ağaçla toplanıyor ve 1024 işlemci var. İş, derinlik ve paralellik nedir? Brent teoremi süre için hangi aralığı verir; hızlanma en az kaçtır?

   Cevap

   $W = 2^{20} - 1 = 1.048.575$, $D = 20$, paralellik ≈ 52.429. Alt sınır $W / p \approx 1024$; üst sınır $W / p + D \approx 1044$ adım. Hızlanma en az $1.048.575 / 1044 \approx 1004$'tür; 1024 işlemciyle neredeyse doğrusal. İşlemci sayısı paralellikten çok küçük olduğunda derinlik terimi önemsizleşir.

8. *1000 × 1000* matrisler satır görevleriyle çarpılıyor. İş, derinlik ve paralellik nedir? 2000 çekirdekli bir makinede hızlanma en fazla kaç olabilir?

   Cevap

   $W = n^3 = 10^9$ çarpma-toplama, $D = n^2 = 10^6$ (tek satırın süresi), paralellik $W / D = 1000$. Yalnız 1000 görev olduğu için çekirdeklerin en çok yarısı çalışır; $T_p \geq D$ olduğundan hızlanma en fazla 1000'dir. Daha fazla paralellik için hücre ya da blok görevleri gerekir.

9. Bir öğrenci bir listenin toplamını `long toplam = 0;` değişkeni ve `liste.parallelStream().forEach(x -> toplam += x)` ile bulmak istiyor. Bu neden yanlıştır? Java bu kodu derler mi? Doğrusu nasıl yazılır?

   Cevap

   Java bu kodu derlemez: lambda içinden yalnız fiilen değişmez (*effectively final*) yerel değişkenler kullanılabilir. Değişken bir alan (*field*) yapılırsa kod derlenir ama `toplam += x` oku, topla, yaz adımlarından oluşur ve iş parçacıkları arasında kayıp güncelleme oluşur. Sonuç çalıştırmaya göre değişir. Doğrusu paylaşılan durum kullanmayan bir indirgemedir: `liste.parallelStream().mapToLong(Long::longValue).sum()`. Her iş parçacığı kendi kısmi toplamını üretir, kısmi toplamlar sonra birleştirilir.

14.14 İleri okuma

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 26: iş, derinlik ve paralellik kavramları ile açgözlü zamanlayıcı sınırı burada işlenir. Paralel birleştirmenin ve paralel birleştirme sıralamasının ayrıntılı çözümlemesi de bu bölümdedir.
- J. JáJá, *An Introduction to Parallel Algorithms* (1992): PRAM modeli, maliyet-optimal ağaç toplamı, önek toplamı ve PRAM türleri arasındaki ilişkiler için kapsamlı bir kaynaktır.
- A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing* (2. baskı, 2003): boru hattı, iletişim maliyetleri, ölçeklenebilirlik ve MPI ile programlama gibi uygulamaya yakın konuları işler.
- D. Lea, "A Java Fork/Join Framework" (2000): Java'daki `ForkJoinPool`'un tasarımını ve iş çalma düzeneğini, onu yazan kişinin anlatımıyla verir.

14.15 Kaynaklar

1. G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS Spring Joint Computer Conference*, 1967.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 26: Parallel Algorithms).

3. J. Dean, S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, *OSDI*, 2004.
4. A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing*, 2. baskı, Addison-Wesley, 2003.
5. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
6. D. Lea, “A Java Fork/Join Framework”, *ACM Java Grande Conference*, 2000.

EK A

Terim Sözlüğü

Kitapta geçen İngilizce terimler ve her biri için kullanılan tek Türkçe karşılık. Sağdaki sayı, terimin ayrıntılı anlatıldığı bölümdür. Sözlüğün aranabilir hâli ders sayfasındadır.

A

activity selection → etkinlik seçimi 7
Çakışmayan en çok etkinliği seçme; en erken biten önce.

adaptive adversary → uyarlanır düşman 13
Algoritmanın rastgele kararlarını görüp sonraki isteği buna göre seçen düşman; buna karşı rastgeleliğin faydası büyük ölçüde kaybolur.

adaptive (sort) → uyarlanabilir 2
Girdi zaten kısmen sıralıysa daha hızlı çalışan (sıralama).

adversary → düşman 13
Algoritmayı en kötü duruma zorlayan girdiyi seçen hayalî rakip.

Amdahl's law → Amdahl yasası 14
Sıralı kalan kısmın, elde edilebilecek en büyük hızlanmayı sınırlaması.

amplification → tekrarla güçlendirme 12
Rastgele bir algoritmayı bağımsız olarak tekrarlayıp hata olasılığını küçültme tekniği.

ant colony optimization → karınca kolonisi optimizasyonu 11
Karıncalardan esinlenen yöntem: kısa tur kuran yapay karıncalar kenarlara daha çok feromon bırakır, feromon zamanla buharlaşır.

approximation algorithm → yaklaşım algoritması 11
NP-zor problemler için polinom sürede, optimuma kanıtlı bir oranla yakın çözüm üreten algoritma.

approximation ratio → yaklaşım oranı 11
Algoritmanın çözümü ile optimum arasındaki en kötü oran (ρ).

approximation scheme (PTAS / FPTAS) → yaklaşım şeması 11
Her ε > 0 için (1 + ε) oranlı çözüm veren algoritma ailesi.

assignment problem → atama problemi 10
n kişiye n işi, her kişiye bir iş düşecek biçimde en az toplam maliyetle dağıtma problemi.

B

backtracking → geri izleme (DP tablosundan çözümü bulmak ise geri okuma (bkz. traceback)) 9
Kısmi çözümü adım adım genişleten, çıkmazda son seçimi geri alıp başka seçeneği deneyen arama.

barrier → bariyer 14
Bütün işlemciler ona ulaşmadan hiçbirinin geçemediği eşgüdüm noktası.

Belady's anomaly → Belady anomalisi 13
FIFO'da çerçeve sayısı artınca sayfa hatasının artabilmesi.

best-first search → en iyi önce arama 10
Sınırı en umut verici olan düğümü önce açan arama stratejisi.

bin packing → kutulara yerleştirme 7
Boyutları verilen nesneleri sabit kapasiteli en az sayıda kutuya yerleştirme problemi; NP-zordur.

binary search → ikili arama 3
Sıralı dizide ortadaki öğeyle karşılaştırıp aralığı her adımda yarıya indiren arama: O(log n).

bound → sınır 10
Bir daldan elde edilebilecek en iyi değerin iyimser tahmini.

branch and bound → dal sınır 10
Durum uzayını dallandırıp her dal için sınır hesaplayan, sınırı bilinen en iyi çözümden kötü olan dalları budayan yöntem.

branching → dallanma 10
Bir alt problemi daha küçük alt problemlere ayırma.

bubble sort → kabarcık sıralaması 2
Komşu öğeleri karşılaştırıp sırasızsa takas eden; her turda en büyük sona çıkar.

bucket → kova 3
Hash tablosundaki dizinin her bir hücresi.

bucket sort → kova sıralaması 2
Öğeleri değer aralıklarına göre kovalara dağıtıp kovaları sıralayan yöntem.

C

cache → önbellek 13
Sık kullanılan verileri tutan küçük, hızlı bellek.

canonical coin system → kanonik para sistemi 7
Açgözlü para üstü algoritmasının her tutar için en az madeni parayı verdiği para sistemi.

Carmichael number → Carmichael sayısı 12
Kendisiyle aralarında asal olan her taban için Fermat testini geçen bileşik sayı.

chromatic number → renk sayısı 9
Bir çizgeyi, kenarla bağlı iki düğüm aynı rengi almayacak biçimde boyamaya yeten en küçük renk sayısı.

closest pair of points → en yakın nokta çifti 6
Düzlemdeki noktalar arasında birbirine en yakın iki noktayı bulma.

coin change problem → para üstü problemi 7
Bir tutarı en az sayıda madeni parayla oluşturma.

collision → çakışma 3
İki farklı anahtarın aynı hash değerine düşmesi.

comparison sort lower bound → karşılaştırma alt sınırı 2
Karşılaştırmaya dayalı her sıralama en kötü durumda Ω(n log n) karşılaştırma yapar.

competitive analysis → rekabetçi analiz 13
Çevrim içi algoritmayı en iyi çevrim dışı algoritmayla (OPT) karşılaştırma.

competitive ratio → rekabet oranı 13
Her girdi için ALG ≤ c · OPT (+ sabit) sağlayan en küçük c.

compulsory miss (cold miss) → zorunlu hata (soğuk hata) 13
Önbellek başlangıçta boş olduğu için ilk farklı sayfalarda hiçbir politikanın önleyemediği sayfa hatası.

concept drift → kavram kayması 13
Verinin dağılımının zamanla değişmesi; artımlı öğrenmede eski veri bu durumda yanıltıcı olur.

constraint satisfaction problem → kısıt sağlama problemi 9
Değişkenlere tüm kısıtları sağlayan değerler atama problemi.

convergence → yakınsama (sayısal yakınsama; approximation algorithm ise yaklaşım algoritması) 11
Yinelemelerin gerçek değere giderek yaklaşması.

convex hull → dışbükey örtü 6
Nokta kümesini içeren en küçük dışbükey çokgen.

counting sort → saymalı sıralama 2
Küçük tam sayı anahtarları sayıp önek toplamıyla yerleştiren, karşılaştırmasız sıralama.

D

data stream → veri akışı 13
Tek geçişte, az bellekle işlenmesi gereken sürekli gelen veri.

dead node → ölü düğüm 10
Genişletilmiş, olursuz bulunmuş ya da sınırı yetersiz olduğu için budanmış düğüm; bir daha ele alınmaz.

decision problem → karar problemi 9
Cevabı evet ya da hayır olan problem.

decision tree → karar ağacı 2
Karşılaştırmalı bir algoritmanın davranışını gösteren ağaç: iç düğümler karşılaştırma, yapraklar ortaya çıkan sıralamadır.

decrease and conquer → azalt ve fethet 3
Problemi tek bir küçük alt probleme indirgeyerek çözme yaklaşımı (ör. ikili arama).

depth-first branch and bound → derinlik öncelikli dal sınır 10
Canlı düğümleri yığınla (derinlik öncelikli) gezen, belleği derinlikle sınırlı dal sınır.

distributed memory → dağıtık bellek 14
Her düğümün kendi belleği olduğu, verinin ağ üzerinden iletiyle taşındığı düzen; iyi ölçeklenir ama iletişim pahalıdır.

divide and conquer → böl ve fethet 6
Problemi küçük alt problemlere bölüp özyinelemeli çözen ve sonuçları birleştiren yaklaşım.

dynamic programming (DP) → dinamik programlama 8
Örtüşen alt problemleri bir kez çözüp sonuçlarını saklayarak kullanan tasarım yöntemi.

E

E-node (expanding node) → E-düğüm 10
O anda çocukları, yani alt problemleri üretilen düğüm.

edge contraction → kenar büzme 12
Bir kenarın iki ucunu tek düğümde birleştirme işlemi (Karger algoritması).

efficiency → verimlilik 14
İşlemci başına hızlanma: hızlanma / p.

entropy → entropi 7
Bir kaynağın sembol başına ortalama bilgi miktarı; önek-özgür kodun ortalama uzunluğu için alt sınır.

eviction policy → çıkarma politikası 13
Önbellek dolunca hangi sayfanın çıkarılacağını belirleyen kural (FIFO, LRU, OPT).

exchange argument → değiştirme argümanı 7
Optimal çözümü açgözlü seçimi içerecek biçimde değiştirerek doğruluğu kanıtlama.

expected running time → beklenen çalışma süresi 12
Rastgele algoritmanın verilen bir girdi için çalışma süresinin beklenen değeri.

expected value → beklenen değer 12
Bir rastgele değişkenin olasılıklarla ağırlıklı ortalaması.

exponential search → üstel arama 3
Aralığı 1, 2, 4, … diye büyütüp sonra ikili arama yapan yöntem.

F

feasible → olurlu 10
Tüm kısıtları sağlayan (çözüm).

Fisher–Yates shuffle → Fisher–Yates karıştırma 12
Her permütasyonu eşit olasılıkla üreten O(n) karıştırma.

forward checking → ileri denetim 9
Bir değişkene değer atanınca bağlı değişkenlerin olası değerlerinden çelişenleri hemen çıkarma.

fractional knapsack → kesirli sırt çantası 7
Eşyaların kesirli alınabildiği, değer/ağırlık oranına göre açgözlü çözülen problem.

G

genetic algorithm → genetik algoritma 11
Bir çözüm popülasyonuyla çalışan yöntem: iyi çözümler daha sık seçilir, çaprazlamayla birleştirilir, mutasyon çeşitliliği korur.

geometric distribution → geometrik dağılım 12
Başarı olasılığı p olan denemelerde ilk başarıya kadar geçen deneme sayısının dağılımı.

global optimum → genel optimum 11
Bütün çözümlerin en iyisi; yerel arama bir yerel optimumda durur ve buna ulaşamayabilir.

graph coloring → çizge boyama 9
Komşu düğümlere farklı renk verecek biçimde düğümleri boyama.

greedy algorithm → açgözlü algoritma 7
Her adımda o an en iyi görünen seçimi yapıp geri dönmeyen algoritma.

greedy-choice property → açgözlü seçim özelliği 7
Yerel en iyi seçimin bir optimal çözüme götürebilmesi.

H

Hamiltonian cycle → Hamilton çevrimi 11
Her düğümden tam bir kez geçip başlangıca dönen çevrim.

Hamiltonian path → Hamilton yolu 9
Her düğümden tam bir kez geçen yol.

hash function → hash fonksiyonu 3
Anahtarı tablo indisine dönüştüren fonksiyon.

hash table → hash tablosu 3
Anahtarları hash fonksiyonuyla bulunan konumlarda saklayan tablo; ortalama O(1) arama.

heap → heap 2
Her düğümün çocuklarından büyük (max-heap) ya da küçük (min-heap) olduğu tam ikili ağaç.

heap sort → heap sort 2
Diziyi max-heap yapıp kökü tekrar tekrar sona alarak sıralayan algoritma.

hit (cache hit) → isabet 13
İstenen sayfanın önbellekte bulunması; maliyeti 0'dır. Bulunmazsa sayfa hatası oluşur.

Huffman coding → Huffman kodlama 7
Sık karakterlere kısa kod veren, önek içermeyen en iyi kodlama.

Hungarian method → Macar yöntemi 10
Atama problemini $O(n^3)$ zamanda çözen algoritma.

I

in-place → yerinde 2
Girdi dışında yalnızca O(1) (ya da çok az) ek bellek kullanan.

incremental learning → artımlı öğrenme 13
Modeli her yeni veriyle, geçmişi yeniden işlemeden güncelleme.

incumbent → enİyi 10
Dal sınırda şimdiye kadar bulunan en iyi tam çözümün değeri.

indicator random variable → gösterge değişkeni 12
Bir olay gerçekleşince 1, gerçekleşmeyince 0 değerini alan rastgele değişken.

insertion sort → eklemeli sıralama 2
Her öğeyi sıralı kısımda doğru yere ekleyerek sıralama.

integer linear programming (ILP) → tam sayılı doğrusal programlama 10
Değişkenlerin tam sayı olması gereken doğrusal programlama.

interpolation search → ara değer arama 3
Aranan değerin konumunu uç değerlere göre tahmin eden arama; düzgün dağılımda O(log log n).

inversion → ters çift 2
i < j iken dizi[i] > dizi[j] olan indis çifti.

J

job sequencing with deadlines → son tarihli iş sıralama 7
Son tarihleri olan işlerden kârı en büyük çizelgeyi seçme.

jump search → atlayarak arama 3
Sıralı dizide √n'lik adımlarla atlayıp bloğun içinde doğrusal arayan yöntem.

K

Karatsuba multiplication → Karatsuba çarpımı 6
Büyük sayıları üç alt çarpımla O(n^1,585) sürede çarpan yöntem.

#

0/1 knapsack → 0/1 sırt çantası 8
Her eşyanın ya bütünüyle alındığı ya hiç alınmadığı, kapasiteyi aşmadan değeri en büyükleyen problem.

L

Las Vegas algorithm → Las Vegas algoritması 12
Her zaman doğru yanıt veren, süresi rastgele olan algoritma.

least recently used (LRU) → LRU (en uzun süredir kullanılmayan) 13
Önbellek dolunca en uzun süredir kullanılmayan sayfayı çıkaran politika; rekabet oranı k'dir.

least-cost (LC) search → en düşük maliyetli (LC) arama 10
Canlı düğümlerden tahmini maliyeti $\hat{c} = f + \hat{g}$ en küçük olanı genişleten arama.

linear congruential generator (LCG) → doğrusal eşlik üreteci 12
Bir sonraki sayıyı $(a \cdot x + c) \bmod m$ formülüyle üreten, en eski ve en basit sözde rastgele sayı üreteci; düşük bitleri kısa dönemlidir.

linear probing → doğrusal yoklama 3
Çakışmada sıradaki boş hücreyi arayarak yerleştirme.

linear search → doğrusal arama 3
Öğeleri baştan sona tek tek inceleyen arama: O(n).

list scheduling → liste çizelgeleme 11
Her işi o an en erken boşalan makineye veren açgözlü çizelgeleme.

list update problem → liste güncelleme problemi 13
Erişilen öğeye göre listeyi yeniden düzenleyerek erişim maliyetini azaltma problemi.

live node → canlı düğüm 10
Dal sınırda üretilmiş ama henüz genişletilmemiş, hâlâ umut vaat eden düğüm; canlı düğümler bir kuyruk yapısında bekler.

load factor → yük faktörü 3
Tablodaki anahtar sayısının hücre sayısına oranı ($\alpha = n/m$).

local optimum → yerel optimum 11
Komşu çözümlerin hiçbirinin daha iyi olmadığı çözüm.

local search → yerel arama 11
Bir çözümden başlayıp küçük değişikliklerle iyileştiren arama (ör. 2-opt).

locality of reference → yerellik 13
Programların belleğe rastgele değil, yakın geçmişte kullandıkları yerlere yeniden erişme eğilimi; LRU'yu FIFO'dan iyi yapan budur.

logarithmic → logaritmik 3
Süresi log n ile büyüyen; örnek: ikili arama.

longest common subsequence (LCS) → en uzun ortak alt dizi 8
İki dizide aynı göreli sırayla geçen (ardışık olması gerekmeyen) en uzun karakter dizisi.

longest common substring → en uzun ortak alt dizgi 8
İki dizgide ardışık olarak ortak geçen en uzun parça.

longest increasing subsequence (LIS) → en uzun artan alt dizi 8
Dizideki öğelerden sırası korunarak seçilebilen en uzun artan dizi.

longest processing time first (LPT) → en uzun iş önce 11
İşleri büyükten küçüğe sıralayıp her birini en az yüklü makineye veren çizelgeleme kuralı.

loop invariant → döngü değişmezi 2
Döngünün her turunun başında doğru olan, algoritmanın doğruluğunu göstermeye yarayan önerme.

lost update → kayıp güncelleme 14
İki iş parçacığının aynı değeri okuyup yazması sonucunda bir yazmanın ötekini ezmesi.

LP relaxation → LP gevşetmesi 10
Tam sayı koşulunu kaldırıp doğrusal programı çözerek sınır elde etme.

LSD / MSD radix sort → LSD / MSD basamak sıralaması 2
En az anlamlı (sağdan) ya da en anlamlı (soldan) basamaktan başlayan basamak sıralaması.

M

makespan → en büyük yük 11
Çizelgede son işin bittiği an (C_max).

marking algorithm → işaretleme algoritması 13
Evrelere bölünen ve her evrede yalnız işaretlenmemiş sayfaları çıkaran çevrim içi önbellek politikası.

Markov's inequality → Markov eşitsizliği 12
Negatif olmayan X için P(X ≥ a) ≤ E[X] / a eşitsizliği.

matching → eşleştirme 11
Hiçbir ikisi ortak uç paylaşmayan kenar kümesi.

matrix chain multiplication → matris zinciri çarpımı 8
Bir matris dizisini en az skaler çarpımla çarpmak için parantezleme sırasını bulma problemi.

matroid → matroid 7
Açgözlü algoritmanın en iyi çözümü bulmasını güvence altına alan küme yapısı.

maximum clique → maksimum klik 10
Her iki düğümü komşu olan en büyük düğüm kümesi.

maximum coverage → en büyük kapsama 7
En fazla k küme seçerek olabildiğince çok öğeyi kapsama problemi; açgözlü seçim en iyinin en az 1 − 1/e katına ulaşır.

median → ortanca 13
Sıralı verinin ortasındaki değer.

median-of-three → üçün ortancası 2
İlk, orta ve son öğenin ortancasını pivot seçme kuralı.

memoization → bellekleme 8
Yukarıdan aşağı özyinelemede hesaplanan sonuçları saklayıp yeniden kullanma.

merge → birleştirme 2
İki sıralı diziyi tek sıralı dizide toplama işlemi.

merge sort → birleştirme sıralaması 2
Diziyi ikiye bölüp her yarıyı sıralayan, sonra sıralı yarıları birleştiren sıralama.

message passing → ileti 14
Dağıtık bellekte işlemcilerin veri alışverişi için gönderdiği bilgi.

metaheuristic → metasezgisel 11
Yerel optimumdan kaçmak için genel arama stratejisi (tavlama, genetik algoritma).

min-conflicts heuristic → en az çakışma sezgiseli 9
Çakışan bir değişkene en az çakışma yaratan değeri veren yerel arama.

minimum cut → minimum kesim 12
Çizgeyi iki parçaya ayıran en az kenarlı (ya da en küçük kapasiteli) kesim.

minimum remaining values (MRV) → MRV (en az kalan değer) 9
Kısıt problemlerinde, en az olası değeri kalan değişkeni önce seçme sezgiseli.

Monte Carlo algorithm → Monte Carlo algoritması 12
Süresi sınırlı, yanıtı küçük bir olasılıkla yanlış olabilen algoritma.

move-to-front (MTF) → öne taşı 13
Erişilen öğeyi listenin başına taşıyan kural; 2-rekabetçidir.

moving average → hareketli ortalama 13
Son k değerin ortalamasını her yeni değerde güncelleme.

N

N-queens problem → n-vezir problemi 9
n × n tahtaya birbirini tehdit etmeyen n vezir yerleştirme.

nearest neighbor heuristic → en yakın komşu 11
Her adımda ziyaret edilmemiş en yakın şehre giden TSP sezgiseli.

NP-complete → NP-tam 9
Hem NP'de hem NP-zor olan problem.

NP-hard → NP-zor 9
NP'deki her problemin polinom sürede indirgenebildiği (en az NP kadar zor) problem.

O

oblivious adversary → habersiz düşman 13
Rastgele algoritmanın kodunu bilen ama yazı-tura sonuçlarını göremeyen, bu yüzden istek dizisini baştan seçen düşman.

offline algorithm → çevrim dışı algoritma 13
Tüm girdiyi baştan bilen algoritma.

online algorithm → çevrim içi algoritma 13
Girdiyi parça parça alıp geleceği bilmeden her adımda karar veren algoritma.

open addressing → açık adresleme 3
Çakışan anahtarın tablonun başka bir boş hücresine yerleştirildiği hash yöntemi.

optimal → optimal 11
Amaç fonksiyonuna göre en iyi (çözüm).

optimal substructure → optimal alt yapı 8
Optimal çözümün, alt problemlerin optimal çözümlerinden kurulabilmesi.

optimization problem → en iyileme problemi 9
Cevabı “evet” ya da “hayır” değil, en iyi değer olan problem; çoğu zaman bir karar problemine çevrilebilir.

overlapping subproblems → örtüşen alt problemler 8
Aynı alt problemin özyinelemede tekrar tekrar ortaya çıkması.

P

page fault → sayfa hatası 13
İstenen sayfanın önbellekte olmaması.

paging → sayfalama 13
Sınırlı önbellekte hangi sayfanın tutulacağına karar verme problemi.

parallel algorithm → paralel algoritma 14
İşi aynı anda çalışan birden çok işlemciye bölen algoritma.

partial solution → kısmi çözüm 9
Geri izlemede adım adım kurulan, henüz tamamlanmamış çözüm.

partition → bölümleme 2
Diziyi pivottan küçükler ve büyükler olarak iki tarafa ayırma işlemi (Lomuto, Hoare).

pipelining → boru hattı 14
İşi aşamalara bölüp farklı verileri aşamalarda üst üste işleme.

pivot → pivot 12
Bölümlemede diziyi ikiye ayırmak için seçilen öğe.

polynomial time → polinom zaman 9
Bir k sabiti için $O(n^k)$ süre.

power iteration → kuvvet yinelemesi 11
Bir vektörü matrisle tekrar tekrar çarpıp en büyük özdeğer ve özvektöre yakınsatan yöntem.

PRAM model → PRAM modeli 14
Ortak belleği paylaşan, eş zamanlı adımlarla çalışan işlemcilerden oluşan soyut model.

prefix sum (scan) → önek toplamı 14
Her konumda o konuma kadarki öğelerin toplamı; paralel olarak O(log n) derinlikte.

prefix-free code → önek-özgür kod 7
Hiçbir kod sözcüğünün bir başka kod sözcüğünün öneki olmadığı kod.

primality test → asallık testi 12
Bir sayının asal olup olmadığını sınama (Miller–Rabin).

primary clustering → birincil kümelenme 3
Doğrusal yoklamada dolu hücrelerin uzun ardışık bloklar oluşturması.

probe → yoklama 3
Bir dizi hücresine bakıp aranan değerle karşılaştırma işlemi.

pruning → budama 9
Çözüme götüremeyeceği anlaşılan dalı aramadan çıkarma.

pseudo-polynomial → sözde-polinom 8
Girdi uzunluğuna değil girdideki sayısal değere (ör. W) göre polinom olan süre.

pseudo-random number → sözde rastgele sayı 12
Deterministik bir üreteçle elde edilen, rastgele görünen sayı.

Q

quadratic → karesel 2
$O(n^2)$ büyüme; çoğunlukla iç içe iki döngü.

quick sort → hızlı sıralama 2
Pivota göre bölümleyip iki tarafı özyinelemeli sıralayan algoritma.

quickselect → rastgele seçme (QuickSelect) 12
Rastgele pivotla k. en küçük öğeyi beklenen O(n) sürede bulma.

R

race condition → yarış durumu 14
Sonucun iş parçacıklarının çalışma sırasına bağlı olması.

radix sort → basamak sıralaması 2
Anahtarları basamak basamak kararlı bir sıralamayla sıralayan yöntem (LSD, MSD).

randomized algorithm → rastgele algoritma 12
Adımlarında rastgele seçim yapan algoritma.

reduced cost matrix → indirgenmiş maliyet matrisi 10
Satır ve sütun en küçükleri çıkarılmış maliyet matrisi; çıkarılan toplam TSP için alt sınırdır.

reduction → indirgeme 11
Bir problemin örneklerini başka bir problemin örneklerine dönüştürme.

reservoir sampling → rezervuar örnekleme 13
Uzunluğu bilinmeyen akıştan k öğelik düzgün rastgele örneklem tutma.

running median → çevrim içi ortanca 13
Akıştaki değerlerin ortancasını iki heap ile güncel tutma.

S

satisfiability (SAT) → mantıksal doyurulabilirlik 9
Mantıksal değişkenlerden kurulmuş bir formülü doğru yapan bir değer ataması olup olmadığını soran problem; ilk NP-tam problemdir.

scheduling → çizelgeleme 11
İşleri makinelere ya da zaman dilimlerine atama problemi.

search problem → arama problemi 3
Bir veri kümesinde aranan değerin olup olmadığını ve yerini bulmak.

seed → tohum 12
Sözde rastgele sayı üretecinin başlangıç değeri.

selection sort → seçimli sıralama 2
Her turda sırasız kısmın en küçüğünü seçip başa getiren sıralama.

self-organizing list → kendini düzenleyen liste 13
Öne taşı ya da transpoz gibi bir kuralla, erişilen öğeyi öne doğru taşıyarak kendini kullanıma uyduran liste.

sentinel → bekçi 3
Döngüde sınır denetimini kaldırmak için sona konan özel değer.

#

(separate) chaining → zincirleme 3
Aynı hücreye düşen anahtarları bir listede tutarak çakışma çözme.

S

shared memory → paylaşımlı bellek 14
Bütün işlemcilerin aynı adres uzayını gördüğü düzen; iletişim ucuzdur ama eşzamanlı erişim denetlenmelidir.

shell sort → Shell sıralaması 2
Aralığı giderek küçülen eklemeli sıralama (Donald Shell).

simulated annealing → benzetimli tavlama 11
Kötüleşen adımları azalan bir olasılıkla kabul eden metasezgisel.

ski rental problem → kayak kiralama problemi 13
Kiralama ile satın alma arasında geleceği bilmeden seçim yapma problemi.

sliding window → kayan pencere 13
Toplamı bir değişkende tutup yeni öğeyi ekleyerek, en eski öğeyi çıkararak son k öğe üzerindeki değeri her adımda güncelleyen yöntem.

sorting → sıralama 2
Öğeleri bir anahtara göre artan ya da azalan düzene koymak.

speedup → hızlanma 14
Sıralı sürenin paralel süreye oranı: T_1 / T_p.

stable (sort) → kararlı 2
Eşit anahtarlı öğelerin göreli sırasını koruyan (sıralama).

stack algorithm → yığın algoritması 13
Daha büyük bellekte sayfa kümesi küçük bellektekini kapsayan sayfa değiştirme politikası; Belady anomalisi göstermez.

state-space tree → durum uzayı ağacı 9
Kısmi çözümlerin düğüm, seçimlerin kenar olduğu arama ağacı.

Strassen's algorithm → Strassen algoritması 6
Matrisleri sekiz yerine yedi alt çarpımla O(n^2,81) sürede çarpan yöntem.

strong liar → güçlü yalancı 12
Bileşik bir sayı için tanık olmayan, onu asal gibi gösteren taban değeri.

subproblem → alt problem 8
Asıl problemin daha küçük bir örneği.

subset sum → alt küme toplamı 8
Toplamı verilen değere eşit bir alt küme olup olmadığı problemi.

swap → takas 2
İki öğenin yerlerini karşılıklı değiştirmek.

T

tabulation → tablolama 8
Alt problemleri küçükten büyüğe (aşağıdan yukarı) bir tabloda doldurma.

throughput → verim 14
Birim zamanda tamamlanan iş sayısı; boru hattında aşama sayısı kadar artar.

Timsort → Timsort 2
Doğal sıralı parçaları eklemeli sıralama ve birleştirmeyle işleyen karma sıralama.

top-down / bottom-up → yukarıdan aşağı / aşağıdan yukarı 8
DP'nin iki biçimi: özyineleme + bellekleme ya da döngüyle tablo doldurma.

traceback (solution reconstruction) → geri okuma (backtracking ise geri izleme) 8
Doldurulmuş DP tablosundan geriye doğru giderek çözümün kendisini bulma.

transpose (heuristic) → transpoz 13
Erişilen öğeyi bir önündekiyle takas eden liste düzenleme kuralı.

travelling salesman problem (TSP) → gezgin satıcı problemi 11
Her şehre bir kez uğrayıp başlangıca dönen en kısa turu bulma.

triangle inequality → üçgen eşitsizliği 11
Her üç nokta için d(a, c) ≤ d(a, b) + d(b, c) koşulu.

U

unbounded knapsack → sınırsız sırt çantası 8
Her eşyadan istenen sayıda alınabilen sırt çantası problemi.

up-sweep / down-sweep → yukarı süpürme / aşağı süpürme 14
Blelloch önek toplamının iki evresi: yukarı süpürme yapraklardan köke kısmi toplamları kurar, aşağı süpürme kökten yapraklara her düğüme solundakilerin toplamını taşır.

V

vertex cover → düğüm kapsama 11
Her kenarın en az bir ucunu içeren düğüm kümesi.

W

witness → tanık 12
Bir sayının bileşik olduğunu kanıtlayan taban değeri (Miller–Rabin).

work / span → iş / derinlik 14
Toplam işlem sayısı (T_1) ve en uzun bağımlılık zinciri (T_∞).

work stealing → iş çalma 14
Boşta kalan iş parçacığının başka birinin kuyruğundan iş alması.

work-efficient → iş-verimli 14
İşi, aynı problemin en iyi sıralı algoritmasının süresiyle aynı mertebede olan paralel algoritma.

EK B

Algoritmalar İçin Ön Bilgiler

ALGORİTMALAR · ÖN BİLGİLER

Algoritmalar dersi Java ile programlamayı, temel veri yapılarını, ayrık matematiği, ispat tekniklerini, toplamları ve logaritmaları, olasılığı ve saymayı bir araya getirir. Bu sayfadaki 33 konu, dersin üzerine kurulduğu ön bilgileri altı grupta toplar.

Her kartta kısa bir açıklama, küçük bir örnek, konunun kullanıldığı slaytlara bağlantılar ve bir kontrol sorusu bulunur; sorunun üzerine tıklanınca yanıtı açılır.

Java ve programlama (9) **Temel veri yapıları (6)** **Ayrık matematik ve ispat (4)**

Analiz araçları (5) **Olasılık ve sayma (3)** **Diğer matematik ve kavramlar (6)**

Java ve programlama

Sunumlardaki kodları okumak, izlemek ve çözümleri Java ile yazmak için gereken bilgiler.

J

Java temelleri: türler, koşullar, döngüler

int · long · double · if · for · while

Sunumlardaki kodlar Java ile yazılmıştır; bu yüzden bir döngünün kaç kez döndüğünü ve her değişkenin nasıl değiştiğini kâğıt üzerinde izleyebilmek gerekir. int türünün yaklaşık ±2,1 milyarlık sınırı ve tam sayı bölmesi (7 / 2 = 3) sık rastlanan hata kaynaklarıdır.

NEDEN ÖNEMLİ

Kod izleme slaytlarının hepsi bu yapıları kullanır; ikili aramadaki klasik taşma hatası da int sınırından kaynaklanır.

KULLANILDIĞI YERLER **Bölüm 2 · Eklemeli: Kod** **Bölüm 1.2 · Adım Sayma**

Bölüm 3 · Uygulama Ayrıntıları

KONTROL SORUSU

▶ **int a = 7, b = 2; double d = a / b; d'nin değeri nedir?**

ÖRNEK (JAVA)

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) sum += i;       // çift sayılar toplanır
}
System.out.println(sum);            // 30
long big = 50_000L * 50_000;        // int olsaydı taşardı
System.out.println(big);            // 2500000000
```

[ ] Diziler ve iki boyutlu diziler

0'dan başlayan indis · length · int[][]

Sıralama, arama ve dinamik programlama tablolarının hepsi diziler üzerinde çalışır. İndislerin 0'dan length − 1'e kadar gittiği, bir dizinin metoda referansla geçtiği ve iki boyutlu bir dizide satır satır gezinildiği bilinmelidir.

NEDEN ÖNEMLİ

Sıralama algoritmaları dizi üzerinde yerinde çalışır, DP tabloları iki boyutlu dizidir, komşuluk matrisi de iki boyutlu bir dizidir.

KULLANILDIĞI YERLER **Bölüm 2 · Kabarcık: Kod** **Bölüm 8 · Tablo Doldurma İzleme**

Bölüm 4.1 · Çizgeyi Bellekte Tutmak

KONTROL SORUSU

▶ **int[] a = {3, 1, 2}; satırından sonra a[3] okunursa ne olur?**

ÖRNEK (JAVA)

```
int[] a = new int[5];               // [0, 0, 0, 0, 0]
a[a.length - 1] = 9;                // son öğe
int[][] dp = new int[3][4];         // 3 satır, 4 sütun
for (int i = 0; i < dp.length; i++)
    for (int j = 0; j < dp[i].length; j++)
        dp[i][j] = i + j;
System.out.println(dp[2][3]);       // 5
```

Aa Dizgiler ve karakterler

String · charAt · length() · char kodları · compareTo

Dizgi algoritmalarının Java kodları String nesneleri üzerinde çalışır. s.length() uzunluğu, s.charAt(i) i. karakteri verir; indisler dizilerde olduğu gibi 0'dan başlar. String değiştirilemez (immutable): s + c her seferinde yeni bir nesne kurar, bu yüzden döngü içinde dizgi biriktirmek için

StringBuilder kullanılır. char türü 16 bitlik bir tam sayıdır; ch - 'a' bir küçük harfi 0–25 aralığında bir indise çevirir ve bir karakter, sayma dizisinde doğrudan indis olarak kullanılabilir. İki dizginin içeriği == ile değil equals ile, sözlük sırası ise compareTo ile karşılaştırılır.

NEDEN ÖNEMLİ

Kaba kuvvet, KMP, Boyer-Moore ve Rabin-Karp kodları charAt ile karakter karşılaştırır; trie düğümleri çocuğunu ch - 'a' ile seçer, hash fonksiyonu ve basamak sıralaması karakter kodlarını sayı gibi kullanır.

KULLANILDIĞI YERLER **Bölüm 5.1 · Kaba Kuvvet: İzleme** **Bölüm 5.3 · Ekleme Algoritması**

Bölüm 3 · Hash Fonksiyonu **Bölüm 5.2 · Örnek: apple · apricot**

KONTROL SORUSU

▶ **"kedi".compareTo("kek") negatif mi, pozitif mi döner?**

ÖRNEK (JAVA)

```
String s = "algoritma";
System.out.println(s.length());                // 9
System.out.println(s.charAt(0));               // a
int idx = s.charAt(2) - 'a';                   // 'g' → 6
StringBuilder sb = new StringBuilder();
for (int i = s.length() - 1; i >= 0; i--) sb.append(s.charAt(i));
System.out.println(idx + " " + sb);            // 6 amtirogla
System.out.println("abc".equals("ab" + "c"));  // true
```

f( ) Metotlar ve parametre aktarımı

dönüş değeri · değerle aktarım · dizi referansı

Algoritmalar genellikle metot olarak yazılır: girdi parametredir, sonuç dönüş değeridir. Java her şeyi değerle aktarır; ancak diziler ve nesneler için kopyalanan değer bir referanstır, bu yüzden metot içindeki bir takas çağıranın dizisini değiştirir.

NEDEN ÖNEMLİ

Bütün sıralama algoritmaları öğeleri takas eden ya da kaydıran yardımcı metotlar kullanır; bölümleme metodu pivotun son konumunu döndürür.

KULLANILDIĞI YERLER **Bölüm 2 · Kabarcık: Kod** **Bölüm 2 · Bölümleme Prosedürü**

KONTROL SORUSU

▶ **Metot içinde a = new int[10]; yazmak çağıranın dizisini değiştirir mi?**

ÖRNEK (JAVA)

```
static void swap(int[] a, int i, int j) {
    int t = a[i]; a[i] = a[j]; a[j] = t;     // çağıranın dizisi değişir
}
static void increment(int x) { x++; }        // çağıranın değişkeni değişmez

int[] d = {5, 7};
swap(d, 0, 1);               // d = [7, 5]
int y = 1;
increment(y);                // y hâlâ 1
System.out.println(d[0] + " " + y);   // 7 1
```

↻

Özyineleme ve çağrı yığını

taban durum · özyinelemeli çağrı · derinlik

Birleştirme sıralaması, hızlı sıralama, DFS, geri izleme ve böl ve fethet algoritmaları özyinelemelidir (recursion). Her çağrının kendi yerel değişkenleriyle çağrı yığınına (call stack) eklendiği, taban durum (base case) unutulursa StackOverflowError oluştuğu ve özyineleme derinliğinin bellek tükettiği bilinmelidir.

NEDEN ÖNEMLİ

Hızlı üs alma, birleştirme sıralaması, özyinelemeli DFS ve geri izleme şablonu doğrudan özyinelemeyle yazılır.

KULLANILDIĞI YERLER **Bölüm 6 · Hızlı Üs Alma: İzleme**

Bölüm 4.1 · Özyinelemeli DFS: Sözde Kod **Bölüm 9 · Genel Şablon**

KONTROL SORUSU

▶ **power(2, 10) çağrısında, ilk çağrı da sayılırsa power metodu kaç kez çalışır?**

ÖRNEK (JAVA)

```
static int power(int a, int n) {          // a üzeri n, n >= 0
    if (n == 0) return 1;                    // taban durum
    int half = power(a, n / 2);              // problem yarıya iner
    return (n % 2 == 0) ? half * half : half * half * a;
}

System.out.println(power(2, 10));         // 1024; derinlik yaklaşık log2(n)
```

NYP

Sınıflar, nesneler ve karşılaştırıcılar

class · Comparable · Comparator · generics

Kenar, düğüm, iş ya da eşya gibi kavramlar küçük sınıflarla gösterilir. Sıralama ve öncelikli kuyruklar öğeleri karşılaştırmak için bir compareTo metoduna ya da bir Comparator'a gerek duyar: Kruskal'da kenarlar ağırlığa göre, etkinlik seçiminde etkinlikler bitiş zamanına göre sıralanır.

NEDEN ÖNEMLİ

Açgözlü algoritmaların çoğu bir sıralamayla başlar; Dijkstra ve Prim öncelikli kuyruğa karşılaştırılabilir nesneler koyar.

KULLANILDIĞI YERLER **Bölüm 4.3 · Kruskal: Adımlar ve Sözde Kod**

Bölüm 7 · Etkinlik Seçimi Problemi

KONTROL SORUSU

▶ **compareTo içinde return w - o.w; yazmak neden risklidir?**

ÖRNEK (JAVA)

```
static class Edge implements Comparable<Edge> {
    final int u, v, w;
    Edge(int u, int v, int w) { this.u = u; this.v = v; this.w = w; }
    public int compareTo(Edge o) { return Integer.compare(w, o.w); }
}

List<Edge> edges = new ArrayList<>();
edges.add(new Edge(0, 1, 7));
edges.add(new Edge(1, 2, 3));
Collections.sort(edges);                          // ağırlığa göre artan
edges.sort((a, b) -> Integer.compare(b.w, a.w));  // azalan: Comparator
System.out.println(edges.get(0).w);               // 7
```

⊞ Java koleksiyonları

ArrayList · HashMap · ArrayDeque · PriorityQueue

Sunumlardaki Java kodları hazır yapıları sık kullanır: komşuluk listeleri için ArrayList, BFS kuyruğu ve DFS yığını için ArrayDeque, Dijkstra ve Prim için PriorityQueue, bellekleme için HashMap. Her temel işlemin maliyetini bilmek, bir algoritmanın toplam çalışma süresini doğru hesaplamanın ön koşuludur.

NEDEN ÖNEMLİ

BFS bir kuyrukla, Dijkstra bir öncelikli kuyrukla, bellekleme bir eşlemeyle (map) yazılır.

KULLANILDIĞI YERLER **Bölüm 4.1 · BFS: Sözde Kod** **Bölüm 4.2 · Dijkstra: Java İzleme**

Bölüm 8 · Bellekleme İzleme

KONTROL SORUSU

▶ **PriorityQueue'da add ve poll ne kadar sürer? contains(x) ve remove(x) ne kadar sürer?**

ÖRNEK (JAVA)

```
Deque<Integer> queue = new ArrayDeque<>();
queue.offer(1); queue.offer(2);
int first = queue.poll();                          // 1 (ilk giren ilk çıkar)

PriorityQueue<int[]> pq = new PriorityQueue<>((x, y) -> Integer.compare(x[1],
y[1]));
pq.add(new int[] {7, 40});
pq.add(new int[] {3, 15});
int[] smallest = pq.poll();                        // [3, 15]

Map<Integer, Long> memo = new HashMap<>();
memo.put(10, 55L);
System.out.println(first + " " + smallest[0] + " " + memo.get(10));   // 1 3
55
```

≡ Sözde kodu okuma ve izleme

← atama · 1'den başlayan indis · izleme tablosu

Sunumlar ve ders kitapları algoritmaları çoğu zaman sözde kodla (pseudocode) verir: ← atamadır, diziler 1'den başlayabilir ve “for j ← 2 to n” son değeri de kapsar. Java'ya çevirirken bu farkları gözetmek ve her değişkenin her adımdaki değerini bir izleme tablosuna yazabilmek gerekir.

NEDEN ÖNEMLİ

Çizge, akış ve dal sınır algoritmaları önce sözde kodla, sonra Java ile verilir.

KULLANILDIĞI YERLER **Bölüm 2 · Eklemeli: Kod** **Bölüm 4.2 · Dijkstra: Sözde Kod**

Bölüm 4.4 · Sözde Kod

KONTROL SORUSU

▶ **Yukarıdaki sözde kod 0'dan başlayan indisli Java'ya çevrilince döngüler nasıl yazılır?**

ÖRNEK

```
INSERTION-SORT(A, n)
  for j ← 2 to n
      key ← A[j]
      i ← j – 1
      while i > 0 and A[i] > key
          A[i + 1] ← A[i]
```

```
        i ← i − 1
    A[i + 1] ← key
```

✓ Test ve hata ayıklama

uç durumlar · doğrulamalar · küçük girdiler

Doğru bir fikir de özenle gerçeklenip test edilmelidir. Boş girdi, tek öğe, tekrarlanan değerler, sıralı ve ters sıralı diziler, bağlantısız çizgeler ve taşmaya yol açabilecek büyük değerlerle test etmek alışkanlık olmalıdır.

NEDEN ÖNEMLİ

Her bölümün "sık yapılan hatalar" slaytı bu tür uç durumları toplar.

KULLANILDIĞI YERLER **Bölüm 2 · Sık Yapılan Hatalar** **Bölüm 3 · Uygulama Ayrıntıları**

KONTROL SORUSU

▶ **İkili aramada mid = (low + high) / 2 ne zaman yanlış sonuç verir?**

ÖRNEK (JAVA)

```
int[] empty = {}, one = {4}, equal = {2, 2, 2}, rev = {5, 4, 3, 2, 1};
for (int[] t : new int[][] {empty, one, equal, rev}) {
    int[] k = t.clone();
    Arrays.sort(k);
    for (int i = 1; i < k.length; i++)
        if (k[i - 1] > k[i]) throw new AssertionError("sıralı değil");
}
System.out.println("tüm testler geçti");
```

Temel veri yapıları

Algoritmaların üzerinde çalıştığı yapılar ve temel işlemlerinin maliyeti. Bu yapılar önkoşul olan Veri Yapıları dersinde ayrıntılı olarak işlenir.

→ Bağlı listeler

düğüm · next · başa O(1) ekleme

Dizi gibi bir sırayı saklar, ancak öğeler bellekte bitişik değildir; her düğüm bir sonrakini gösterir. Başa ekleme O(1), i. öğeye erişim O(n) sürer. Komşuluk listeleri, hash tablolarındaki zincirleme ve LRU önbelleğinin O(1) gerçeklemesi bağlı listelere dayanır.

NEDEN ÖNEMLİ

Zincirlemeli hash tabloları ve O(1) LRU önbelleği bağlı liste kullanır.

KULLANILDIĞI YERLER **Bölüm 3 · Zincirleme ile Ekleme** **Bölüm 13 · LRU: O(1) Uygulama**

KONTROL SORUSU

▶ **Tek yönlü bağlı listede son öğeyi silmek neden O(n) sürer?**

ÖRNEK (JAVA)

```
static class Node {
    int value; Node next;
    Node(int value, Node next) { this.value = value; this.next = next; }
}

Node head = null;
for (int x = 1; x <= 3; x++) head = new Node(x, head);    // 3 → 2 → 1
for (Node n = head; n != null; n = n.next) System.out.print(n.value + " ");
```

⇅ Yığınlar ve kuyruklar

LIFO · FIFO · push/pop · offer/poll

Yığın (stack) son giren ilk çıkar (LIFO), kuyruk (queue) ilk giren ilk çıkar (FIFO) ilkesiyle çalışır; ikisinde de temel işlemler O(1) sürer. DFS bir yığınla (ya da özyinelemenin çağrı yığınıyla), BFS bir kuyrukla çalışır; geri izleme seçimleri yığın sırasıyla geri alır.

NEDEN ÖNEMLİ

Yinelemeli DFS açık bir yığın kullanır; BFS ve Edmonds-Karp kuyruk kullanır.

KULLANILDIĞI YERLER **Bölüm 4.1 · Yığınla DFS İzleme** **Bölüm 4.1 · BFS: Fikir**

Bölüm 4.4 · BFS ile Artış Yolu (Java)

KONTROL SORUSU

▶ **1, 2, 3 bu sırayla hem bir kuyruğa hem bir yığına ekleniyor. Her birinden ilk hangi öğe çıkar?**

ÖRNEK (JAVA)

```
Deque<Character> stack = new ArrayDeque<>();
for (char c : "abc".toCharArray()) stack.push(c);
StringBuilder sb = new StringBuilder();
while (!stack.isEmpty()) sb.append(stack.pop());
System.out.println(sb);                    // cba
```

▲ Heap ve öncelikli kuyruklar

tam ikili ağaç · dizi yerleşimi · O(log n)

Bir min-heap'te her düğüm çocuklarından küçüktür, bu yüzden en küçük öğe köktedir. Dizi olarak saklandığında i. düğümün çocukları 2i + 1 ve 2i + 2'de, ebeveyni (i − 1) / 2'dedir. Ekleme ve en küçüğü çıkarma O(log n) sürer; heap sort, Dijkstra, Prim, Huffman ve çevrim içi ortanca bu yapıya dayanır.

NEDEN ÖNEMLİ

Heap sort, Dijkstra'nın karmaşıklığı, Huffman ağacının kurulması ve çevrim içi ortanca heap'e dayanır.

KULLANILDIĞI YERLER **Bölüm 2 · Heap Sort** **Bölüm 4.2 · Dijkstra: Karmaşıklık** **Bölüm 7 · Huffman: Adımlar** **Bölüm 13 · Çevrim İçi Ortanca**

KONTROL SORUSU

▶ **n öğeyi tek tek ekleyerek heap kurmak O(n log n) sürer. Aşağıdan yukarı heap kurma ne kadar sürer?**

ÖRNEK

```
dizi:   [2, 5, 3, 9, 6, 4]
indis:   0  1  2  3  4  5
2'nin çocukları: 5 ve 3   (indis 1, 2)
5'in çocukları:  9 ve 6   (indis 3, 4)
3'ün çocuğu:     4        (indis 5)
```

Hash tabloları

hash fonksiyonu · çakışma · ortalama O(1)

Hash tablosu bir anahtarı hash fonksiyonuyla (hash function) dizi indisine dönüştürür; ekleme ve arama ortalamada O(1) sürer. İki anahtar aynı hücreye düştüğünde çakışma (collision) zincirlemeyle ya da doğrusal yoklamayla çözülür; yük faktörü (load factor) arttıkça performans düşer. Bellekleme, “ziyaret edildi” kümeleri ve Rabin-Karp'ın hash karşılaştırması bu fikre dayanır.

NEDEN ÖNEMLİ

Hash ile arama, Rabin-Karp ve bellekleme hash tablosu kullanır.

KULLANILDIĞI YERLER **Bölüm 3 · Hash ile Arama: Fikir** **Bölüm 3 · Yük Faktörü** **Bölüm 5.1 · Rabin-Karp**

KONTROL SORUSU

▶ **Hash tablosunda en kötü durumdaki arama süresi nedir ve ne zaman ortaya çıkar?**

ÖRNEK (JAVA)

```
int m = 7;
int[] keys = {10, 17, 24};
for (int k : keys) System.out.print(k % m + " ");   // 3 3 3 → çakışma
Set<String> visited = new HashSet<>();
visited.add("A");
System.out.println(visited.contains("A"));          // true
```

A

Ağaçlar ve ikili arama ağaçları

kök · ebeveyn · çocuk · yaprak · yükseklik

n düğümlü bir ağacın n − 1 kenarı vardır ve herhangi iki düğüm arasında tam olarak bir yol bulunur. Kök, ebeveyn, çocuk, yaprak, derinlik ve yükseklik terimleri DFS/BFS ağaçlarında, MST'lerde, union-find yapısında, trie'lerde ve durum uzayı ağaçlarında aynen kullanılır. İkili arama ağacı küçük anahtarları sol alt ağaçta, büyük anahtarları sağ alt ağaçta tutar; ağaç dengeliyse arama O(log n) sürer.

NEDEN ÖNEMLİ

BFS ağacı, kapsayan ağaçlar, trie'ler ve durum uzayı ağaçları aynı terimlerle anlatılır.

KULLANILDIĞI YERLER **Bölüm 4.1 · BFS Ağacı ve En Kısa Yol**

Bölüm 4.3 · Kapsayan Ağaç ve MST **Bölüm 5.3 · Temel Bileşenler**

Bölüm 9 · Durum Uzayı Ağacı

KONTROL SORUSU

▶ **Yüksekliği h olan bir ikili ağaçta en çok kaç düğüm bulunabilir?**

ÖRNEK

```
        8            yükseklik 2; yapraklar 1, 6, 14
      /   \
     3     10        3'ün ebeveyni 8, çocukları 1 ve 6
    / \      \
   1   6      14     ortada sıralı (inorder) gezinme: 1 3 6 8 10 14
```

◇

Çizge gösterimleri

G = (V, E) · komşuluk listesi · komşuluk matrisi

Çizge düğümlerden ve kenarlardan oluşur; kenarlar yönlü ya da yönsüz, ağırlıklı ya da ağırlıksız olabilir. Komşuluk matrisi (adjacency matrix) V^2 bellek kullanır ve bir kenar sorgusunu O(1) sürede yanıtlar; komşuluk listesi (adjacency list) O(V + E) bellek kullanır ve seyrek çizgelerde ge-

zinmeyi O(V + E) sürede yapar. Derece, yol, döngü, bağlantılılık ve iki parçalılık terimleri tanıdık olmalıdır.

NEDEN ÖNEMLİ
Bütün çizge bölümleri ve dal sınırdaki TSP örnekleri bir çizge gösterimini varsayar.

KULLANILDIĞI YERLER **Bölüm 4.1 · Çizgeyi Bellekte Tutmak** **Bölüm 4.1 · Çizge Türleri**

Bölüm 4.2 · Çizge ve En Kısa Yol

KONTROL SORUSU

▶ **10.000 düğümlü ve 30.000 kenarlı bir yol ağına hangi gösterim uygundur?**

ÖRNEK (JAVA)

```
int V = 4;
List<List<Integer>> adj = new ArrayList<>();
for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
int[][] edges = {{0, 1}, {0, 2}, {2, 3}};
for (int[] e : edges) {                    // yönsüz: iki yön de eklenir
    adj.get(e[0]).add(e[1]);
    adj.get(e[1]).add(e[0]);
}
System.out.println(adj.get(0));            // [1, 2]
```

Ayrık matematik ve ispat

Algoritmaları tanımlamak ve doğruluklarını kanıtlamak için kullanılan dil.

∈ Kümeler, fonksiyonlar ve bağıntılar

∪ ∩ \ · |A| · f: A → B · bağıntı

Algoritmalar küme diliyle anlatılır: G = (V, E), ziyaret edilmiş düğümler kümesi, MST'deki (S, V − S) kesimi, sırt çantasında seçilen eşyaların alt kümesi. Birleşim, kesişim, fark, Kartezyen çarpım ve alt kümeler; bir fonksiyonun tanım ve değer kümesi; bağıntıların yansıma, simetri ve geçişme özellikleri bilinmelidir.

NEDEN ÖNEMLİ
Kesim özelliği, Floyd-Warshall'ın geçişli kapanışı ve dal sınırdaki olurlu çözümler kümelerle tanımlanır.

KULLANILDIĞI YERLER **Bölüm 4.3 · Kesim ve Döngü Özelliği**

Bölüm 4.2 · Floyd-Warshall Algoritması **Bölüm 10 · Temel Kavramlar**

KONTROL SORUSU

▶ **n öğeli bir kümenin kaç alt kümesi vardır ve hangi kaba kuvvet yöntemi bu kadar maliyetlidir?**

ÖRNEK

```
A = {1, 2, 3},  B = {2, 3, 4}
A ∪ B = {1, 2, 3, 4}    A ∩ B = {2, 3}    A \ B = {1}
|A × B| = 3 · 3 = 9     A'nın alt küme sayısı: 2³ = 8
"ulaşılabilir" bağıntısı yansıyan ve geçişlidir; yönsüz çizgede simetrik de
olur.
```

∀

Mantık ve niceleyiciler

∧ ∨ ¬ ⇒ · ∀ ∃ · De Morgan

Döngü koşulları, budama koşulları ve tanımlar mantıksal önermelerdir. “Her $n \geq n_0$ için $f(n) \leq c \cdot g(n)$” gibi niceleyicili cümleleri okuyabilmek, bir önermenin değilini doğru almak (De Morgan) ve “p ⇒ q” ile “q ⇒ p” önermelerini ayırt edebilmek gerekir.

NEDEN ÖNEMLİ

Büyük O niceleyicilerle tanımlanır; eklemeli sıralamanın iç döngüsü bileşik bir koşulda durur.

KULLANILDIĞI YERLER **Bölüm 1.2 · Büyük-O Tanımı** **Bölüm 2 · Eklemeli: Kod**

KONTROL SORUSU

▶ **“Her düğümün en az bir komşusu vardır” önermesinin değili nedir?**

ÖRNEK

```
¬(a ∧ b)  ≡  ¬a ∨ ¬b                          (De Morgan)
while (i > 0 && A[i] > x) döngüsünden çıkış koşulu:  i <= 0 || A[i] <= x
f(n) = O(g(n))  ⇔  ∃c > 0, ∃n₀, ∀n ≥ n₀ : f(n) ≤ c · g(n)
```

∴

İspat teknikleri: doğrudan ispat, çelişki, karşı örnek

doğrudan ispat · çelişkiyle ispat · karşı örnek

Bir algoritmanın her girdide doğru çalıştığını ispatla, yanlış olduğunu ise tek bir karşı örnekle gösteririz. Açgözlü algoritmaların doğruluğu çoğunlukla çelişkiyle ya da değiştirme argümanıyla (exchange argument) ispatlanır; “açgözlü para üstü her zaman en iyisini verir” iddiası ise bir karşı örnekle çürür.

NEDEN ÖNEMLİ

Açgözlü seçimlerin doğruluğu, MST'nin kesim özelliği ve 2-yaklaşım oranları ispatla gösterilir.

KULLANILDIĞI YERLER **Bölüm 7 · Para Üstü Problemi** **Bölüm 7 · Neden En Erken Biten?** **Bölüm 11 · 2-Yaklaşım Kanıtı**

KONTROL SORUSU

▶ **$\sqrt{2}$'nin irrasyonel olduğunun ispatında hangi teknik kullanılır?**

ÖRNEK

```
İddia: açgözlü para üstü her madeni para sisteminde en az sayıda para verir.
Karşı örnek: paralar {1, 3, 4}, tutar 6
  açgözlü: 4 + 1 + 1  → 3 para
  en iyi:  3 + 3      → 2 para        ⇒ iddia yanlıştır.
```

n+1

Tümevarım ve döngü değişmezleri

taban durum · tümevarım adımı · başlangıç–koruma–sonlanma

Tümevarım, P(1)'in doğru olduğunu ve P(k) ⇒ P(k + 1) olduğunu göstererek bir önermeyi her n için kanıtlar. Döngü değişmezi (loop invariant) bunun algoritmalara uyarlanmış biçimidir: döngüden önce doğru olan (başlangıç), her yinelemede korunan (koruma) ve döngü bitince sonucu veren (sonlanma) bir önermedir. Özyinelemeli algoritmaların doğruluğu da tümevarımla ispatlanır.

NEDEN ÖNEMLİ

Sıralama algoritmalarının doğruluğu ve yineleme bağıntılarının çözümleri tümevarımla gösterilir.

KULLANILDIĞI YERLER **Bölüm 2 · Eklemeli: Adımlar** **Bölüm 1.2 · f3: Bağıntıyı Açma**

KONTROL SORUSU

▶ **$1 + 3 + 5 + \ldots + (2n - 1) = n^2$ eşitliği tümevarımla nasıl gösterilir?**

ÖRNEK

```
Eklemeli sıralama, değişmez: "j. yinelemenin başında A[0..j-1] sıralıdır."
  Başlangıç: j = 1 → A[0..0] tek öğedir, sıralıdır.
  Koruma:    A[j] yerine kaydırılır → A[0..j] sıralıdır.
  Sonlanma:  j = n → A[0..n-1], yani dizinin tamamı sıralıdır.
```

Analiz araçları

Çalışma sürelerini hesaplamak için toplamlar, logaritmalar ve büyüme hızları.

Σ Toplam gösterimi ve seriler

aritmetik · geometrik · harmonik

İç içe döngülerin ve özyineleme ağaçlarının maliyeti bir toplam olarak yazılır. Üç seri temeldir: aritmetik seri karesel büyür; oranı 1'den küçük geometrik seri bir sabite yakınsar, oranı 1'den büyük olan ise en büyük teriminin mertebesindedir; harmonik seri logaritmik büyür.

NEDEN ÖNEMLİ

Döngü sayma, özyineleme ağacının düzeylerini toplama ve rastgele hızlı sıralamanın beklenen karşılaştırma sayısı bu serileri kullanır.

KULLANILDIĞI YERLER **Bölüm 1.2 · f2: Toplam** **Bölüm 1.2 · Döngü Kalıpları**

Bölüm 12 · Beklenen Karşılaştırma

KONTROL SORUSU

▶ **for (i = 1; i < n; i *= 2) for (j = 0; j < i; j++) döngüsünün gövdesi kaç kez çalışır?**

ÖRNEK

```
Σ (i = 1..n) i       = n(n + 1) / 2          = Θ(n²)
Σ (i = 0..k) 2^i     = 2^(k+1) − 1             (1 + 2 + 4 + … + 2^k)
Σ (i = 0..∞) (1/2)^i = 2
H(n) = Σ (i = 1..n) 1/i ≈ ln n               = Θ(log n)
```

log Logaritma ve üs kuralları

$\log_2$ n · üs kuralları · taban değiştirme

Girdisini yarıya indiren her algoritma (ikili arama, birleştirme sıralaması, heap) logaritmik derinlik üretir: n'yi 1'e indirmek $\log_2$ n kez yarılamak demektir. Üs ve logaritma kuralları, logaritma tabanının asimptotik olarak önemsiz olduğu ve $2^{10} \approx 10^3$ yaklaşımı bilinmelidir.

NEDEN ÖNEMLİ

İkili aramanın adım sayısı, ana teoremdeki n^(log_b a) terimi ve Karatsuba'nın n^1,585 üssü logaritmayla hesaplanır.

KULLANILDIĞI YERLER **Bölüm 3 · Kaç Adım?** **Bölüm 1.2 · Üç Durum**

Bölüm 6 · Karatsuba Analizi

KONTROL SORUSU

▶ **Bir milyar öğeli sıralı bir dizide ikili arama en çok yaklaşık kaç karşılaştırma yapar?**

ÖRNEK

```
log2(1024) = 10        log2(10^6) ≈ 20       log2(10^9) ≈ 30
a^(m+n) = a^m · a^n     (a^m)^n = a^(m·n)     log(x·y) = log x + log y
n^(log_b a) = a^(log_b n)              ana teoremde karşımıza çıkar
log_3 n = log_2 n / log_2 3            taban yalnız sabit bir çarpanı
değiştirir
```

⌊⌋ Taban ve tavan

⌊x⌋ · ⌈x⌉ · tam sayı bölmesi

Bir dizi ikiye bölündüğünde yarılarda ⌊n/2⌋ ve ⌈n/2⌉ öğe bulunur; Java'da n / 2 pozitif sayılar için tabanı verir. Taban ve tavan ikili aramanın orta indisinde, heap'teki ebeveyn indisinde, ikili ağacın yüksekliğinde ve yaklaşım oranlarında karşımıza çıkar.

NEDEN ÖNEMLİ

Birleştirme sıralamasının bölme adımı, ikili aramanın orta indisi ve heap'in indis hesapları taban ve tavanı kullanır.

KULLANILDIĞI YERLER **Bölüm 2 · Merge Sort: Özyineleme**

Bölüm 3 · İkili Arama Algoritması **Bölüm 2 · Heap Sort: Örnek**

KONTROL SORUSU

▶ **Java'da −7 / 2 kaçtır? ⌊−7/2⌋ kaçtır?**

ÖRNEK

```
⌊7/2⌋ = 3      ⌈7/2⌉ = 4      ⌊n/2⌋ + ⌈n/2⌉ = n
n düğümlü heap'in yüksekliği ⌊log2 n⌋:  n = 10 → 3
Java: 7 / 2 == 3,  (7 + 2 − 1) / 2 == 4     pozitif a, b için: ⌈a/b⌉ = (a + b
− 1) / b
```

O Asimptotik büyüme

$1 < \log n < n < n \log n < n^2 < 2^n < n!$

Büyük n için baskın terim belirleyicidir: $3n^2 + 100n + 7 = \Theta(n^2)$. Büyüme sınıflarını sıralayabilmek ve saniyede kabaca 10^8–10^9 basit işlem varsayarak hangi karmaşıklığın hangi n için uygulanabilir olduğunu kestirebilmek gerekir. O üst sınır (upper bound), Ω alt sınır (lower bound), Θ ise sıkı sınırdır (tight bound).

NEDEN ÖNEMLİ

Her bölüm algoritmaları karmaşıklıklarına göre karşılaştırır; büyüme sınıfları hangi yöntemin hangi girdi boyutuna uygun olduğunu belirler.

KULLANILDIĞI YERLER **Bölüm 1.2 · Büyüme Eğrileri** **Bölüm 1.2 · Asimptotik Kurallar**

Bölüm 1.2 · Yaygın Büyüme Sınıfları

KONTROL SORUSU

▶ **Sonunda hangisi daha hızlı büyür: n^{100} mü, $1{,}01^n$ mi?**

ÖRNEK

```
n = 10^6 için yaklaşık işlem sayıları
  n log2 n ≈ 2 · 10^7     → bir saniyenin altında
  n²       = 10^12        → saatler
  2^n      : n = 60 bile ≈ 10^18 eder → uygulanamaz
```

T(n) Yineleme bağıntıları

T(n) = aT(n/b) + f(n) · özyineleme ağacı

Özyinelemeli bir algoritmanın çalışma süresi kendi cinsinden yazılır: birleştirme sıralaması iki yarıyı çözer ve doğrusal sürede birleştirir, bu yüzden T(n) = 2T(n/2) + n olur. Bir bağıntıyı açarak ya da özyineleme ağacının her düzeyindeki işi toplayarak çözmeye alışık olmak, ana teoremi izlemeyi çok kolaylaştırır.

NEDEN ÖNEMLİ

Yineleme bağıntıları ana teoremle çözülür; her böl ve fethet algoritması bir bağıntıyla analiz edilir.

KULLANILDIĞI YERLER **Bölüm 1.2 · Yineleme Bağıntısı** **Bölüm 1.2 · f4: Özyineleme Ağacı**

Bölüm 1.2 · Böl ve Fethet Bağıntısı

KONTROL SORUSU

▶ **İkili aramanın yineleme bağıntısı ve çözümü nedir?**

ÖRNEK

```
T(n) = 2T(n/2) + n
  düzey 0:  n                  = n
  düzey 1:  n/2 + n/2          = n
  düzey 2:  4 · n/4            = n
  … log2 n düzey               ⇒ T(n) = Θ(n log n)
T(n) = T(n − 1) + 1  ⇒ Θ(n)      T(n) = 2T(n − 1) + 1  ⇒ Θ(2^n)
```

Olasılık ve sayma

Arama uzaylarının büyüklüğünü ve rastgele algoritmaların davranışını ölçmek için.

n!

Kombinatorik: permütasyonlar ve kombinasyonlar

çarpma kuralı · n! · C(n, k) · 2^n

Kaba kuvvetin, geri izlemenin ve dal sınırın maliyeti arama uzayının büyüklüğüdür; bunu saymak için çarpma kuralı, n! permütasyon, C(n, k) kombinasyon ve 2^n alt küme gerekir. Örneğin n şehirli TSP'de başlangıç şehri sabitlenince (n − 1)! tur vardır.

NEDEN ÖNEMLİ

Permütasyon üretme, TSP'nin kaba kuvvet çözümü ve n-vezir arama ağacının büyüklüğü sayarak bulunur.

KULLANILDIĞI YERLER **Bölüm 1.1 · O(n!): Permütasyonlar**

Bölüm 11 · Kaba Kuvvetle Karşılaştırma **Bölüm 9 · Kaba Kuvvet ve Budama**

KONTROL SORUSU

▶ **8 × 8 bir tahtaya her satırda bir vezir olacak biçimde 8 vezir kaç yolla yerleştirilir? Sütunlar da farklı olmak zorundaysa kaç yolla?**

ÖRNEK

```
P(n, k) = n! / (n − k)!          C(n, k) = n! / (k! · (n − k)!)
C(5, 2) = 10       5! = 120        2^10 = 1024
n = 10 şehir: 9! = 362.880 tur      n = 20 şehir: 19! ≈ 1,2 · 10^17 tur
```

P

Olasılığın temelleri

örnek uzay · bağımsızlık · koşullu olasılık

Rastgele algoritmaların başarı ve hata olasılıklarını hesaplamak için örnek uzay, olay, tümleyen, bağımsız olaylar ve koşullu olasılık gerekir. Önemli bir kalıp: tek bir deneme p olasılıkla başarısız oluyorsa, k bağımsız denemenin hepsi p^k olasılıkla başarısız olur; Monte Carlo algoritmalarının hatası böyle küçültülür.

NEDEN ÖNEMLİ

Miller–Rabin testinin ve Karger algoritmasının hata analizi tam olarak bu kalıbı kullanır.

KULLANILDIĞI YERLER **Bölüm 12 · Olasılık Hatırlatması**

Bölüm 12 · Hata Olasılığını Küçültme **Bölüm 12 · Karger Minimum Kesim**

KONTROL SORUSU

▶ **Bir algoritma tek çalıştırmada doğru yanıtı 1/2 olasılıkla buluyor. 10 bağımsız çalıştırmadan en az birinin doğru yanıtı bulma olasılığı nedir?**

ÖRNEK

```
P(A ∪ B) = P(A) + P(B) − P(A ∩ B)
A ve B bağımsız  ⇔  P(A ∩ B) = P(A) · P(B)
Miller–Rabin: tur başına hata ≤ 1/4  ⇒  20 turdan sonra ≤ (1/4)^20 ≈ 9 ·
10^−13
```

E

Beklenen değer ve beklenen değerin doğrusallığı

rastgele değişken · E[X] · gösterge değişkeni

Rastgele hızlı sıralamanın beklenen O(n log n) çalışma süresi gibi sonuçlar beklenen değer (expected value) olarak ifade edilir. Beklenen değerin doğrusallığı, E[X + Y] = E[X] + E[Y], değişkenler bağımlı olsa bile geçerlidir; karmaşık bir sayımı 0/1 gösterge değişkenlerinin toplamına ayırıp olasılıklarını toplamak standart bir tekniktir.

NEDEN ÖNEMLİ

Rastgele hızlı sıralamanın, QuickSelect'in ve rastgele sayfalamanın analizleri beklenen değerle yapılır.

KULLANILDIĞI YERLER **Bölüm 12 · Beklenen Karşılaştırma** **Bölüm 12 · QuickSort Analizi** **Bölüm 13 · Rastgele Sayfalama**

KONTROL SORUSU

▶ **Rastgele hızlı sıralamanın en kötü durumu $O(n^2)$'dir. Neden "beklenen O(n log n)" denir?**

ÖRNEK

```
Zar: E[X] = (1 + 2 + … + 6) / 6 = 3,5
n öğenin rastgele bir permütasyonunda yerinde kalan öğe sayısı X:
  X = X1 + … + Xn,  Xi = 1 (i. öğe yerinde kalırsa),  P(Xi = 1) = 1/n
  E[X] = n · (1/n) = 1
Başarı olasılığı p olan bir denemeyi ilk başarıya kadar tekrarlamak: E[deneme
sayısı] = 1/p
```

Diğer matematik ve kavramlar

Belirli bölümlerde kullanılan matrisler, modüler aritmetik, bit işlemleri, optimizasyon ve eş zamanlılığın temelleri.

M

Matris işlemleri

satır–sütun çarpımı · birim matris · blok biçim

İki n × n matrisin doğrudan çarpımı, iç içe üç döngüde n^3 çarpma yapar; Strassen bu sayıyı blok matrislerle azaltır. Komşuluk matrisleri, Floyd-Warshall'ın uzaklık tablosu ve kuvvet yinelemesindeki (power iteration) matris–vektör çarpımı için satır–sütun çarpımını, birim matrisi ve bir matrisi bloklara bölmeyi bilmek gerekir.

NEDEN ÖNEMLİ

Matris çarpımı karmaşıklık analizindeki $O(n^3)$ örneği, Strassen algoritmasının konusu ve kuvvet yinelemesinin temel adımıdır.

KULLANILDIĞI YERLER **Bölüm 1.1 · $O(n^3)$: Matris Çarpımı** **Bölüm 6 · Matris Çarpımı**

Bölüm 11 · Power Iteration **Bölüm 4.2 · Floyd-Warshall: Adımlar**

KONTROL SORUSU

▶ **[[1, 2], [3, 4]] · [[0, 1], [1, 0]] çarpımı nedir?**

ÖRNEK (JAVA)

```
static int[][] multiply(int[][] a, int[][] b) {
    int n = a.length;
    int[][] c = new int[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];    // toplam n³ çarpma
    return c;
}

int[][] c = multiply(new int[][] {{1, 2}, {3, 4}}, new int[][] {{0, 1}, {1,
0}});
System.out.println(Arrays.deepToString(c));      // [[2, 1], [4, 3]]
```

xy Düzlem geometrisi ve uzaklık ölçüleri

Öklid uzaklığı · Manhattan uzaklığı · üçgen eşitsizliği · çapraz çarpım

Düzlemdeki iki nokta arasındaki Öklid uzaklığı $\sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$, ızgarada yalnız yatay ve dikey adımlarla ölçülen Manhattan uzaklığı ise $|x_1 - x_2| + |y_1 - y_2|$'dir. Bir uzaklık ölçüsü üçgen eşitsizliğini sağlıyorsa d(a, c) ≤ d(a, b) + d(b, c) olur: üçüncü bir noktaya uğramak yolu hiçbir zaman kısaltmaz. A'dan B'ye ve A'dan P'ye giden vektörlerin çapraz çarpımı (B.x − A.x)(P.y − A.y) − (B.y − A.y)(P.x − A.x) tam sayılarla hesaplanır; işareti P'nin AB doğrusunun solunda mı, sağında mı, yoksa üzerinde mi olduğunu söyler. Uzaklıkları yalnız karşılaştırmak için karekök almaya gerek yoktur, karelerini karşılaştırmak aynı sonucu verir.

NEDEN ÖNEMLİ

A* algoritmasının sezgiselleri Öklid ve Manhattan uzaklığıdır; dışbükey örtü yönelim testine, en yakın nokta çifti Öklid uzaklığına, metrik TSP'nin 2-yaklaşımı üçgen eşitsizliğine dayanır.

KULLANILDIĞI YERLER **Bölüm 4.2 · Sezgisel Fonksiyonlar** **Bölüm 6 · Yönelim Testi** **Bölüm 6 · En Yakın Nokta Çifti** **Bölüm 11 · Metrik TSP**

KONTROL SORUSU

▶ **A = (0, 0) ve B = (3, 4) için Öklid ve Manhattan uzaklıkları nedir? P = (1, 5) noktası AB doğrusunun hangi tarafındadır?**

ÖRNEK (JAVA)

```
static double oklid(int x1, int y1, int x2, int y2) {
    return Math.hypot(x1 - x2, y1 - y2);
}
static int manhattan(int x1, int y1, int x2, int y2) {
    return Math.abs(x1 - x2) + Math.abs(y1 - y2);
}
static long capraz(int ax, int ay, int bx, int by, int px, int py) {
    return (long) (bx - ax) * (py - ay) - (long) (by - ay) * (px - ax);
}
System.out.println(oklid(0, 0, 3, 4));          // 5.0
System.out.println(manhattan(0, 0, 3, 4));      // 7
System.out.println(capraz(0, 0, 3, 4, 1, 5));   // 11 (P solda)
```

mod

Modüler aritmetik ve OBEB

a mod m · (a · b) mod m · OBEB · asal sayı

Hash fonksiyonları, Rabin-Karp'ın kayan hash'i, sözde rastgele sayı üreteçleri ve Miller–Rabin testi modüler aritmetikle çalışır. Toplama ve çarpmaların her adımında kalanı almanın sonucu değiştirmediğini, OBEB'in Öklid algoritmasıyla nasıl hesaplandığını ve asal sayının tanımını bilmek gerekir.

NEDEN ÖNEMLİ

Öklid algoritması, kayan hash, Miller–Rabin ve sözde rastgele sayılar modüler aritmetik kullanır.

KULLANILDIĞI YERLER **Bölüm 1.1 · Öklid Algoritması** **Bölüm 5.1 · Rabin-Karp: İşleyiş** **Bölüm 12 · Miller–Rabin: Kod**

KONTROL SORUSU

▶ **(17 · 23) mod 5 değeri, çarpmadan önce kalanlar alınarak nasıl hesaplanır?**

ÖRNEK (JAVA)

```
static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

System.out.println(gcd(48, 18));                          // 6
long M = 1_000_000_007L, h = 0;
for (char c : "abc".toCharArray()) h = (h * 31 + c) % M;  // kayan hash'in
temeli
System.out.println(h);                                     // 96354
```

01 İkili sayılar ve bit işlemleri

& | ^ ~ << >> · 2'nin kuvvetleri · alt küme maskeleri

Bilgisayarlar sayıları ikili tabanda saklar; 2'nin kuvvetleri, kaydırmalar ve maskeler pek çok algoritmayı kısaltır. Bir tam sayının bitleri bir alt kümeyi gösterebilir; 1 << k ifadesi 2^k değerini verir, x & (x − 1) ise en düşük 1 bitini sıfırlar. n-vezir probleminde O(1) güvenlik testi ve küçük n için bütün alt kümeler üzerinde gezinmek bit maskeleriyle yapılır.

NEDEN ÖNEMLİ

Basamak sıralaması basamak basamak çalışır; n-vezir çözümü bit maskeleri kullanır.

KULLANILDIĞI YERLER

Bölüm 9 · O(1) Güvenlik Testi **Bölüm 2 · Basamak Sıralama**

KONTROL SORUSU

▶ **13'ün ikili gösterimi nedir? 13 >> 1 ve 13 & 1 kaçtır?**

ÖRNEK (JAVA)

```
int n = 3;
for (int mask = 0; mask < (1 << n); mask++) {          // 2³ = 8 alt küme
    StringBuilder s = new StringBuilder("[");
    for (int i = 0; i < n; i++)
        if ((mask & (1 << i)) != 0) s.append(i);
    System.out.print(s.append("] "));
}
// [] [0] [1] [01] [2] [02] [12] [012]
System.out.println((16 & (16 - 1)) == 0);             // true: 16, 2'nin bir
kuvvetidir
```

opt Karar ve optimizasyon problemleri

amaç fonksiyonu · kısıt · olurlu çözüm · alt/üst sınır

Açgözlü, dinamik programlama, yaklaşım ve dal sınır algoritmaları optimizasyon problemlerini çözer: kısıtları sağlayan çözümler (olurlu çözümler) arasından amaç fonksiyonunu en büyük ya da en küçük yapanı ararız. Aynı problemin karar biçimi ("değeri en az K olan bir çözüm var mı?") P, NP ve NP-tamlık tartışmasında kullanılır. Bir enbüyükleme probleminde her olurlu çözümün değeri, optimum için bir alt sınırdır.

NEDEN ÖNEMLİ

Sırt çantası problemi, NP-zorluk, yaklaşım oranları ve dal sınırdaki sınırlar hep bu dille ifade edilir.

KULLANILDIĞI YERLER **Bölüm 8 · 0/1 Sırt Çantası Problemi** **Bölüm 9 · P, NP ve NP-Zor** **Bölüm 10 · Sınır Nasıl Seçilir?**

KONTROL SORUSU

▶ **Bir enbüyükleme probleminde açgözlü bir algoritma değeri 40 olan olurlu bir çözüm buluyor; kesirli gevşetmenin değeri 46. Optimum hangi aralıktadır?**

ÖRNEK

```
0/1 sırt çantası, kapasite W = 5
  eşyalar (ağırlık, değer): 1 = (2, 3), 2 = (3, 4), 3 = (4, 5)
  olurlu: {1, 2} → ağırlık 5, değer 7       olurlu değil: {2, 3} → ağırlık 7
  optimizasyon: en büyük değer nedir?               → 7
  karar: değeri ≥ 8 olan olurlu bir seçim var mı?   → hayır
```

‖ Eş zamanlılığın temelleri

iş parçacığı · paylaşılan değişken · eş zamanlama

Paralel algoritmalar bölümü, işin birkaç iş parçacığına (thread) nasıl bölündüğünü ve sonuçların nasıl birleştirildiğini Java ile gösterir. İş parçacığının ne olduğunu, aynı değişkeni korumasız güncelleyen iki iş parçacığının yarış durumuna (race condition) yol açtığını ve programın sıralı kısmının hızlanmayı sınırladığını sezgisel olarak bilmek yeterlidir.

NEDEN ÖNEMLİ

Yarış durumu, fork/join ile paralel sıralama ve Amdahl yasası bu fikirlerin üzerine kurulur.

KULLANILDIĞI YERLER **Bölüm 14 · Yarış Durumu** **Bölüm 14 · Fork/Join Adım Adım** **Bölüm 14 · Amdahl Yasası**

KONTROL SORUSU

▶ **Bir programın %20'si sıralı kalıyorsa, sınırsız sayıda işlemciyle elde edilebilecek en büyük hızlanma nedir?**

ÖRNEK (JAVA)

```
final AtomicInteger counter = new AtomicInteger();
Runnable job = () -> { for (int i = 0; i < 100_000; i++)
counter.incrementAndGet(); };
Thread t1 = new Thread(job), t2 = new Thread(job);
t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter.get());   // her zaman 200000; korumasız int++ daha
az verebilir
```

EK C

Algoritmalarda Önemli İsimler

ALGORİTMALAR · İSİMLER

Dersin on dört bölümündeki algoritmaların ve fikirlerin arkasındaki 76 isim: Öklid algoritmasından ve Turing makinesinden Dijkstra ile Kruskal'a, KMP'den ve Huffman kodlamadan Cook'un NP-tamlık teoremine, rekabetçi analize, Karatsuba çarpımına ve dal sınır yöntemine kadar.

Her kartta kişinin yaşadığı yıllar, ülkesi ve kurumu, en bilinen katkısı ve kısa bir özet yer alır; bağlantılar bu katkının işlendiği slaytlara gider. Kartlar bölüme göre süzülebilir, aranabilir, katkı yılına ya da soyadına göre sıralanabilir.

EU

Euclid

MÖ y. 300 · Antik Yunan · İskenderiye

OBEB algoritması · Elements

Elements (Elemanlar) adlı yapıtında iki sayının en büyük ortak bölenini art arda çıkarma yoluyla bulmayı anlattı; bugün aynı yöntemi bölme ve kalanla yazıyoruz. gcd(a, b) = gcd(b, a mod b) özdeşliğine dayanan Öklid algoritması (Euclid's algorithm), hâlâ kullanılan en eski algoritmalardan biridir. Büyük sayı her iki adımda en az yarıya indiği için adım sayısı O(log min(a, b)) ile sınırlıdır.

DERSTE **Bölüm 1.1 · Öklid Algoritması** **Bölüm 1.1 · Öklid: Karmaşıklık**

OBEB sonlanma logaritmik adım sayısı

AK

Al-Khwarizmi

y. 780 – y. 850 · Bağdat · Beytülhikme

Hint-Arap rakamlarıyla hesap · cebir

Bağdat'taki Beytülhikme'de çalışan bir matematikçiydi. Hint-Arap rakamlarıyla hesap yapmayı anlatan kitabı Latince çevirisiyle (Algoritmi de numero Indorum) Avrupa'ya yayıldı. “Algoritma” sözcüğü adının Latinceleşmiş biçiminden, “cebir” sözcüğü ise al-Kitāb al-mukhtaṣar fī ḥisāb al-

jabr wa-l-muqābala adlı yapıtının başlığından gelir. Problemleri açık ve tekrarlanabilir adımlarla çözme fikrinin simgesidir.

DERSTE **Bölüm 1.1 · Algoritma Nedir?**

“algoritma” sözcüğünün kökeni adım adım yöntem

LF

Leonardo Fibonacci

y. 1170 – y. 1250 · İtalyan · Pisa

Liber Abaci · 1202

Liber Abaci (1202) ile Hint-Arap rakamlarını Avrupa'ya tanıttı; kitaptaki tavşan problemi 1, 1, 2, 3, 5, 8, … dizisini üretir. F(n) = F(n − 1) + F(n − 2) bağıntısını düz özyinelemeyle çözmek üstel zaman alır, çünkü aynı alt problemler yeniden ve yeniden hesaplanır. Bu örnek, dinamik programlamanın ve bellekleme (memoization) tekniğinin en bilinen başlangıç noktasıdır.

DERSTE **Bölüm 1.1 · $O(2^n)$: Fibonacci** **Bölüm 8 · Fibonacci: Düz Özyineleme**

Bölüm 8 · Tekrarlanan Hesaplar

yineleme bağıntısı örtüşen alt problemler

PF

Pierre de Fermat

ö. 1665 · Fransız · Toulouse

Fermat'nın küçük teoremi · 1640

1640'ta Frénicle de Bessy'ye yazdığı bir mektupta, bugün küçük teoremi diye bilinen sonucu bildirdi: n asal ve 1 < a < n ise $a^{n-1} \equiv 1 \pmod{n}$ olur. Teoremin yayımlanan ilk kanıtını 1736'da Euler verdi. Fermat asallık testi bu eşitliği rastgele seçilen tabanlarla dener; eşitlik bozulursa sayı kesinlikle bileşiktir. Tersi ise her zaman doğru değildir: 561 = 3 · 11 · 17 gibi Carmichael sayıları, kendileriyle aralarında asal her tabanda eşitliği sağlar. Robert Carmichael 561'i 1910'da yayımladı. Miller–Rabin testi bu açığı karekök özelliğiyle kapatır.

DERSTE **Bölüm 12 · Miller–Rabin Asallık Testi** **Bölüm 12 · Miller–Rabin Örnekleri**

Fermat testi Carmichael sayıları modüler aritmetik

LE

Leonhard Euler

1707–1783 · İsviçreli · St. Petersburg, Berlin

Königsberg köprüleri · 1736

1736'da Königsberg'in yedi köprüsünün her birinden tam bir kez geçen bir yürüyüş olmadığını kanıtladı; kara parçalarını düğüm, köprüleri kenar olarak ele alması çizge kuramının başlangıcı sayılır. Böyle bir yürüyüş için tek dereceli düğüm sayısının 0 ya da 2 olması gerektiğini gösterdi. Her kenarı bir kez kullanan Euler turu, Christofides algoritmasının da bir adımıdır.

DERSTE **Bölüm 4.1 · Çizgeler Her Yerde** **Bölüm 11 · Metrik TSP**

çizge kuramı Euler turu

CG Carl Friedrich Gauss

1777–1855 · Alman · Göttingen

Aritmetik seri · 1 + 2 + … + n

Okul yıllarına ait bilinen bir öyküye göre 1'den 100'e kadar olan sayıları, iki uçtan terimleri eşleyerek bir anda topladı ve $n(n + 1)/2$ sonucuna ulaştı. Bu toplam, iç içe döngüler $O(n^2)$ olarak sayıldığında her seferinde karşımıza çıkar. İki karmaşık sayıyı dört yerine üç gerçel çarpmayla çarpma hilesi de ona atfedilir; Karatsuba yöntemi benzer bir fikri büyük sayılara uygular.

DERSTE **Bölüm 1.2 · f2: Toplam** **Bölüm 6 · Büyük Sayı Çarpımı**

aritmetik seri döngü sayma

GL Gabriel Lamé

1795–1870 · Fransız · Paris

Öklid algoritmasının adım sayısı · 1844

1844'te Öklid algoritmasının yaptığı bölme sayısının, küçük sayının onluk basamak sayısının beş katını geçemeyeceğini kanıtladı. Kanıtta Fibonacci sayılarını kullandı: belli sayıda bölme gerektiren en küçük girdi ardışık iki Fibonacci sayısıdır, çünkü bu girdilerde her bölümün sonucu 1 çıkar ve sayılar en yavaş biçimde küçülür. Teorem, bir algoritmanın en kötü durumunu inceleyen en eski çalışmalardan biri sayılır ve $O(\log \min(a, b))$ sınırının kesin biçimidir.

DERSTE **Bölüm 1.1 · Öklid: Karmaşıklık**

Lamé teoremi Öklid algoritması Fibonacci sayıları

AC

Arthur Cayley

1821–1895 · İngiliz · Cambridge

Cayley formülü · 1889

1889'da n düğümlü tam çizgenin tam olarak n^{n-2} kapsayan ağacı olduğunu gösteren formülü yayımladı; aynı sayıyı Carl Wilhelm Borchardt 1860'ta determinantla bulmuştu. Sayı çok hızlı büyür: 4 düğümde 16, 10 düğümde 100 milyon kapsayan ağaç vardır. Bu yüzden en ucuz kapsayan ağacı bütün ağaçları tek tek deneyerek bulmak pratik değildir; Kruskal ve Prim gibi açgözlü algoritmalar aynı sonuca polinom sürede ulaşır. Cayley matris cebirinin ve soyut grup kavramının da kurucularındandır.

DERSTE **Bölüm 4.3 · Kapsayan Ağaç ve MST**

Cayley formülü kapsayan ağaç sayısı tam çizge

OB

Otakar Borůvka

1899–1995 · Çek · Brno

İlk MST algoritması · 1926

1926'da Moravya'nın elektrik şebekesini en düşük maliyetle kurmak için ilk minimum kapsayan ağaç (minimum spanning tree, MST) algoritmasını yayımladı. Her turda her bileşen, kendisinden çıkan en hafif kenarı seçer; bileşen sayısı her turda en az yarıya indiği için O(log V) tur yeter. Bir turdaki seçimler birbirinden bağımsız olduğundan yöntem paralel ve dağıtık ortamlara da uygundur.

DERSTE **Bölüm 4.3 · Borůvka Algoritması**

minimum kapsayan ağaç bileşen

AT

Alan Turing

1912–1954 · İngiliz · Cambridge, Manchester

Turing makinesi · 1936

1936 tarihli “On Computable Numbers” makalesinde algoritma kavramını Turing makinesiyle biçimselleştirdi ve durma problemini (halting problem) hiçbir algoritmanın çözemeyeceğini gösterdi. Alonzo Church'ün aynı yıl lambda hesabıyla elde ettiği sonuçla birlikte bu tanım, “etkin biçimde hesaplanabilir” kavramının anlamı olarak kabul edilir (Church–Turing tezi). P ve NP sınıfları da aynı modelin zamanla sınırlanmış bir biçimiyle tanımlanır.

DERSTE **Bölüm 1.1 · Algoritma Nedir?** **Bölüm 9 · P, NP ve NP-Zor**

hesaplanabilirlik durma problemi Church–Turing tezi

FY

Ronald A. Fisher ve Frank Yates

1890–1962 · 1902–1994 · İngiliz · Rothamsted

Fisher–Yates karıştırma · 1938

1938'de Statistical Tables for Biological, Agricultural and Medical Research adlı kitapta, bir listenin rastgele permütasyonunu rastgele sayı tablosu yardımıyla kâğıt kalemle üretmenin yolunu anlattılar. Richard Durstenfeld 1964'te yöntemi bilgisayara uyarlayarak O(n) süreli takas sürümünü verdi; bu sürümü Knuth yaygınlaştırdı. Fisher, deney tasarımının ve modern istatistiğin kurucularındandır.

DERSTE **Bölüm 12 · Örnek: Karıştırma** **Bölüm 12 · Fisher–Yates İzleme**

Fisher–Yates karıştırma (Fisher–Yates shuffle) düzgün dağılımlı permütasyon

JN

John von Neumann

1903–1957 · Macar asıllı Amerikalı · Princeton IAS

Birleştirme sıralaması · 1945

1945'te EDVAC için yazdığı programlarda birleştirme sıralamasını (merge sort) anlattı; Knuth bu programı bilgisayar için yazılmış ilk sıralama programları arasında sayar. Böl ve fethet (divide and conquer) yaklaşımının klasik örneği olan yöntem en kötü durumda bile O(n log n) sürede çalışır ve kararlıdır. Stanisław Ulam ile Monte Carlo yönteminin geliştirilmesine katkıda bulundu; bugünkü bilgisayar mimarisi de onun adını taşır.

DERSTE **Bölüm 2 · Birleştirme Sıralaması** **Bölüm 6 · Birleştirme Sıralaması**

Bölüm 12 · Monte Carlo ile π

böl ve fethet kararlı sıralama Monte Carlo

SU

Stanisław Ulam

1909–1984 · Polonya doğumlu · ABD, Los Alamos

Monte Carlo yöntemi · 1946–1949

1946'da Los Alamos'ta, bir solitaire oyununu kazanma olasılığını rastgele denemelerle kestirme fikrinden yola çıkarak Monte Carlo yöntemini önerdi; von Neumann ile birlikte yöntemi ENIAC üzerinde nötron yayınımı hesaplarına uyguladı. Yöntemin adını taşıyan ilk makale 1949'da

Nicholas Metropolis ile yayımlandı. Rastgele noktalarla π sayısını kestirmek, derste kullanılan en yalın örnektir.

DERSTE **Bölüm 12 · Monte Carlo ile π** **Bölüm 12 · Las Vegas ve Monte Carlo**

Monte Carlo rastgele örnekleme

GD George B. Dantzig

1914–2005 · Amerikalı · RAND, Stanford

Simpleks yöntemi · 1947

1947'de doğrusal programları çözmek için simpleks yöntemini geliştirdi; dal sınır algoritmalarında her düğümdeki sınır çoğunlukla böyle bir LP gevşetmesiyle (LP relaxation) hesaplanır. 1954'te Fulkerson ve Johnson ile birlikte 49 şehirlik bir gezgin satıcı problemini (travelling salesman problem, TSP) kesme düzlemleriyle en iyi biçimde çözdü. Öğrenciyken çözülmemiş iki istatistik problemini ödev sanıp çözdüğü öykü de ünlüdür.

DERSTE **Bölüm 10 · Tam Sayılı Programlama** **Bölüm 10 · TSP: Alt Sınır**

doğrusal programlama simpleks TSP

CS Claude Shannon

1916–2001 · Amerikalı · Bell Labs, MIT

Bilgi kuramı · 1948

1948 tarihli "A Mathematical Theory of Communication" makalesiyle bilgi kuramını kurdu ve entropiyi, kayıpsız sıkıştırmada simge başına ortalama bit sayısının alt sınırı olarak tanımladı. Bir Huffman kodunun ortalama uzunluğu entropiyi en çok 1 bit aşar. Boole cebirini anahtarlama devrelerine uyguladığı 1937 tarihli yüksek lisans tezi de sayısal devre tasarımının temellerindendir.

DERSTE **Bölüm 5.1 · Dizgi Sıkıştırma** **Bölüm 7 · Kodlar ve Kazanç**

entropi sıkıştırma sınırı

DL Derrick H. Lehmer

1905–1991 · Amerikalı · UC Berkeley

Lehmer üreteci (1949) · "backtrack" terimi

1949'da bugün Lehmer üreteci olarak anılan sözde rastgele sayı (pseudo-random number) yöntemini, yani çarpımsal doğrusal eşlik üretecini x(k+1) = a · x(k) mod m tanıttı. 1950'lerde kısmi bir çözümü genişleten ve çıkmaza girince geri adım atan arama için “backtrack” terimini kullandı; bu yaklaşım bugün geri izleme (backtracking) adıyla bilinir. Sayılar kuramında asallık testleri ve hesap makineleri üzerine de öncü çalışmalar yaptı.

DERSTE **Bölüm 12 · Sözde Rastgele Sayılar** **Bölüm 9 · Geri Dönüp Dene**

sözde rastgele sayılar geri izleme

RH Richard Hamming

1915–1998 · Amerikalı · Bell Labs

Hamming uzaklığı · 1950

1950'de Bell System Technical Journal'da yayımladığı “Error Detecting and Error Correcting Codes” makalesinde hata düzelten kodları ve bugün Hamming uzaklığı diye anılan ölçüyü tanıttı. Hamming uzaklığı, eşit uzunluktaki iki dizginin farklı karakter taşıdığı konumların sayısıdır ve tek bir taramayla O(n) sürede hesaplanır; örneğin “karolin” ile “kathrin” arasındaki uzaklık 3'tür. Uzunlukları farklı dizgilerde tanımsızdır; ekleme ve silmeye de izin veren Levenshtein uzaklığı bu durumda kullanılır. 1968'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 5.2 · Benzerlik Ölçüleri**

Hamming uzaklığı hata düzelten kod dizgi benzerliği

SK Stephen C. Kleene

1909–1994 · Amerikalı · University of Wisconsin

Düzenli ifadeler · 1951

1951'de hazırladığı bir RAND raporunda (yayımlanışı 1956) düzenli ifadeleri (regular expressions) tanımladı ve bunların sonlu otomatların tanıdığı dilleri tam olarak betimlediğini gösterdi (Kleene teoremi). “Sıfır ya da daha çok tekrar” anlamına gelen * işlemi onun adıyla Kleene yıldızı diye anılır. Özyinelemeli fonksiyonlar kuramıyla birlikte hesaplanabilirlik kuramının kurucularındandır.

DERSTE **Bölüm 5.2 · Düzenli İfadeler** **Bölüm 5.2 · Niceleyiciler ve Gruplama**

düzenli ifade Kleene yıldızı

DH

David A. Huffman

1925–1999 · Amerikalı · MIT, UC Santa Cruz

Huffman kodlama · 1952

MIT'de lisansüstü öğrenciyken, Robert Fano'nun dersinde final sınavının yerine geçen dönem ödevi olarak en kısa önek kodunu bulma problemini çözdü ve sonucu 1952'de yayımladı. En az sıklıktaki iki düğümü tekrar tekrar birleştiren açgözlü yöntem, karakter başına beklenen bit sayısını en aza indirir ve hocasının Shannon–Fano yöntemini geride bırakır. Huffman kodlama (Huffman coding) bugün DEFLATE ve JPEG gibi biçimlerin bir aşamasıdır.

DERSTE **Bölüm 7 · Huffman Kodlama** **Bölüm 7 · Huffman Neden En İyi?**

Bölüm 5.1 · Dizgi Sıkıştırma

Huffman kodlama önek kodu açgözlü seçim

HL

Hans Peter Luhn

1896–1964 · Alman asıllı Amerikalı · IBM

Anahtarları kovalara dağıtma · 1953

1953'te IBM'de, anahtarları bir fonksiyonla “kovalara” dağıtarak aramayı hızlandırmayı önerdi; bu, hash tablolarının (hash table) ve zincirlemenin (chaining) bilinen en eski örneklerinden biridir. Kredi kartı numaralarını doğrulamakta hâlâ kullanılan Luhn denetim basamağı algoritması ve bağlam içinde anahtar sözcük (KWIC) dizinlemesi de onundur.

DERSTE **Bölüm 3 · Hash ile Arama: Fikir** **Bölüm 3 · Zincirleme ile Ekleme**

hash tablosu zincirleme

RB

Richard Bellman

1920–1984 · Amerikalı · RAND Corporation

Dinamik programlama · 1950'ler

1950'lerde RAND Corporation'da çok aşamalı karar problemleri için “dinamik programlama” (dynamic programming) adını ve optimallik ilkesini ortaya koydu: optimal bir çözümün her parçası, kendi alt problemi için de optimaldir. 1958'de negatif kenar ağırlıklarına izin veren tek kaynaklı en kısa yol yöntemini yayımladı (Bellman-Ford). “Boyutluluk laneti” (curse of dimensionality) deyimi de ona aittir.

DERSTE **Bölüm 8 · Dinamik Programlama Nedir?** **Bölüm 8 · İki Temel Özellik**

Bölüm 4.2 · Bellman-Ford Algoritması

dinamik programlama optimal alt yapı Bellman-Ford

GA

Gene Amdahl

1922–2015 · Amerikalı · IBM

Doğrusal yoklama (1954) · Amdahl yasası (1967)

1954'te Elaine McGraw ve Arthur Samuel ile birlikte IBM 701 için yazılan bir çeviricide, çakışma olunca bir sonraki boş hücreye geçen doğrusal yoklamayı (linear probing) kullandı. IBM System/360'ın baş mimarlarından biriydi. 1967'deki bir konferans konuşmasında, programın sıralı kısmının paralel hızlanmayı sınırladığını savundu; bu gözlem Amdahl yasası (Amdahl's law) olarak bilinir.

DERSTE **Bölüm 3 · Doğrusal Yoklama: Ekleme** **Bölüm 14 · Amdahl Yasası**

doğrusal yoklama hızlanma Amdahl yasası

FF

L. R. Ford Jr. ve D. R. Fulkerson

1927–2017 · 1924–1976 · Amerikalı · RAND Corporation

Maksimum akış · 1956

1956'da RAND Corporation'da, artık ağda (residual network) artış yolu (augmenting path) kalmayana kadar akışı artıran Ford-Fulkerson yöntemini ve maksimum akış minimum kesim teoremini yayımladılar. Ford aynı yıllarda kenarları tekrar tekrar gevşeten en kısa yol yöntemini de verdi; algoritmanın Bellman-Ford diye anılması bundandır. 1962 tarihli Flows in Networks kitapları ağ akışını ayrı bir alan hâline getirdi.

DERSTE **Bölüm 4.4 · Ford-Fulkerson Yöntemi** **Bölüm 4.4 · Maks-Akış Min-Kesim**

Bölüm 4.2 · Bellman-Ford Algoritması

Ford-Fulkerson artış yolu minimum kesim

JK

Joseph Kruskal

1928–2010 · Amerikalı · Princeton, Bell Labs

Kruskal algoritması · 1956

1956 tarihli kısa bir makalede kenarları ağırlık sırasıyla inceleyip döngü oluşturmayanları ekleyen MST algoritmasını verdi; aynı makale ters silme (reverse-delete) yöntemini de anlatır. Bugün döngü denetimi union-find yapısıyla yapılır ve algoritma O(E log E) sürede çalışır. Çok boyutlu ölçekleme gibi katkılarıyla istatistikte de önemli bir yeri vardır.

DERSTE **Bölüm 4.3 · Kruskal Algoritması** **Bölüm 4.3 · Ters Silme Algoritması**

Kruskal algoritması union-find ters silme

EM Edward F. Moore

1925–2003 · Amerikalı · Bell Labs

Labirentte en kısa yol (BFS) · 1957

Bell Labs'ta labirentteki en kısa yolu bulmak için genişlik öncelikli aramayı (breadth-first search, BFS) anlattı (“The Shortest Path Through a Maze”, sunuluşu 1957, yayımlanışı 1959). Çalışma, ağırlıksız çizgelerde en kısa yolların katman katman bulunabileceğini gösterir; ağırlıklı durum için verdiği yöntem yüzünden Bellman-Ford kimi zaman “Bellman–Ford–Moore” diye anılır. Sonlu durum makinelerinin Moore makinesi türü de onun adını taşır.

DERSTE **Bölüm 4.1 · BFS: Fikir** **Bölüm 4.1 · BFS ile Labirent** **Bölüm 5.2 · FSM Türleri**

BFS Moore makinesi

RP Robert C. Prim

1921–2021 · Amerikalı · Bell Labs

Prim algoritması · 1957

1957'de Bell Labs'ta, bir düğümden başlayıp ağacı her adımda dışarı çıkan en hafif kenarla büyüten minimum kapsayan ağaç algoritmasını yayımladı. Aynı yöntemi Vojtěch Jarník 1930'da bulmuş, Dijkstra da 1959'da yeniden keşfetmişti; bu yüzden Jarník–Prim algoritması diye de anılır. Öncelikli kuyrukla O(E log V) sürede çalışır ve yapısı Dijkstra algoritmasına çok benzer.

DERSTE **Bölüm 4.3 · Prim Algoritması** **Bölüm 4.3 · Prim ve Dijkstra**

Prim algoritması kesim özelliği

DS

Donald L. Shell

1924–2015 · Amerikalı · General Electric

Shell sıralaması · 1959

1959'da yayımladığı yöntemde öğeler önce geniş aralıklarla, sonra giderek daralan aralıklarla eklemeli sıralamadan geçirilir. Böylece uzak ters çiftler (inversion) erkenden düzelir ve eklemeli sıralamanın $O(n^2)$ davranışı uygulamada büyük ölçüde iyileşir. Karmaşıklık aralık dizisine bağlıdır ve kimi diziler için hâlâ açık bir sorudur.

DERSTE **Bölüm 2 · Shell Sıralaması** **Bölüm 2 · Shell: Adımlar**

Shell sıralaması (Shell sort) ters çift

ED

Edsger W. Dijkstra

1930–2002 · Hollandalı · Amsterdam, Eindhoven; Austin, ABD

En kısa yollar · 1959

Kendi anlatımına göre en kısa yol algoritmasını 1956'da Amsterdam'da yirmi dakika kadar içinde tasarladı ve 1959'da yayımladı; aynı makale minimum kapsayan ağaç için Prim yöntemini de bağımsız olarak verir. Geçici uzaklığı en küçük olan düğümü kesinleştirip kenarlarını gevşetmek (relaxation) negatif olmayan ağırlıklarda doğru sonuç verir ve öncelikli kuyrukla (priority queue) $O((V + E) \log V)$ sürer. Yapısal programlama ve semaforlarla da tanınır; 1972'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 4.2 · Dijkstra Algoritması** **Bölüm 4.2 · Gevşetme (Relaxation)** **Bölüm 4.3 · Prim ve Dijkstra**

Dijkstra algoritması gevşetme öncelikli kuyruk

MR

Michael O. Rabin

d. 1931 · İsrailli · Kudüs İbrani Üniversitesi; Harvard

Miller–Rabin (1980) · Rabin-Karp (1987)

1959'da Dana Scott ile deterministik olmayan sonlu otomatları tanımladı; bu çalışma 1976 Turing Ödülü'nü getirdi. 1980'de Gary Miller'ın asallık testini (1976) koşulsuz bir olasılıksal teste dönüştürdü: Miller–Rabin testinde bileşik bir sayı bir turu en çok 1/4 olasılıkla geçer. Karp ile kayan hash (rolling hash) kullanan Rabin-Karp dizgi eşleştirme algoritmasını 1987'de yayımladı; rastgeleliği algoritma tasarımında bilinçli olarak kullanmanın öncülerindendir.

DERSTE **Bölüm 12 · Miller–Rabin Asallık Testi** **Bölüm 5.1 · Rabin-Karp**

Bölüm 5.2 · FSM Tanımı

asallık testi kayan hash sonlu otomat

TH

C. A. R. (Tony) Hoare

1934–2026 · İngiliz · Oxford

Hızlı sıralama · 1959–1961

Hızlı sıralamayı (quicksort) 1959–1960'ta Moskova Devlet Üniversitesi'nde konuk öğrenciyken tasarladı; algoritma 1961'de yayımlandı. Bir pivot çevresinde bölümleyip iki yanı özyinelemeyle sıralamak ortalamada O(n log n), kötü pivotlarla O(n^2) sürer. Aynı yıl k. en küçük öğe için Find yöntemini (bugünkü QuickSelect) yayımladı; program doğruluğu için Hoare mantığını kurdu ve 1980'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 2 · Hızlı Sıralama** **Bölüm 2 · Bölümleme Prosedürü**

Bölüm 12 · Rastgele Seçme (QuickSelect)

hızlı sıralama bölümleme QuickSelect

EF

Edward Fredkin

1934–2023 · Amerikalı · MIT, Carnegie Mellon

Trie · 1960

1960'ta dizgileri ortak önekleri paylaşan bir ağaçta saklama fikrini yayımladı ve yapıya “retrieval” sözcüğünün ortasından “trie” adını verdi; René de la Briandais aynı yapıyı bir yıl önce (1959) anlatmıştı. Fredkin tersinir hesaplama ve Fredkin kapısıyla da tanınır; MIT'deki Project MAC'i yönetti.

DERSTE **Bölüm 5.3 · Trie (Önek Ağacı)** **Bölüm 5.3 · Neden Trie?**

trie önek sorgusu

KA

Anatoly Karatsuba

1937–2008 · Sovyet · Moskova

Karatsuba çarpımı · 1960/1962

1960'ta Kolmogorov'un seminerinde iki n basamaklı sayıyı çarpmanın $\Omega(n^2)$ işlem gerektirdiği sanısı tartışılırken, bir hafta içinde üç alt çarpımla O(n^1,585) sürede çalışan yöntemi buldu; o sırada 23 yaşındaydı. Kolmogorov sonucu 1962'de Karatsuba ve Ofman adlarıyla yayımlattı. Böl ve fethet yaklaşımının varsayılan bir alt sınırı aşabildiğini gösteren ilk örneklerden biridir.

DERSTE **Bölüm 6 · Büyük Sayı Çarpımı** **Bölüm 6 · Karatsuba Analizi**

Karatsuba çarpımı böl ve fethet

LD Ailsa Land ve Alison Doig

1927–2021 · d. 1929 · İngiliz · London School of Economics

Dal sınır · 1960

London School of Economics'te yazdıkları 1960 tarihli Econometrica makalesinde tam sayılı doğrusal programları çözmek için dal sınır (branch and bound) yöntemini önerdiler: çözüm uzayı dallara ayrılır ve her dal için LP gevşetmesiyle bir sınır hesaplanır. Sınırı bilinen en iyi çözümden kötü olan dallar budanır. Bu fikir bugünkü tam sayılı programlama çözücülerinin hâlâ çekirdeğindedir; Doig daha sonra Alison Harcourt adıyla University of Melbourne'de çalıştı.

DERSTE **Bölüm 10 · Dal Sınır Fikri** **Bölüm 10 · Tam Sayılı Programlama**

dal sınır LP gevşetmesi

RF Robert W. Floyd

1936–2001 · Amerikalı · Carnegie Mellon, Stanford

Floyd-Warshall · 1962

1962'de tüm çiftler arasındaki en kısa yolları (all-pairs shortest paths) $O(V^3)$ sürede bulan algoritmayı yayımladı; ara düğümleri birer birer ekleyen bu dinamik program bugün Floyd-Warshall diye bilinir. 1964'te heap'i O(n) sürede kuran yöntemi verdi; listelerde döngü bulmak için “kaplumbağa ve tavşan” yöntemi de ona atfedilir. Program doğrulama alanındaki öncü çalışmaları için 1978'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 4.2 · Floyd-Warshall Algoritması** **Bölüm 2 · Heap Sort: Heap Kurma**

tüm çiftler arası en kısa yollar heap kurma

SW Stephen Warshall

1935–2006 · Amerikalı

Geçişli kapanış · 1962

1962 tarihli "A Theorem on Boolean Matrices" makalesinde bir bağıntının geçişli kapanışını (transitive closure) Boole matrisi üzerinde $O(V^3)$ sürede hesaplayan yöntemi yayımladı: i'den k'ye ve k'den j'ye yol varsa i'den j'ye de yol vardır. Floyd'un aynı yılki en kısa yol algoritması aynı üçlü döngüyü kullandığı için ikisi birlikte anılır. Bernard Roy benzer bir sonucu 1959'da yayımlamıştı.

DERSTE **Bölüm 4.2 · Floyd-Warshall: Adımlar** **Bölüm 4.2 · Floyd-Warshall: Sözde Kod**

geçişli kapanış komşuluk matrisi

AK

Arthur B. Kahn

Communications of the ACM 5(11), 1962

Topolojik sıralama · 1962

1962 tarihli "Topological Sorting of Large Networks" makalesinde, yönlü ve döngüsüz bir çizgenin topolojik sırasını giriş derecesi sıfır olan düğümleri art arda çıkararak bulan algoritmayı verdi. Giriş derecesi 0 olan düğümler kuyruğa konur; kuyruktan çıkan düğüm sıraya yazılır ve ardıllarının giriş derecesi bir azaltılır, derecesi sıfıra inen düğüm kuyruğa katılır. Algoritma O(V + E) sürede çalışır. Bütün düğümler çıkarılamıyorsa çizgede döngü vardır; böylece aynı yöntem döngü denetimi de yapar.

DERSTE **Bölüm 4.1 · Topolojik Sıralama** **Bölüm 4.1 · Kahn Algoritması: Kod İzleme**

topolojik sıralama giriş derecesi DAG

HK

Michael Held ve Richard M. Karp

Karp: d. 1935 · *Journal of the SIAM*, 1962

Gezgin satıcı için dinamik programlama · 1962

1962 tarihli "A Dynamic Programming Approach to Sequencing Problems" makalesinde gezgin satıcı problemini dinamik programlamayla çözdüler. Durum bir küme ile bir düğümden oluşur: dp[S][i], başlangıçtan çıkıp S kümesindeki şehirlerin hepsine uğrayan ve i'de biten en kısa yoldur. Algoritma $O(n^2 \cdot 2^n)$ sürede ve $O(n \cdot 2^n)$ bellekle çalışır; n! turu tek tek denemekten çok hızlıdır ama yine üsteldir. Richard Bellman aynı yöntemi aynı yıl bağımsız olarak yayımladı.

DERSTE **Bölüm 8 · Diğer Klasik DP Problemleri** **Bölüm 11 · TSP: Çözüm Yaklaşımları**

Held–Karp gezgin satıcı alt küme DP

LM

John D. C. Little, Katta G. Murty, Dura W. Sweeney ve Caroline Karel

Little: 1928–2024 · Amerikalı · MIT

Gezgin satıcı için dal sınır · 1963

1963'te Operations Research dergisinde yayımladıkları "An Algorithm for the Traveling Salesman Problem" makalesinde gezgin satıcı problemi için bir dal sınır algoritması verdiler; "dal sınır" (branch and bound) adı ilk kez bu makalede geçer. Alt sınır indirgenmiş maliyet matrisiyle bulunur: her satırdan ve her sütundan en küçük öğe çıkarılır, çıkarılan değerlerin toplamı her turun maliyetine bir alt sınırdır. Yöntem simetri gerektirmez, d[i][j] ≠ d[j][i] olan asimetrik problemlerde de çalışır. Little, kuyruk kuramındaki Little yasasıyla (1961) da tanınır.

DERSTE **Bölüm 10 · İndirgenmiş Maliyet Matrisi** **Bölüm 10 · TSP: 6 Şehirli Örnek**

dal sınır indirgenmiş maliyet matrisi gezgin satıcı

JW

J. W. J. Williams

1930–2012 · İngiliz-Kanadalı

Heap sort · 1964

1964'te heap sort algoritmasını ve onun dayandığı ikili heap yapısını Communications of the ACM'de "Algorithm 232: Heapsort" başlığıyla yayımladı. Diziyi max-heap'e dönüştürüp kökü tekrar tekrar sona taşımak, en kötü durumda O(n log n) süre ve O(1) ek bellek gerektirir. Aynı yıl Robert Floyd heap'i O(n) sürede kuran iyileştirmeyi verdi.

DERSTE **Bölüm 2 · Heap Sort** **Bölüm 2 · Heap Sort: Heap Kurma**

heap yerinde sıralama

JE

Jack Edmonds

d. 1934 · Amerika doğumlu · Kanada, Waterloo

Çiçek algoritması (1965) · Edmonds-Karp (1972)

1965 tarihli "Paths, Trees, and Flowers" makalesinde genel çizgelerde maksimum eşleştirmeyi polinom zamanda bulan çiçek (blossom) algoritmasını verdi ve "verimli algoritma" ifadesinin polinom zamanlı algoritma anlamına gelmesi gerektiğini savundu. 1972'de Karp ile, artış yolları BFS ile seçildiğinde Ford-Fulkerson yönteminin $O(V \cdot E^2)$ sürede çalıştığını gösterdi. 1971'de açgözlü algoritmanın tam olarak matroidler üzerinde optimal olduğunu gösterdi.

DERSTE **Bölüm 4.4 · Edmonds-Karp Algoritması** **Bölüm 7 · Ne Zaman Doğru Çalışır?**

Edmonds-Karp polinom zaman matroid

VL

Vladimir Levenshtein

1935–2017 · Sovyet-Rus · Keldysh Uygulamalı Matematik Enstitüsü

Levenshtein uzaklığı · 1965

1965'te bir dizgiyi diğerine dönüştürmek için gereken en az ekleme, silme ve değiştirme sayısını tanımladı; bu ölçü onun adını taşır. Amacı hata düzelten kodlardı; ölçü bugün yazım denetiminde, DNA dizilerini karşılaştırmada ve yaklaşık aramada kullanılır. Uzaklığı dinamik programlamayla O(m·n) sürede hesaplamak (Wagner–Fischer, 1974), derste kullanılan tablo yöntemidir.

DERSTE **Bölüm 5.2 · Levenshtein Uzaklığı** **Bölüm 5.2 · DP Bağıntısı**

düzenleme uzaklığı (edit distance) DP tablosu

SG

Solomon W. Golomb

1932–2016 · Amerikalı · University of Southern California

Backtrack Programming · 1965

Leonard Baumert ile yazdığı 1965 tarihli Journal of the ACM makalesi “Backtrack Programming”, geri izlemeyi genel bir arama yöntemi olarak sistemli biçimde anlatan ilk çalışmalardandır; budamanın (pruning) arama ağacını nasıl küçülttüğünü örneklerle gösterir. 1953'te poliominoları tanımlaması, bulmaca ve döşeme problemlerinin bilgisayarla aranmasına esin kaynağı oldu. Golomb kodlaması da onun katkısıdır.

DERSTE **Bölüm 9 · Durum Uzayı Ağacı** **Bölüm 9 · Kaba Kuvvet ve Budama**

geri izleme budama

RG

Ronald Graham

1935–2020 · Amerikalı · Bell Labs, UC San Diego

Liste çizelgeleme (1966) · Graham taraması (1972)

1966'da her işi ilk boşalan makineye veren liste çizelgelemenin (list scheduling) en büyük yükünün (makespan) en iyi değerin en çok (2 − 1/m) katı olduğunu kanıtladı; bu, yaklaşım oranının (approximation ratio) ilk örneklerindendir. 1969'da işleri uzundan kısaya sıralayan LPT kuralı-

nın oranının 4/3 − 1/(3m) olduğunu gösterdi. 1972'de düzlemdeki noktaların dışbükey örtüsünü (convex hull) O(n log n) sürede bulan Graham taramasını yayımladı.

DERSTE **Bölüm 11 · Liste Çizelgeleme** **Bölüm 11 · LPT Kuralı** **Bölüm 6 · Dışbükey Örtü**

liste çizelgeleme yaklaşım oranı dışbükey örtü

LB

László Bélády

1928–2021 · Macaristan doğumlu · ABD, IBM

OPT (MIN) · Belady anomalisi

1966'da IBM'de, gelecekte en geç kullanılacak sayfayı çıkaran ve en az sayfa hatası veren çevrim dışı algoritmayı (MIN, bugün OPT) tanımladı; çevrim içi politikalar bu ideal ile karşılaştırılır. Nelson ve Shedler ile, FIFO kullanıldığında çerçeve sayısı artınca sayfa hatası sayısının da artabileceğini gösterdi (Belady anomalisi). 1961'de ABD'ye göç etmişti.

DERSTE **Bölüm 13 · OPT Adım Adım** **Bölüm 13 · Belady Anomalisi**

sayfalama çıkarma politikası

MF

Michael J. Flynn

1934–2025 · Amerikalı · Stanford

Flynn sınıflandırması · 1966

1966'da Proceedings of the IEEE'de yayımlanan “Very High-Speed Computing Systems” makalesinde bilgisayarları aynı anda işledikleri komut ve veri akışlarının sayısına göre dört sınıfa ayırdı: SISD (tek komut, tek veri), SIMD (tek komut, çok veri), MISD (çok komut, tek veri) ve MIMD (çok komut, çok veri). Klasik tek çekirdekli işlemci SISD, vektör birimleri SIMD, çok çekirdekli işlemciler ve kümeler MIMD sınıfındadır. Sınıflandırma paralel makineleri anlatmanın ortak dili oldu.

DERSTE **Bölüm 14 · Flynn Sınıflandırması**

SIMD MIMD paralel mimari

AV

Andrew Viterbi

d. 1935 · İtalya doğumlu · ABD, UCLA, Qualcomm

Viterbi algoritması · 1967

1967'de evrişimli kodları çözmek için her adımda her duruma ulaşan en iyi yolu saklayan bir dinamik programlama algoritması önerdi. Gizli Markov modellerinde en olası durum dizisini bulan bu yöntem mobil ve uydu iletişiminde, konuşma tanımada ve biyoinformatikte kullanılır. 1985'te Irwin Jacobs ile Qualcomm'u kurdu.

DERSTE **Bölüm 8 · Diğer Klasik DP Problemleri** **Bölüm 8 · DP Tasarım Adımları**

dinamik programlama en olası yol

DK

Donald E. Knuth

d. 1938 · Amerikalı · Stanford

The Art of Computer Programming · 1968–

Çok ciltli The Art of Computer Programming (Bilgisayar Programlama Sanatı) ile algoritma analizini bir disipline dönüştürdü; işlem sayılarını tam olarak hesaplamayı ve ortalama durum analizini yaygınlaştırdı. 1976'da O, Ω ve Θ gösterimlerinin bugünkü anlamlarını önerdi. Morris ve Pratt ile KMP dizgi eşleştirme algoritmasını yayımladı (1977), TeX dizgi sistemini geliştirdi ve 1974'te Turing Ödülü'nü aldı.

DERSTE **Bölüm 1.2 · Ω ve Θ** **Bölüm 5.1 · Knuth-Morris-Pratt**

algoritma analizi asimptotik gösterim KMP

A*

Peter Hart, Nils Nilsson ve Bertram Raphael

Nilsson: 1933–2019 · Amerikalı · Stanford Research Institute (SRI)

A* algoritması · 1968

SRI'da Shakey robotunun yol planlaması için A* algoritmasını 1968'de yayımladılar. Nilsson'un sezgisel aramasına o ana kadarki maliyeti eklemeyi Raphael önerdi; kabul edilebilir (admissible) ve tutarlı sezgisel kavramlarını Hart getirdi. f = g + h değerine göre ilerleyen A*, kabul edilebilir bir sezgiselle en kısa yolu bulur ve çoğu zaman Dijkstra algoritmasından çok daha az düğüm açar.

DERSTE **Bölüm 4.2 · A* Algoritması** **Bölüm 4.2 · Sezgisel Fonksiyonlar**

A* algoritması kabul edilebilir sezgisel

VS

Volker Strassen

d. 1936 · Alman · Zürih, Konstanz

Strassen algoritması · 1969

1969 tarihli “Gaussian Elimination is not Optimal” makalesinde matrisleri dört bloğa ayırıp sekiz yerine yedi blok çarpımıyla çarptı ve O(n^2,81) süreli matris çarpımını elde etti; matrislerin n^3'ten hızlı çarpılabildiğini gösteren ilk sonuç buydu. 1971'de Schönhage ile çok büyük tam sayıların hızlı çarpımını, 1977'de Solovay ile olasılıksal bir asallık testini geliştirdi.

DERSTE **Bölüm 6 · Strassen'in 7 Çarpımı** **Bölüm 6 · Strassen Analizi**

Strassen algoritması matris çarpımı

YD Yefim Dinitz

günümüz · Sovyet-İsrailli · Ben-Gurion Üniversitesi

Dinic algoritması · 1970

1969'da Moskova'da, düzey çizgesinde (level graph) engelleyici akışlar (blocking flow) bularak ilerleyen maksimum akış algoritmasını tasarladı ve 1970'te yayımladı. Batı'da Shimon Even'ın sunumu aracılığıyla “Dinic algoritması” adıyla tanındı. Algoritma $O(V^2 \cdot E)$ sürede çalışır; birim kapasiteli ağlarda ve iki parçalı eşleştirmede çok daha hızlıdır.

DERSTE **Bölüm 4.4 · Dinic Algoritması** **Bölüm 4.4 · Karmaşıklık: $O(V^2 \cdot E)$**

düzey çizgesi engelleyici akış

MP James H. Morris ve Vaughan Pratt

d. 1941 · d. 1944 · Amerikalı · Berkeley, Stanford (Pratt Avustralya doğumlu)

Knuth-Morris-Pratt · 1970/1977

Morris 1970'te örüntünün kendi öneklerinden yararlanarak metinde hiç geri gitmeyen bir eşleştirme yöntemi buldu ve bunu Pratt ile bir teknik raporda yayımladı; birkaç hafta sonra Knuth aynı algoritmaya otomatlar kuramından bağımsız olarak ulaştı. Üçü “Fast Pattern Matching in Strings” makalesini 1977'de birlikte yayımladı. LPS tablosu sayesinde metin göstergesi hiç geri gitmez ve arama O(n + m) sürer.

DERSTE **Bölüm 5.1 · Knuth-Morris-Pratt** **Bölüm 5.1 · LPS Tablosu**

KMP LPS tablosu önek

SC

Stephen Cook

d. 1939 · Amerikalı-Kanadalı · University of Toronto

Cook teoremi · 1971

1971 tarihli “The Complexity of Theorem-Proving Procedures” makalesinde NP'deki her problemin polinom zamanda SAT'a indirgendiğini, yani SAT'ın NP-tam (NP-complete) olduğunu gösterdi. Bu sonuç P = NP sorusunu biçimsel olarak ortaya koydu ve NP-tamlık kuramını başlattı. 1982'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 9 · P, NP ve NP-Zor**

NP-tam SAT P = NP?

RT

Robert E. Tarjan

d. 1948 · Amerikalı · Princeton

Güçlü bağlantılı bileşenler · 1972

1972'de yönlü bir çizgenin güçlü bağlantılı bileşenlerini (strongly connected components) tek bir DFS ile doğrusal sürede bulan algoritmayı yayımladı. 1975'te yol kısaltma (path compression) ve rütbeye göre birleştirme (union by rank) kullanan union-find yapısının amortize olarak neredeyse sabit süre aldığını kanıtladı ve amortize analizi (amortized analysis) sistemli hâle getirdi. Sleator ile splay ağaçlarını ve çevrim içi liste güncelleme analizini (1985), Goldberg ile itme–yeniden etiketleme (push–relabel) yöntemini (1988) geliştirdi; 1986 Turing Ödülü'nü Hopcroft ile paylaştı.

DERSTE **Bölüm 4.1 · Tarjan: İlkeler ve Adımlar** **Bölüm 4.3 · Union-Find (Ayrık Kümeler)**

Bölüm 1.2 · Amortize Analiz **Bölüm 4.4 · İtme–Yeniden Etiketleme**

DFS union-find amortize analiz

RK

Richard M. Karp

d. 1935 · Amerikalı · Berkeley

21 NP-tam problem · 1972

1972'de SAT'tan başlayarak klik, düğüm kapsama (vertex cover), Hamilton döngüsü ve sırt çantası gibi 21 problemin NP-tam olduğunu polinom zamanlı indirgemelerle (reduction) gösterdi ve NP-tamlığın ne kadar yaygın olduğunu ortaya koydu. Edmonds ile Edmonds-Karp algoritmasını (1972), Hopcroft ile iki parçalı eşleştirme algoritmasını (1973), Rabin ile Rabin-Karp dizgi eşleştirme algoritmasını (1987) geliştirdi. 1985'te Turing Ödülü'nü aldı.

DERSTE **Bölüm 9 · P, NP ve NP-Zor** **Bölüm 11 · 3-SAT → Düğüm Kapsama**

Bölüm 4.4 · Edmonds-Karp Algoritması **Bölüm 5.1 · Rabin-Karp**

NP-tam indirgeme Rabin-Karp

JH

John Hopcroft

d. 1939 · Amerikalı · Cornell

Hopcroft–Karp eşleştirmesi · 1973

Tarjan ile birlikte derinlik öncelikli aramanın (depth-first search, DFS) çizge algoritmalarında ne kadar güçlü olduğunu gösterdi; düzlemsellik testini birlikte doğrusal sürede çözdüler. Karp ile 1973'te iki parçalı bir çizgede maksimum eşleştirmeyi O(E√V) sürede bulan algoritmayı yayımladı. Ullman ile yazdığı otomatlar ve biçimsel diller kitabı uzun yıllar temel kaynak oldu; 1986 Turing Ödülü'nü Tarjan ile paylaştı.

DERSTE **Bölüm 4.1 · DFS: Analiz ve Kullanım** **Bölüm 4.4 · Uygulamalar**

DFS iki parçalı eşleştirme

LL

Leonid Levin

d. 1948 · Sovyet-Amerikalı · Boston University

Cook–Levin teoremi · 1973

SSCB'de, Cook'tan bağımsız olarak arama problemleri için “evrensel problemler” kavramını geliştirdi ve 1973'te yayımladı; teoremin Cook–Levin teoremi diye anılması bundandır. Bu problemler için neredeyse optimal bir “evrensel arama” algoritması da verdi. Daha sonra ABD'ye göç etti ve 1980'den beri Boston University'de çalışıyor.

DERSTE **Bölüm 9 · P, NP ve NP-Zor** **Bölüm 11 · 3-SAT → Düğüm Kapsama**

NP-tam indirgeme

RB

Richard P. Brent

d. 1946 · Avustralyalı · Avustralya Ulusal Üniversitesi

Brent teoremi · 1974

1974'te Journal of the ACM'de yayımladığı “The Parallel Evaluation of General Arithmetic Expressions” makalesinde, çok işlemci için yazılmış bir hesabın daha az işlemciyle ne kadar sürede yapılabileceğini sınırladı. İş–derinlik diliyle sonuç $T_p \le W/p + D$ biçimindedir: W toplam iş,

D derinlik, p işlemci sayısıdır. $T_p \geq W/p$ ve $T_p \geq D$ alt sınırlarıyla birlikte okunduğunda, p işlemcili süre en iyinin iki katını geçmez. Brent sayısal kök bulmadaki Brent yöntemiyle (1973) de tanınır.

DERSTE **Bölüm 14 · İş ve Derinlik**

Brent teoremi iş derinlik

AC Alfred V. Aho ve Margaret J. Corasick

Aho: d. 1941 · Bell Labs

Aho–Corasick algoritması · 1975

1975'te Communications of the ACM'de yayımladıkları “Efficient String Matching: An Aid to Bibliographic Search” makalesinde birçok örüntüyü metinde tek geçişte arayan algoritmayı verdiler. Örüntüler bir trie'ye yerleştirilir; her düğüme, eşleşme bozulduğunda geçilecek en uzun özel soneki gösteren hata bağlantısı (failure link) eklenir. Arama süresi metin uzunluğu, örüntülerin toplam uzunluğu ve bulunan eşleşme sayısıyla doğrusaldır. Algoritma Unix'in fgrep komutunun temelidir. Aho, AWK dilinin ve “Dragon Book” diye anılan derleyici kitaplarının da yazarlarındandır; 2020 Turing Ödülü'nü Jeffrey Ullman ile paylaştı.

DERSTE **Bölüm 5.3 · Aho–Corasick** **Bölüm 5.3 · Trie Ailesi**

çoklu örüntü arama hata bağlantısı trie

NC Nicos Christofides

1942–2019 · Kıbrıslı-İngiliz · Imperial College London

Christofides algoritması · 1976

1976'da üçgen eşitsizliğini sağlayan TSP (metrik TSP) için en iyi turun en çok 3/2 katı uzunlukta bir tur bulan algoritmayı verdi: bir MST kurulur, tek dereceli düğümler üzerine minimum mükemmel eşleştirme eklenir ve Euler turunda kısayollarla gezinilir. Bu oran 2020'ye kadar metrik TSP için bilinen en iyi garantiydi. Anatoliy Serdyukov aynı sonuca SSCB'de hemen hemen aynı zamanda ulaştı.

DERSTE **Bölüm 11 · Metrik TSP** **Bölüm 11 · MST Tabanlı 2-Yaklaşım**

gezgin satıcı problemi yaklaşım oranı

BM

Robert S. Boyer ve J Strother Moore

Boyer: d. 1946 · Amerikalı · University of Texas at Austin

Boyer-Moore · 1977

1977'de örüntüyü sağdan sola karşılaştıran ve kötü karakter (bad character) ile iyi sonek (good suffix) kurallarıyla metinde büyük sıçramalar yapan algoritmayı yayımladılar; uzun örüntülerde metin karakterlerinin çoğuna hiç bakılmaz. Horspool 1980'de yalnız kötü karakter kuralını kullanan daha basit bir sürüm verdi. İkili ayrıca Boyer–Moore teorem kanıtlayıcısını (ACL2'nin atası) ve çoğunluk oyu algoritmasını geliştirdi.

DERSTE **Bölüm 5.1 · Boyer-Moore** **Bölüm 5.1 · Kötü Karakter Kuralı**

Bölüm 5.1 · Horspool Basitleştirmesi

kötü karakter kuralı iyi sonek kuralı

LZ

Abraham Lempel ve Jacob Ziv

1936–2023 · 1931–2023 · İsrailli · Technion

LZ77 · LZ78

Technion'da 1977'de kayan bir pencere içindeki önceki tekrarlara geri başvuran LZ77'yi, 1978'de görülen öbekleri bir sözlükte toplayan LZ78'i yayımladılar. Terry Welch 1984'te LZ78'i sadeleştirerek LZW'yi elde etti; GIF biçimi LZW'yi kullanır, ZIP ve PNG'nin kullandığı DEFLATE ise LZ77'yi Huffman kodlamayla birleştirir. Bu yöntemler, önceden olasılık tablosu gerektirmeyen “evrensel” sıkıştırmanın temelidir.

DERSTE **Bölüm 5.1 · Lempel-Ziv-Welch (LZW)** **Bölüm 5.1 · LZ78: LZW'nin Atası**

sözlük tabanlı sıkıştırma LZW

JK

Joseph B. (Jay) Kadane

d. 1941 · Amerikalı · Carnegie Mellon

En büyük toplamlı alt dizi · 1977

Ulf Grenander'ın 1977'de görüntü analizi için ortaya koyduğu en büyük toplamlı alt dizi problemini Michael Shamos, Carnegie Mellon'daki bir seminerde anlattı. Seminerdeki istatistikçi Kadane, problemi tek taramada çözen O(n) süreli algoritmayı buldu: V(i) = max(a[i], V(i−1) + a[i]) değeri i'de biten en iyi alt dizinin toplamıdır, cevap bu değerlerin en büyüğüdür. Öykü ve algoritma Jon Bentley'in 1984'teki Programming Pearls yazısıyla yaygınlaştı. Kadane Bayesçi istatistik alanındaki çalışmalarıyla tanınır.

DERSTE **Bölüm 8 · Diğer Klasik DP Problemleri**

Kadane algoritması en büyük toplamlı alt dizi O(n)

SK S. Rao Kosaraju

günümüz · Hint asıllı Amerikalı · Johns Hopkins

Kosaraju algoritması · 1978

1978'de yönlü bir çizgenin güçlü bağlantılı bileşenlerini iki DFS ile bulan yöntemi önerdi, ama yayımlamadı; Micha Sharir aynı algoritmayı bağımsız olarak bulup 1981'de yayımladı. Birinci DFS düğümlerin bitiş sırasını çıkarır. İkinci DFS kenarları ters çevrilmiş çizgede aramaları düğümlerin bitiş zamanlarının azalan sırasıyla başlatır; bu DFS'in her ağacı bir güçlü bağlantılı bileşendir. Komşuluk listesiyle süre O(V + E)'dir. Tarjan'ın tek DFS'li yönteminden daha kolay anlaşılır, ama çizgeyi iki kez gezer.

DERSTE **Bölüm 4.1 · Karmaşıklık ve Kosaraju** **Bölüm 4.1 · Kosaraju: Kod İzleme**

güçlü bağlantılı bileşenler iki DFS ters çizge

GJ Michael Garey ve David S. Johnson

d. 1945 · 1945–2016 · Amerikalı · Bell Labs

Computers and Intractability · 1979

Bell Labs'ta yazdıkları Computers and Intractability: A Guide to the Theory of NP-Completeness (1979), yüzlerce NP-tam problemi listeleyen ve indirgeme tekniklerini öğreten temel başvuru kitabı oldu. Johnson 1974'te kutulama ve küme örtüsü gibi problemler için yaklaşım algoritmalarının ilk sistemli analizlerinden birini yaptı. Bir problemin NP-tam olduğunu kanıtlamanın yaklaşım algoritmalarına (approximation algorithm) ya da sezgisel aramaya yönelmeyi haklı çıkardığı görüşü bu çalışmalarla yerleşti.

DERSTE **Bölüm 11 · Neden Yaklaşım Algoritmaları?** **Bölüm 9 · P, NP ve NP-Zor**

NP-tamlık yaklaşım algoritması

LV Leslie Valiant

d. 1949 · Macaristan doğumlu · Birleşik Krallık; ABD, Harvard

#P · PAC · BSP · 2010 Turing Ödülü

1979'da sayma problemleri için #P sınıfını tanımladı ve permanent hesabının #P-tam olduğunu gösterdi. Makine öğrenmesinin kuramsal temellerinden PAC öğrenme modelini 1984'te, paralel hesaplama için toplu eş zamanlı paralel (bulk synchronous parallel, BSP) köprü modelini 1990'da önerdi. Vazirani ile rastgele indirgemeler üzerinde çalıştı ve 2010'da Turing Ödülü'nü aldı.

DERSTE **Bölüm 14 · PRAM Modeli** **Bölüm 12 · Rastgele Algoritmalar**

#P paralel model rastgele indirgeme

JB Jon Bentley

d. 1953 · Amerikalı · Carnegie Mellon, Bell Labs

Böl ve fethet bağıntıları · 1980

Haken ve Saxe ile 1980'de T(n) = aT(n/b) + f(n) biçimindeki böl ve fethet bağıntılarını çözmek için genel bir yöntem yayımladı; ana teorem (master theorem) bu çalışmayla yaygınlaştı. Yao ile 1976'da sınırsız arama için neredeyse optimal bir yöntem verdi; bu yöntem üstel aramanın (exponential search) temelidir. 1975'te k-d ağacını tanıttı, Programming Pearls yazılarıyla verimli programlamayı geniş bir kitleye ulaştırdı.

DERSTE **Bölüm 1.2 · Böl ve Fethet Bağıntısı** **Bölüm 3 · Üstel Arama: Fikir**

ana teorem üstel arama

DS Daniel Sleator

d. 1953 · Amerikalı · Carnegie Mellon

Rekabetçi analiz · 1985

Tarjan ile yazdığı 1985 tarihli “Amortized Efficiency of List Update and Paging Rules” makalesinde bir çevrim içi algoritmayı (online algorithm) geleceği bilen en iyi çevrim dışı algoritmayla karşılaştırmayı önerdi; öne taşı (move-to-front, MTF) kuralının 2-rekabetçi, LRU'nun k-rekabetçi olduğunu gösterdiler. “Rekabetçi analiz” (competitive analysis) adı Karlin, Manasse, Rudolph ve Sleator'ın 1988 tarihli makalesinde kondu. Tarjan ile splay ağaçlarını ve link-cut ağaçlarını da geliştirdi.

DERSTE **Bölüm 13 · Rekabet Oranı** **Bölüm 13 · MTF 2-Rekabetçidir**

Bölüm 13 · Üst Sınır: LRU k-Rekabetçi

rekabetçi analiz öne taşı sayfalama

JV Jeffrey S. Vitter

d. 1955 · Amerikalı · Brown University

Rezervuar örnekleme · 1985

1985 tarihli “Random Sampling with a Reservoir” makalesinde, uzunluğu bilinmeyen bir veri akışından (data stream) tek geçişte k öğelik düzgün rastgele örnek tutan yöntemleri analiz etti; temel yönteme Algorithm R adını verdi ve daha hızlı Algorithm Z'yi geliştirdi. Doktorasını Knuth'un yanında yaptı; dış bellek algoritmalarının ve veri sıkıştırmanın da öncülerindendir.

DERSTE **Bölüm 13 · Rezervuar Örnekleme** **Bölüm 13 · Veri Akışı Modeli**

rezervuar örnekleme (reservoir sampling) veri akışı

HS W. Daniel Hillis ve Guy L. Steele Jr.

d. 1956 · d. 1954 · Amerikalı · Thinking Machines

Paralel önek toplamı · 1986

1986'da Communications of the ACM'de yayımladıkları “Data Parallel Algorithms” makalesinde, binlerce işlemcili Connection Machine için veri paralel algoritmalar anlattılar. Önek toplamında $d = 1, 2, 4, \ldots$ için her $i \geq d$ konumu aynı anda $x[i] \leftarrow x[i - d] + x[i]$ işlemini yapar ve hesap $\lceil \log n \rceil$ turda biter. Toplam iş $O(n \log n)$ olduğu için yöntem iş-verimli değildir; Blelloch'un yöntemi $O(\log n)$ derinlikte $O(n)$ iş yapar. Hillis Connection Machine'i tasarladı; Steele, Scheme dilinin tasarımcılarından ve Java dil belirtiminin yazarlarındandır.

DERSTE **Bölüm 14 · Paralel Önek Toplamı** **Bölüm 14 · Önek Toplamı Adım Adım**

önek toplamı veri paralelliği iş-verimlilik

JG John L. Gustafson

d. 1955 · Amerikalı · *Communications of the ACM* 31(5), 1988

Gustafson yasası · 1988

1988'de Edwin Barsis ile yazdığı “Reevaluating Amdahl's Law” yazısında Amdahl yasasının varsayımını sorguladı. Amdahl problem boyutunu sabit tutar, bu yüzden sıralı kısım hızlanmayı $1/(1 - f)$ ile sınırlar. Gerçekte işlemci sayısı arttıkça çoğu zaman daha büyük problemler çözülür; problem p ile büyürse ölçeklenmiş hızlanma $S = (1 - f) + f \cdot p$ olur ve işlemci sayısıyla neredeyse doğrusal artar. Aynı yıl ilk Gordon Bell Ödülü'nü aldı.

DERSTE **Bölüm 14 · Amdahl Yasası**

Gustafson yasası ölçeklenmiş hızlanma Amdahl yasası

MM

Udi Manber ve Gene Myers

günümüz · d. 1953 · Amerikalı · University of Arizona

Sonek dizisi · 1990

1990'da (dergi sürümü 1993) sonek ağacından çok daha az bellek kullanan sonek dizisini (suffix array) tanıttılar ve önek uzunluğunu her turda ikiye katlayarak sonekleri O(n log n) sürede sıralayan yöntemi verdiler. Myers daha sonra BLAST'ın geliştirilmesine ve insan genomunun dizilenmesine katkıda bulundu; Manber Yahoo!, Amazon ve Google'da arama teknolojileri üzerinde çalıştı.

DERSTE **Bölüm 5.2 · Sonek Dizisi** **Bölüm 5.2 · Önek İkiye Katlama**

sonek dizisi önek ikiye katlama

CL

Charles E. Leiserson

d. 1953 · Amerikalı · MIT

Cilk · iş çalma · CLRS

Cormen, Rivest ve Stein ile yazdığı Introduction to Algorithms (ilk baskı 1990), dünyada en çok kullanılan algoritma ders kitaplarından biri ve bu dersin ana kaynaklarındandır. MIT'de geliştirdiği Cilk dili, boşta kalan işlemcinin başka bir işlemcinin kuyruğundan iş aldığı, verimliliği kanıtlanmış bir iş çalma (work stealing) zamanlayıcısı kullanır; Java'nın fork/join çatısı da aynı fikre dayanır. İş/derinlik (work/span) analizinin yaygınlaşmasında da büyük payı vardır.

DERSTE **Bölüm 14 · İş ve Derinlik** **Bölüm 14 · Fork/Join Adım Adım**

iş çalma iş / derinlik

GB

Guy Blelloch

günümüz · Amerikalı · Carnegie Mellon

Önek toplamı (scan) · NESL

1990 tarihli “Prefix Sums and Their Applications” çalışmasında önek toplamını (prefix sum, scan) paralel algoritmaların temel yapı taşı olarak ele aldı; O(n) iş ve O(log n) derinlikle çalışan yukarı tarama/aşağı tarama yöntemi onun adıyla anılır. İş–derinlik maliyet modeline dayanan paralel programlama dili NESL'i geliştirdi. Carnegie Mellon'da paralel çizge işleme kütüphaneleri üzerinde çalışıyor.

DERSTE **Bölüm 14 · Paralel Önek Toplamı** **Bölüm 14 · Önek Toplamı Adım Adım**

önek toplamı iş / derinlik

DG

David Karger

d. 1967 · Amerikalı · Stanford, MIT

Rastgele minimum kesim · 1993

Stanford'da doktora öğrencisiyken, rastgele seçilen kenarları iki düğüm kalana kadar büzen minimum kesim algoritmasını 1993'te yayımladı. Tek bir çalıştırma minimum kesimi en az 2/(n(n − 1)) olasılıkla bulur; yeterince tekrarla hata olasılığı istenildiği kadar küçülür. Stein ile geliştirdiği Karger–Stein sürümü yöntemi çok daha hızlı hâle getirdi.

DERSTE **Bölüm 12 · Karger Minimum Kesim** **Bölüm 12 · Hata Olasılığını Küçültme**

minimum kesim Monte Carlo algoritması

BW

Michael Burrows ve David Wheeler

d. 1963 · 1927–2004 · İngiliz · Cambridge; DEC SRC, ABD

Burrows–Wheeler dönüşümü · 1994

Wheeler'ın 1983'te bulduğu dönüşümü Burrows ve Wheeler 1994'te DEC Systems Research Center'ın 124 numaralı araştırma raporu olarak yayımladı. Bir dizginin sıralanmış döndürmelerinin son sütunu benzer karakterleri bir araya toplar ve geri çevrilebilir; bzip2 sıkıştırıcısı ve FM-index tabanlı DNA hizalama araçları bu dönüşüme dayanır. Wheeler daha önce Cambridge'de EDSAC için alt program çağrısının ilk biçimlerinden birini geliştirmişti.

DERSTE **Bölüm 5.2 · Burrows–Wheeler Dönüşümü** **Bölüm 5.2 · BWT'yi Tersine Çevirme**

BWT sıkıştırma

VV

Vijay V. Vazirani

günümüz · Hindistan doğumlu · ABD, Georgia Tech, UC Irvine

Approximation Algorithms · 2001

Approximation Algorithms (2001) kitabı, yaklaşım algoritmalarını LP gevşetmesi ve primal–dual şeması gibi ortak tekniklerle anlatan temel ders kitabıdır. Micali ile genel çizgelerde maksimum eşleştirme için bilinen en hızlı algoritmalardan birini verdi (1980). Valiant ile 1986'da tek çözümlü SAT'ın da rastgele indirgemeler altında zor olduğunu gösterdi (Valiant–Vazirani teoremi).

DERSTE **Bölüm 11 · Yaklaşım Oranı** **Bölüm 11 · Yaklaşım Teknikleri**

yaklaşım algoritması LP gevşetmesi

Yıllar ve katkılar özgün yayınlara ve ansiklopedik kaynaklara dayanır. “Günümüz” ibaresi, doğum yılı yayımlanmış kaynaklarda verilmeyen yaşayan kişileri gösterir. Fermat'nın doğum yılı kaynaklarda tartışmalı olduğu için yalnız ölüm yılı verilmiştir; yaşam yılları güvenilir kaynaklarda bulunamayan kişilerde ise yalnız katkının yılı ve yayımlandığı dergi verilmiştir.