DERS KİTABI

İleri Algoritmalar

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

İleri Algoritmalar

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

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

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

© Sercan Külcü. Tüm hakları saklıdır.

Notların 16 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Bu kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen İleri Algoritmalar dersinin notlarından oluşur ve *Algoritmalar* kitabının devamıdır. Algoritma karmaşıklığı, sıralama, arama, çizge ve dizgi algoritmaları o kitapta anlatılır; burada bilindikleri varsayılır. Kitap, algoritma tasarımının temel yöntemlerini, zor problemlerle başa çıkma yollarını ve sıralı, belirlenimci modelin dışındaki hesaplama modellerini inceler. Bölümler dersteki sırayı izler; örnekler Java ile yazılmıştır.

Kitap üç kısımdır. Kısımlar sırasıyla şunlardır: **Tasarım Paradigmaları** (Bölüm 1–3), **Zor Problemler** (Bölüm 4–6) ve **Diğer Hesaplama Modelleri** (Bölüm 7–9).

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/advanced-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 · TASARIM PARADİGMALARI

1 Böl ve Fethet Yaklaşımı 4

1.1 Böl ve fethet paradigması 5
1.2 Yineleme bağıntıları 11
1.3 Temel örnekler 15
1.4 Dışbükey örtü ve Quickhull 21
1.5 En yakın nokta çifti 31
1.6 Karatsuba çarpımı 38
1.7 Strassen matris çarpımı 42
1.8 Değerlendirme 45
1.9 Sık yapılan hatalar 46
1.10 Özet 46
1.11 Kavram listesi 47
1.12 Kendini Dene 47
1.13 İleri okuma 49
1.14 Kaynaklar 49

2 Açgözlü Algoritmalar 50

2.1 Açgözlü yaklaşım 51
2.2 Para üstü problemi 56
2.3 Etkinlik seçimi 61
2.4 Son tarihli iş sıralama 67
2.5 Kesirli sırt çantası 71
2.6 Minimum kapsayan ağaçta açgözlü seçim 75
2.7 Huffman kodlama 79
2.8 Diğer açgözlü algoritmalar 88
2.9 Karşılaştırma ve sık yapılan hatalar 89
2.10 Özet 90
2.11 Kavram listesi 91
2.12 Kendini Dene 91
2.13 İleri okuma 94
2.14 Kaynaklar 94

3 Dinamik Programlama 95

3.1 Temel kavramlar 96
3.2 Fibonacci: bellekleme ve tablolama 102
3.3 0/1 sırt çantası 107
3.4 Sınırsız sırt çantası 116
3.5 Alt küme toplamı 120

3.6 En uzun artan alt dizi 124
3.7 En uzun ortak alt dizgi 132
3.8 En uzun ortak alt dizi 135
3.9 Diğer klasik DP problemleri 139
3.10 Sık yapılan hatalar 144
3.11 Özet 145
3.12 Kavram listesi 146
3.13 Kendini Dene 146
3.14 İleri okuma 148
3.15 Kaynaklar 149

KISIM II · ZOR PROBLEMLER

4 Geri İzleme Algoritmaları 151

4.1 Zor problemler: P, NP ve NP-zor 152
4.2 Geri izlemenin temel fikri 155
4.3 n-vezir problemi 166
4.4 En az çakışma sezgiseli 179
4.5 Sudoku 183
4.6 Çizge boyama 190
4.7 Hamilton yolu 196
4.8 Kelime arama 201
4.9 Diğer uygulamalar 204
4.10 Analiz 207
4.11 Sık yapılan hatalar 209
4.12 Özet 210
4.13 Kavram listesi 211
4.14 Kendini Dene 211
4.15 İleri okuma 213
4.16 Kaynaklar 214

5 Dal Sınır Algoritmaları 215

5.1 Dal sınır fikri 216
5.2 Paradigma 219
5.3 0/1 sırt çantası 225
5.4 Gezgin satıcı problemi 236
5.5 Atama problemi 247
5.6 Tam sayılı doğrusal programlama 250
5.7 Maksimum klik 252
5.8 Diğer problemler ve A* ile ilişki 256
5.9 Tasarım paradigmalarının karşılaştırması 257
5.10 Sık yapılan hatalar 258
5.11 Özet 259
5.12 Kavram listesi 259
5.13 Kendini Dene 260
5.14 İleri okuma 262

5.15 Kaynaklar 262

6 Yaklaşım Algoritmaları 264

6.1 Yaklaşım algoritmaları 265
6.2 Gezgin satıcı problemi 270
6.3 Yerel arama: 2-opt 280
6.4 İş çizelgeleme 288
6.5 Düğüm kapsama 294
6.6 Sırt çantası 302
6.7 Kuvvet yinelemesi 304
6.8 Karşılaştırma ve sık yapılan hatalar 310
6.9 Özet 311
6.10 Kavram listesi 311
6.11 Kendini Dene 312
6.12 İleri okuma 314
6.13 Kaynaklar 314

KISIM III · DİĞER HESAPLAMA MODELLERİ

7 Rastgele Algoritmalar 316

7.1 Rastgelelik ve algoritmalar 317
7.2 Las Vegas ve Monte Carlo 321
7.3 Karıştırma 324
7.4 BogoSort 328
7.5 Rastgele QuickSort 332
7.6 Rastgele seçme (QuickSelect) 339
7.7 Monte Carlo ile π 342
7.8 Miller–Rabin asallık testi 345
7.9 Karger minimum kesim algoritması 351
7.10 Diğer uygulamalar 357
7.11 Sözde rastgele sayılar 358
7.12 Sık yapılan hatalar 359
7.13 Özet 361
7.14 Kavram listesi 361
7.15 Kendini Dene 362
7.16 İleri okuma 364
7.17 Kaynaklar 364

8 Çevrim İçi Algoritmalar 365

8.1 Çevrim içi problem modeli 366
8.2 Rekabetçi analiz 370
8.3 Kayak kiralama 372
8.4 Sayfalama 379
8.5 Liste güncelleme 396
8.6 Veri akışları 400
8.7 Artımlı öğrenme 409

8.8 Problemler ve sonuçlar 411
8.9 Sık yapılan hatalar 411
8.10 Özet 412
8.11 Kavram listesi 413
8.12 Kendini Dene 413
8.13 İleri okuma 415
8.14 Kaynaklar 415

9 Paralel Algoritmalar 416

9.1 Paralel algoritma fikri 417
9.2 Paralel hesaplama modelleri 420
9.3 Başarım ölçütleri 423
9.4 PRAM ile toplama 429
9.5 Paralel önek toplamı 434
9.6 Boru hattı 439
9.7 Boru hatlı ağaç 444
9.8 Paralel algoritma türleri ve Java 446
9.9 Paylaşılan durum ve yarış 454
9.10 Sık yapılan hatalar 459
9.11 Özet 460
9.12 Kavram listesi 461
9.13 Kendini Dene 461
9.14 İleri okuma 463
9.15 Kaynaklar 463

EKLER

Ek A Terim Sözlüğü 465

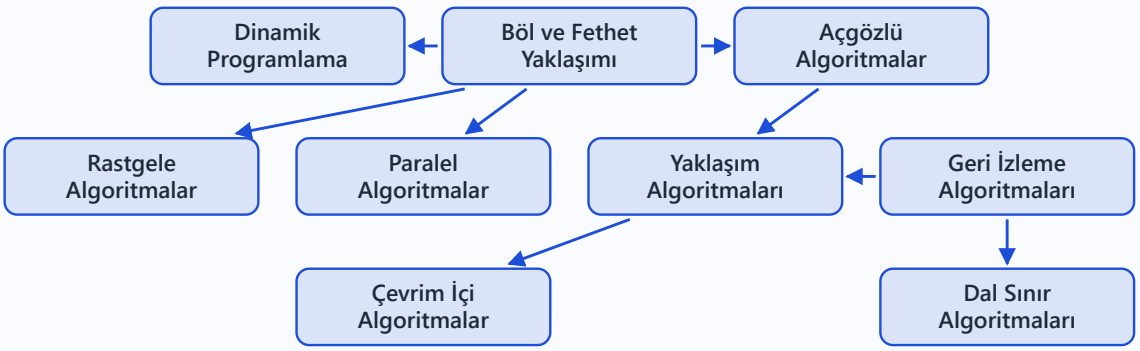
Ek B İleri Algoritmalar İçin Ön Bilgiler 470

Ek C İleri Algoritmalarda Önemli İsimler 485

İLERİ 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. Üst satırda tasarım paradigmaları, sağda zor problemler, solda diğer hesaplama modelleri yer alır.

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
Böl ve Fethet Yaklaşımı (1)	—	Birleştirme sıralaması ve hızlı sıralamanın ortak şablonu; yineleme bağıntısı ve Ana Teorem (Algoritmalar, Bölüm 2–3).
Açgözlü Algoritmalar (2)	Böl ve Fethet Yaklaşımı	Her adımda en iyi görünen seçim; Prim, Kruskal ve Dijkstra açgözlü çizge algoritmalarıdır (Algoritmalar, Bölüm 6–7).
Dinamik Programlama (3)	Böl ve Fethet Yaklaşımı	Böl ve fethetin örtüşen alt problemli hâli: bellekleme ve tablolama; dizgilerde LCS ve düzenleme uzaklığı (Algoritmalar, Bölüm 10).
Geri İzleme Algoritmaları (4)	—	Durum uzayı ağacında derinlik öncelikli arama (Algoritmalar, Bölüm 5); P, NP ve NP-zor problemlere giriş.
Dal Sınır Algoritmaları (5)	Geri İzleme Algoritmaları	Geri izlemeye sınır eklenir: en iyi çözümü içeremeyecek dallar budanır.

Konu (bölüm)	Dayandığı konu	Bağ
Yaklaşım Algoritmaları (6)	Açgözlü Algoritmalar, Geri İzleme Algoritmaları	NP-zor problemlerde kanıtlanmış oranlı çözüm: MST tabanlı 2-yaklaşım, LPT, düğüm kapsama.
Rastgele Algoritmalar (7)	Böl ve Fethet Yaklaşımı	Rastgele pivotlu hızlı sıralama ve QuickSelect; Las Vegas ve Monte Carlo türleri.
Çevrim İçi Algoritmalar (8)	Açgözlü Algoritmalar, Yaklaşım Algoritmaları	Geleceği bilmeden karar: rekabet oranı yaklaşım oranının çevrim içi karşılığıdır; sayfalama.
Paralel Algoritmalar (9)	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: **tasarım paradigmaları → zor problemler → diğer modeller**. Ders, Algoritmalar dersindeki analiz, sıralama, arama, çizge ve dizgi algoritmalarının üzerine kurulur. Böl ve fethet sıralamadan doğar ve dinamik programlamaya, rastgele ve paralel algoritmalara açılır. Açgözlü yöntem çizge algoritmalarından tanıdıktır. 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

Tasarım Paradigmaları

Bölüm 1–3

KISIM I · BÖLÜM 1

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/advanced-algorithms-java/Bolum01/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.

1.1 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.

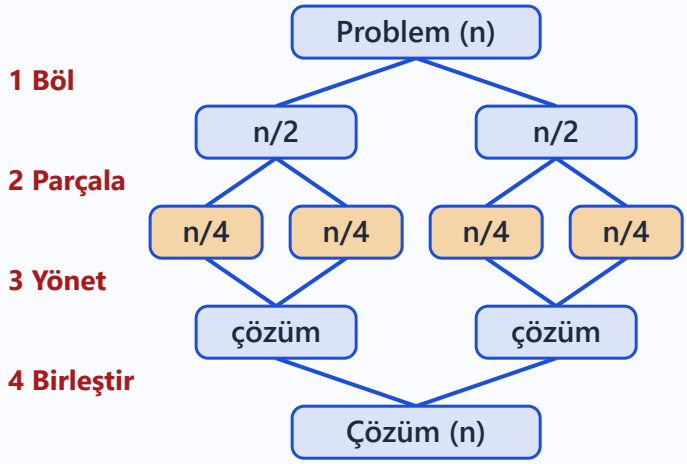
1.1.1 Böl, parçala, yönet, birleştir

Tanım 1.1

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 1.1: 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 1.1 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

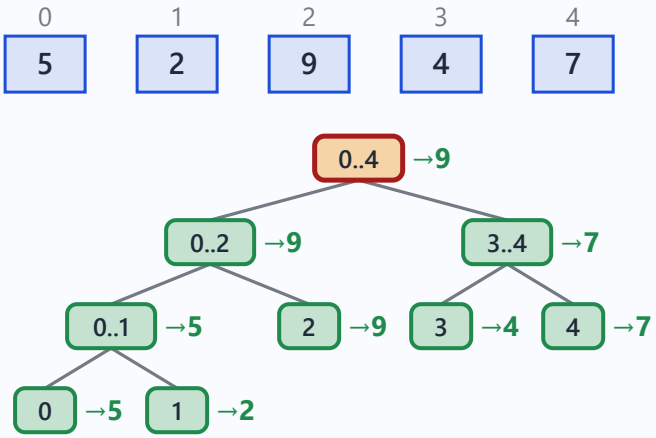
Bölme yeni dizi oluşturmaz; yalnız iki indis hesaplar. Bu yüzden bölme ve birleştirme birlikte sabit sürelidir: $f(n) = O(1)$. Bağıntı $T(n) = 2T(n/2) + O(1)$ olur ve çözümü $\Theta(n)$'dir.

1.1.5 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 1.1 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 1.1: `enBuyuk(a, 0, 4)` izlemesi, `a = [5, 2, 9, 4, 7]`. Çağrılar `(alt, ust)` çiftiyle gösterilir; çağrı yığını soldan sağa büyür.



Şekil 1.2: `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).

1.1.6 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 1.1 (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?

1.2 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.

1.2.1 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 1.1

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.

1.2.2 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 1.2 (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 1.2

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.

1.2.3 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 1.2 $n = 16$ 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 1.2: $n = 16$ 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.

1.2.4 Bu bölümün algoritmaları ve ana teorem

Tablo 1.3 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 1.3: 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 $\log_2 7 \approx 2{,}807$.

1.2.5 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.

1.3 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.

1.3.1 Klasik böl ve fethet algoritmaları

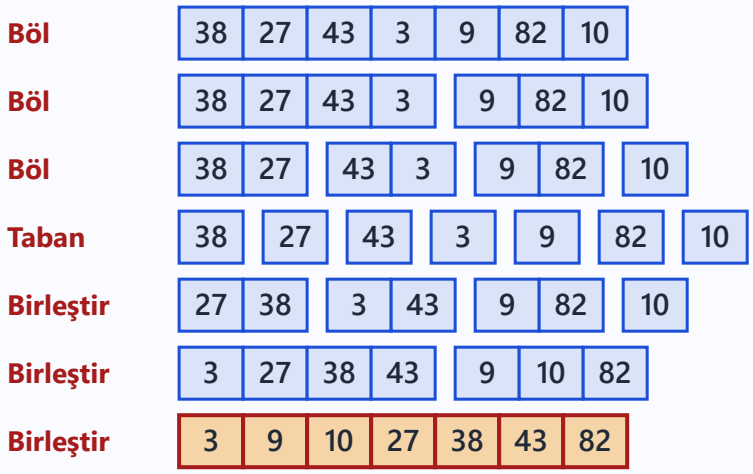
Tablo 1.4 bu notta adı geçen algoritmaların böl ve birleştir adımlarını ve bağıntılarını tek tabloda toplar. Tabloya satır satır değil, sütun sütun bakmak öğreticidir. “Birleştir” sütununda iki algoritmanın karşısında “yok” yazar: ikili aramada ve hızlı sıralamada iş ya bölme anında yapılır ya da sonuç 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 1.4: Klasik böl ve fethet algoritmaları.

1.3.2 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.



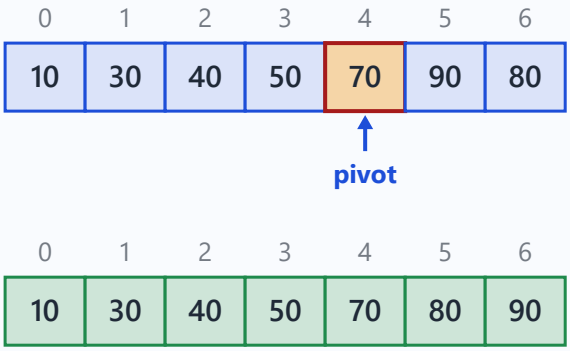
Şekil 1.3: `[38, 27, 43, 3, 9, 82, 10]` dizisinin birleştirme sıralaması. Üstteki üç satır bölmeyi, "Taban" satırı tek öğeli dizileri, alttaki üç satır birleştirmeleri gösterir. Son satır sıralı dizidir: `[3, 9, 10, 27, 38, 43, 82]`.

Şekil 1.3 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.

1.3.3 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 1.4: `[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 1.4'teki örnekte Lomuto bölümlemesi kullanılır: aralığın son öğesi pivottur. Tablo 1.5 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 1.5: `[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.

1.3.4 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.

1.3.5 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.

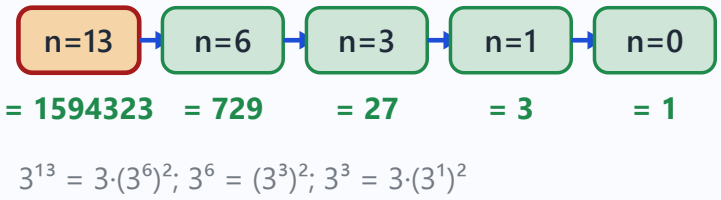
1.3.6 Kod izleme: us(3, 13)

`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 1.6 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 1.6: `us(3, 13)` izlemesi. Satırlar dönüş sırasındadır: en derin çağrı önce döner.



Şekil 1.5: `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

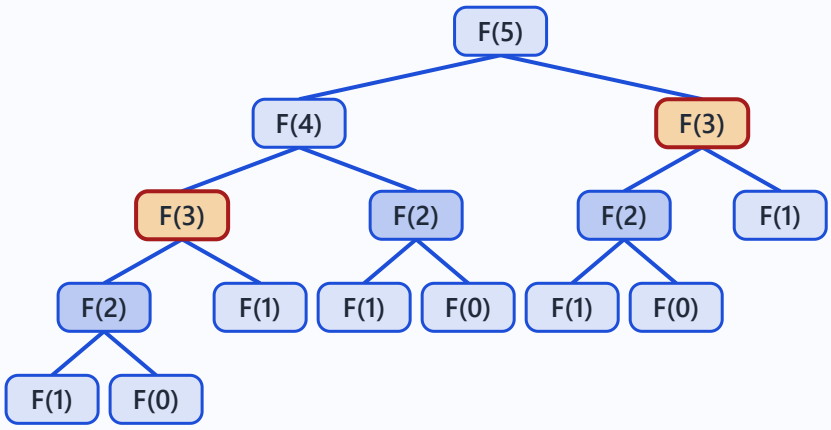
`y` değişkenini kullanmayıp yarım üssü her kullanımda yeniden hesaplamak, örneğin çift n için `return us(x, n / 2) * us(x, n / 2);`, tek n için `return x * us(x, n / 2) * us(x, n / 2);` yazmak. Kod yine doğru sonucu verir, ama her çağrı artık iki özyinelemeli çağrı yapar. Bağıntı $T(n) = 2T(n/2) + O(1)$ 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.

1.3.7 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 1.6: `fib(5)` çağrı ağacı. Vurgulu düğümler aynı alt problemin tekrarlarıdır: *F(3)* iki kez, *F(2)* üç kez hesaplanır.

Şekil 1.6'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.

1.4 Dışbükey örtü ve Quickhull

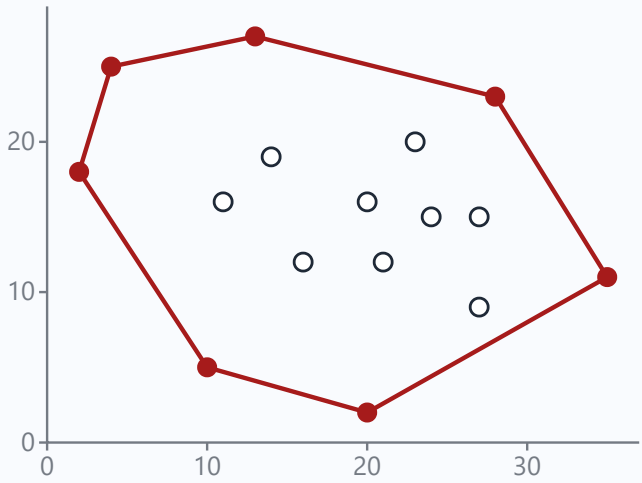
Hesaplamalı geometri, noktalar, doğrular ve çokgenler üzerinde çalışan algoritmaları inceler. Bu alanın en temel problemlerinden biri dışbükey örtüdür. Problemin böl ve fethetle çözümü olan Quickhull, adını hızlı sıralamaya benzerliğinden alır: bir "pivot" seçilir, noktalar bu pivota göre bölümlenir ve parçalar özyinelemeyle işlenir.

1.4.1 Dışbükey örtü problemi

Tanım 1.2

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 1.7: 16 noktalı bir küme ve 7 köşeli dışbükey örtüsü. Dolu noktalar örtünün köşeleridir; içi boş noktalar örtünün içinde kalır.

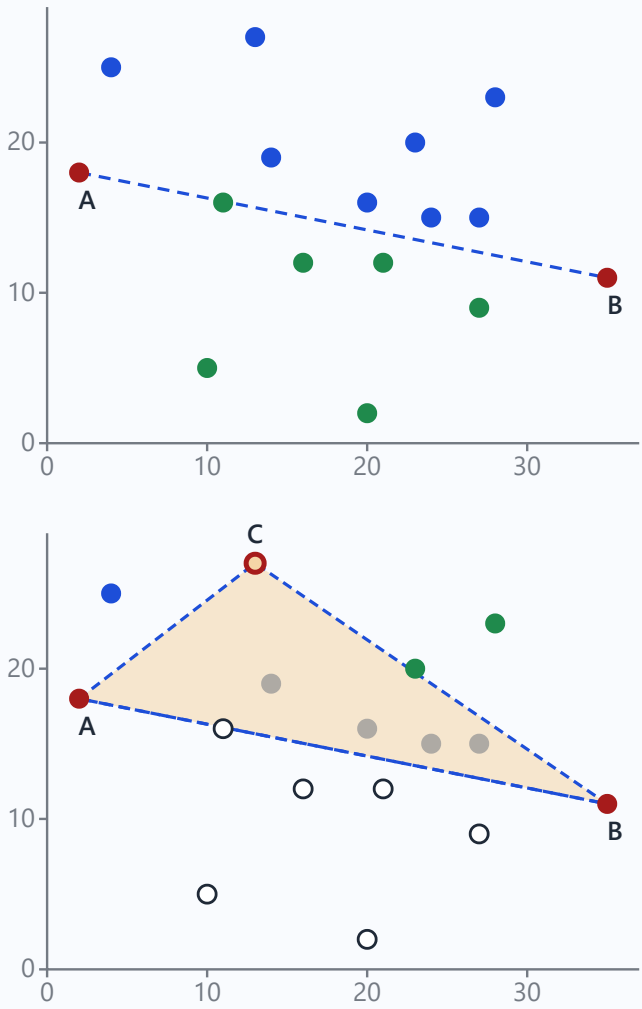
Algoritmanın çıktısı, örtünün köşelerinin sıralı listesidir; geleneksel olarak saat yönünün tersine sıralanır. Yalnız “hangi noktalar örtüdedir?” sorusunu yanıtlamak yetmez, çünkü çokgeni çizmek ya da üzerinde hesap yapmak için köşelerin sırası gerekir. Şekil 1.7'deki kümenin örtüsü yedi köşelidir; kalan 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.

1.4.2 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 1.8: 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.

1.4.3 Yönelim testi ve uzaklık

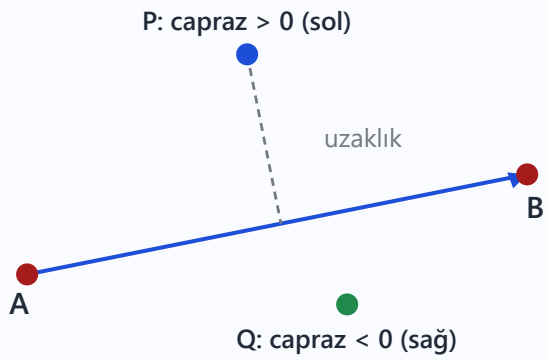
Algoritma iki geometrik soruyu sürekli sorar: bir *P* noktası *A* → *B* doğrusunun solunda mı, sağında mı? Ve *P*'nin *AB*'ye uzaklığı ne kadar? İki soru da tek bir formülle yanıtlanır. Formül yalnız tam sayı çarpımı ve çıkarması kullanır; açı, karekök ya da bölme gerekmez.

```
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 1.9: Yönelim testi. P, $A \to B$'nin solundadır ve `capraz` pozitiftir; Q sağındadır ve `capraz` negatiftir. Kesikli çizgi P'nin doğruya dik uzaklığıdır.

Örnek 1.3

$A = (2, 18)$ 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.

1.4.4 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.

1.4.5 Kod izleme: 16 nokta

İzlemede Şekil 1.7'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 1.7 `bul` çağrılarını sırayla verir.

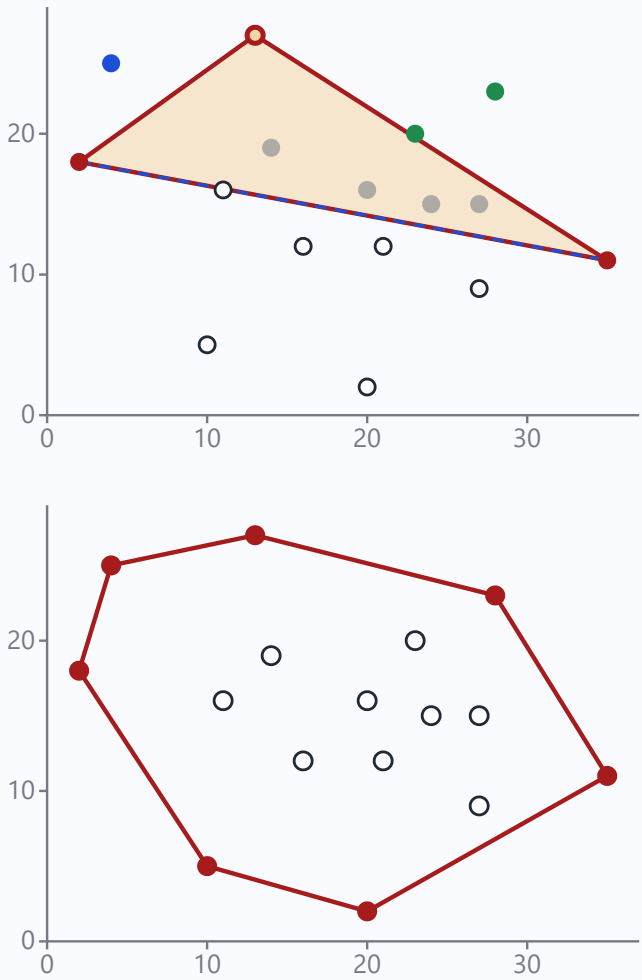
Çağrı	Doğru	$\|S\|$	En uzak C	Elenen	AC dışı	CB dışı
1	(2,18) → (35,11)	8	(13,27)	4	1	2
2	(2,18) → (13,27)	1	(4,25)	0	0	0
3	(2,18) → (4,25)	0	S boş: örtü kenarı			
4	(4,25) → (13,27)	0	S boş: örtü kenarı			
5	(13,27) → (35,11)	2	(28,23)	1	0	0
6	(13,27) → (28,23)	0	S boş: örtü kenarı			
7	(28,23) → (35,11)	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 1.7: 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 1.10: 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.

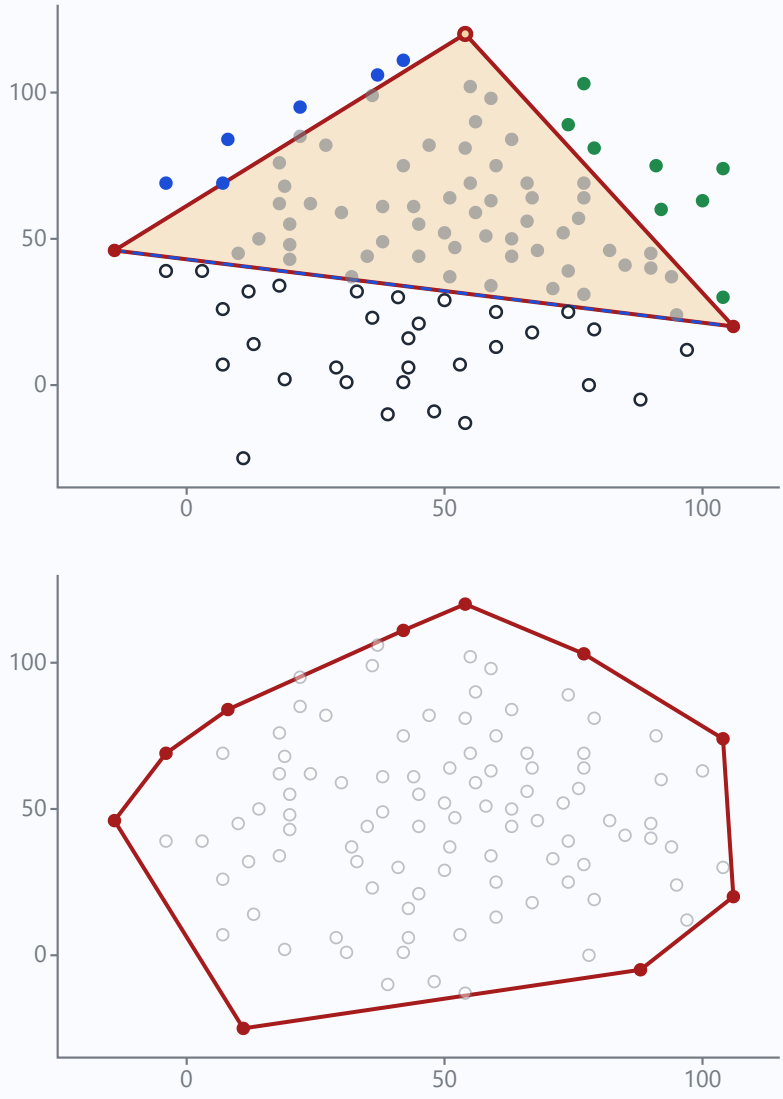
1.4.6 104 noktalı örnek

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 1.8 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 1.8: 104 noktalı kümede Quickhull'un dolu çağrıları.



Şekil 1.11: 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.

1.4.7 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 1.9 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 1.9: 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.

1.5 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.

1.5.1 Problem ve kaba kuvvet

Tanım 1.3

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 1.12: 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.

1.5.2 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.

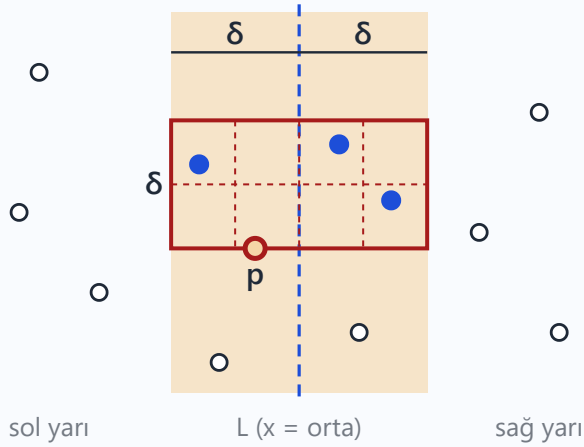
1.5.3 Şeritte neden en çok yedi komşu?

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 1.3 (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 1.13: 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.

1.5.4 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);
```

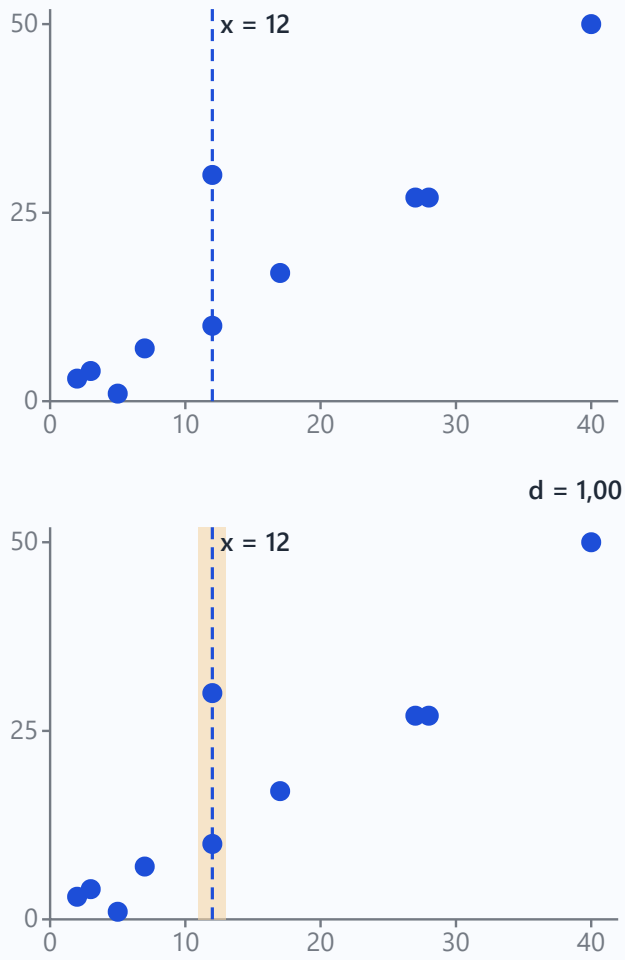
Bu parça `Ysol` ve `Ysag`'ı tek geçişte kurar. Sol yarının noktaları bir `HashSet`'e konur; `Y` sırayla gezilir ve her nokta, sol yarıda olup olmamasına göre iki listeden birine eklenir. `Nokta` sınıfı `equals` metodunu yeniden tanımlamadığı için küme nesnelerin kimliğiyle çalışır; aynı koordinatlı iki farklı nokta bile karışmaz. Geçiş beklenen *O(n)* sürer ve listeler *y* sırasını korur. Yeniden sıralama yapılmaz.

1.5.5 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 1.10: `enYakin(X, Y)` izlemesi. Uzaklıklar iki ondalık basamağa yuvarlanmıştır.



Şekil 1.14: 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 1.4

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 $3 \geq 2{,}83$; 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.

1.5.6 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 1.11: 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 $n \leq 3$ yerine $n \leq 1$ 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.

1.6 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.

1.6.1 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.

1.6.2 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:

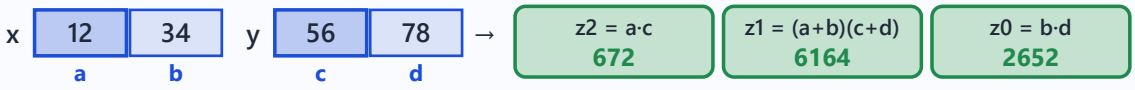
$$z_2 = ac, \quad z_0 = bd, \quad z_1 = (a + b)(c + d), \quad x \cdot y = z_2 \cdot 10^{2m} + (z_1 - z_2 - z_0) \cdot 10^m + z_0$$

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 1.5

$1234 \cdot 5678$ 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 1.15: 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.

1.6.3 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();
}
```

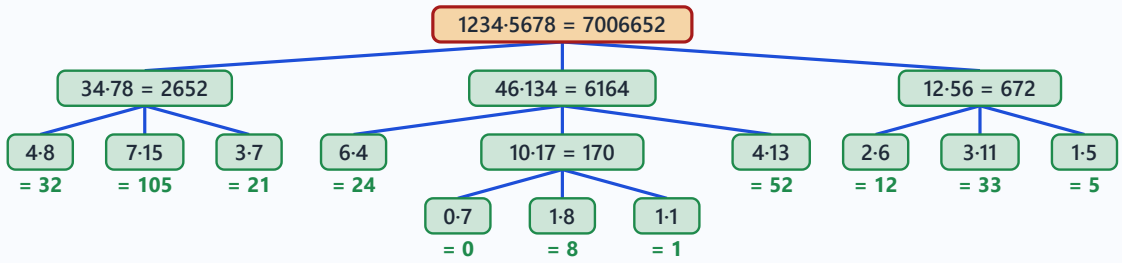
Bu Java kodu algoritmayı göstermek içindir. `long` türü en çok 19 basamaklı sayıları tuttuğu için büyük sayılarda kullanılamaz. Gerçek uygulamalar sayıları basamak dizileri olarak tutar; Java'da bu iş `BigInteger` sınıfıyla yapılır. Kodun doğruluğu yine de denetlenebilir: rastgele seçilen 100.000 sayı çiftinde `karatsuba(x, y)` ile `x * y` aynı sonucu vermiştir.

1.6.4 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 1.12 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 1.12: `karatsuba(1234, 5678)` izlemesi. Girintiler çağrı derinliğini gösterir.



Şekil 1.16: `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.

1.6.5 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 1.13: 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.

1.7 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.

1.7.1 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];
```

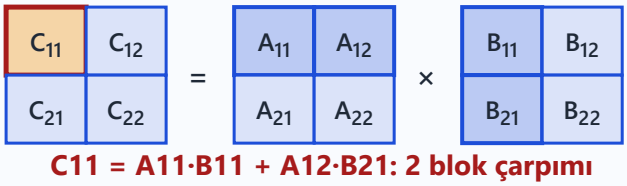
n^2 hücrenin her biri için n çarpım yapılır: toplam n^3 çarpım. Kodda `C`'nin başlangıçta sıfırlarla dolu olduğu varsayılır; Java'da yeni oluşturulan dizi zaten sıfırlarla doludur.

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 1.17: 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.

1.7.2 Sekiz yerine yedi çarpım

Strassen, yedi ara çarpımın uygun toplamlarıyla dört sonuç bloğunun da elde edilebileceğini gösterdi. Tablo 1.14 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 1.14: 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.

1.7.3 Adım adım: 2 × 2 örnek

Örnek 1.6

$A = [[1, 3], [7, 5]]$ 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.

1.7.4 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.

1.8 Değerlendirme

Tablo 1.15 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 1.15: 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.

1.9 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.

1.10 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.

1.11 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*

1.12 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.

1.13 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.

1.14 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 I · BÖLÜM 2

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/advanced-algorithms-java/Bolum02/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.

2.1 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.

2.1.1 Adım adım karar vermek

> **Tanım 2.1**
>
> **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.

2.1.2 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 2.1: 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 2.1 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.

2.1.3 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 2.2

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 2.3

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.

2.1.4 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 2.1

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.

2.1.5 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.

2.1.6 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 2.1 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 2.1: 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.

2.2 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.

2.2.1 Problem ve açgözlü ölçüt

Tanım 2.4

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.

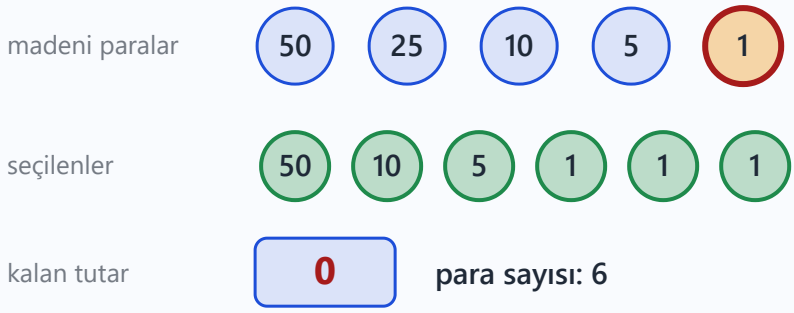
2.2.2 Kanonik bir sistemde adım adım

Sunumdaki ilk örnekte paralar *{50, 25, 10, 5, 1}*, tutar 68'dir. Tablo 2.2 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 2.2: *{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 2.2: 68 tutarının açgözlü ödenmesinin son durumu. Seçilenler 50, 10, 5, 1, 1, 1; kalan tutar 0, para sayısı 6.

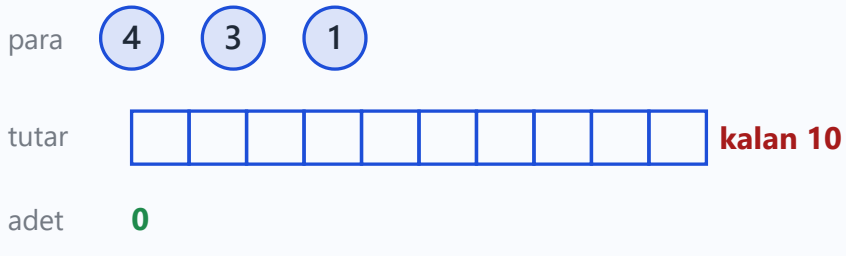
Açgözlü sonucun her tutar için en iyi olduğu para sistemlerine **kanonik** para sistemi denir. *{50, 25, 10, 5, 1}* ve *{25, 10, 5, 1}* kanonik sistemlerdir. Türk lirasının madeni paralarına karşılık gelen *{100, 50, 25, 10, 5, 1}* kuruş da kanoniktir. Euro sisteminin *{200, 100, 50, 20, 10, 5, 2, 1}* 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.

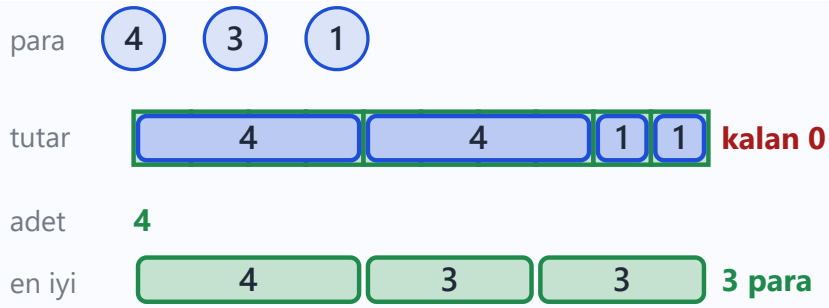
2.2.3 Kod izleme: {4, 3, 1} ve 10

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 2.3 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 2.3: `paraUstu({4, 3, 1}, 10)` izlemesi.





Şekil 2.3: `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.

2.2.4 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 2.4 bu iki sonucu gösterir.



Şekil 2.4: *{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 2.2

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.

2.2.5 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 2.4 *{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 2.4: *{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.

2.3 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.

2.3.1 Problemin tanımı

Tanım 2.5

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 2.5).

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 2.5: Sunumdaki on etkinlik.

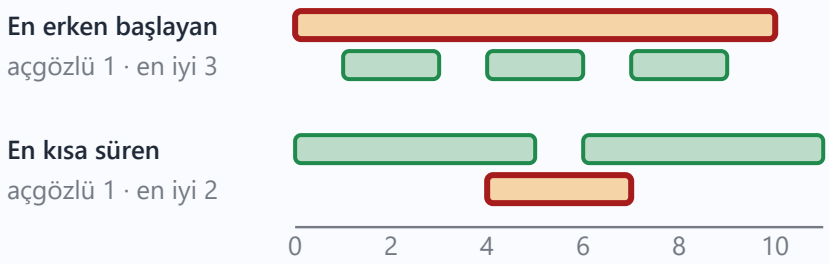
Kaba kuvvetle bütün alt kümeleri denemek 2^n küme demektir; on etkinlik için 1024 küme, kırk etkinlik için bir trilyonu aşan küme. Açgözlü algoritma ise etkinlikleri bir kez sıralar ve tek geçişte seçer.

2.3.2 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 2.5 ilk iki ölçüt için sunumdaki karşı örnekleri gösterir.



Şekil 2.5: 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 2.6 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 2.6: “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.

2.3.3 Kod ve izleme

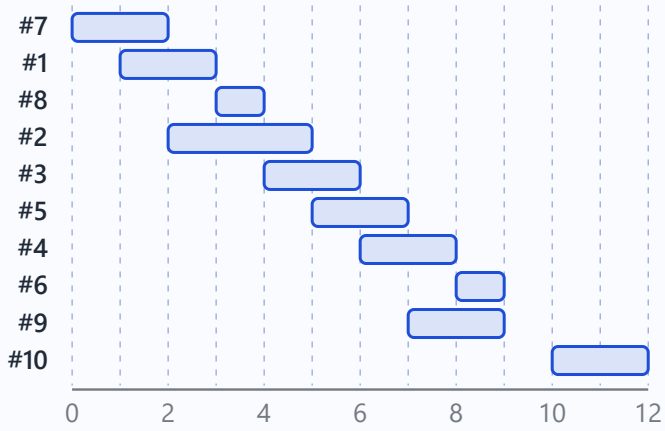
Sunumdaki `sec` metodu etkinlikleri bitiş zamanına göre sıralar, ilkini seçer ve sonra her etkinliği son seçilenin bitişiyle karşılaştırır:

```
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 2.7 döngünün her adımını verir.



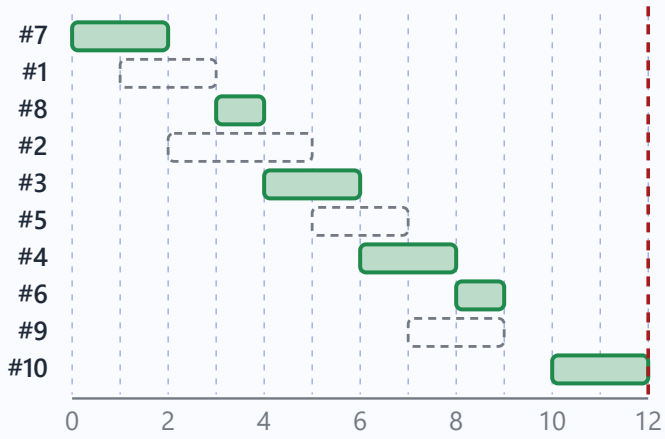
Şekil 2.6: Bitiş zamanına göre sıralanmış on etkinlik (yukarıdan aşağı). Yatay eksen zamandır.

`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 2.7: `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 2.7: 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 2.3

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.

2.3.4 En erken biten neden doğrudur?

Teorem 2.1 (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 2.5'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.

2.4 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.

2.4.1 Problemin tanımı

Tanım 2.6

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 2.8).

İş	1	2	3	4	5	6
son tarih	4	1	1	1	3	2
kâr	20	10	40	30	50	30

Tablo 2.8: 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.

2.4.2 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.

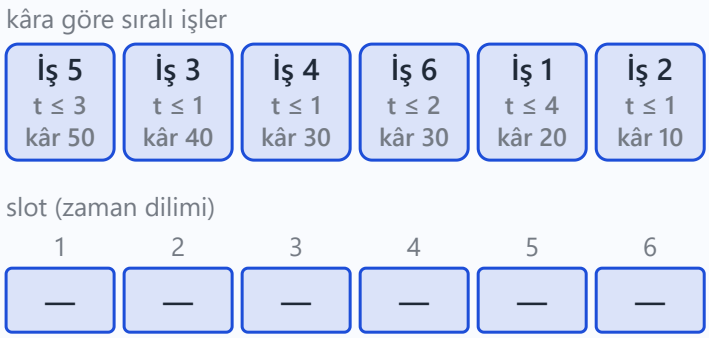
İkinci kararın gerekçesi şudur. Bir işi gereğinden erken bir dilime koymak, son tarihi daha yakın olan bir işin yerini kapatabilir. Örneğin son tarihi 3 olan bir iş birinci dilime konursa, sonradan gelecek ve son tarihi 1 olan bir işin gidebileceği tek yer dolmuş olur. Aynı iş üçüncü dilime konsaydı birinci dilim boş kalırdı. En geç boş dilim, erken dilimleri son tarihi daha yakın işlere bırakır ve esnekliği korur.

2.4.3 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.

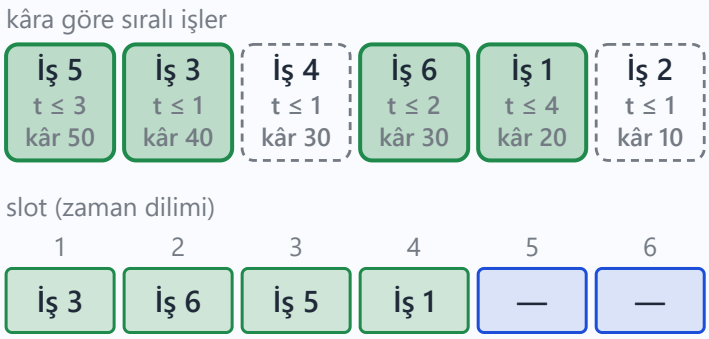


Şekil 2.8: İşler kâra göre azalan sırada; altı dilim boş. Eşit kârlı İş 4 ve İş 6 ilk sıralarını korur.

Kâra göre sıralanmış dizi İş 5 (50), İş 3 (40), İş 4 (30), İş 6 (30), İş 1 (20), İş 2 (10) şeklindedir. Tablo 2.9 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 2.9: `isSirala(is, 6)` izlemesi.



Şekil 2.9: 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 2.4

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.

2.4.4 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.

2.4.5 Doğruluk ve maliyet

Algoritmanın doğruluğu iki parçada gösterilir. Birinci parça, en geç boş dilim kuralının bir işi ancak gerçekten sığmıyorsa reddettiğidir.

Teorem 2.2 (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.

2.5 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.

2.5.1 Problem ve oran ölçütü

Tanım 2.7

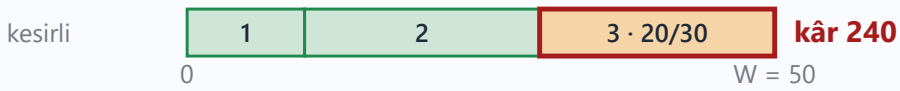
Kesirli sırt çantası (*fractional knapsack*): n nesne ve bir W kapasitesi verilir; i. nesnenin kârı k_i, ağırlığı w_i'dir. Her nesneden $0 \le x_i \le 1$ oranında bir parça alınabilir. Toplam ağırlık $\sum x_i\, w_i \le W$ koşulunu sağlayan ve toplam kârı $\sum x_i\, k_i$ en büyük yapan seçim bulunmak 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 2.10).

Nesne	Kâr	Ağırlık	Oran
1	60	10	6
2	100	20	5
3	120	30	4

Tablo 2.10: 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 2.10: 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.

2.5.2 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 2.11: `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 2.11: 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 2.5

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.

2.5.3 Oran ölçütü neden en iyidir?

Teorem 2.3 (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.

2.5.4 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 2.12: 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.

2.6 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.

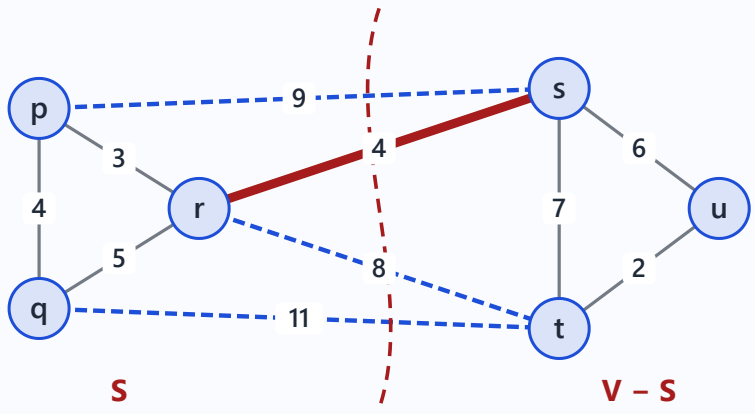
2.6.1 Kesim özelliği

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 2.4 (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 2.12: 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 2.12'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 2.6

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.

2.6.2 Kruskal, Prim ve Borůvka: üç kesim

Üç MST algoritması aynı özelliği farklı kesimlerle kullanır. Fark, her adımda hangi kesimin seçildiğidir. Tablo 2.13 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 2.13: 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.

2.7 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.

2.7.1 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.

2.7.2 Önek-özgür kodlar

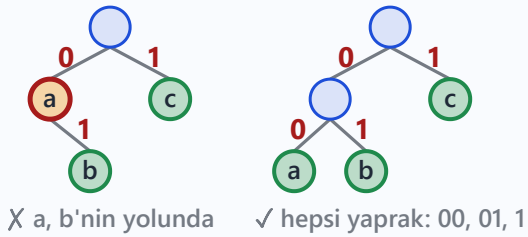
Kodlar a = 0, b = 01, c = 1 olsun. “01” bit dizisi alındığında bunun “ab” mi yoksa yalnız “b” mi olduğu anlaşılamaz. Sorunun kaynağı, a'nın kodu 0'ın, b'nin kodu 01'in başında yer almasıdır. Kod sözcükleri arasına bir ayırıcı konarak bu sorun çözülebilir, ama ayırıcı da bit harcar. Daha iyi bir çözüm, hiçbir kodun bir başka kodun başlangıcı olmamasıdır.

Tanım 2.8

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 2.13: 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 2.13'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 2.7

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.

2.7.3 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 2.14'te verilmiştir.

Karakter	F	O	R	G	E	T
Frekans	2	3	4	4	5	7

Tablo 2.14: 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.

2.7.4 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();
}
```

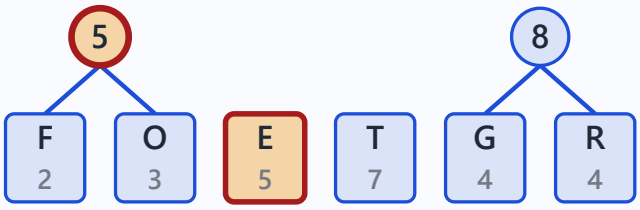
`Dugum` sınıfının dört alanı vardır: karakter `h`, frekans `f`, sol ve sağ çocuk. Yapraklarda çocuklar `null`'dır. Kuyruk, `(a, b) -> a.f - b.f` karşılaştırıcısıyla kurulan bir `PriorityQueue`'dur. Java'nın bu sınıfı bir en küçük heap'tir (*min-heap*); `poll()` her çağrıldığında frekansı en küçük düğümü 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.

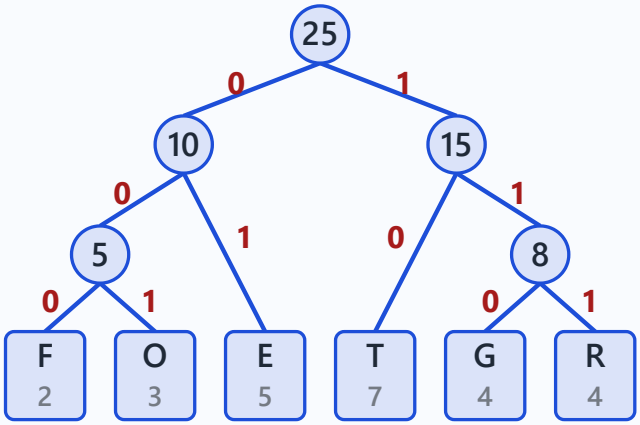
Tur	Sol (1. `poll`)	Sağ (2. `poll`)	Yeni düğüm	Tur sonunda kuyruk
başlangıç	—	—	—	F2, O3, R4, G4, E5, T7
1	F2	O3	(5) = 2 + 3	R4, G4, E5, (5), T7
2	G4	R4	(8) = 4 + 4	E5, (5), T7, (8)
3	(5)	E5	(10) = 5 + 5	T7, (8), (10)
4	T7	(8)	(15) = 7 + 8	(10), (15)
5	(10)	(15)	(25) = 10 + 15	(25)

Tablo 2.15: `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 2.14: Üçüncü turun başı: (5) ve (8) iç düğümleri kurulmuştur. Sıradaki en küçük iki düğüm (5) ve E5'tir (vurgulu).



Şekil 2.15: 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 2.8

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.

2.7.5 Kodları üretmek: kod ve izleme

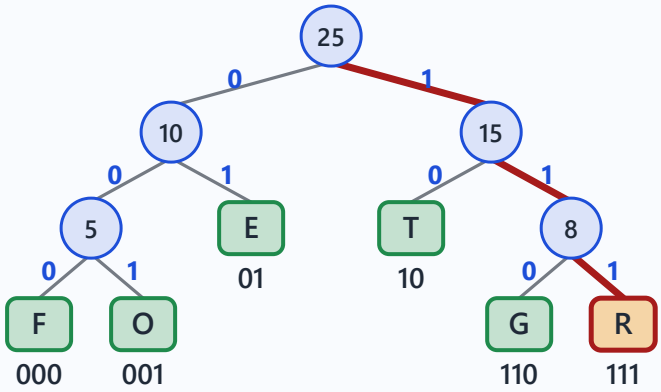
Ağaç kurulduktan sonra her karakterin kodu ağacın bir kez gezilmesiyle bulunur. Sunumdaki özyinelemeli `kodUret` metodu bunu yapar:

```
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 2.16: `kodUret(kok, "")` izlemesi (çağrı sırasıyla).



Şekil 2.16: 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.

2.7.6 Kodlama, çözme ve kazanç

Tablo 2.17 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 2.17: 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.

2.7.7 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.

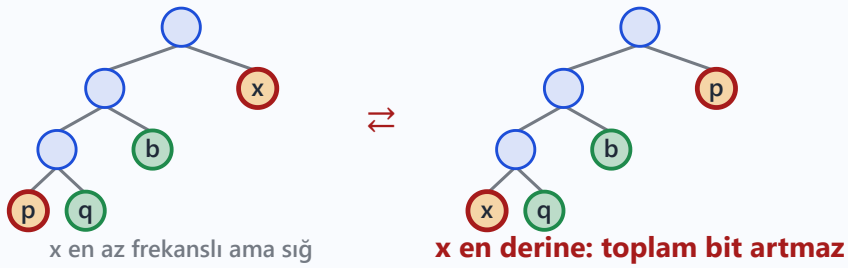
Teorem 2.5 (Huffman'da açgözlü seçim)

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 2.17: 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 2.6 (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.

2.7.8 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.

2.8 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.

2.9 Karşılaştırma ve sık yapılan hatalar

Tablo 2.18 notta incelenen problemleri açgözlü ölçüt, süre ve en iyilik bakımından karşılaştırır.

Problem	Açgözlü ölçüt	Zaman	En iyi mi?
Para üstü	sığan en büyük para	$O(k \cdot adet)$	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 2.18: 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.

2.10 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.

2.11 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*

2.12 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.

2.13 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.

2.14 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 I · BÖLÜM 3

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/advanced-algorithms-java/Bolum03/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.

3.1 Temel kavramlar

3.1.1 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 3.1

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.

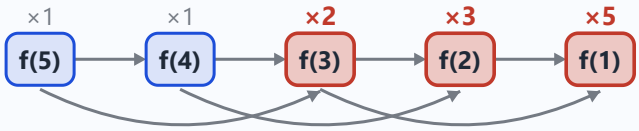
3.1.2 Örtüşen alt problemler

Bir problemin dinamik programlamadan yarar görmesi için iki özellik gerekir. Birincisi alt problemlerin örtüşmesidir.

Tanım 3.2

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 3.1 bu tekrarları gösterir.



Şekil 3.1: 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.

3.1.3 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 3.3

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.

3.1.4 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: $O(n^3)$.

3.1.5 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 3.2: 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 3.1: 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.

3.1.6 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 3.2: 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.

3.1.7 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.

3.2 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.

3.2.1 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);
}
```

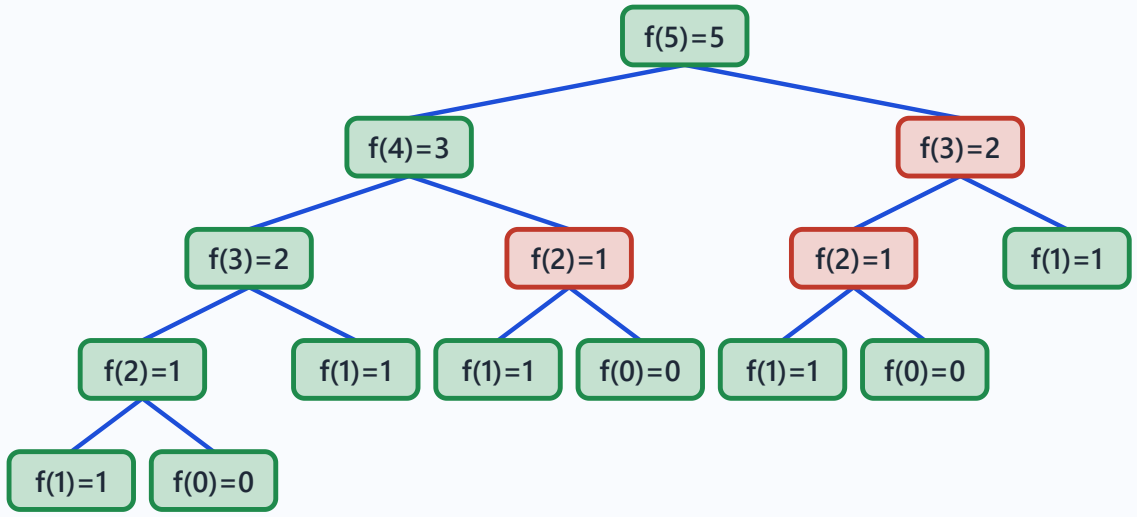
Metodun yaptığı çağrı sayısı $T(n)$ olsun. Her çağrı kendisi için 1 sayılır ve iki alt çağrı yapar: $T(n) = T(n-1) + T(n-2) + 1$, $T(0) = T(1) = 1$. Bu bağıntının çözümü $T(n) = 2 \cdot F(n+1) - 1$'dir; tümevarımla kolayca doğrulanır. Fibonacci sayıları altın oranın kuvvetleri gibi büyüdüğünden çağrı sayısı $\Theta(\varphi^n)$'dir; burada $\varphi \approx 1{,}618$'dir. Tablo 3.3 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 3.3: 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 3.3: fib(5) özyineleme ağacı, bütün çağrılar bittikten sonra. 15 çağrı yapılır; kırmızı kutular daha önce çözülmüş bir alt problemin yeniden hesaplandığı yerlerdir.

Şekil 3.3'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.

3.2.2 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];

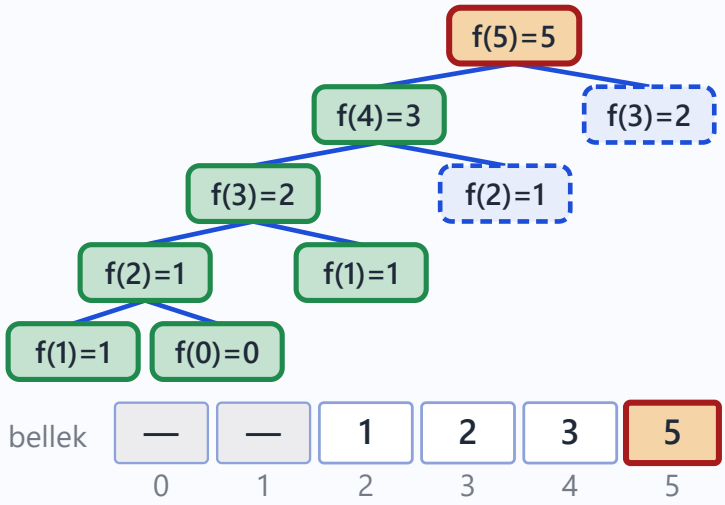
static long fib(int n) {
  if (n <= 1) return n;          // taban
  if (bellek[n] != 0) return bellek[n];
  long a = fib(n - 1);
  long b = fib(n - 2);
  bellek[n] = a + b;
  return bellek[n];
}
```

`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 3.4, 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 3.4: Bellekleme ile fib(5) izlemesi. "Bellek" sütunu adımdan sonraki `bellek[2..5]` değerlerini verir (— hesaplanmadı).



Şekil 3.4: Bellekleme ile fib(5): gerçekte açılan çağrılar ve dolmuş bellek dizisi. Kesikli kutular, tablodan hazır okunan iki çağrıdır.

Şekil 3.4'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 3.1

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.

3.2.3 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 3.5: 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.

3.2.4 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 3.5: 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.

3.3 0/1 sırt çantası

3.3.1 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 3.4

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 3.6: 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 3.2

Şekil 3.6'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.

3.3.2 Kaba kuvvet ve açgözlü seçim neden yetmez?

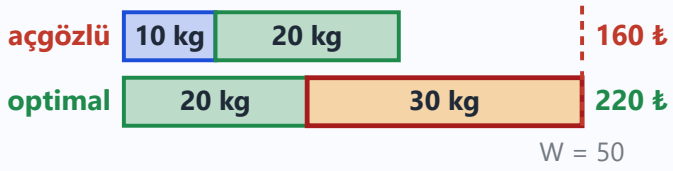
Problemin en doğrudan çözümü bütün seçimleri denemektir. Her eşya için iki seçenek vardır; n eşya için 2^n alt küme oluşur. $n = 50$ için bu sayı $2^{50} \approx 1{,}13 \cdot 10^{15}$'tir. Saniyede bir milyar alt küme denense bile on üç günden uzun sürer. Kaba kuvvet ancak çok küçük girdilerde kullanılabilir.

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 3.3

$W = 50$ 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 3.7: 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.

3.3.3 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.

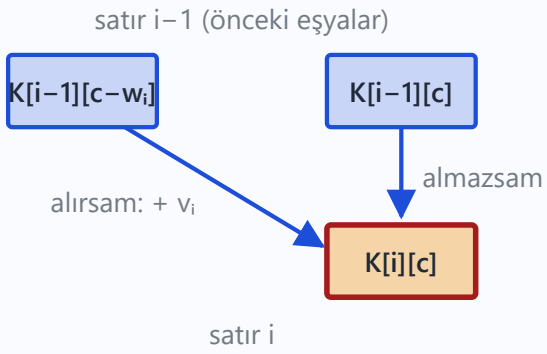
Tanım 3.5

$K[i][c]$, 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 3.8: $K[i][c]$ 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 3.8 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.

3.3.4 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 3.6 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 3.6: *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 3.9: 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.

3.3.5 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 3.7: *K[4][6] = 9* hücresinden geri okuma.



Şekil 3.10: 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.

3.3.6 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 3.6**
>
> 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.

3.3.7 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 3.8: 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 3.11: Dört eşya işlendikten sonra tek satırlık tablo: `dp[6] = 9`, iki boyutlu tablonun sağ alt hücresiyle aynı.

Örnek 3.4

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.

3.3.8 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.

3.4 Sınırsız sırt çantası

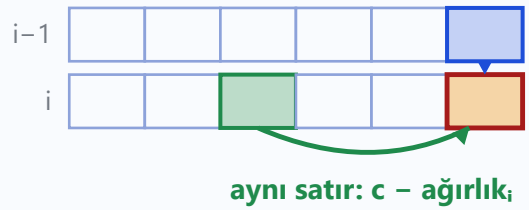
3.4.1 Problem ve bağıntı

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 3.12: 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.

3.4.2 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 3.9: 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*.



Şekil 3.13: 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.

3.4.3 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 3.5

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.

3.4.4 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 3.10: `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.

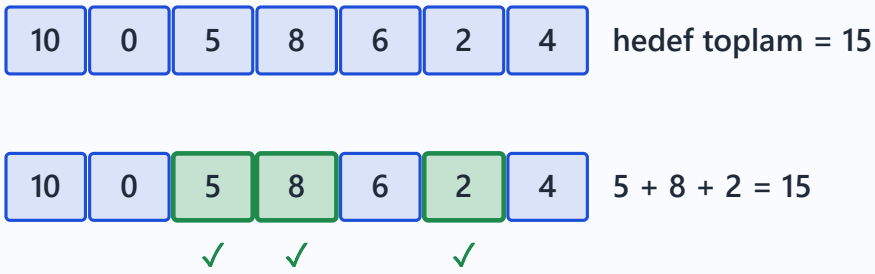


Şekil 3.14: İzlemenin sonunda `dp` dizisi; `dp[8] = 140`.

3.5 Alt küme toplamı

3.5.1 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 3.15: 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 3.7

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.

3.5.2 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 3.11: 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 3.16: 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.

3.5.3 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 3.6

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

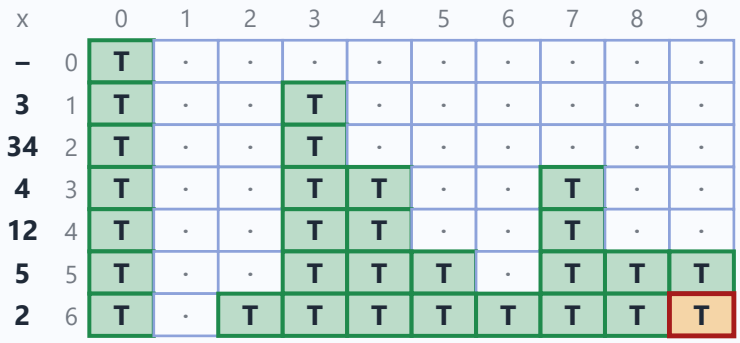
Sütun 0 doğru olarak başlatılmaz. Tablo baştan sona `false` kalır ve hiçbir toplam ulaşılabilir görünmez; her girdide cevap "hayır" olur. Boş kümenin toplamı 0'dır ve her satırda geçerlidir.

3.5.4 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 3.12: `altKume({3, 34, 4, 12, 5, 2}, 9)` izlemesi; her satırda doğru olan sütunlar.



Şekil 3.17: 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.

3.6 En uzun artan alt dizi

3.6.1 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 3.8

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 3.9

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 3.18: 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.

3.6.2 $O(n^2)$ 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 3.10

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.

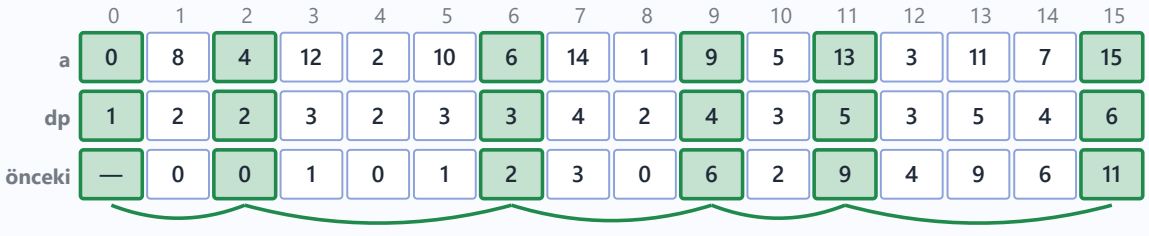
Bağıntının anlamı şudur. *a[i]*'de biten artan bir alt dizide *a[i]*'den önceki öğe, solda yer alan ve *a[i]*'den küçük bir *a[j]*'dir. O öğede biten en uzun dizinin sonuna *a[i]* eklenir. Bütün adaylar denenir ve en uzunu seçilir. Önünde daha küçük öğe yoksa *a[i]* tek başına 1 uzunluklu bir dizidir. Cevap son hücre değildir: LIS dizinin herhangi bir yerinde bitebilir.

Tablo 3.13 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 3.13: 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 3.19: 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 3.18'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.

3.6.3 Kod ve izleme

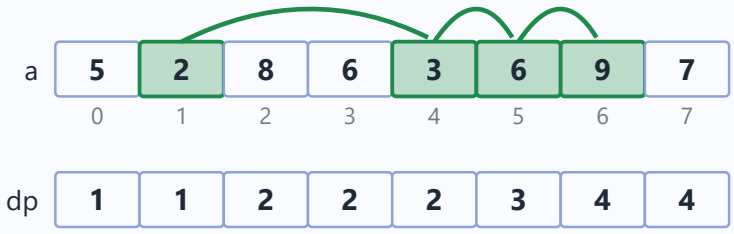
```
static int lis(int[] a) {
  int n = a.length, enUzun = 0;
  int[] dp = new int[n];
  for (int i = 0; i < n; i++) {
    dp[i] = 1;               // tek başına
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;   // uzat
    enUzun = Math.max(enUzun, 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 3.14: `lis({5, 2, 8, 6, 3, 6, 9, 7})` izlemesi.



Şekil 3.20: 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).

3.6.4 Aktif listeler

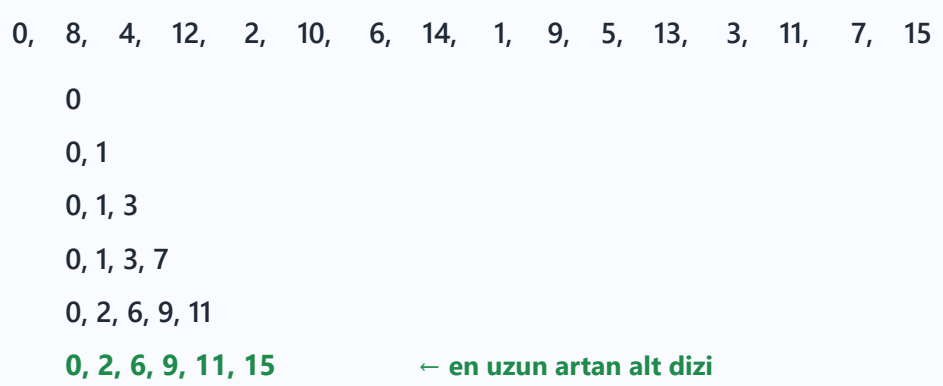
$O(n^2)$ algoritma her öğede soldaki bütün öğelere bakar. Daha hızlı bir yöntem için şu gözlem kullanılır: aynı uzunlukta iki artan dizi varsa, sonu küçük olan her zaman daha iyidir. Sonradan gelecek bir sayı, sonu büyük olanı uzatabiliyorsa sonu küçük olanı da uzatabilir; tersi doğru değildir. O hâlde 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 3.15: 16 öğeli dizide aktif listeler. Her satır, öğe işlendikten sonraki en uzun liste ve liste sayısıdır.



Şekil 3.21: 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 3.1 (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.

3.6.5 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 3.16: `lisHizli({5, 2, 8, 6, 3, 6, 9, 7})` izlemesi; "karş." ikili aramadaki karşılaştırma sayısıdır.



Şekil 3.22: 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

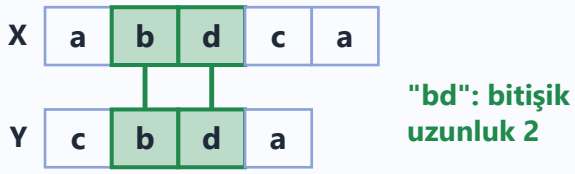
İkili aramada `son[orta] < x` yerine `son[orta] <= x` yazılır. Bu durumda eşit bir öğe, eşit sonun yerine değil, sağına yazılır ve liste uzar. Algoritma artık kesin artan değil, azalmayan (eşitliğe izin veren) en uzun alt diziyi bulur. {2, 2, 2} için doğru cevap 1, hatalı kodun cevabı 3'tür.

3.7 En uzun ortak alt dizgi

3.7.1 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 3.23: “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 3.11

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.

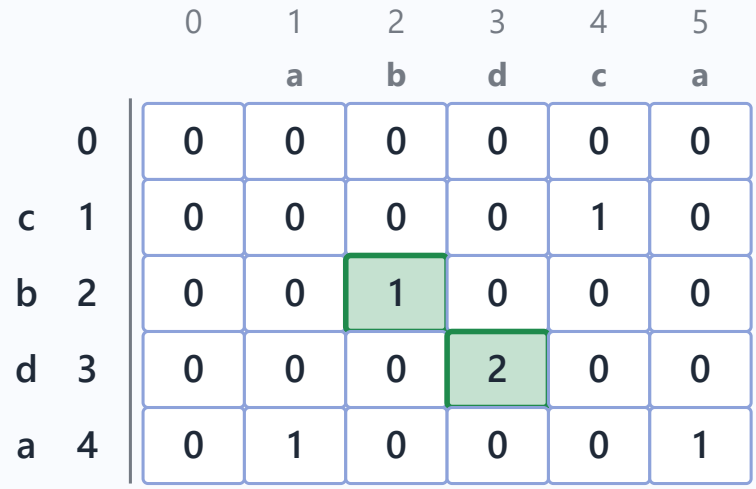
3.7.2 Tablonun doldurulması

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 3.17: 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 3.24: 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.

3.7.3 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.

3.8 En uzun ortak alt dizi

3.8.1 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 3.2 (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 3.18 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 3.18: 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.

3.8.2 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];
}
```

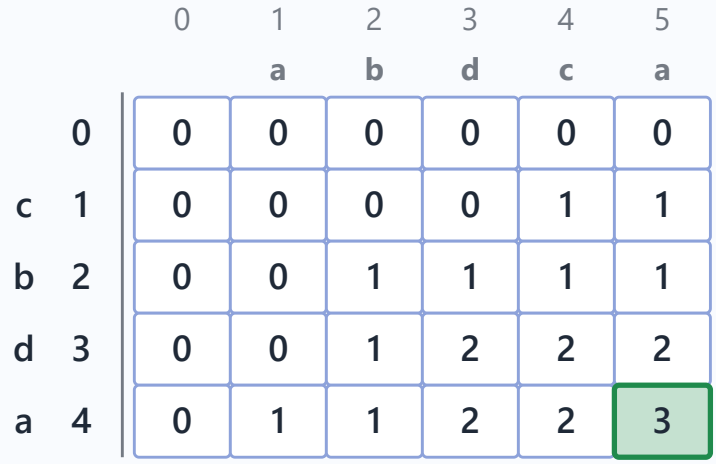
Kod, bağıntıyı iç içe iki döngüyle uygular. Dizgilerin karakterlerine `charAt(i - 1)` ve `charAt(j - 1)` ile erişilir, çünkü tablonun *i*. satırı *Y*'nin *i*. karakterini, yani 0 tabanlı *i − 1* indisli karakterini temsil eder. Aynı “abdca” ve “cbda” örneği için tablo şöyle dolar:

	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 3.19: 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 3.25: 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.

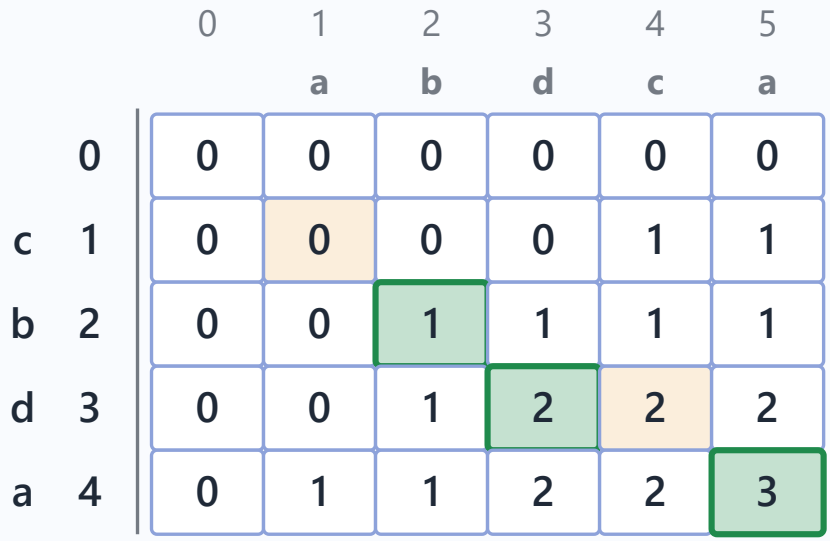
3.8.3 Geri okuma

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 3.20: “abdca” ve “cbda” için LCS geri okuması.



Şekil 3.26: Geri okuma yolu. Yeşil hücreler eşleşmelerdir: LCS = “bda”, uzunluk 3. Geri okuma *O(m + n)* adım sürer.

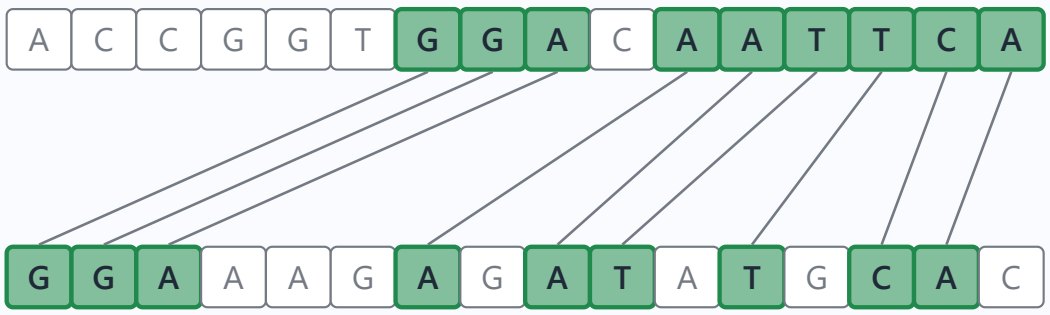
Her adımda ya satır ya sütun ya da ikisi birden bir azalır; geri okuma en çok *m + n* adım sürer. Aşağıdaki metot tabloyu doldurur ve LCS'nin kendisini döndürür.

```
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.

3.8.4 DNA örneği

Biyoinformatikte iki DNA dizisinin ne kadar benzediği LCS'nin uzunluğuyla ölçülebilir. Şekil 3.27'deki diziler ACCGGTGGACAATTCA ve GGAAAGAGATATGCAC'tir. Her ikisi de 16 bazdan oluşur.



Şekil 3.27: 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.

3.9 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 3.21 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 3.21: Diğer klasik dinamik programlama problemleri.

3.9.1 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 3.7

"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.

3.9.2 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 3.8

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.

3.9.3 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 3.9

[−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.

3.9.4 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 3.10

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.

3.9.5 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.

3.10 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.

3.11 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 3.22: 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.

3.12 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*

3.13 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.

3.14 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.

3.15 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 II

Zor Problemler

Bölüm 4–6

KISIM II · BÖLÜM 4

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/advanced-algorithms-java/Bolum04/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.

4.1 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.

4.1.1 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.

4.1.2 P ve NP

> **Tanım 4.1**
>
> **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.

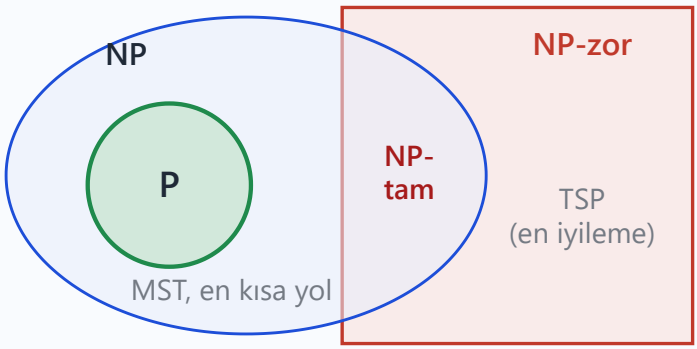
4.1.3 İndirgeme, NP-zor ve NP-tam

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 4.2**
>
> 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 4.1: *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 4.1'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 4.1: Notta geçen problemlerin sınıfları.

4.1.4 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.

4.2 Geri izlemenin temel fikri

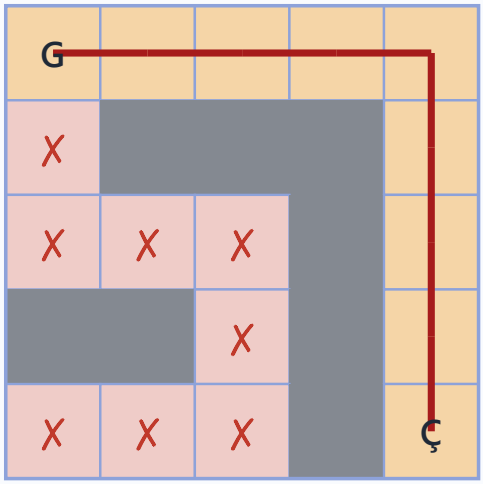
4.2.1 Geri dönüp denemek

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 4.3**
>
> **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.

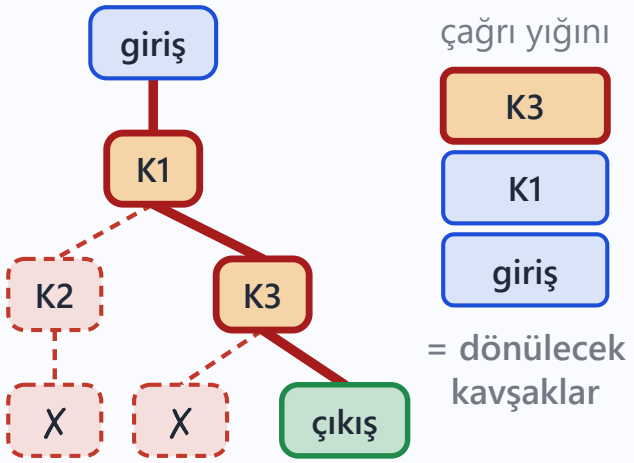


Şekil 4.2: Bir labirentte geri izleme. G giriş, Ç çıkıştır. Turuncu hücreler bulunan yolu, X işaretli hücreler denenip bırakılan çıkmazları gösterir.

4.2.2 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.

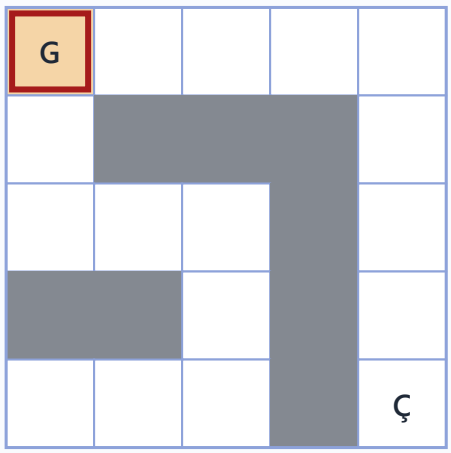


Şekil 4.3: Kavşaklar ve çağrı yığını. Girişten K1'e, oradan K3'e geçilmiştir. K2 dalı ve K3'ün ilk dalı çıkmazdır. Yığında giriş, K1 ve K3 durur: bunlar geri dönülecek kavşaklardır.

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?

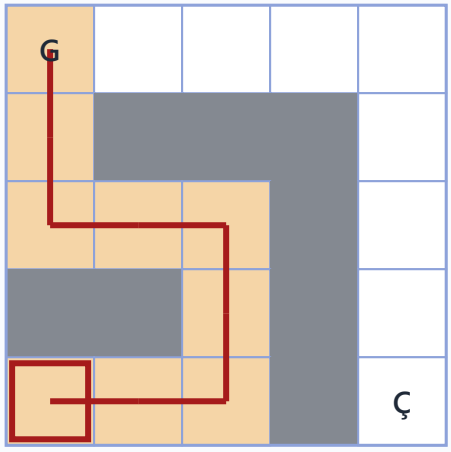
4.2.3 Labirentte geri dönüş adım adım

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.



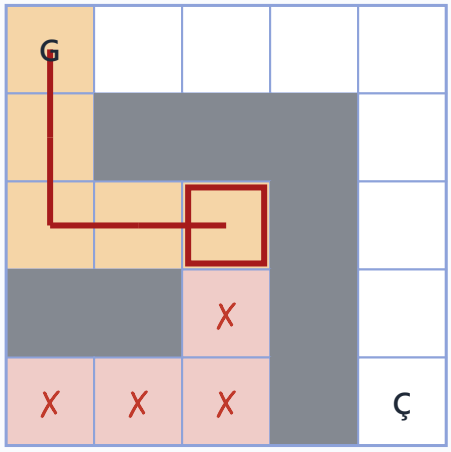
Şekil 4.4: Başlangıç: giriş (G) hücresindeyiz. Yön sırası aşağı, sağ, yukarı, soldur.

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.



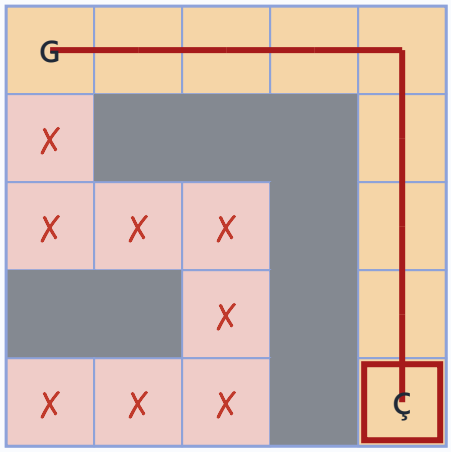
Şekil 4.5: Aşağı ve sola ilerleyen koridor (4, 0) hücresinde çıkmaza girer.

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 4.6: 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 4.7: Çıkışa ulaşıldı. Yol 9 hücredir; 8 hücre çıkmaz sokak olarak denenip bırakılmıştır.

Örnek 4.1

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?

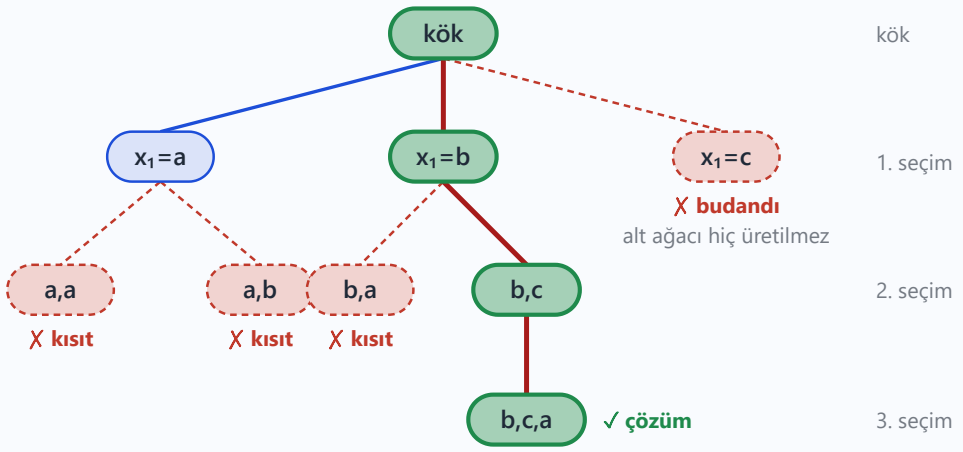
Çözüm. Sol alt bölge, yani (1, 0), (2, 0), (2, 1), (2, 2), (3, 2) ve (4, 0)–(4, 2) hücreleri, labirentin geri kalanına yalnız girişten bağlıdır. (2, 3), (3, 3), (4, 3) ve (1, 2) duvar olduğu için bu bölgeden sağ sütuna geçilemez. Çıkışa varan tek bağlantı üst koridor ve sağ sütundur. Bu yüzden yalnız 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.

4.2.4 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 4.4

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 4.8: 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 4.8 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.

4.2.5 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.

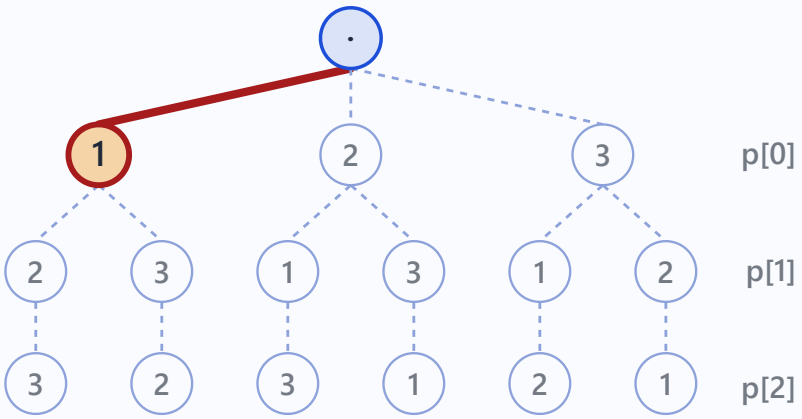
4.2.6 Şablonun uygulaması: permütasyonlar

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 4.9: İlk adım: `k = 0`'da aday 1 seçildi. Kesikli düğümler henüz üretilmemiştir.

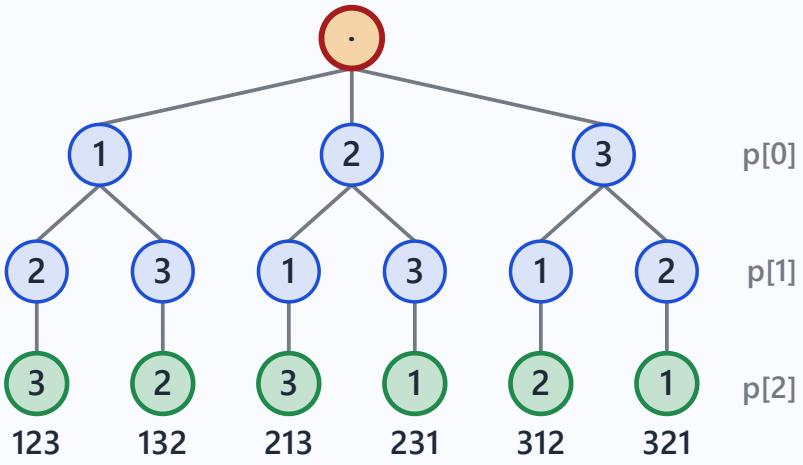
Adım	k	İşlem	p	Kullanılan	Çıktı
1	0	a = 1 seçilir	[1, –, –]	{1}	
2	1	a = 1 kullanılmış; a = 2 seçilir	[1, 2, –]	{1, 2}	
3	2	yalnız 3 kullanılmamış; seçilir	[1, 2, 3]	{1, 2, 3}	
4	3	`k == p.length` : çözüm	[1, 2, 3]	{1, 2, 3}	[1, 2, 3]
5	1	3 ve 2 geri alınır; a = 3 seçilir	[1, 3, 3]	{1, 3}	
6	2	yalnız 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 4.2: `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 4.10: Arama bitince durum uzayı ağacının tamamı. Altı yaprak altı permütasyondur; her yaprağın altında kökten o yaprağa giden yolun okunuşu yazılıdır.

Örnek 4.2

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.

4.2.7 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 4.3: 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.

4.2.8 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.

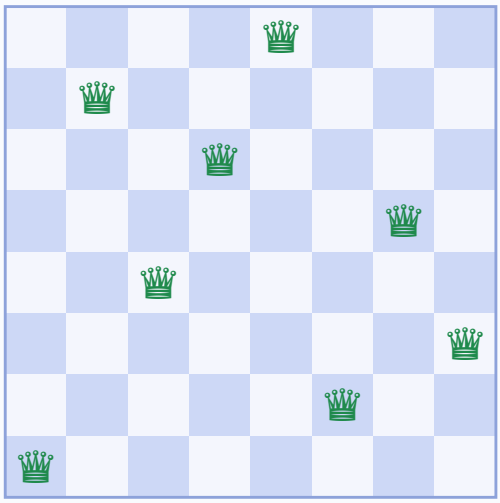
4.3 n-vezir problemi

4.3.1 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 4.5**
>
> **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.



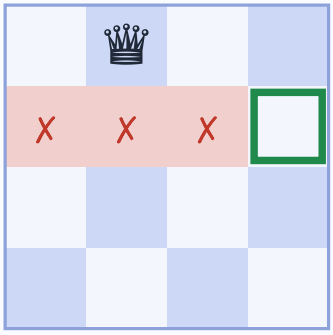
Şekil 4.11: 8 vezir probleminin bir çözümü: a1, b7, c4, d6, e8, f2, g5, h3. Satranç gösteriminde harf sütunu, sayı satırı belirtir; 1. satır tahtanın en altındadır.

Şekil 4.11'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.

4.3.2 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.



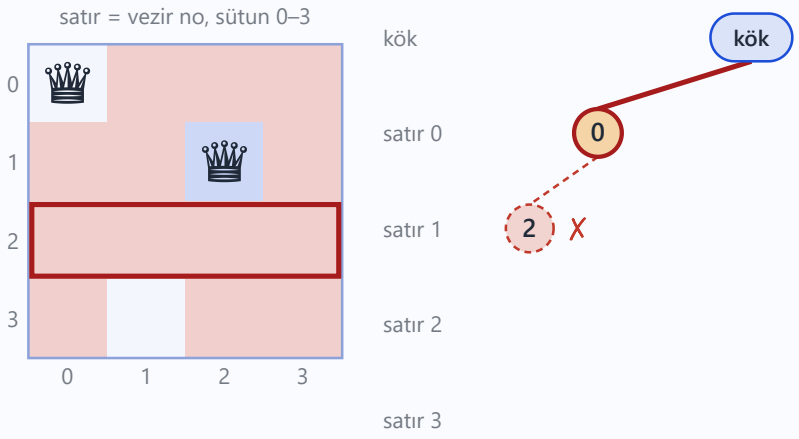
Şekil 4.12: 4 × 4 tahtada 0. satırın 1. sütununda bir vezir varken 1. satır: sütun 0, 1 ve 2 tehdit altındadır, sütun 3 güvenlidir.

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.

4.3.3 4 vezir: tahta ve durum uzayı ağacı

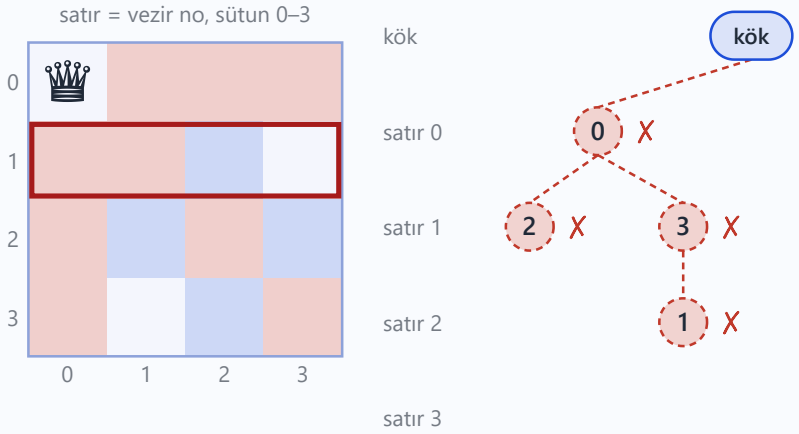
Algoritmanın işleyişi en iyi küçük bir örnekte görülür. $n = 4$ için vezirler satır satır yerleştirilir; her satırda sütun 0, 1, 2, 3 sırayla denenir. Durum uzayı ağacında kökün altındaki her düzey bir satırdır; bir düğüm, o satırda seçilen sütunu gösterir. Tahtadaki kırmızı hücreler, yerleşmiş vezirlerin 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.



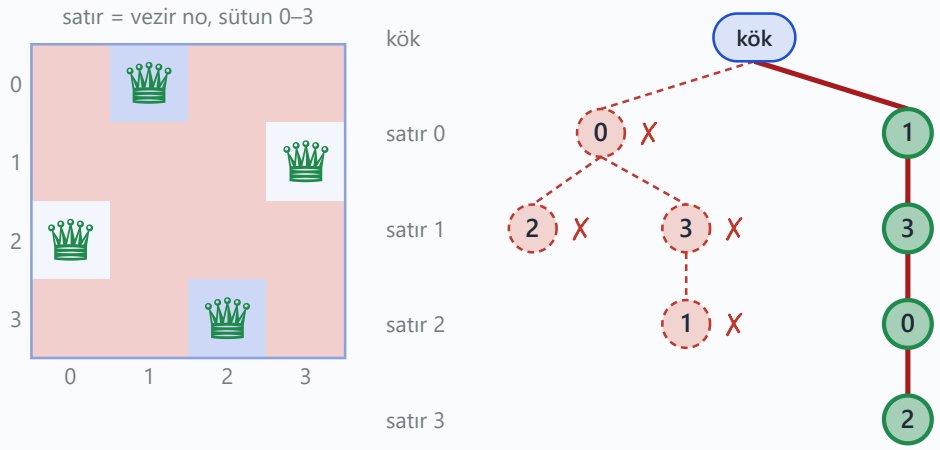
Şekil 4.13: İlk çıkmaz. Vezirler (0, 0) ve (1, 2)'deyken satır 2'nin bütün kareleri tehdit altındadır. Ağaçta 2 düğümü kesikli çizilmiştir: bu dal geri alınacaktır.

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 4.14: 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 4.15: 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 4.3

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.

4.3.4 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.

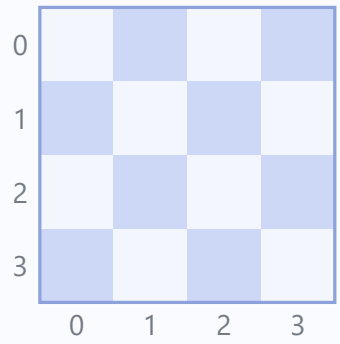
Maliyet iki parçadan oluşur. `guvenliMi` en çok üç döngü çalıştırır; her döngü en çok n adımdır. Bu yüzden bir çağrı $O(n)$ sürer. `coz` için en kötü durum, durum uzayı ağacının tamamının gezilmesidir. İlk satırda n, ikincide en çok $n - 1$ güvenli sütun vardır, çünkü en azından bir sütun tutulmuştur. 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.

4.3.5 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 4.16: `coz(t, 0)` çağrıldı; tahta boş, `satir = 0`.

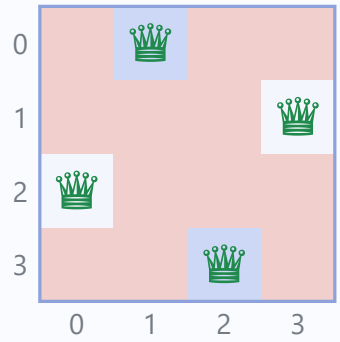
#	Kare	`guvenliMi`	Neden / işlem	Yer	Yığın derinliği
1	(0, 0)	true	`t[0][0] = 1`, `coz(t, 1)`	[0, –, –, –]	2
2	(1, 0)	false	aynı sütun: (0, 0)	[0, –, –, –]	2
3	(1, 1)	false	sol üst çapraz: (0, 0)	[0, –, –, –]	2

#	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 4.4: `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 4.17: `satir == n` olunca `true` değeri yığın boyunca yukarı taşınır; tahtada çözüm [1, 3, 0, 2] kalır.

Örnek 4.4

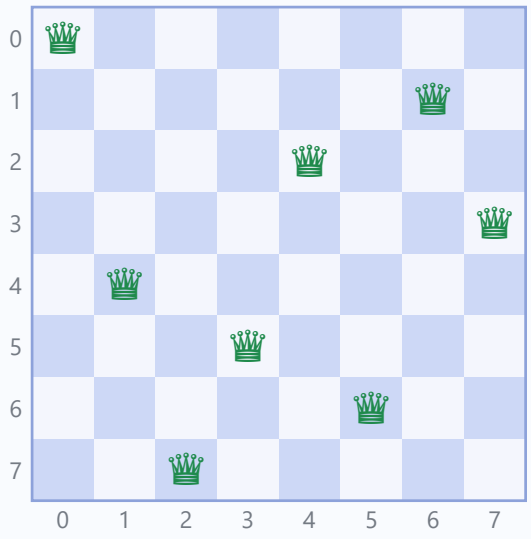
Soru. $n = 5$ için `coz(t, 0)` ilk çözümü nasıl bulur? Kaç yerleştirme ve kaç geri dönüş yapılır?

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.

4.3.6 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 4.18: 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 4.11'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.

4.3.7 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 4.5: n-vezir için çözüm sayısı, kaba kuvvetin tam yerleşim sayısı ve geri izlemenin yaptığı yerleştirmeler.

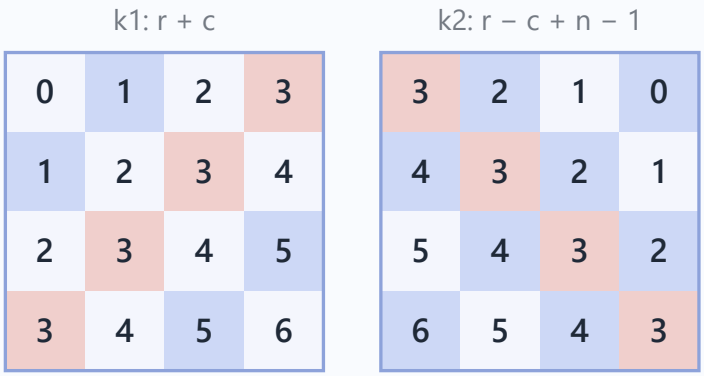
Tablodan üç sonuç çıkar.

- *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.

4.3.8 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 4.19: 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 4.5

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.

4.3.9 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.

4.4 En az çakışma sezgiseli

4.4.1 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 4.6**
>
> **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.

4.4.2 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 4.6

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?

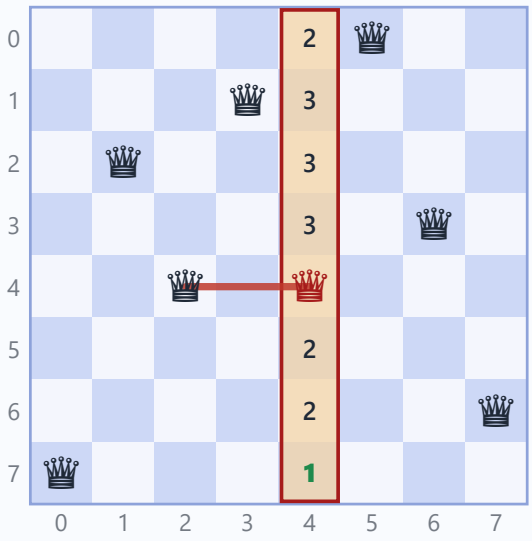
Çözüm. Vezirler (satır, sütun) olarak (1, 0), (3, 1), (0, 2) ve (3, 3)'tedir. Sütun 1 ile sütun 3'teki vezirler aynı satırdadır (3); başka çakışma yoktur. Sütun 3 için hesap:

- 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.

4.4.3 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.



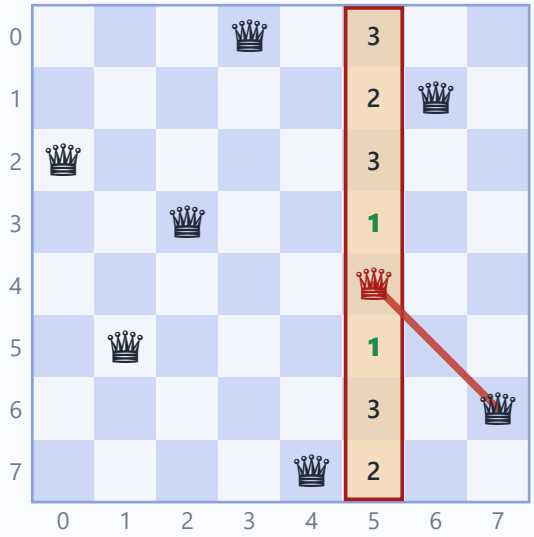
Şekil 4.20: Açgözlü başlangıç. Sütun 4'teki vezir seçildi. Sütunundaki çakışma sayıları satır 0'dan 7'ye 2, 3, 3, 3, (kendi satırı), 2, 2, 1'dir; en küçüğü satır 7'dedir.

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

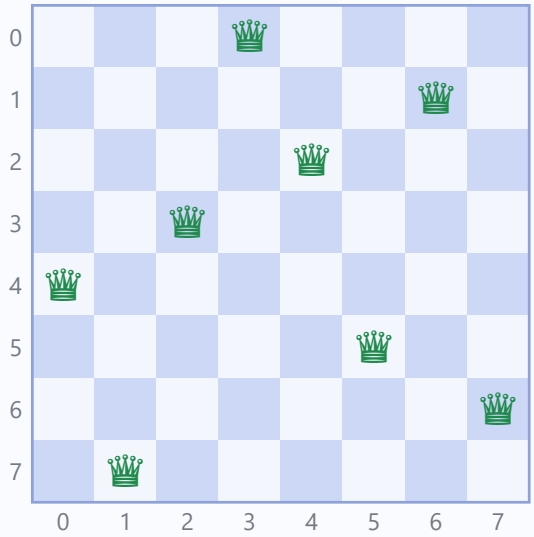
Tablo 4.6: En az çakışma sezgiselinin bir çalışması ($n = 8$). `yer[k]`, k. sütundaki vezirin satırıdır.

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 4.21: Plato. Sütun 5'teki vezir için satır 3, 4 ve 5'in hepsinde çakışma 1'dir; taşıma durumu iyileştirmez.



Şekil 4.22: 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.

4.4.4 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.

4.5 Sudoku

4.5.1 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 4.7

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

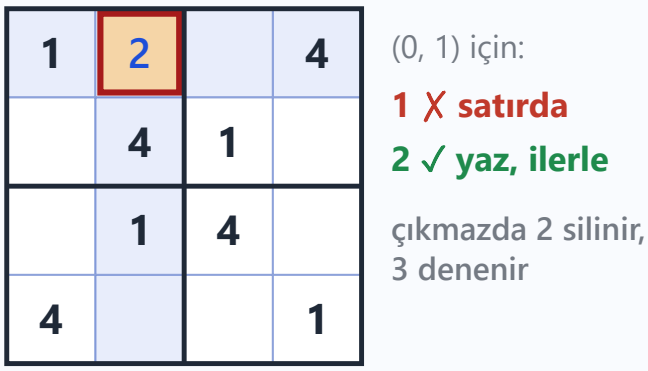
Şekil 4.23: Örnek bulmaca. 30 hücre ipucudur, 51 hücre boştur.

4.5.2 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 4.24: 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.

4.5.3 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;
}
```

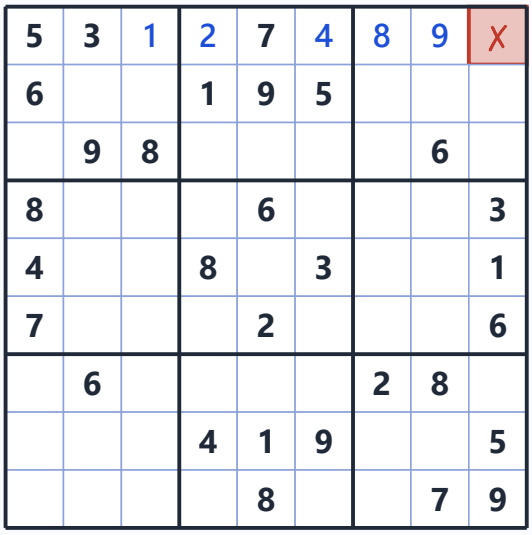
`gecerliMi`, *r* rakamının *(s, k)* hücresine yazılıp yazılamayacağını denetler. İlk döngü tek geçişte hem satırı (`t[s][i]`) hem sütunu (`t[i][k]`) tarar. Ardından hücrenin kutusu taranır. Kutunun sol üst köşesi `bs = s / 3 * 3` ve `bk = k / 3 * 3` ile bulunur. Tam sayı bölmesi aşağı yuvarlar; 3 ile 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.

4.5.4 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 4.25: 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 4.7: 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 4.26: Çözüm bulundu. Siyah rakamlar ipucu, mavi rakamlar algoritmanın yazdıklarıdır; turuncu hücre son yazılan hücredir.

4.5.5 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 4.27: 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 4.8: 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.

4.6 Çizge boyama

4.6.1 Problem

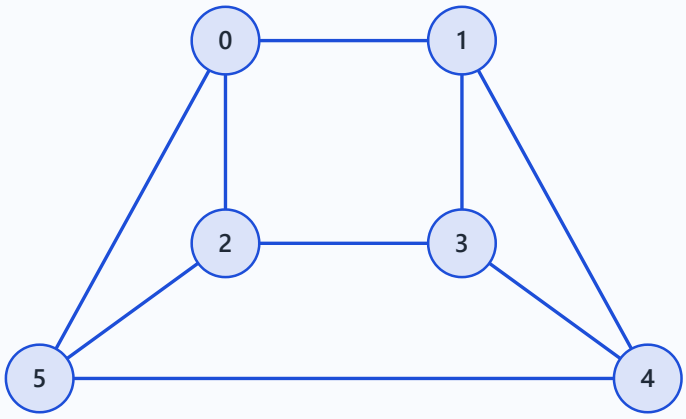
Bir haritada komşu ülkelerin farklı renklerle boyanması istenir. Ülkeler düğüm, ortak sınırlar kenar olarak düşünülürse bu soru bir çizge problemine dönüşür. Aynı problem başka kılıklarda da karşımıza çıkar. Sınav çizelgesinde aynı öğrencinin girdiği iki sınav aynı saate konamaz; sınavlar düğüm, çakışmalar kenar, saatler renktir. Derleyicide aynı anda canlı olan iki değişken aynı yazmacı paylaşamaz; değişkenler düğüm, yazmaçlar renktir.

> **Tanım 4.8**
>
> **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.

4.6.2 Adım adım: altı düğümlü çizge

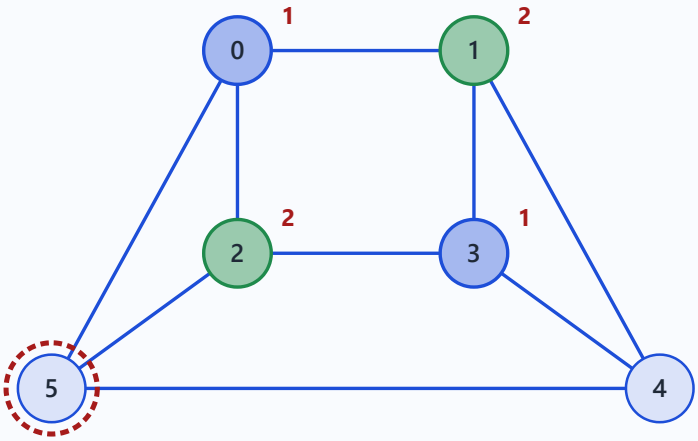
Çizgenin altı düğümü ve dokuz kenarı vardır: 0–1, 0–2, 0–5, 1–3, 1–4, 2–3, 2–5, 3–4 ve 4–5. Üç renk kullanılır: 1 mavi, 2 yeşil, 3 turuncu. Her düğümde renkler 1, 2, 3 sırasıyla denenir.



Şekil 4.28: Boyanacak altı düğümlü çizge, *m = 3*.

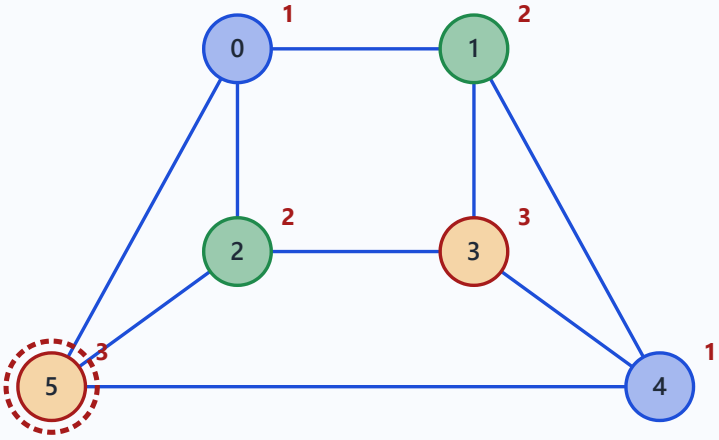
Adım	Düğüm	Komşu renkleri	Karar	`renk[0..5]`
1	0	—	renk 1	[1, 0, 0, 0, 0, 0]
2	1	0: 1	1 dolu; renk 2	[1, 2, 0, 0, 0, 0]
3	2	0: 1	1 dolu; renk 2	[1, 2, 2, 0, 0, 0]
4	3	1: 2, 2: 2	renk 1	[1, 2, 2, 1, 0, 0]
5	4	1: 2, 3: 1	1, 2 dolu; renk 3	[1, 2, 2, 1, 3, 0]
6	5	0: 1, 2: 2, 4: 3	çıkmaz; düğüm 4'ün rengi 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 4.9: Altı düğümlü çizgenin 3-boyaması. “Komşu renkleri” sütunu, o an boyalı komşuların renklerini verir.



Şekil 4.29: 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 4.30: Altı düğüm de boyandı. Her düğümün yanındaki sayı rengidir: [1, 2, 2, 3, 1, 3].

4.6.3 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
```

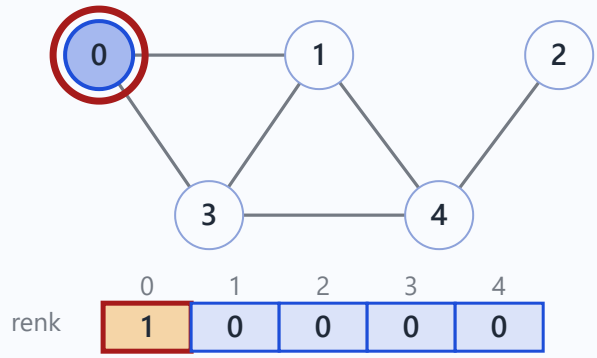
`boya(v)`, düğüm 0'dan *v* – *1*'e kadarının boyandığını varsayar ve *v*. düğümden itibaren boyamayı sürdürür. `v == n` ise bütün düğümler boyanmıştır. Döngü renkleri 1'den *m*'ye kadar dener. Uygun bir renk `renk[v]`'ye yazılır ve bir sonraki düğüme geçilir. Alt çağrı başarısız olursa renk 0 yapılır; 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.

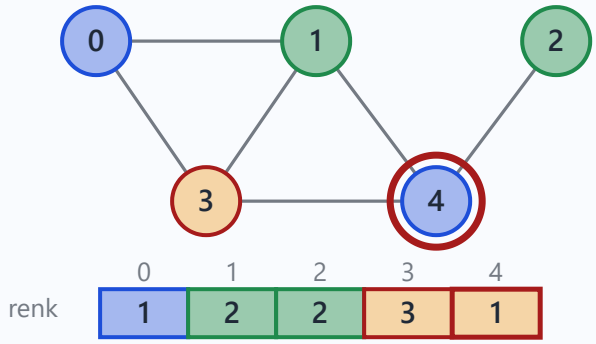


Şekil 4.31: `boya(0)` başlıyor. Düğüm 0'a renk 1 atandı; `renk` dizisinin öteki hücreleri 0'dır.

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 4.10: `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 4.32: `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.

4.6.4 Tek uzunluklu döngü ve en az renk sayısı

Örnek 4.7

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.

4.7 Hamilton yolu

4.7.1 Problem ve adım adım çözüm

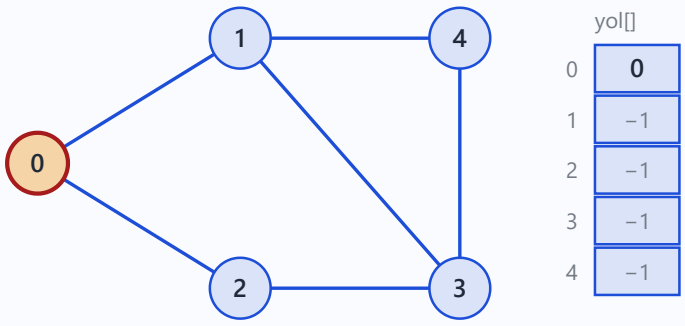
Tanım 4.9

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.

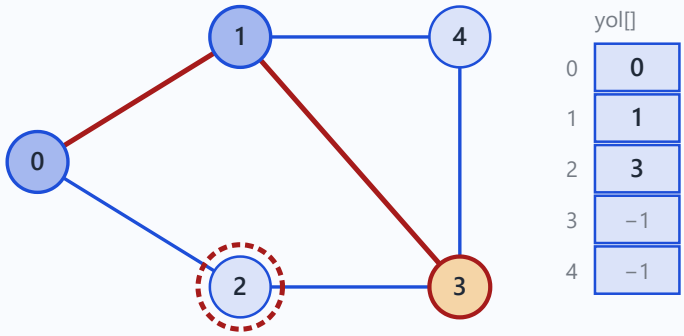
Örnek çizgenin beş düğümü ve altı kenarı vardır: 0–1, 0–2, 1–3, 1–4, 2–3 ve 3–4. Yol düğüm 0'dan başlar; `yol[0] = 0`. Her konumda düğümler 0, 1, 2, … sırasıyla denenir.



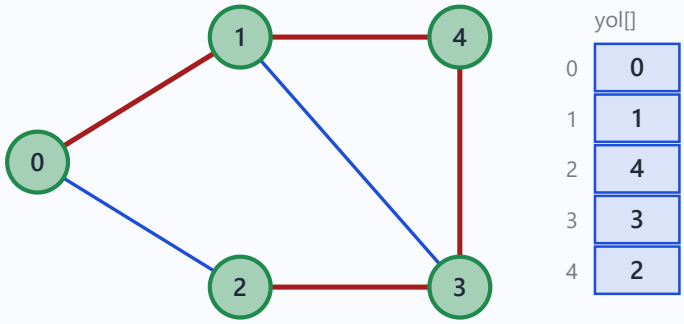
Şekil 4.33: Başlangıç: `yol[0] = 0`, öteki konumlar −1 (boş).

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 4.34: 4. adım: yol 0 → 1 → 3 → 2. Son düğüm 2'nin bütün komşuları yolda; 2 çıkarılacak.



Şekil 4.35: Hamilton yolu bulundu: 0 → 1 → 4 → 3 → 2.

4.7.2 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;
}
```

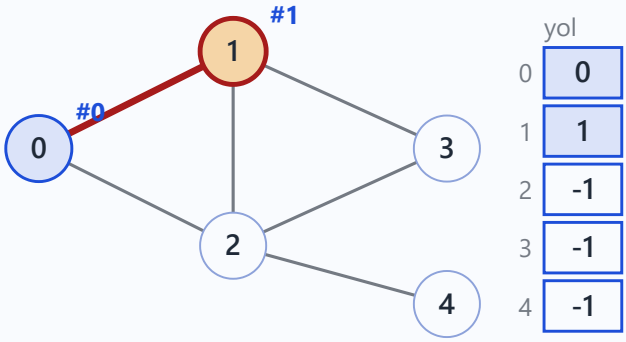
`graf` komşuluk matrisidir; `graf[u][v] == 1`, *u* ile *v* arasında kenar olduğunu gösterir. Çağıran kod `yol` dizisini −1 ile doldurur, `yol[0] = 0` yapar ve `bul(graf, yol, 1)`'i çağırır. `konum == graf.length` hedef testidir: yolda *n* düğüm vardır. Döngü her düğümü aday olarak dener. `uygunMu` iki 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.

4.7.3 Kod izleme: ikinci çizge

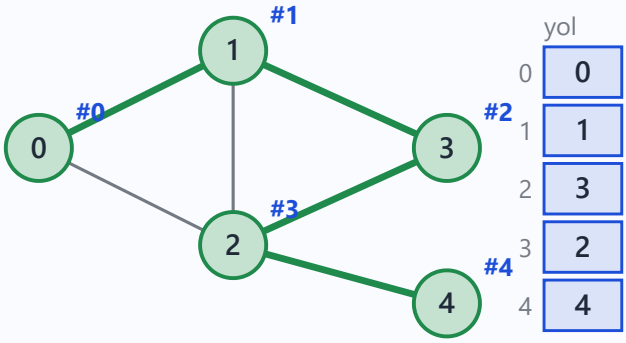
Kod ikinci bir çizgede izlensin. Beş düğüm ve altı kenar vardır: 0–1, 0–2, 1–2, 1–3, 2–3 ve 2–4. Düğüm 4'ün tek komşusu 2'dir. Bu yüzden düğüm 4 bir Hamilton yolunda ancak uçta bulunabilir.



Şekil 4.36: 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 4.11: `bul(graf, yol, 1)` çağrısının izlemesi (ikinci çizge).



Şekil 4.37: 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.

4.7.4 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.

4.8 Kelime arama

4.8.1 Problem

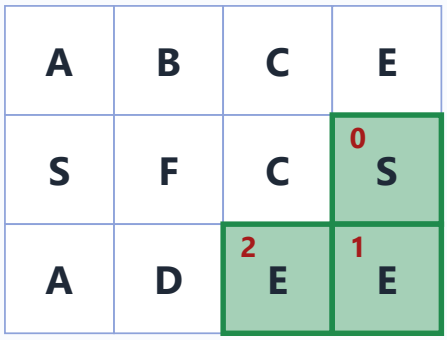
Bir harf ızgarasında bir kelime aranır. Kelime, komşu hücrelerden geçen bir yol boyunca okunmalıdır. Komşuluk dört yönlüdür: yukarı, aşağı, sol ve sağ. Bir hücre aynı yolda en çok bir kez kullanılabilir. Soru şudur: kelimeyi oluşturan böyle bir yol var mı?

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 4.38: 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 4.39: "SEE" bulundu. Küçük sayılar harfin kelimedeki indisidir: (1, 3) S, (2, 3) E, (2, 2) E.

4.8.2 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 4.12: ABCE / SFCS / ADEE ızgarasında bazı kelimeler.

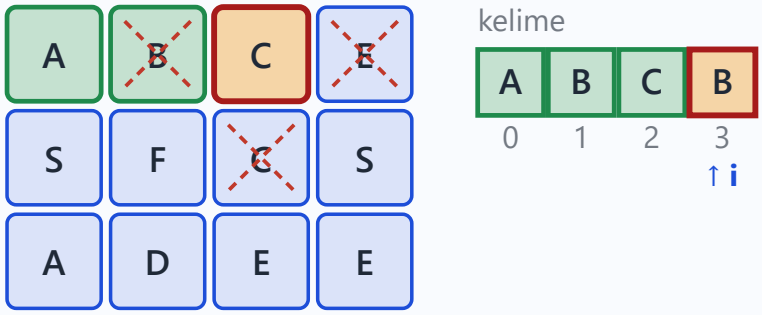
“SADE” araması da bir geri dönüş içerir. (1, 0)'daki S'den önce yukarıdaki (0, 0) = 'A' denenir. Bu A'nın komşuları B, S (ziyaretli) ve ızgara dışıdır; 'D' bulunamaz. İşaret kaldırılır ve aşağıdaki (2, 0) = 'A' denenir. Oradan (2, 1) = 'D' ve (2, 2) = 'E' ile kelime tamamlanır.

4.8.3 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, …)` `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 4.40: 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.

4.9 Diğer uygulamalar

4.9.1 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 4.13: `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 4.8

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.

4.9.2 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 4.14: 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.

4.10 Analiz

4.10.1 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 4.15: 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 4.9

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.

4.10.2 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.

4.10.3 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 4.16: 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.

4.11 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.

4.12 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.

4.13 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*

4.14 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.

4.15 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.

4.16 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 II · BÖLÜM 5

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/advanced-algorithms-java/Bolum05/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.

5.1 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.

5.1.1 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 5.1 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 5.1: 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.

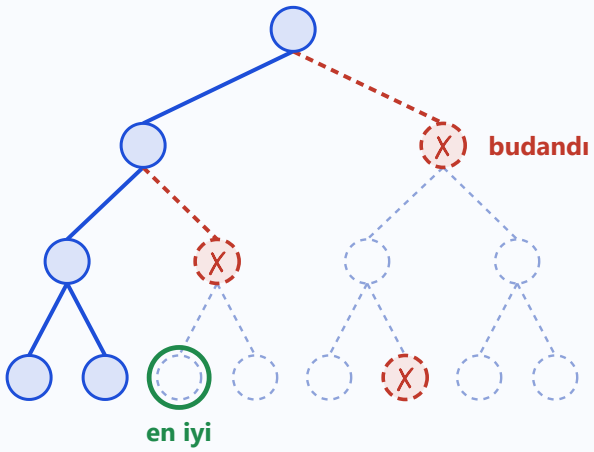
5.1.2 Dallanma, sınırlama, budama

Tanım 5.1

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 5.1: 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 5.1 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.

5.1.3 Temel kavramlar

Dal sınır algoritmaları bir ağaç üzerinde çalışır ve bu ağacın düğümleri çeşitli durumlarda bulunur. Kavramların adlarını baştan koymak, sonraki izlemeleri okumayı kolaylaştırır.

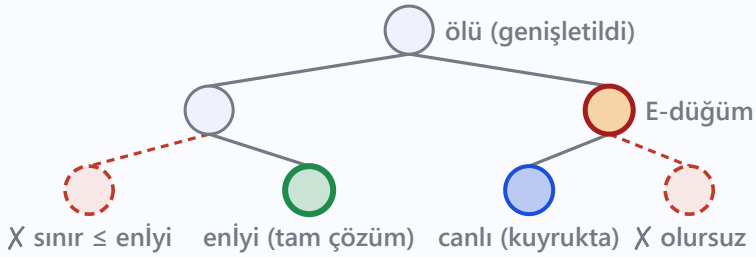
Tanım 5.2

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 5.2: 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 5.2 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.

5.2 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.

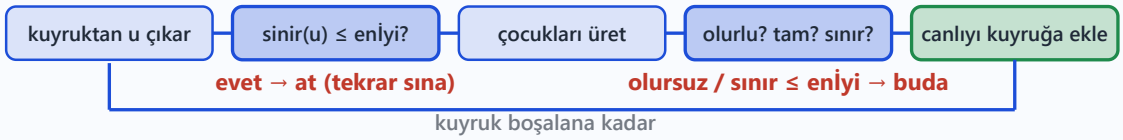
5.2.1 Genel şema

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 5.3: 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.

5.2.2 Sınır nasıl seçilir?

Dal sınırın bütün başarısı sınır fonksiyonuna bağlıdır. İyi bir sınırın dört özelliği vardır. Bunlardan biri zorunludur, öbür üçü verimlilikle ilgilidir.

Tanım 5.3

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.

5.2.3 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.

5.2.4 Arama stratejileri

Genel şemada "kuyruktan bir canlı düğüm çıkarılır" denmişti. Hangi canlı düğümün çıkacağı, kuyruk yapısının türüne bağlıdır. Üç yaygın strateji vardır (Tablo 5.2).

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 5.2: 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.

5.2.5 En düşük maliyetli arama ve maliyet tahmini

LC aramanın genel biçiminde her düğüme bir tahmini maliyet atanır:

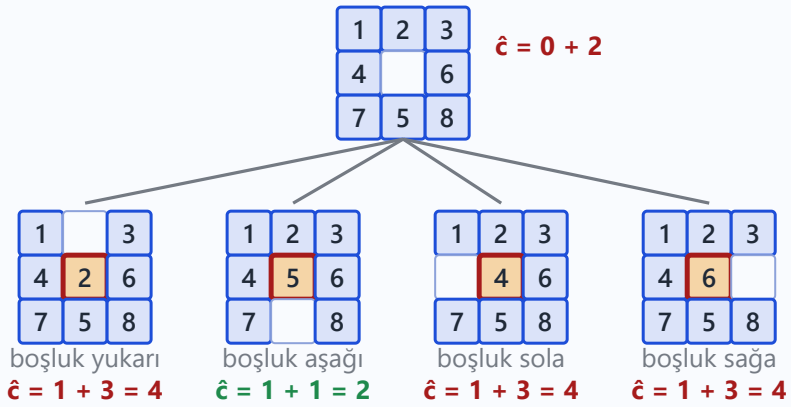
> $$\hat{c}(x) = f(x) + \hat{g}(x)$$

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 5.4: 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 5.4 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.

5.2.6 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 5.3).

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 5.3: 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.

5.2.7 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.

5.3 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.

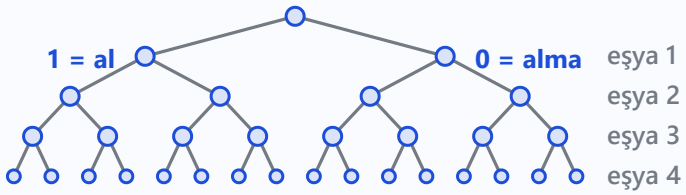
5.3.1 Problem ve durum uzayı ağacı

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

> $$\Sigma\, w_i\, x_i \le W \text{ koşuluyla } \Sigma\, p_i\, x_i \text{ en büyük}$$

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 5.5: 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 5.4'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 5.4: 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.

5.3.2 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:

$$\mathrm{sinir}(u) = k + \sum_{j=i+1}^{m-1} p_j + (W - a - \sum_{j=i+1}^{m-1} w_j) \cdot p_m / w_m$$

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 5.1 (Kesirli sınır geçerlidir)

Eşyalar p_i/w_i'ye göre azalan sıradaysa, her u düğümü için $\mathrm{sinir}(u)$, 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 $\mathrm{sinir}(u)$ 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 $\mathrm{sinir}(\text{kök}) = 70 + 45 = 115$ 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.

5.3.3 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;
}
```

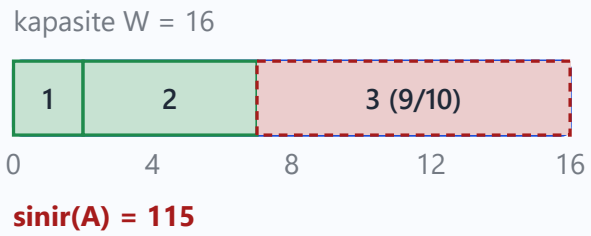
Satırların görevi şöyledir:

- `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 5.5 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 5.5: `sinir(kök)` izlemesi (*W = 16*; kök: düzey −1, kazanç 0, ağırlık 0).



Şekil 5.6: 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 5.6 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 5.1

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.

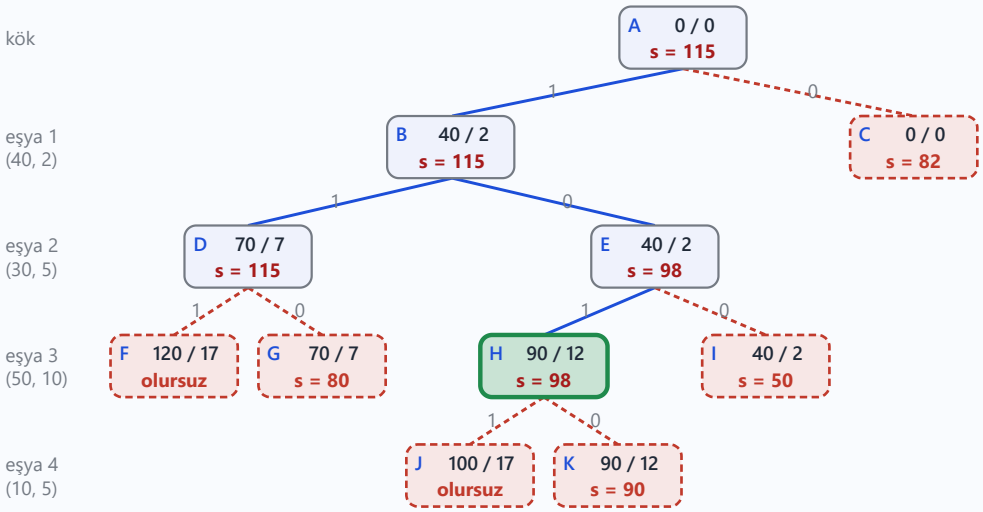
Optimum eşya 1 ve 3'tür: kazanç 55, ağırlık 9. Oran sırasıyla açgözlü seçim ise 1 ve 2'yi alır ve 50'de kalır. Optimum bulunduktan sonra "eşya 1 alınmadı" düğümünün sınırı *50 ≤ 55* olur ve düğüm budanır: eşya 1'siz hiçbir çözüm 55'i geçemez.

5.3.4 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 5.7: 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 5.7 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.

5.3.5 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.

5.3.6 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.

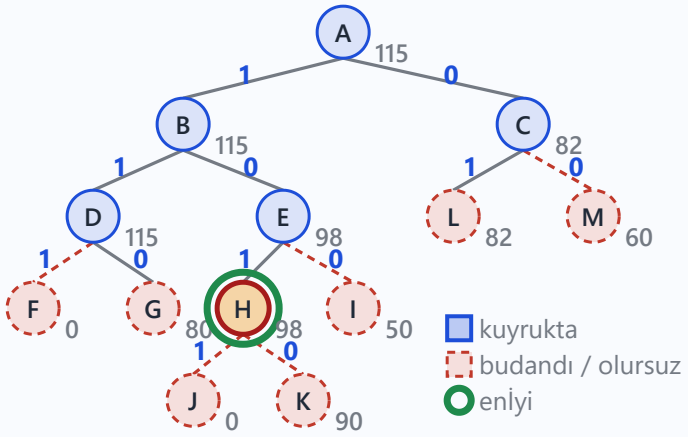
```
int fifoDalSinir() {
  ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
  kuyruk.add(kok);  int enIyi = 0;
  while (!kuyruk.isEmpty()) {
    Dugum u = kuyruk.poll();          // ilk giren
    if (u.sinir <= enIyi || u.seviye == n - 1) continue;
    for (int al = 1; al >= 0; al--) { // önce al
      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 5.6 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 5.6: FIFO dal sınır izlemesi ($W = 16$). Düğümler "kazanç / ağırlık, sınır" biçimindedir.

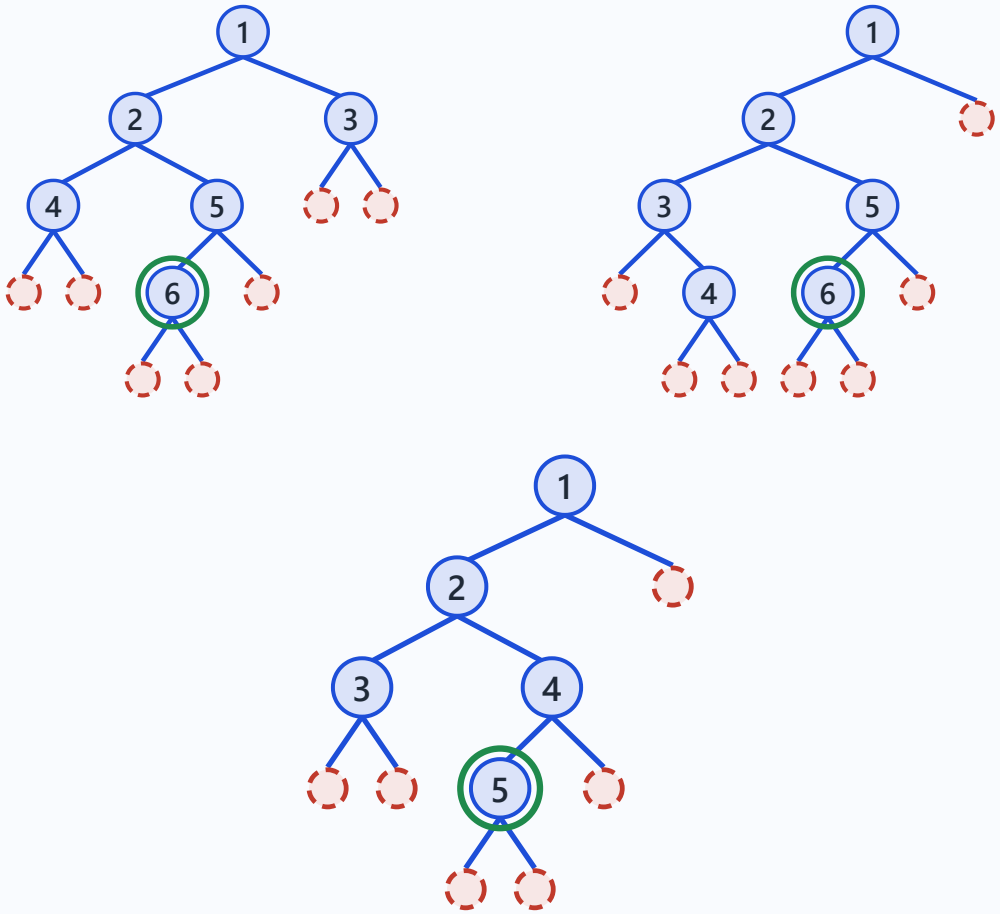
Sonuç yine 90'dır. FIFO 13 düğüm üretmiş ve 6 düğüm genişletmiştir: A, B, C, D, E, H. Kuyruk en çok 3 düğüm tutmuştur. En iyi önce stratejisiyle farkın kaynağı C düğümüdür. En iyi önce, C'nin sınırı (82) kuyruktaki öbür düğümlerden küçük olduğu için C'yi sona bırakmıştı; o sıraya gelindiğinde `enIyi` 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 5.8: 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).

5.3.7 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 5.9: 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 5.9 üç aramayı yan yana gösterir; Tablo 5.7 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 5.7: 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 5.2

Örnek 5.1'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.

5.4 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.

5.4.1 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 dönülünce 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 5.2 (En ucuz çıkış sınırı geçerlidir)

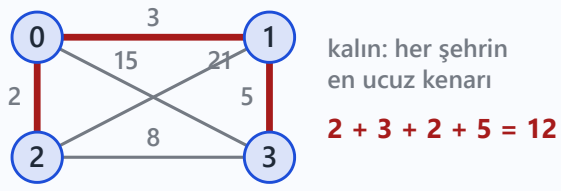
0'dan başlayan her kısmi yol için, bu yolu tamamlayan her turun maliyeti *alt(yol)*'dan büyük ya da ona eşittir.

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 5.8'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 5.8: Dört şehirli simetrik uzaklık matrisi *d* ve `enKucuk`.



Şekil 5.10: 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 5.10'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.

5.4.2 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:

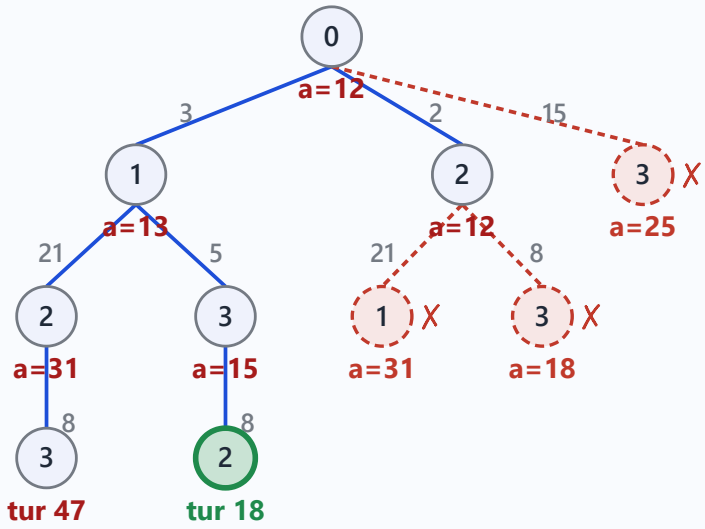
- `if (sayac == n)` taban durumdur. Bütün şehirler yoldadır; son şehirden 0'a dönülür ve tur tamamlanır. Tur `enIyi` 'den kısaysa `enIyi` güncellenir.
- `for` döngüsü dallanmadır. Ziyaret edilmemiş her şehir `s` bir çocuktur. Ziyaret edilmiş şehirler `continue` ile atlanır; bu, olursuzluk kuralıdır.
- `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 5.9 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 5.9: `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 5.11: `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.

5.4.3 Kod izleme: altSinir(3, 8)

Sınır fonksiyonu kısadır:

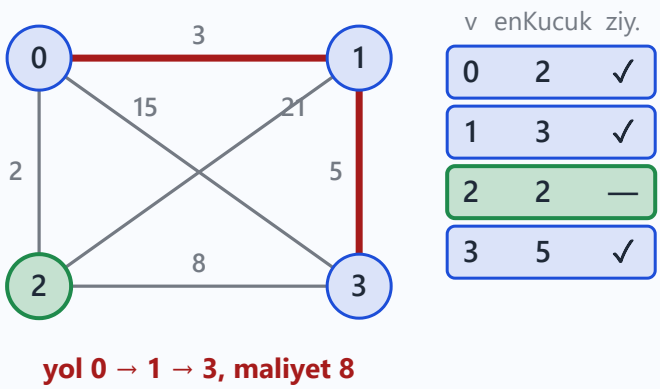
```
int altSinir(int son, int maliyet) {
  int alt = maliyet + enKucuk[son];
  for (int v = 0; v < n; v++)
    if (!ziyaret[v])       // gidilmemiş
      alt += enKucuk[v];
  return alt;
}
```

`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 5.10: `altSinir(3, 8)` izlemesi.



Şekil 5.12: `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 5.3

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.

5.4.4 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 5.3 (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 $\Sigma_v\, t(v) = 2 \cdot maliyet(T)$ 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 $2 \cdot maliyet(T) \geq \Sigma_v\, e(v)$ elde edilir. ■

Kökte bu sınırı hesaplayan kod şöyledir:

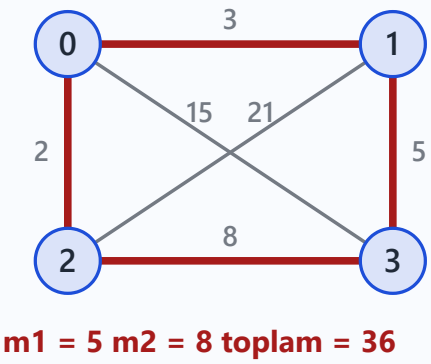
```
double ikiKenarSinir() {
  int toplam = 0;
  for (int v = 0; v < n; v++) {
    int m1 = INF, m2 = INF;     // en küçük iki
    for (int u = 0; u < n; u++) {
      if (u == v) continue;
      if (d[v][u] < m1) { m2 = m1; m1 = d[v][u]; }
      else if (d[v][u] < m2) m2 = d[v][u];
    }
    toplam += 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 5.11: `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 5.13: 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 5.13'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.

5.4.5 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 5.12: `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 5.13 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 5.13: 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 5.4

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.

5.4.6 Üç sınırın karşılaştırması

Aynı dört şehirli örnekte üç alt sınır kökte üç farklı değer verir (Şekil 5.14).



Şekil 5.14: 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 5.14: 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.

5.4.7 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 5.15).

	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 5.15: 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 5.16: 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.

5.5 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.

5.5.1 Problem ve alt sınır

> **Tanım 5.4**
>
> **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 5.17: 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.

5.5.2 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 5.18: `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 5.15: 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 5.19'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
1	5	6	2	9

kişi \ iş	0	1	2	3
2	4	8	5	3
3	6	2	7	5

Tablo 5.19: 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.

5.6 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.

5.6.1 Problem ve LP gevşetmesi

> **Tanım 5.5**
>
> **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.

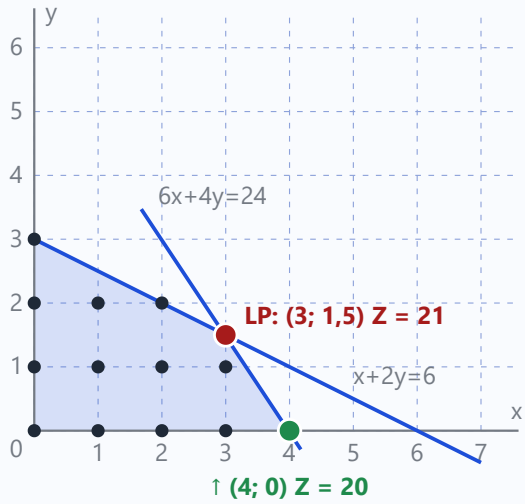
Tam sayı koşulu olmasaydı problem sıradan bir **doğrusal program** (*linear program*, LP) olurdu. Doğrusal programlar simpleks yöntemiyle pratikte çok hızlı, iç nokta yöntemleriyle de polinom sürede çözülür. İki değişkenli küçük örneklerde çözüm grafikle bulunur: kısıtlar düzlemde bir çokgen (olurlu 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



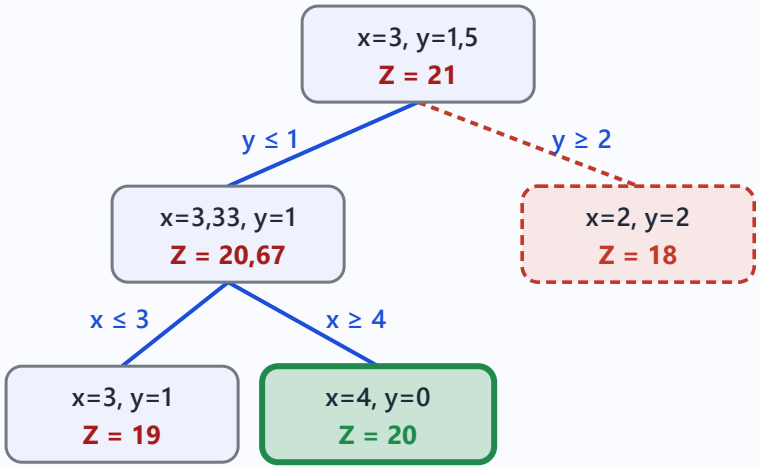
Şekil 5.16: Gölgeli alan LP gevşetmesinin olurlu bölgesidir; noktalar tam sayı çözümlerdir. LP optimumu (3; 1,5) noktasındadır ve $Z = 21$'dir. Tam sayı optimum (4; 0) noktasındadır ve $Z = 20$'dir.

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 5.16 bölgeyi ve iki optimumu gösterir.

5.6.2 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 5.17: 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.

5.7 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.

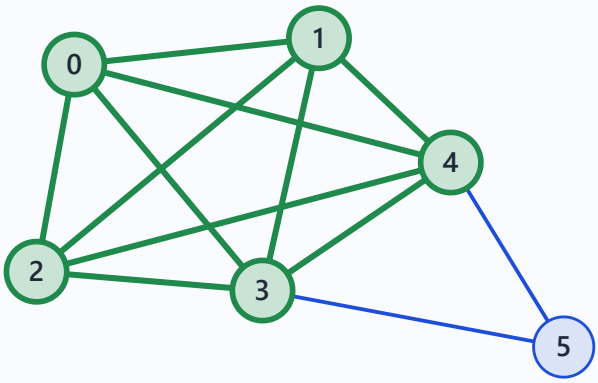
5.7.1 Problem ve sınır

Tanım 5.6

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 5.18: 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.

5.7.2 Kod izleme: klik({}, 0)

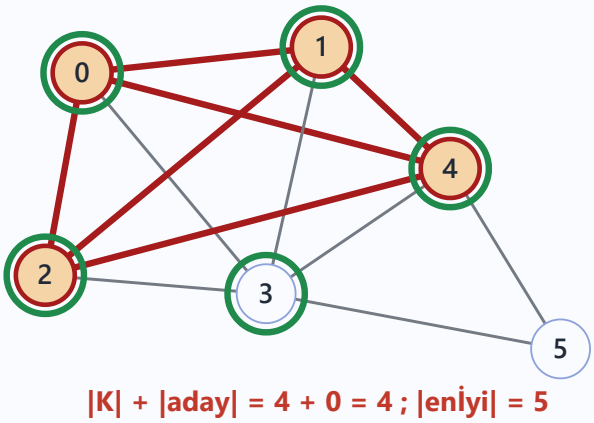
Arama boş klik ve `bas` = 0 ile başlar. Tablo 5.20 çağrıları sırayla verir.

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 5.20: `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.



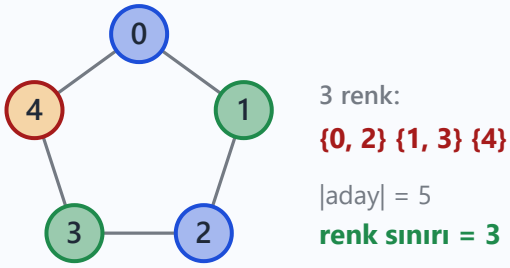
Şekil 5.19: 7. çağrı: *K = {0, 1, 2, 4}* (turuncu), aday yok. Yeşil halkalar en iyi kliği (5 düğüm) gösterir. *|K| + |aday| = 4 + 0 ≤ 5*; dal budanır.

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.

5.7.3 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.



Şekil 5.20: Beş adaylı bir halka. Açgözlü boyama 3 renk kullanır: {0, 2}, {1, 3}, {4}. *|aday| = 5*, renk sınırı 3'tür.

Şekil 5.20'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.

5.8 Diğer problemler ve A* ile ilişki

5.8.1 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.

5.8.2 Dal sınır, A* ve derinlik öncelikli dal sınır

En iyi önce dal sınır ile yol bulmadaki A* algoritması aynı fikrin iki yüzüdür. Tablo 5.21 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 5.21: 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.

5.9 Tasarım paradigmalarının karşılaştırması

Dal sınır, algoritma tasarım paradigmaları arasında en son ele alınanlardandır ve ötekilerin bazı fikirlerini birleştirir. Tablo 5.22 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 5.22: 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.

5.10 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.

5.11 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.

5.12 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*

5.13 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 ≤ 7*, *3x + y ≤ 9*, *x, y ≥ 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 ≤ 2*: (2,33; 2), *Z = 17*; *x* kesirlidir. Onun altında *x ≤ 2*: (2; 2), *Z = 16*, tam sayı, `enIyi` = 16. *x ≥ 3*: (3; 0), *Z = 9 ≤ 16*, budanır. Dal *y ≥ 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.

5.14 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.

5.15 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 II · BÖLÜM 6

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/advanced-algorithms-java/Bolum06/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.

6.1 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.

6.1.1 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 6.1

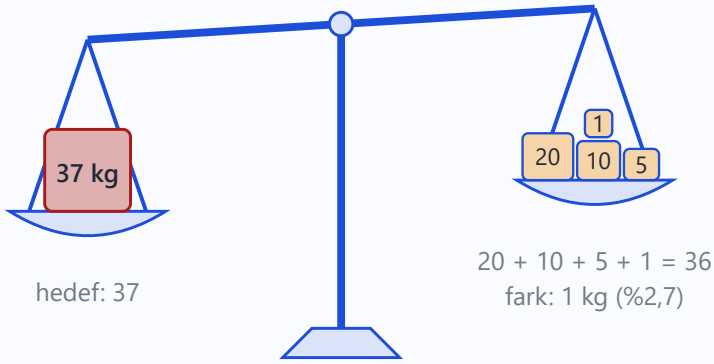
Yaklaşım algoritması (*approximation algorithm*), bir en iyileme problemi için polinom sürede çalışan ve her girdide, en iyi çözüme kanıtlanmış bir oran içinde yakın olan uygun bir çözüm üreten algoritmadır.

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.

6.1.2 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 6.1: Terazi benzetmesi. Solda 37 kilogramlık yük, sağda 20 + 10 + 5 + 1 = 36 kilogram. Tam denge kurulamaz; fark 1 kilogramdır (%2,7).

Şekil 6.1 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.

6.1.3 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.

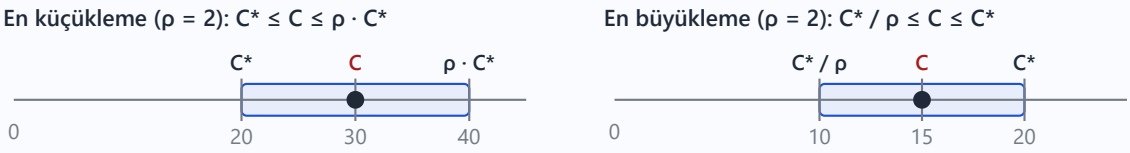
Tanım 6.2

Bir algoritma, n boyutlu her girdi için

$$\max(\, C / C^*,\ C^* / C \,) \le \rho(n)$$

koşulunu sağlıyorsa bir **ρ(n)-yaklaşım algoritmasıdır**. *ρ(n)*'ye algoritmanın **yaklaşım oranı** (*approximation ratio*) denir. Oran her zaman $\rho \ge 1$'dir; $\rho = 1$ ise algoritma en iyi çözümü bulur.

Tanımdaki *max* iki durumu birleştirir. En küçüklemede $C \ge C^*$ olduğu için büyük oran C / C^*'dır ve koşul $C \le \rho \cdot C^*$ 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 $C \le C^*$ olduğu için büyük oran C^* / C'dir ve koşul $C \ge C^* / \rho$ 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 6.2: Sayı doğrusunda yaklaşım oranı ($\rho = 2$). Solda en küçükleme: $C^* = 20$ ise C 20 ile 40 arasındadır. Sağda en büyükleme: $C^* = 20$ ise C 10 ile 20 arasındadır.

Şekil 6.2 iki durumu sayı doğrusunda gösterir. En küçüklemede C'nin bulunabileceği aralık $[C^*, \rho \cdot C^*]$, en büyüklemede $[C^* / \rho, C^*]$'dır. Şekildeki örnekte $C^* = 20$ ve $\rho = 2$ 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 $20 / 15 \approx 1{,}33$'tür. İkisi de 2 sınırının içindedir.

Örnek 6.1

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 $30 \le 2 \cdot 20 = 40$'tır.

Ters yönden bakılabilir. 2-yaklaşım bir algoritma 30 bulduysa en iyi maliyet hakkında ne söylenebilir? $C \le 2 \cdot C^*$ eşitsizliğinden $C^* \ge 15$ çıkar. Ayrıca en iyi çözüm, bulunan çözümden kötü olamaz: $C^* \le 30$. 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.

6.1.4 Yaklaşım algoritması, sezgisel ve yaklaşım şemaları

Zor bir en iyileme problemine uygulanabilecek yöntemler, verdikleri garantiye göre sıralanabilir. Tablo 6.1 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 6.1: 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 6.3

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.

6.1.5 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.

6.2 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.

6.2.1 Problemin tanımı

Tanım 6.4

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 6.2'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 6.2: 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.

6.2.2 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 6.3: Simetrik TSP'de farklı tur sayısı ve saniyede 10^9 tur denendiğinde gereken süre.

Tablo 6.3 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.

6.2.3 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.

6.2.4 Kod izleme: beş şehir

`enYakinKomsu(d, 0)` çağrısı Tablo 6.2'deki matrisle çalıştırılsın. Başlangıçta `ziyaret[0] = true`, `tur = [0]` ve `toplam = 0`'dır. Tablo 6.4 her dış döngü adımını gösterir.

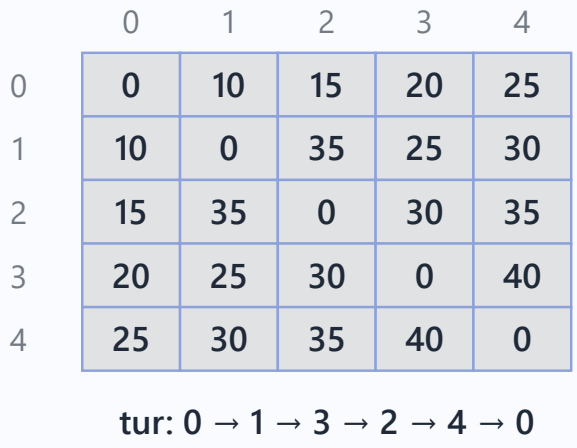


Şekil 6.3: En yakın komşu izlemesinin başlangıcı. Uzaklık matrisi; tur yalnız 0 şehrinden oluşur.

`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 6.4: `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 6.4: İzlemenin sonu. Tur *0 → 1 → 3 → 2 → 4 → 0*, uzunluk 125.

Sonuç $0 \to 1 \to 3 \to 2 \to 4 \to 0$ turudur ve uzunluğu 125'tir (Şekil 6.4). Doğrulama betiği aynı kodu beş başlangıç şehrinin hepsiyle çalıştırmıştır. Her başlangıçta bulunan turun uzunluğu yine 125'tir. Örneğin 2'den başlanırsa tur $2 \to 0 \to 1 \to 3 \to 4 \to 2$ olur: $15 + 10 + 25 + 40 + 35 = 125$.

6.2.5 Kaba kuvvetle karşılaştırma

Beş şehirde $(5 - 1)!/2 = 12$ farklı tur vardır. Hepsi denenince en yakın komşunun sonucu değerlendirilebilir. Tablo 6.5 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 6.5: 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.

6.2.6 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 6.1 (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.

6.2.7 Metrik TSP için MST tabanlı 2-yaklaşım

Tanım 6.5

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.

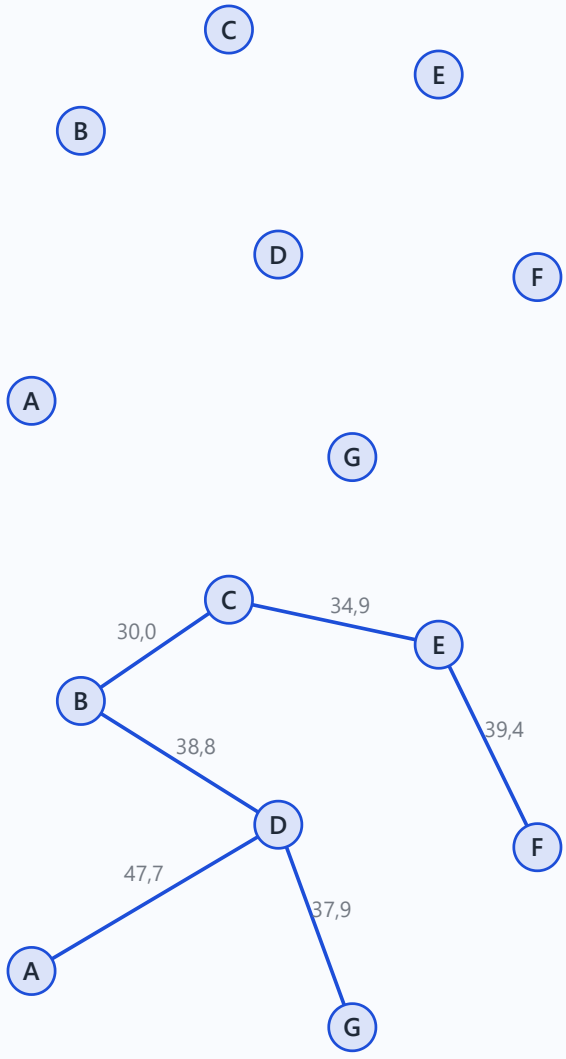
Gerçek hayattaki uzaklıkların çoğu metriktir. Düzlemdeki noktalar arasındaki Öklid uzaklığı, bir yol ağındaki en kısa yol uzaklıkları ve uçuş süreleri bu türdendir. Metrik TSP hâlâ NP-zordur, ama artık sabit oranlı bir algoritma mümkündür. En basiti minimum kapsayan ağaca (*minimum spanning tree*, MST) 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.

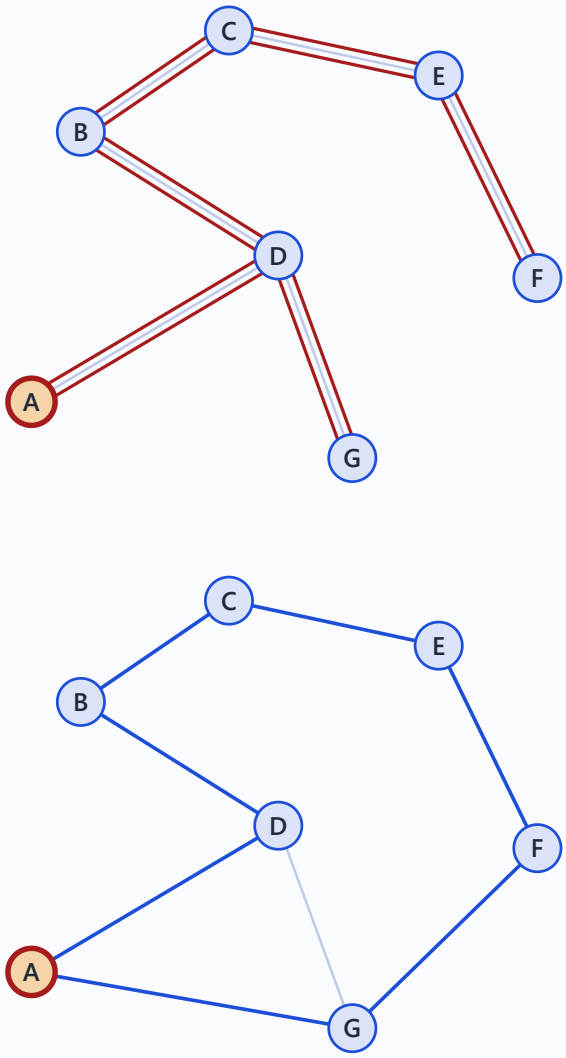
Sunumdaki örnekte düzlemde yedi şehir vardır ve uzaklıklar Öklid uzaklığıdır. Doğrulama betiği şehirlerin koordinatlarını şekildeki çizimden geri hesaplamıştır: A (10, 20), B (18, 68), C (42, 86), D (50, 46), E (76, 78), F (92, 42), G (62, 10). Bütün uzaklıklar bu koordinatlardan bulunur.



Şekil 6.5: 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 6.5). 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 6.6: 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 6.6'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 6.6: 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 6.2 (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 6.2

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 6.2'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.

6.2.8 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 6.7 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).
En yakın komşu	$\Theta(\log n)$	Metrik durumda bile sabit oran garanti etmez.

Tablo 6.7: TSP için üçgen eşitsizliğine göre bilinen yaklaşım sonuçları.

6.3 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.

6.3.1 Yerel arama fikri

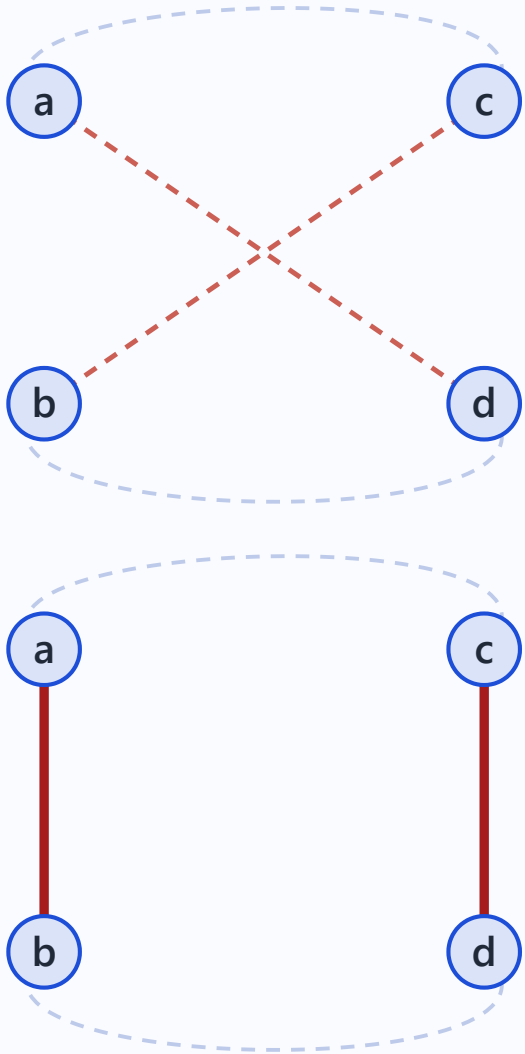
Tanım 6.6

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.

6.3.2 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 … c* arasındaki alt yol ters yönde gezilir. Bu değişikliğe **2-opt hamlesi** denir.



Şekil 6.7: 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 6.3

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.

6.3.3 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.

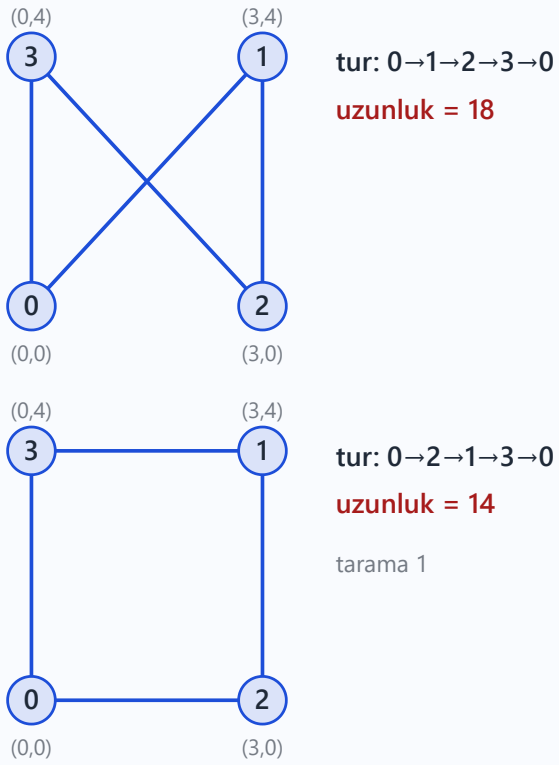
Hamle yapılınca `tersCevir(tur, i + 1, j)` çağrılır. Bu yardımcı metot sunumda verilmez; `tur[i + 1]` ile `tur[j]` arasındaki öğeleri, iki sınır dahil, yerinde ters çevirir. İki uçtan ortaya doğru yer değiştiren bir döngüyle yazılabilir:

```
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.

6.3.4 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 6.8: 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 6.8: `ikiOpt` izlemesi, dört nokta (*n = 4*).

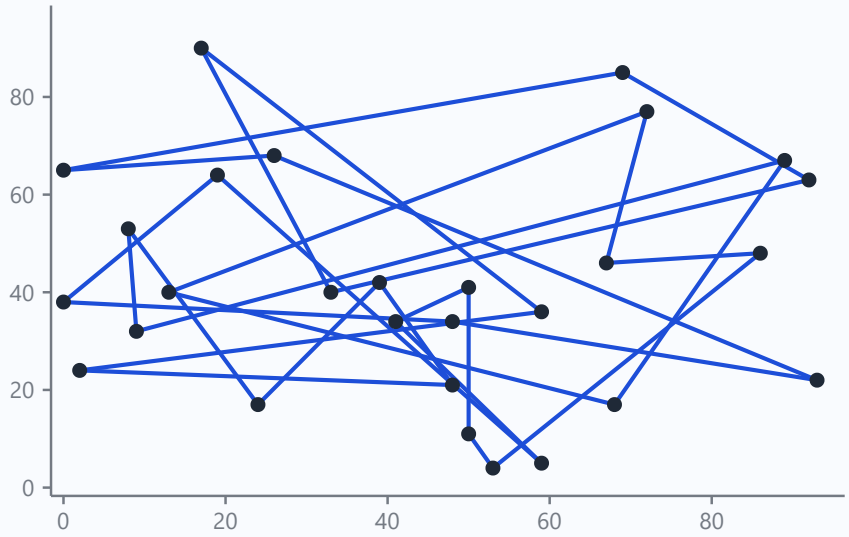
Tablo 6.8 izlemeyi verir. *n = 4* olduğu için `i` 0 ile 2 arasında, `j` de `i + 2` ile 3 arasında değişir. Geçerli çiftler (0, 2), (0, 3) ve (1, 3)'tür. İlk taramada (0, 2) çifti *(0, 1)* ve *(2, 3)* köşegenlerini çıkarıp *(0, 2)* ve *(1, 3)* kenarlarını önerir. Bu kenarlar 3 birimlik kısa kenarlardır: *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.

6.3.5 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.

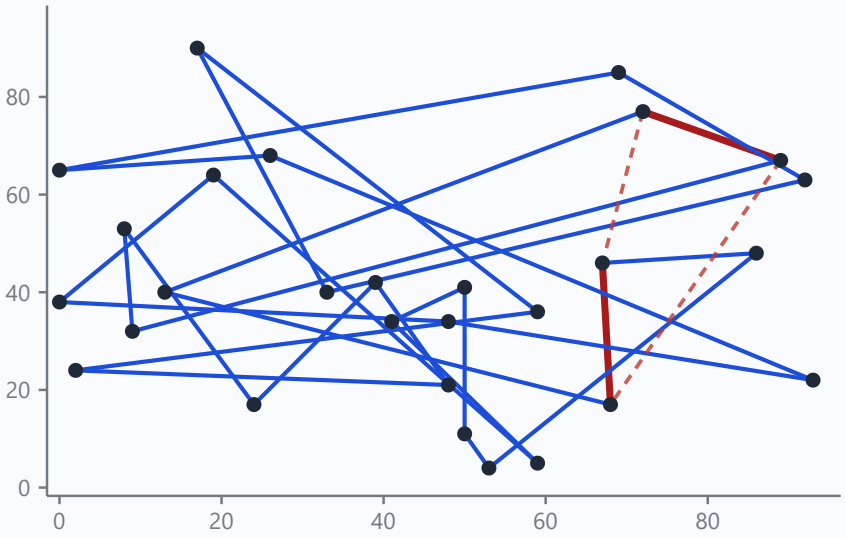


Şekil 6.9: 28 şehir ve rastgele başlangıç turu; uzunluk 1269,8. Kenarlar birbirini 66 noktada keser.

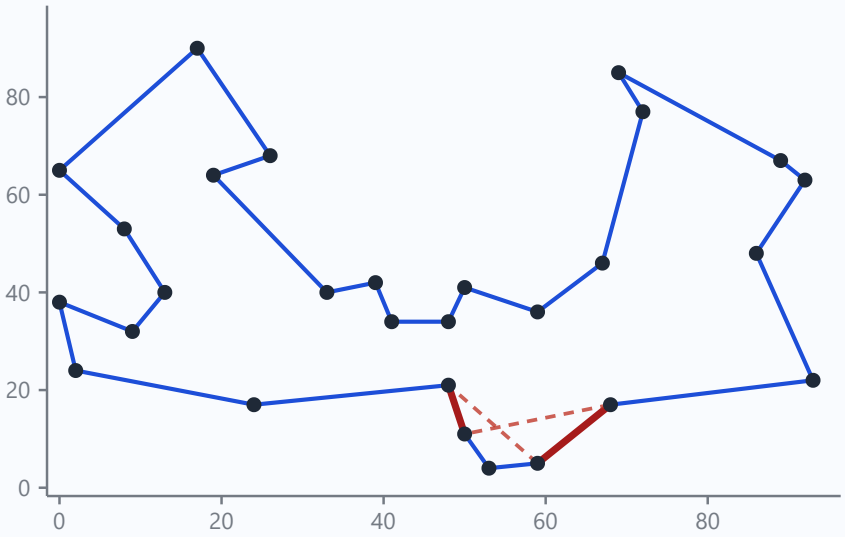
Hamle	Kazanç Δ	Tur uzunluğu
başlangıç	—	1269,8
1	−36,9	1232,9
2	−8,9	1224,0
4	−27,6	1187,3
8	−32,6	1111,7
14	−14,9	1005,2
22	−37,0	852,8
32	−11,8	595,9
48 (son)	−13,2	430,8

Tablo 6.9: 28 şehirde ilk iyileştiren hamle stratejisiyle 2-opt; seçilmiş hamlelerden sonra kazanç ve tur uzunluğu.

Tablo 6.9'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 6.10: İlk iyileştiren hamle. Kesikli kenarlar çıkarılır, kalın kenarlar eklenir; $\Delta = -36{,}9$, yeni uzunluk 1232,9.



Şekil 6.11: 48. ve son hamle. Artık hiçbir 2-opt hamlesi turu kısaltmaz; tur yerel optimumdur, uzunluk 430,8. Kesişen kenar kalmamıştır.

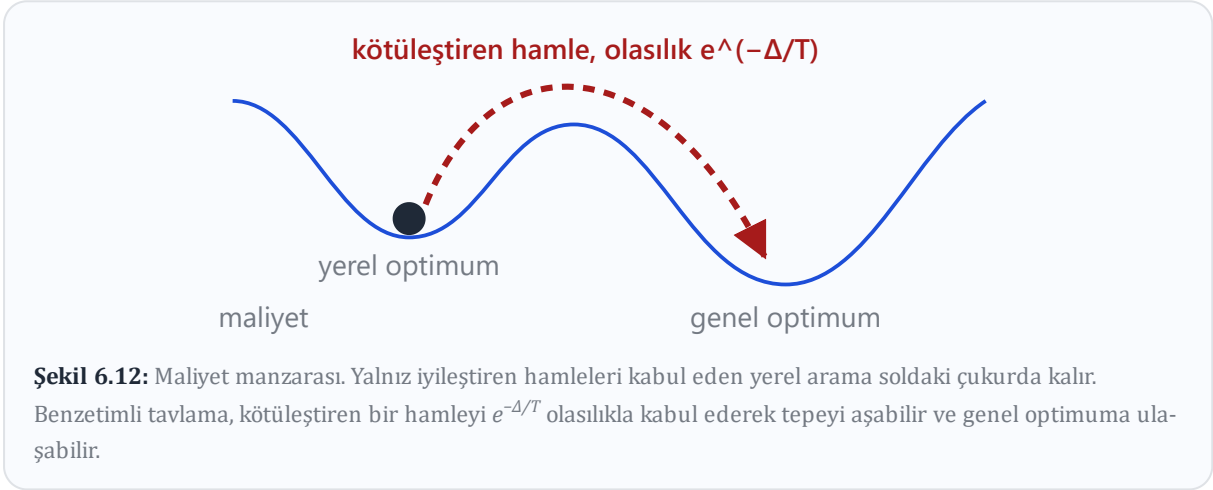
Son turda kesişen kenar yoktur (Şekil 6.11). Bu, 2-opt hamlesinin geometrik yorumundan beklenen sonuçtur: 2-opt yerel optimumu olan bir Öklid turunun kenarları birbirini kesmez. Ancak kesişme olmaması turun en iyi olduğu anlamına gelmez. Kesişmesiz birçok tur vardır ve bunların uzunlukları

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.

6.3.6 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 6.12: 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
1	5	≈ 0,0067

Tablo 6.10: $\Delta = 5$ kadar kötüleştiren bir hamlenin kabul olasılığı $e^{-\Delta/T}$.

Tablo 6.10 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.

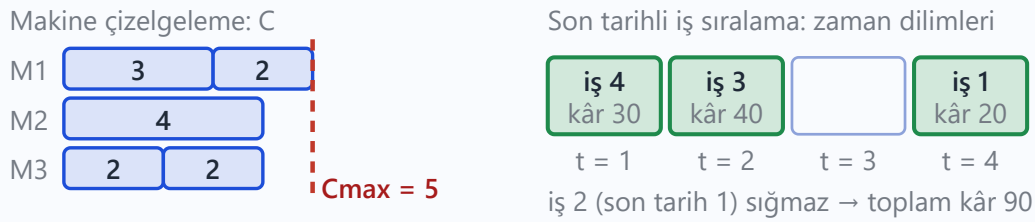
6.4 İş çizelgeleme

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.

6.4.1 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 6.13: 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.

6.4.2 Liste çizelgeleme

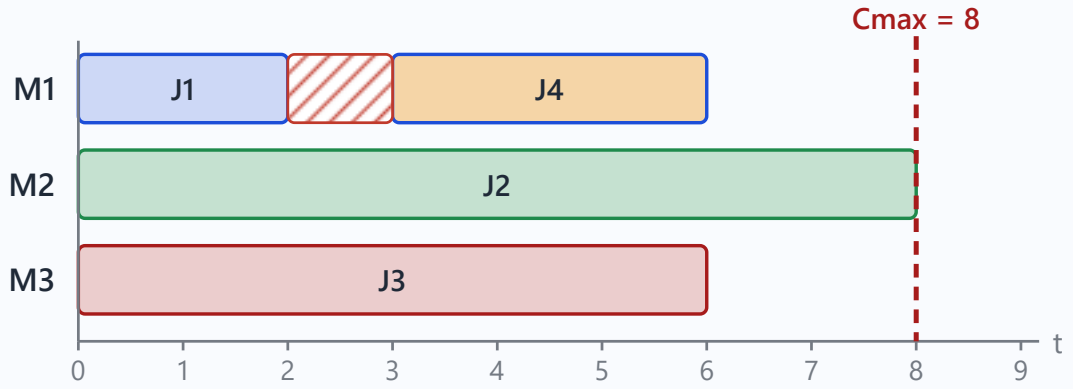
Tanım 6.7

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 6.11: Varış zamanlı liste çizelgeleme örneği.



Şekil 6.14: 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 6.14). 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.

6.4.3 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 6.3 (Graham)

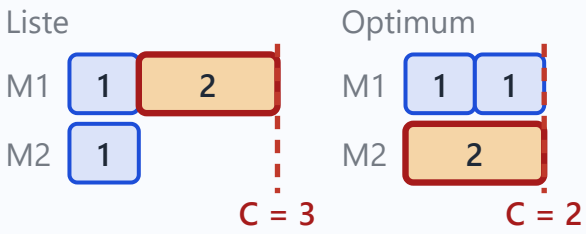
Bütün işler başta hazırsa liste çizelgeleme, işlerin sırası ne olursa olsun, $C_{max} \leq (2 - 1/m) \cdot OPT$ koşulunu sağlayan bir çizelge üretir.

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.



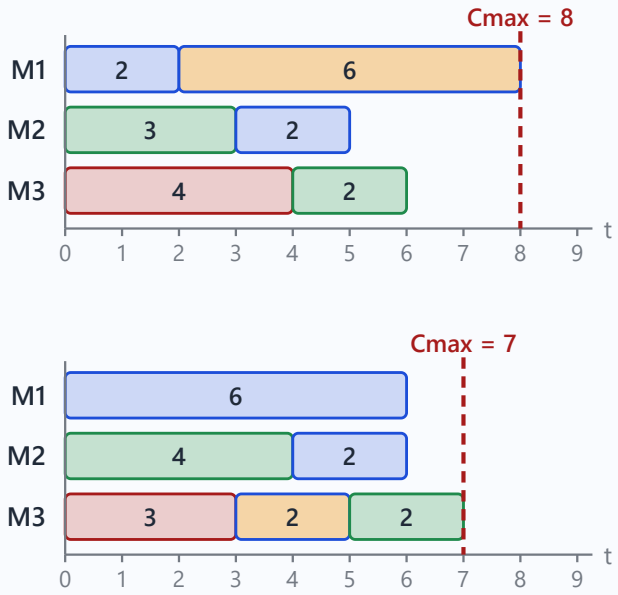
Şekil 6.15: Sıkı örnek, $m = 2$ ve işler 1, 1, 2. Liste çizelgeleme $C = 3$, en iyi çizelge $C = 2$ bulur.

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.

6.4.4 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 6.16: 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 6.4 (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 + ε* oranına polinom sürede ulaşılabilir, ama süre *1/ε*'da üsteldir.

6.4.5 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 6.12: `lpt({2, 3, 4, 6, 2, 2}, 3)` izlemesi; sıralama sonrası işler 6, 4, 3, 2, 2, 2.



Şekil 6.17: LPT izlemesinin sonu. Makine yükleri 6, 6 ve 7; $C_{max} = 7$.

Tablo 6.12 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 $\lceil 19/3 \rceil = 7$ olduğu için bu çizelge en iyidir.

6.5 Düğüm kapsama

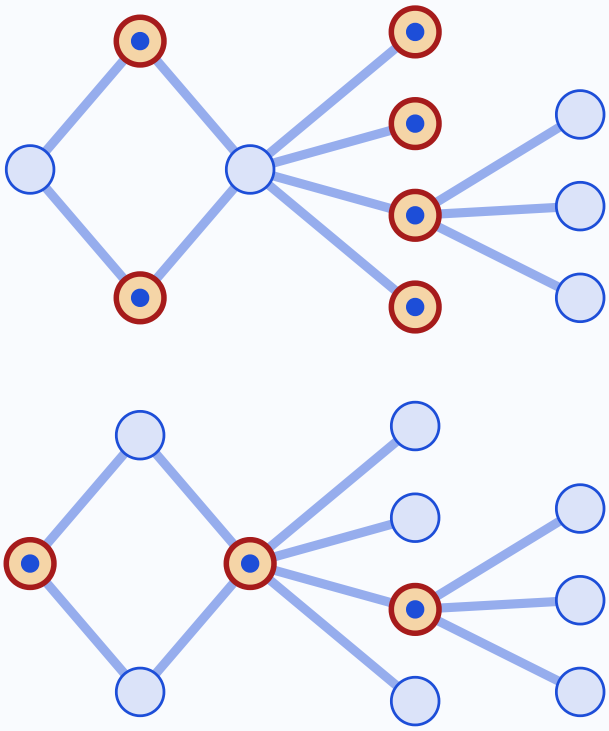
Bir kampüs ağında her bağlantının izlenmesi istensin. İzleme yazılımı bir yönlendiriciye kurulunca o yönlendiriciye bağlı bütün hatları görür. En az kaç yönlendiriciye yazılım kurulmalıdır? Bu soru, çizge kuramının klasik düğüm kapsama problemidir.

6.5.1 Problemin tanımı

Tanım 6.8

$G = (V, E)$ 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.

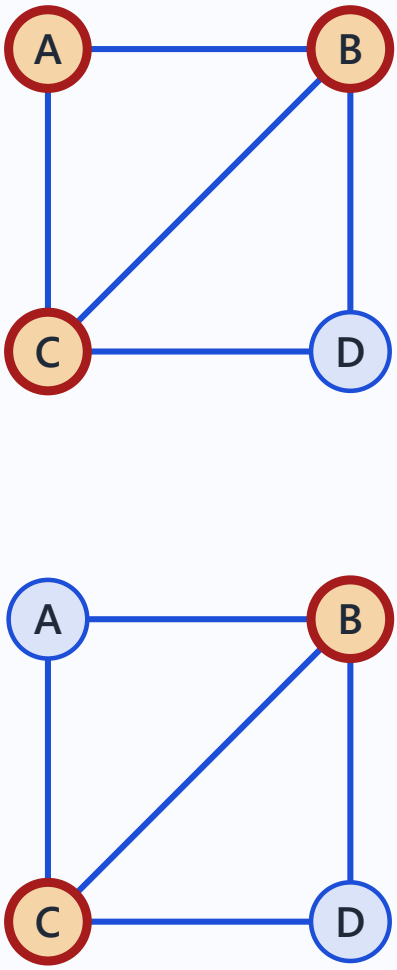
Kapsama kümesi bir bekçi düzeni gibi düşünülebilir: her düğümdeki bekçi o düğüme değen kenarları görür ve her kenarı en az bir bekçi görmelidir. Problemin karar sürümü, yani “k düğümlük bir kapsama var mı?” sorusu, NP-tamdır.



Şekil 6.18: 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 6.18'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.

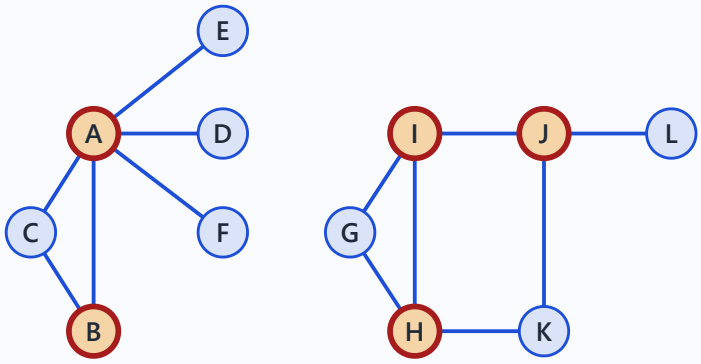


Şekil 6.19: Dört düğümlü çizge. Solda *{A, B, C}*, geçerli ve boyutu 3. Sağda *{B, C}*, en küçük ve boyutu 2.

6.5.2 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.

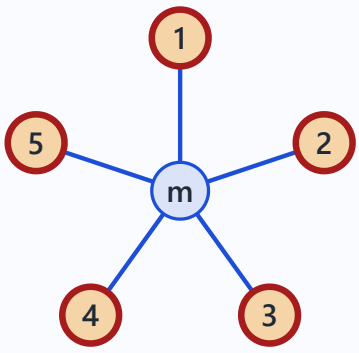


Şekil 6.20: A'dan L'ye 12 düğümlü çizge. En küçük kapsamalardan biri *{A, B, H, I, J}*, 5 düğüm.

Şekil 6.20'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.

6.5.3 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 6.21: Yıldız çizge. Her kenarda yaprak ucu seçilirse 5 düğüm seçilir; en iyi çözüm yalnız merkezdir, *{m}*.

Şekil 6.21 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.

6.5.4 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;
}
```

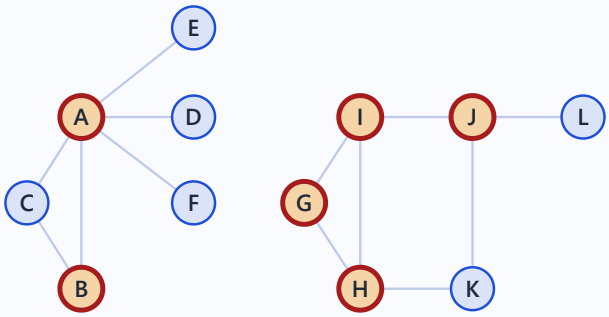
Her kenar iki karakterlik bir dizidir: `k[0]` ve `k[1]` uçlardır. Kapsama bir `HashSet`'te tutulur; `contains` ve `add` işlemleri ortalama *O(1)* sürer. Döngü her kenarda iki ucun da kapsamada olmadığını denetler. Öyleyse kenar henüz kapsanmamıştır ve iki ucu birden eklenir. Uçlardan biri zaten kapsamadaysa 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 6.13: `dugumKapsama` izlemesi, 12 düğümlü çizge.

Tablo 6.13, kodu Şekil 6.20'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 6.22: 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.

6.5.5 Oran neden en fazla 2'dir?

Teorem 6.5 (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 6.20'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.

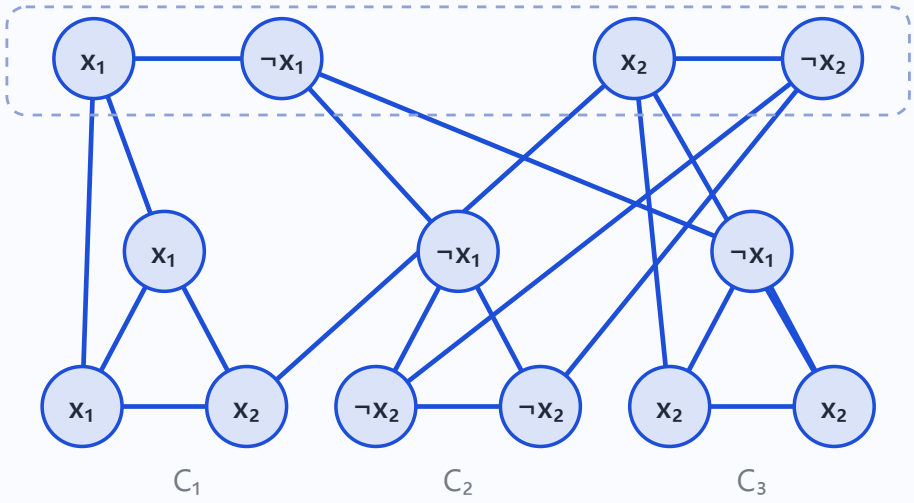
6.5.6 NP-tamlık: 3-SAT'tan indirgeme

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.

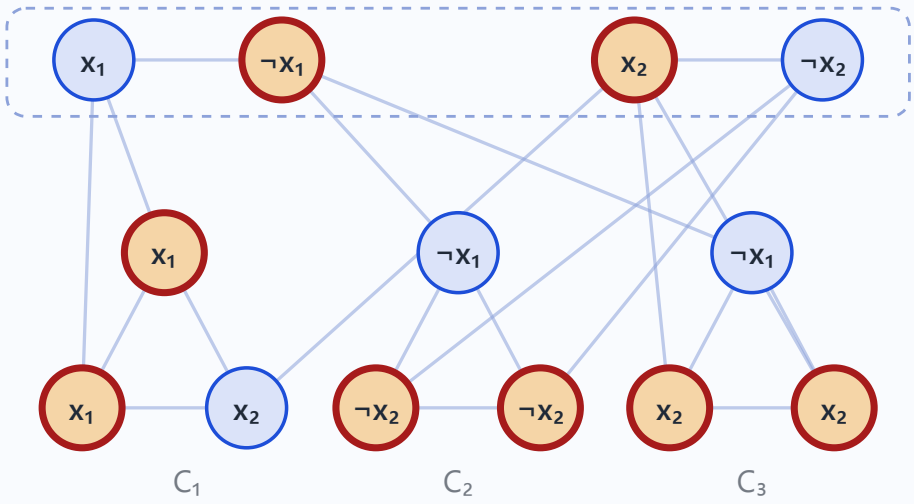
Aranan kapsama boyutu $k = l + 2m$'dir: her değişken kenarından bir düğüm, her üçgenden iki düğüm. Çizgede $2l + 3m$ düğüm ve $l + 6m$ kenar vardır; kurulum polinom süredir.



Şekil 6.23: $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$ 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 6.24: 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 6.24). 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.

6.6 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.

6.6.1 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 6.14: Kapasite $W = 10$ için oran sıralı açgözlü kural.

Tablo 6.14 sunumdaki yedi eşyalı örneği verir. En yüksek oranlı eşya 6'dır (20/7 ≈ 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 6.4

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.

6.6.2 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 6.6 (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

$$\mathrm{OPT} \leq v_1 + \dots + v_{k-1} + v_k$$

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 6.5

$W = 15$ 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.

6.6.3 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.

6.7 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.

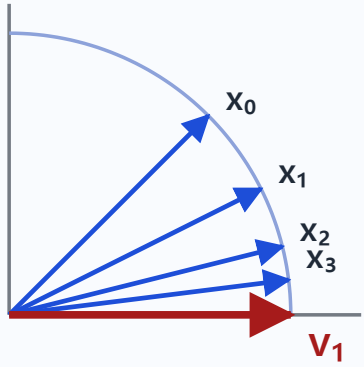
6.7.1 Baskın özdeğer ve özvektör

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 6.9

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 6.25: $A = [[2, 0], [0, 1]]$ 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 6.25). 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.

6.7.2 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.

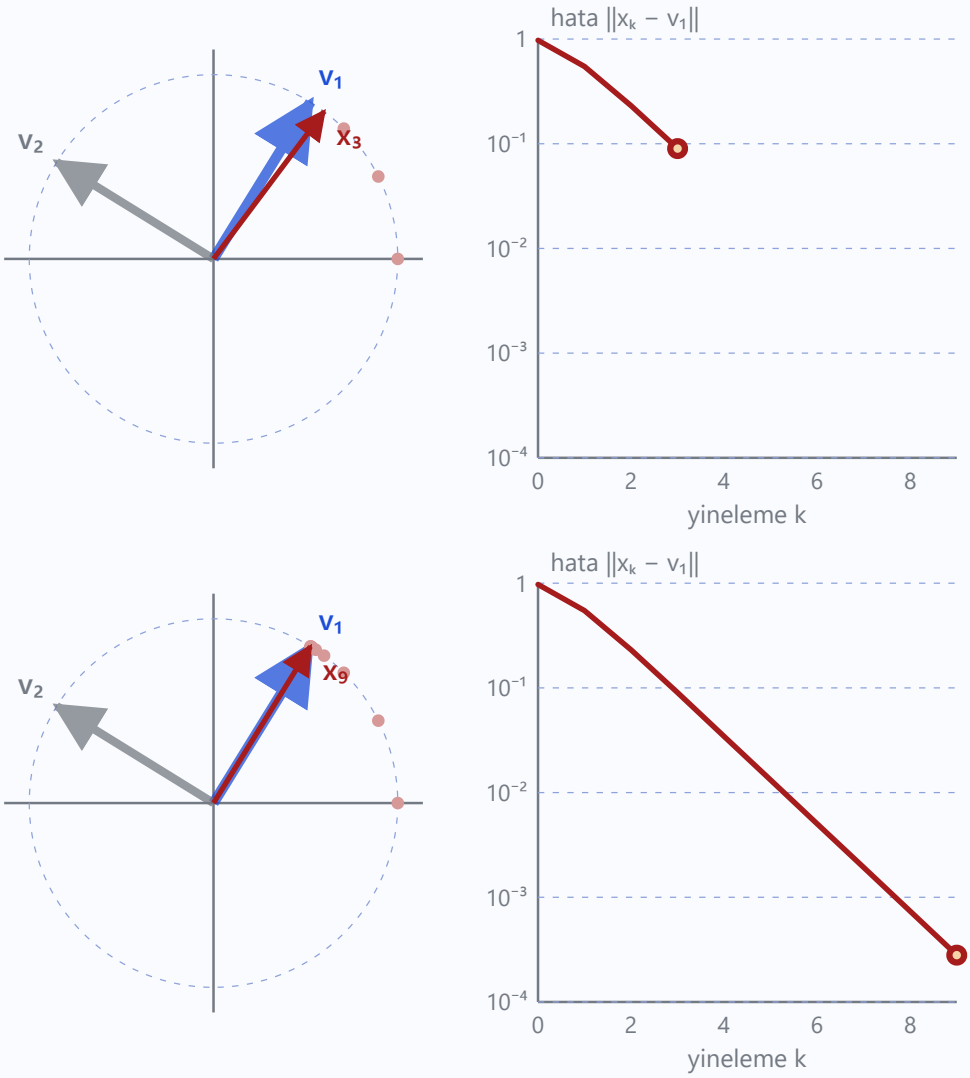
6.7.3 Adım adım yakınsama

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 6.15: $A = [[2, 1], [1, 3]]$, $x_0 = (1, 0)$ ile kuvvet yinelemesi. λ, Rayleigh bölümüdür; hata $\|x_k - v_1\|$.

Tablo 6.15 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 6.26: 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 6.26). 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.

6.7.4 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.

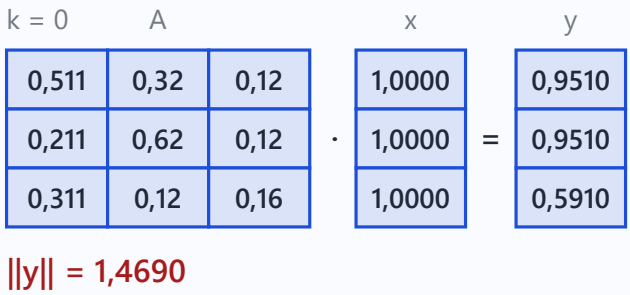
`k`	`y = A · x`	`norm` = $\|y\|$	**Yeni** `x`
0	(0,9510; 0,9510; 0,5910)	1,4690	(0,6474; 0,6474; 0,4023)
1	(0,5862; 0,5862; 0,3434)	0,8974	(0,6533; 0,6533; 0,3827)
2	(0,5888; 0,5888; 0,3428)	0,9005	(0,6539; 0,6539; 0,3807)

`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 6.16: `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 6.27: 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.

6.7.5 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.

6.8 Karşılaştırma ve sık yapılan hatalar

Tablo 6.17 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 6.17: 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.

6.9 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 $|\lambda_1| > |\lambda_2|$.

6.10 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*

6.11 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.

6.12 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.

6.13 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 III

Diğer Hesaplama Modelleri

Bölüm 7–9

KISIM III · BÖLÜM 7

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/advanced-algorithms-java/Bolum07/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.

7.1 Rastgelelik ve algoritmalar

7.1.1 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 7.1: 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.

7.1.2 Rastgele algoritma ve beklenen süre

Tanım 7.1

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 7.2 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 7.2: 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 7.2

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 7.1

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.

7.1.3 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.

7.1.4 Olasılık araçları

Rastgele algoritmaların analizinde birkaç temel olasılık aracı tekrar tekrar kullanılır. Tablo 7.1 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 7.1: 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.

7.2 Las Vegas ve Monte Carlo

7.2.1 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 7.3

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 7.2: 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.

7.2.2 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.

7.2.3 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 7.1 (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 7.3 sınırın tur sayısıyla nasıl düştüğünü logaritmik ölçekte gösterir: her yeni tur sınırı 4'e böler.



Şekil 7.3: 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 7.2

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.

7.2.4 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 7.3

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.

7.3 Karıştırma

7.3.1 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 7.3: 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.

7.3.2 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.

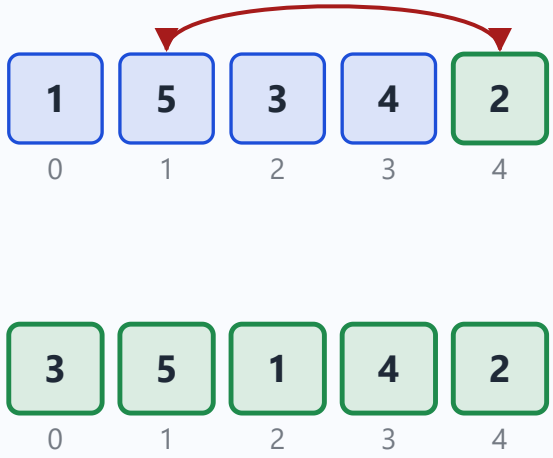
> **Örnek 7.4**
>
> $a = [1, 2, 3, 4, 5]$ dizisi karıştırılsın. Rastgele sayı üreteci sırasıyla $j = 1, 3, 0, 1$ değerlerini versin. Tablo 7.4 her adımı verir.

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 7.4: 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 7.4: 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.

7.3.3 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 7.2 (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 karıştı-

rıldı. 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.

7.3.4 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 ≥ 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 7.5: [1, 2, 3] dizisinde saf karıştırmanın 27 çekiliş dizisi tek tek sayıldı. Son sütun, tohumu 42 olan 600.000 denemelik benzetimin sonucudur.

Tablo 7.5'te 132, 213 ve 231 sonuçları beşer yolla, diğer üçü dörder yolla elde edilir. Beklenen sayılar *600.000 · 5/27 ≈ 111.111* ve *600.000 · 4/27 ≈ 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.

7.3.5 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 $j \in \{0, 1\}$, $i = 1$ turunda ise $j = 0$ zorunludur. Yalnız $2 \cdot 1 = 2$ 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 7.6: `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.

7.4 BogoSort

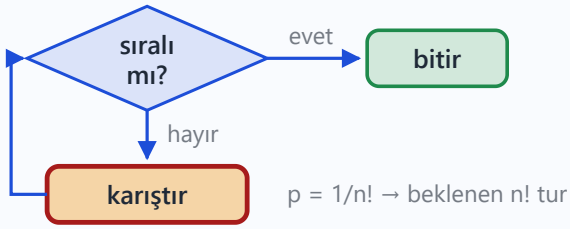
7.4.1 Algoritma

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 7.5: BogoSort akışı: sıralıysa biter, değilse karıştırıp yeniden sorar. Her turun başarı olasılığı $p = 1/n!$, beklenen tur sayısı $n!$'dir.

7.4.2 Bir çalıştırma

Örnek 7.5

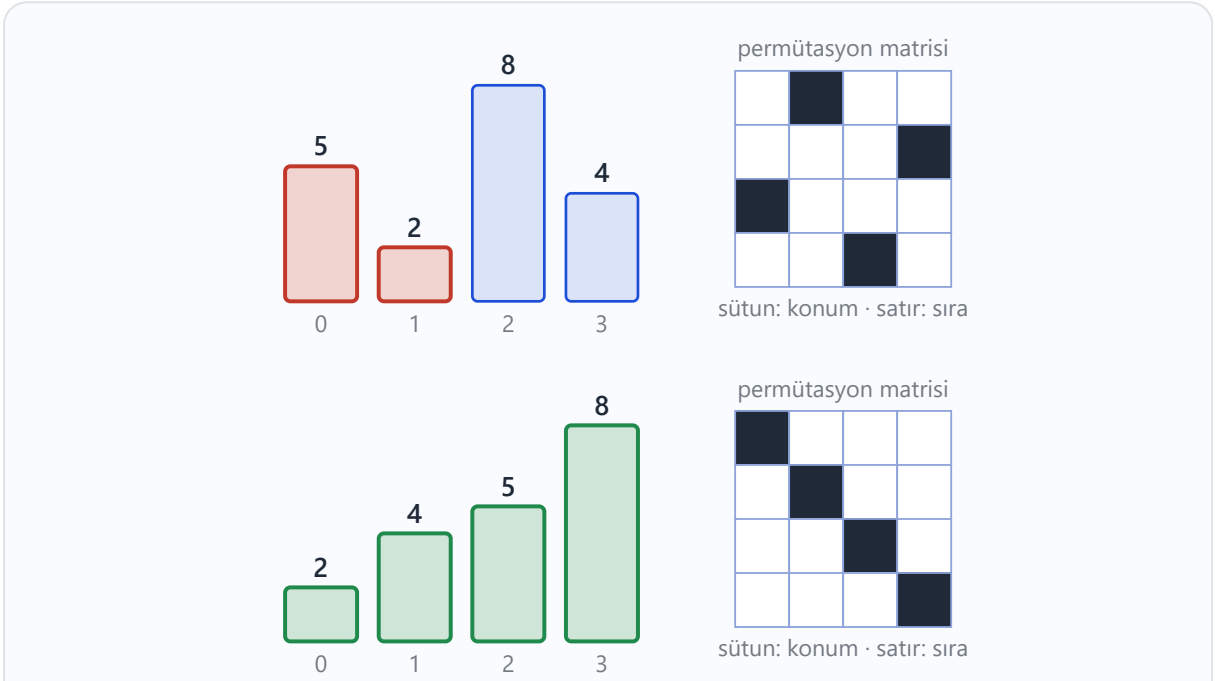
[5, 2, 8, 4] listesi BogoSort ile sıralansın. Her karıştırma Fisher–Yates ile yapılır. Tablo 7.7, tohumlu bir üreteçle yapılan bir çalıştırmanın bütün adımlarını verir. Her satırda `siraliMi`'nin bulduğu ilk ters sıralı komşu çift gösterilmiştir.

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
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

karıştırma	dizi	ilk ters çift	sonuç
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 7.7: [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 7.6: 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.

7.4.3 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 7.8: 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 7.6

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.

7.5 Rastgele QuickSort

7.5.1 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 7.9 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 7.9: 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.

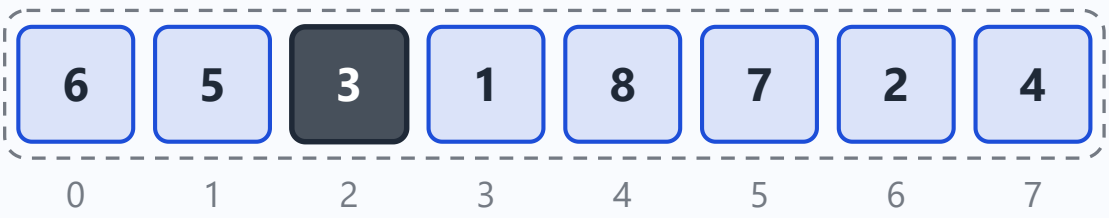
7.5.2 Rastgele pivotla bir çalıştırma

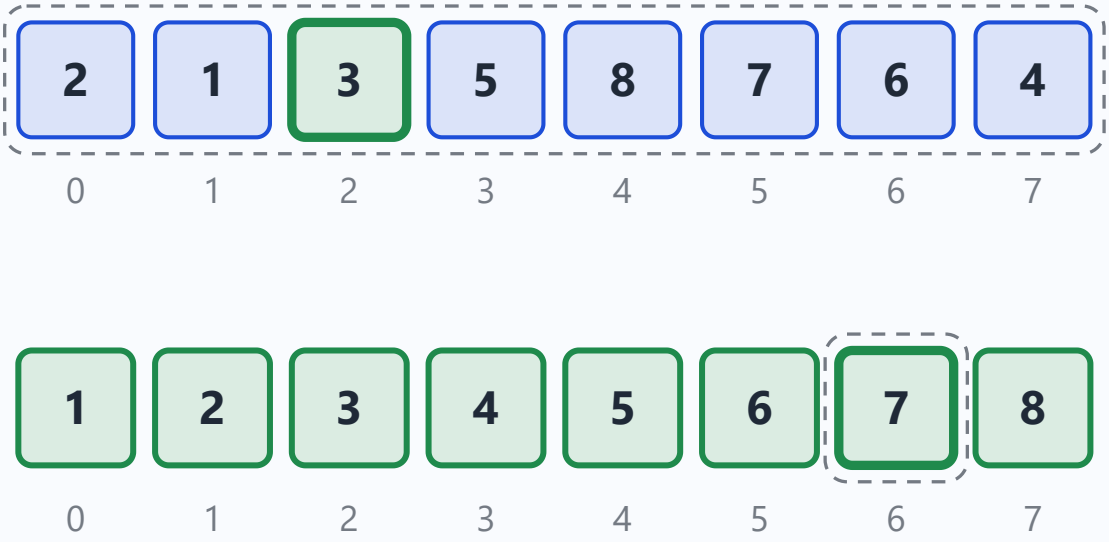
Rastgele pivotlu hızlı sıralama [6, 5, 3, 1, 8, 7, 2, 4] dizisinde çalıştırılsın. Bu çalıştırmada pivotlar sırasıyla 3, 1, 4, 6 ve 8 olarak seçilmiş olsun. İzlemede bölümleme iki işaretçiyle yapılır. *i* soldan ilerler ve pivottan küçük öğeleri atlar; ilk büyük ya da eşit öğede durur. *j* sağdan ilerler 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 7.10: [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 7.7: 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.

7.5.3 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 7.7**
>
> *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 7.11 bölümlemenin her adımını verir.

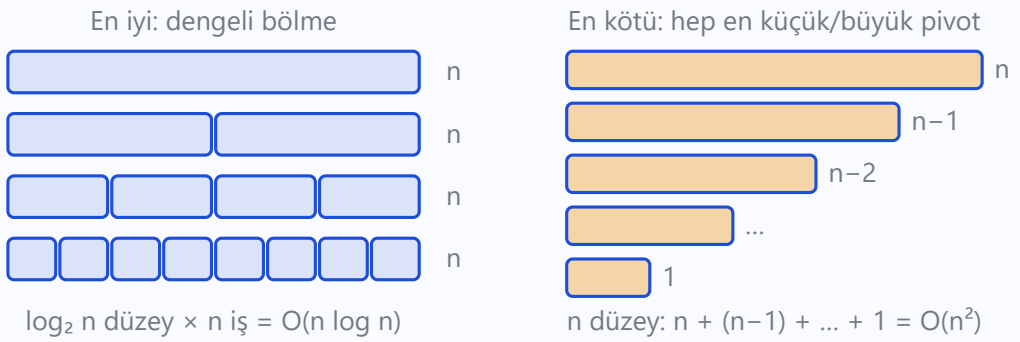
adım	i	a[i] ≤ 3?	işlem	k	dizi
pivot sona	—	—	a[2] ↔ a[7]	0	[6, 5, 4, 1, 8, 7, 2, 3]
1	0	6: hayır	—	0	değişmez
2	1	5: hayır	—	0	değişmez
3	2	4: hayır	—	0	değişmez
4	3	1: evet	a[3] ↔ a[0]	1	[1, 5, 4, 6, 8, 7, 2, 3]
5	4	8: hayır	—	1	değişmez
6	5	7: hayır	—	1	değişmez

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 7.11: 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.

7.5.4 En iyi, en kötü ve beklenen durum



Şekil 7.8: 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 $n + (n - 1) + ... + 1 = O(n^2)$.

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 7.12: 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.

7.5.5 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 7.3 (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 7.9: $\{z_2, \ldots, z_6\}$ 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 $E[X] < 2nH_n$'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 7.13: 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 7.13'teki 24,4 değeri, sıralı dizi dahil 10 öğeli her girdi için geçerlidir.

7.5.6 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.

7.6 Rastgele seçme (QuickSelect)

7.6.1 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.

7.6.2 Kod izleme

Örnek 7.8

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 7.14 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 7.14: `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 7.10: 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.

7.6.3 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.

7.7 Monte Carlo ile π

7.7.1 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*.

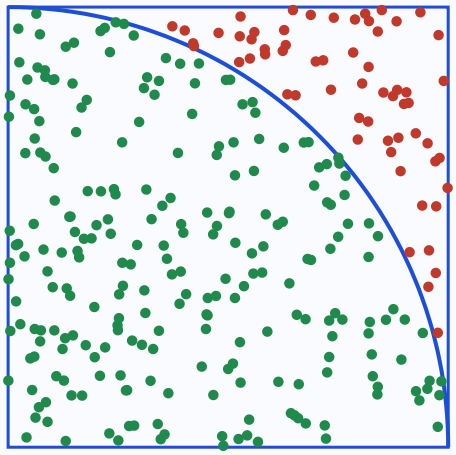
Bir nokta *(x, y)*, ancak orijine uzaklığı en çok 1 ise çeyrek dairenin içindedir: $x^2 + y^2 \le 1$. Karekök almaya gerek yoktur. *n* nokta atılır ve içeri düşenler sayılır. İç nokta oranı *π/4*'ün tahminidir; dolayısıyla

> $\pi \approx 4 \cdot \text{iç} / n$

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 7.11: Birim kareye atılan 300 nokta. 239'u çeyrek dairenin içinde (yeşil): $\pi \approx 4 \cdot 239 / 300 \approx 3{,}187$.

7.7.2 Kod izleme

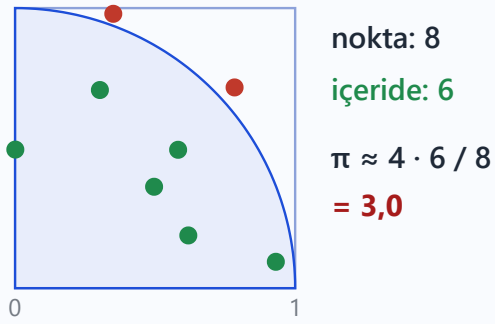
Örnek 7.9

`zar = new Random(2024)` ile `piTahmin(8)` çağrılsın. Tohum sabit olduğu için çalıştırma her seferinde aynı noktaları üretir. Tablo 7.15 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 7.15: `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 7.12: `piTahmin(8)` bittiğinde: 8 nokta, 6'sı içeride (yeşil), 2'si dışarıda (kırmızı). Tahmin *4 · 6 / 8 = 3,0*.

7.7.3 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 7.16: 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 7.10

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.

7.8 Miller–Rabin asallık testi

7.8.1 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.

7.8.2 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.

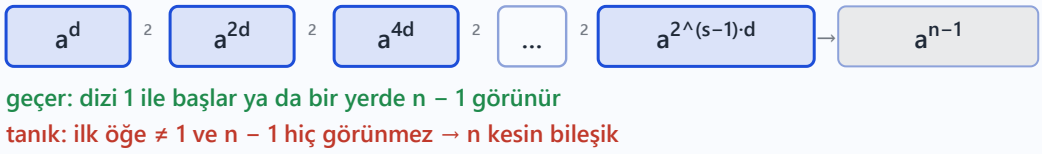
Test şöyle kurulur. n tek bir sayı olsun. $n-1$ çifttir ve $n - 1 = 2^s \cdot d$ biçiminde yazılır; burada d tektir. Rastgele bir a tabanı için şu dizi hesaplanır:

> $$a^d,\ a^{2d},\ a^{4d},\ \ldots,\ a^{2^{s-1}d} \pmod n$$

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 7.4

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.



geçer: dizi 1 ile başlar ya da bir yerde n − 1 görünür
tanık: ilk öğe ≠ 1 ve n − 1 hiç görünmez → n kesin bileşik

Şekil 7.13: 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.

7.8.3 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.

7.8.4 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 7.17: `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.

7.8.5 561 sayısının izlemesi

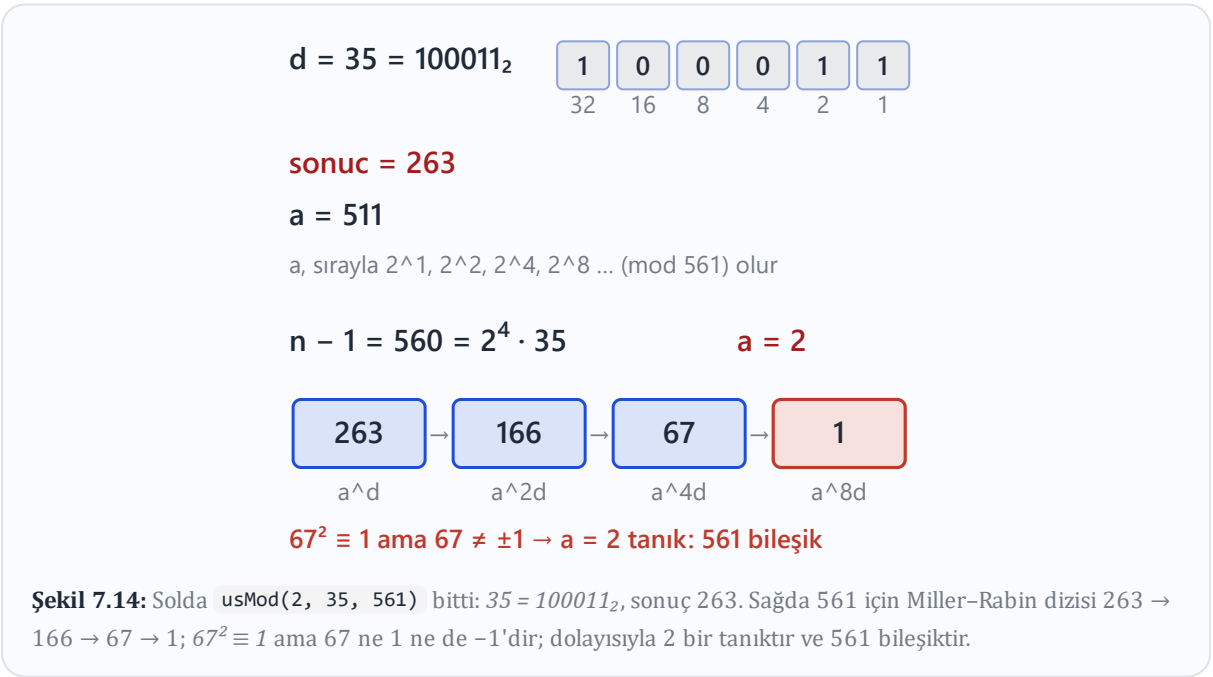
Örnek 7.11

`asalMi(561, 1)` çağrılsın ve rastgele taban $a = 2$ çıksın. 561 tektir ve 4'ten büyüktür; küçük durum denetimleri atlanır. Tablo 7.18 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 7.18: `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 7.14: 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.

7.8.6 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 7.19: 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.

7.8.7 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.

7.9 Karger minimum kesim algoritması

7.9.1 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 7.5

$G = (V, E)$ 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.

7.9.2 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.

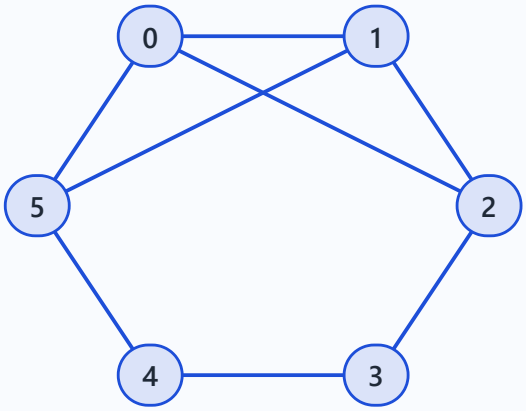
Örnek 7.12

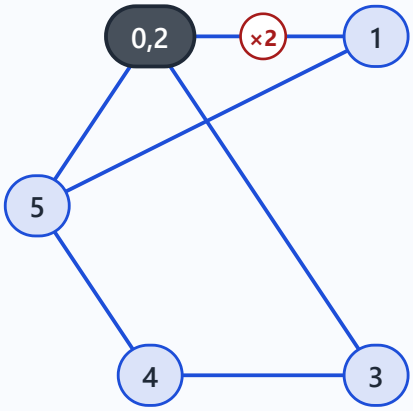
Altı düğümlü ve sekiz kenarlı bir çizge ele alınsın. Kenarlar (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (0, 5), (0, 2) ve (1, 5)'tir. Minimum kesim 2'dir. 3 ve 4 düğümleri çizgeye yalnız (2, 3) ve (4, 5) kenarlarıyla bağlıdır; {3, 4} kümesi bu iki kenarla ayrılır. Tek başına {3} ya da tek başına {4} de 2 kenarlı kesimdir. Bu çizgede üç minimum kesim vardır. Algoritma sırasıyla (0, 2), (1, 5), (3, 4) ve (0, 1) kenarlarını seçsin. Tablo 7.20 büzmeleri verir.

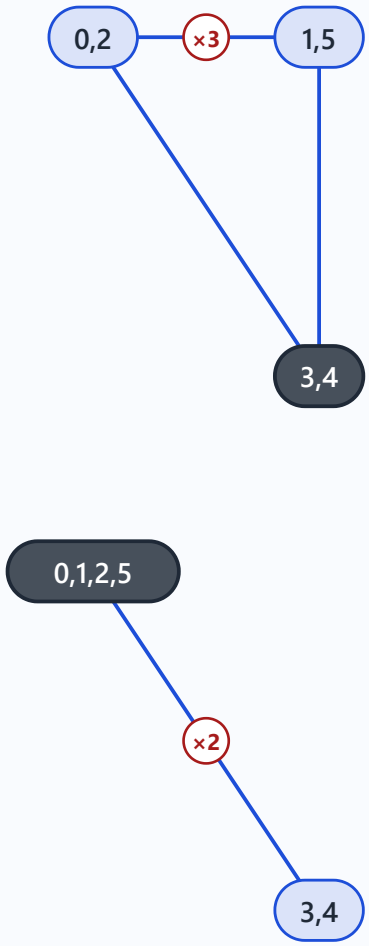
adım	büzülen kenar	süper düğümler	silinen öz-döngüler	kalan kenar
0	—	{0} {1} {2} {3} {4} {5}	—	8
1	(0, 2)	{0, 2} {1} {3} {4} {5}	(0, 2)	7
2	(1, 5)	{0, 2} {1, 5} {3} {4}	(1, 5)	6
3	(3, 4)	{0, 2} {1, 5} {3, 4}	(3, 4)	5
4	(0, 1)	{0, 1, 2, 5} {3, 4}	(0, 1), (1, 2), (0, 5)	2

Tablo 7.20: 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 7.15: 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.

7.9.3 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.

7.9.4 Başarı olasılığı

Teorem 7.4 (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 7.13

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.

7.9.5 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 7.14

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.

7.10 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.

7.11 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 7.15

$a = 5$, $c = 3$, $m = 16$ 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 7.21: 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.

7.12 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.

7.13 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 7.22: 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.

7.14 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*

7.15 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: $91 = 7 \cdot 13$ 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.

7.16 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.

7.17 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 III · BÖLÜM 8

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/advanced-algorithms-java/Bolum08/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.

8.1 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.

8.1.1 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 8.1

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.

8.1.2 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 8.1 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 8.1: 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.

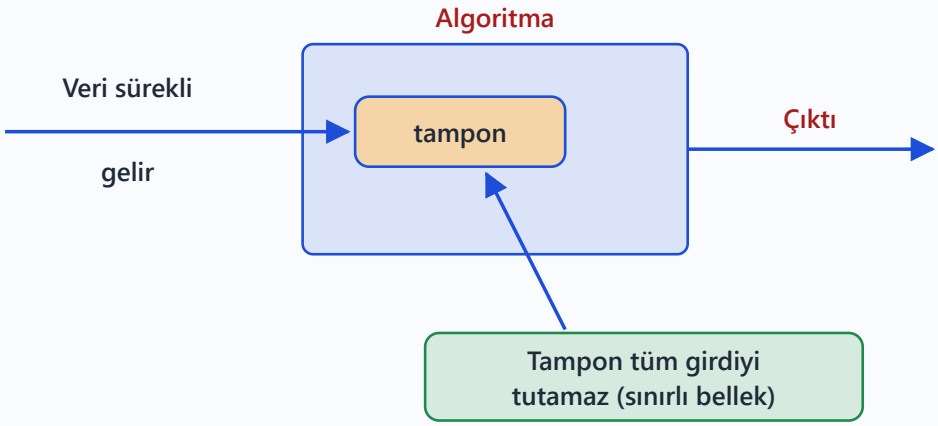
Örnek 8.1

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.

8.1.3 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 8.1 bu durumu gösterir: veri algoritmaya sürekli gelir, algoritma küçük bir tamponla (*buffer*) çalışır ve hemen çıktı üretir.



Şekil 8.1: 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.

8.1.4 Uygulama alanları

Çevrim içi problemler bilgisayar biliminin hemen her alanında karşımıza çıkar. Tablo 8.2 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 8.2: 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.

8.1.5 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.

8.1.6 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.

8.2 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.

8.2.1 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 8.2

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 8.2

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.

8.2.2 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.

8.3 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.

8.3.1 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.

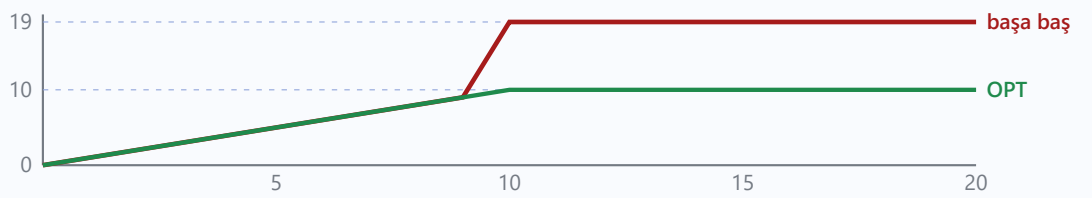
Tanım 8.3

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 8.2, $B = 10$ için OPT'nin maliyetini ve aşağıda incelenen başa baş stratejisinin maliyetini gösterir.



Şekil 8.2: *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*.

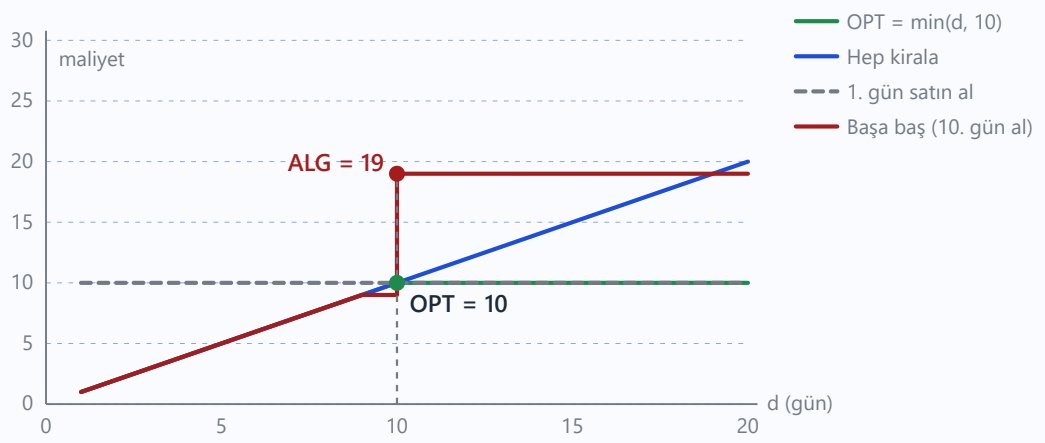
Problemin yapısı yalnız kayakla sınırlı değildir. Bir bulut sunucusu saatlik kiralanabilir ya da yıllık olarak satın alınabilir. Bir TCP bağlantısı açık tutulabilir ya da kapatılıp gerektiğinde yeniden açılabilir. Açık tutmak her saniye küçük bir maliyet doğurur, yeniden açmak ise tek seferlik büyük 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?

8.3.2 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 8.3 bu üç stratejinin maliyetini OPT ile birlikte, *B = 10* için gün sayısının fonksiyonu olarak çizer. Tablo 8.3 aynı değerleri birkaç *d* için sayılarla verir.



Şekil 8.3: *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 8.3: *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.

8.3.3 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.

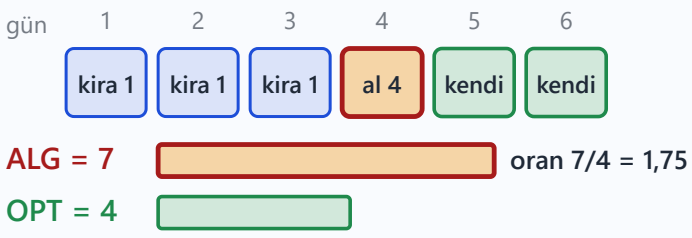
```
int basaBas(int B, int d) {  // d bilinmez
  int maliyet = 0;
  for (int gun = 1; gun <= d; gun++) {
    if (gun < B) {
      maliyet += 1;            // kirala
    } else {
      maliyet += B;            // satın al
      break;                   // ücret biter
    }
  }
  return maliyet;
}
```

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 8.4, *B = 4* ve *d = 6* için kodun izlemesini verir. Şekil 8.4 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 8.4: `basaBas(4, 6)` çağrısının izlemesi. Sezon gerçekte 6 gün sürüyor; algoritma bunu bilmiyor.



Şekil 8.4: *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.

8.3.4 Üst sınır: başa baş stratejisinin oranı

İzlemedeki oran rastlantı değildir. Başa baş stratejisinin oranı hiçbir *d* için *2 − 1/B*'yi aşmaz.

Teorem 8.1 (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.

8.3.5 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 8.2 (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 8.5, $B = 10$ 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 8.5: $B = 10$; 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 8.3

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.

8.3.6 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 8.4

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 8.6 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 8.6: 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.

8.4 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.

8.4.1 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 8.5 bu anı gösterir.



Şekil 8.5: 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 8.4

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.

8.4.2 Sayfa çıkarma politikaları

Çıkarma politikaları geçmişten farklı bilgiler kullanır. Tablo 8.7 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 8.7: 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.

8.4.3 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 8.6 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 8.8'deki kuyruk sütunundadır.



Şekil 8.6: 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 8.8: 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.

8.4.4 LRU adım adım

LRU her isabette sayfanın son kullanım zamanını günceller. Bu küçük fark, sık kullanılan sayfaların önbellekte kalmasını sağlar. Şekil 8.7 aynı dizinin LRU ile işlenişini, Tablo 8.9 ise her adımdaki kullanım sırasını gösterir.



Şekil 8.7: 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 8.9: 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.

8.4.5 OPT adım adım

OPT her hatada geleceğe bakar. Önbellekteki her sayfa için bir sonraki isteğin sırası bulunur; en uzak olan çıkarılır. Şekil 8.8 OPT'nin çerçevelerini, Tablo 8.10 ise her çıkarmada hesaplanan sonraki kullanım sıralarını verir.



Şekil 8.8: 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 8.10: 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.

8.4.6 Üç politikanın karşılaştırması

Tablo 8.11 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 8.11: 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.

8.4.7 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 8.12, 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 8.12: `lruHata(new int[]{1, 2, 3, 1, 4, 2}, 3)` çağrısının izlemesi. `onb` en eskiden en yeniye yazılmıştır.



Şekil 8.9: İzlemenin sonu: liste [1, 4, 2]. Altı istekte 5 sayfa hatası ve 1 isabet.

Metot 5 döndürür (Şekil 8.9). 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.

8.4.8 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 8.5

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 8.12 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?

8.4.9 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 8.13). 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 8.13: `fifoHata` ile 1 2 3 4 1 2 5 1 2 3 4 5, *k = 3*. Kuyruk en eskiden en yeniye yazılmıştır.

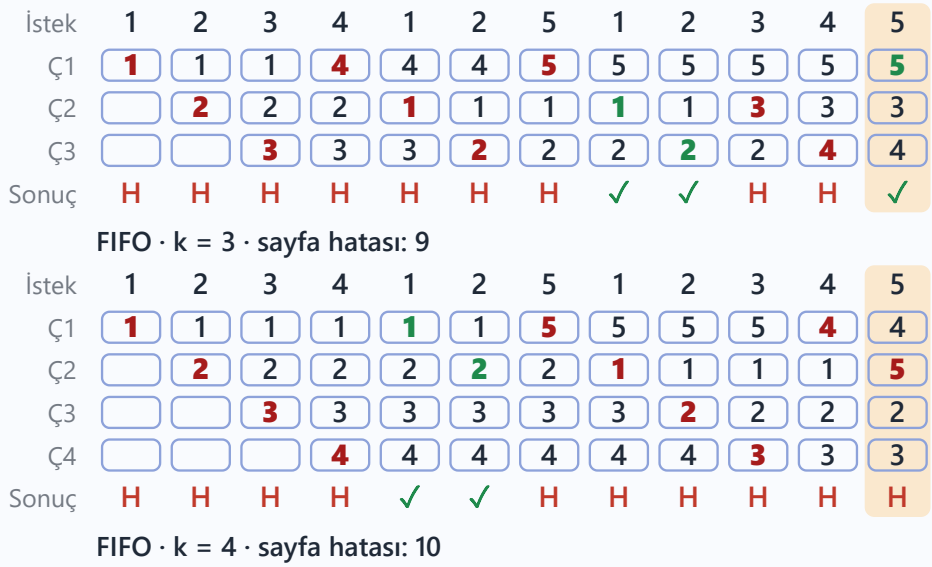


Şekil 8.10: Belady dizisinde FIFO'nun son durumu: kuyruk [5, 3, 4], 9 hata.

Metot 9 döndürür (Şekil 8.10). Aynı kod *k = 4* ile çalıştırıldığında 10 döndürür. Önbellek büyüdü, hata sayısı arttı.

8.4.10 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 8.11 aynı diziyi FIFO ile önce 3, sonra 4 çerçeveyle işler.



Şekil 8.11: 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 8.5

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 8.6

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.

8.4.11 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 8.3 (Sleator ve Tarjan, 1985)

k sayfalık önbellekle çalışan hiçbir deterministik sayfalama algoritmasının rekabet oranı k'den küçük olamaz.

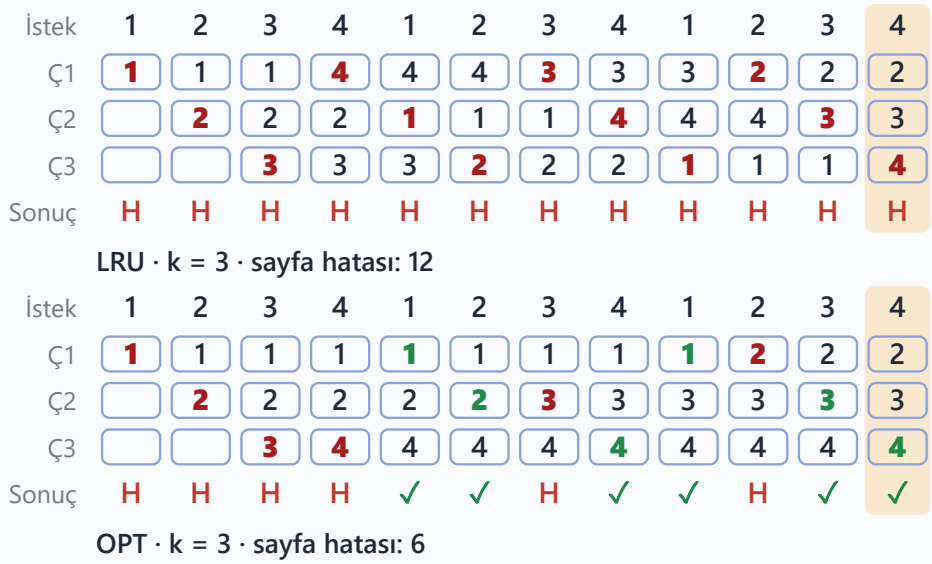
İspat. Düşman yalnız $k + 1$ farklı sayfa kullanır. Önbellek k sayfa tuttuğu için bu sayfalardan en az biri her an önbellek dışındadır. Düşman algoritmanın kodunu bildiği için hangi sayfanın dışarıda olduğunu her adımda hesaplayabilir. Her adımda tam o sayfayı ister. Böylece algoritma *her* istekte hata

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 8.12 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 8.12: 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.

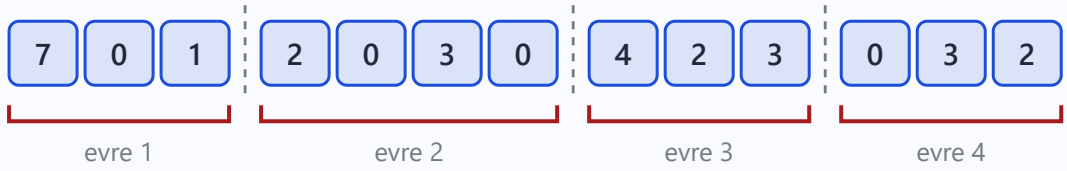
Aynı dizide FIFO da 12 hata yapar. Bu dizide FIFO ile LRU aynı davranır, çünkü hiç isabet yoktur. Dizi uzadıkça oran k'ye yaklaşır. Örneğin 5 sayfa ve $k = 4$ ile 1 2 3 4 5 dizisi üç kez tekrarlanırsa LRU 15, OPT 7 hata yapar.

8.4.12 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 8.6

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 8.13: 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 8.4 (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 8.13'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.

8.4.13 Rastgele sayfalama: işaretleme

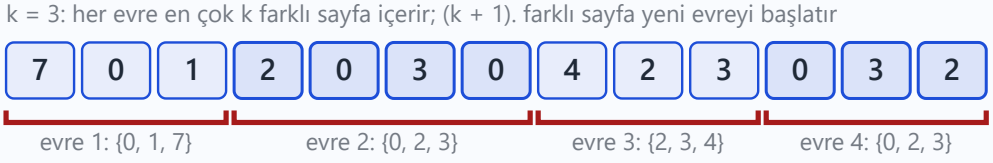
Deterministik alt sınır kanıtının anahtarı, düşmanın algoritmanın önbelleğini bilmesidir. Algoritma çıkaracağı sayfayı rastgele seçerse habersiz düşman hangi sayfanın dışarıda olduğunu bilemez. Bu fikir **işaretleme algoritmasına** (*marking algorithm*) götürür.

Tanım 8.7

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 8.14 örnek dizinin evrelerini ve her evrede istenen sayfa kümelerini gösterir.



Şekil 8.14: 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:

$$H_k = 1 + 1/2 + 1/3 + \dots + 1/k \approx \ln k$$

Aynı çalışma, hiçbir rastgele algoritmanın H_k'den iyi olamayacağını da gösterir. Tablo 8.14 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 8.14: 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.

8.4.14 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;
}
```

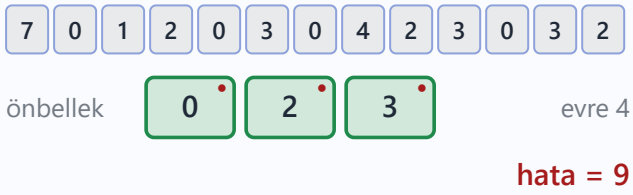
Sayfa önbellekte yoksa hata sayacı artar. Önbellek doluysa önce evre denetlenir: işaretli sayfa sayısı k'ye eşitse önbellekteki her sayfa işaretlidir. Yeni evre başlar ve `isr.clear()` bütün işaretleri siler. Ardından `aday` listesi kurulur: önbellekteki sayfalardan işaretli olanlar çıkarılır.

`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 8.15, 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 8.15: `isaretleme` ile 7 0 1 2 0 3 0 4 2 3 0 3 2, *k* = *3*.
"Adaylar" işaretsiz sayfalardır; "•" işaretli sayfayı gösterir.



Şekil 8.15: 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 8.15). 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.

8.5 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.

8.5.1 Problem ve üç kural

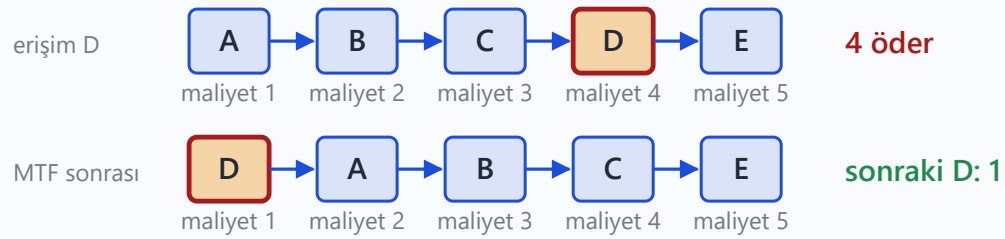
n öğeli bir bağlı liste verilsin. Bir öğeye erişmek için liste baştan taranır. i. konumdaki öğeye erişmenin maliyeti i'dir. Erişilen öğe, erişimden sonra ücretsiz olarak listenin önüne doğru taşınabilir. İki komşu öğenin başka bir biçimde yer değiştirmesi ise 1 birim tutar; bu tür takaslara *ücretli takas* denir.

Tanım 8.8

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 8.16: 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.

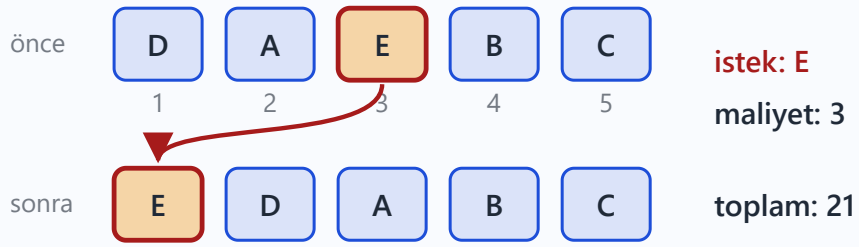
8.5.2 Öne taşı adım adım

Başlangıç listesi A B C D E, istekler E D E A D E olsun. Tablo 8.16 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 8.16: Öne taşı ile A B C D E listesinde E D E A D E istekleri.



Şekil 8.17: Son istek: E 3. konumda, maliyet 3. Liste E D A B C olur; toplam 21.

Aynı dizide öteki kurallar daha çok öder (Tablo 8.17). 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 8.17: 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 8.7

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.

8.5.3 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 8.5 (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: $\text{gerçek} + \Delta\Phi$. 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 8.18, 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 8.18: 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.

8.5.4 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}, ... 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.

8.6 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.

8.6.1 Veri akışı modeli

Tanım 8.9

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 8.19, 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 8.19: 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.

8.6.2 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;

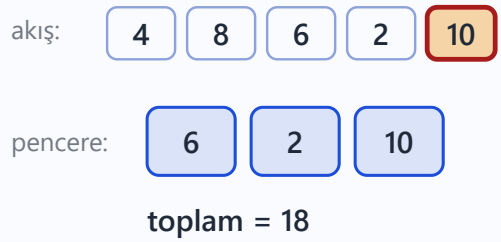
  HareketliOrtalama(int k) { this.k = k; }

  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 8.20, *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 8.20: `new HareketliOrtalama(3)` ile 4, 8, 6, 2, 10 akışı.



Şekil 8.18: Son adım: pencere [6, 2, 10], toplam 18, ortalama 6.

Sık yapılan hata

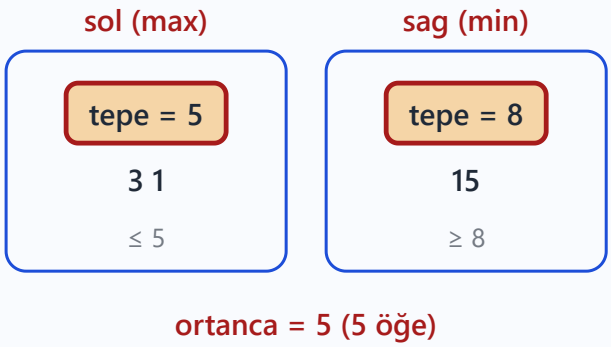
Çıkarma ile eklemenin sırasını karıştırmak. Önce eklenip sonra `pencere.size() > k` denetlenirse kod yine doğru çalışır. Ama önce eklenip sonra `size() == k` denetlenirse pencere hiç *k*'ye ulaşmadan küçültülür. Bu durumda yalnız son *k − 1* öğenin ortalaması alınır. Bir başka hata, `toplam`'ı `double` tutmaktır. Uzun akışlarda ekleme ve çıkarma yuvarlama hatası biriktirir; tam sayı verilerde toplam tam sayı tutulmalıdır.

8.6.3 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 8.19: 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 8.21, 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 8.21: Çevrim içi ortanca ile 5, 15, 1, 3, 8, 7. Heap içerikleri tepe önce yazılmıştır.



Şekil 8.20: 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 8.8

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.

8.6.4 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 8.10

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 8.6 (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 8.21: *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*.

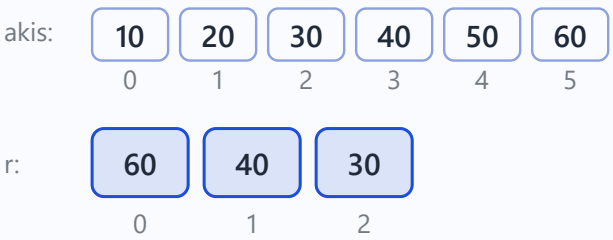
```
List<Integer> rezervuar(int[] akis, int k) {
  List<Integer> r = new ArrayList<>();
  for (int i = 0; i < k; i++) r.add(akis[i]);
  for (int i = k; i < akis.length; i++) {
    int j = rastgele.nextInt(i + 1); // 0..i
    if (j < k)
      r.set(j, akis[i]);
  }
  return r;
}
```

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 8.22 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 8.22: `rezervuar` ile 10, 20, 30, 40, 50, 60 ve *k = 3*.



Şekil 8.22: 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 8.9

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.

8.6.5 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:

$$h_{yeni} = (h_{eski} - c_{çıkan} \cdot B^{m-1}) \cdot B + c_{giren}$$

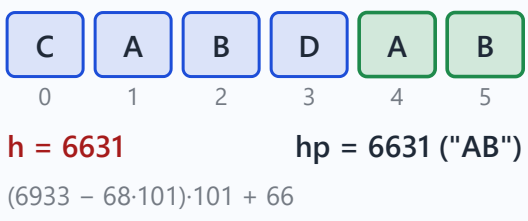
```
void ara(String akis, String oruntu) {
  int m = oruntu.length(), B = 101, h = 0, hp = 0;
  int gucu = 1;                          // B^(m-1)
  for (int i = 0; i < m - 1; i++) gucu *= B;
  for (int i = 0; i < m; i++)
    hp = hp * B + oruntu.charAt(i);
  for (int t = 0; t < akis.length(); t++) {
    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 8.23, "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 8.23: `ara("CABDAB", "AB")` çağrısında kayan hash.



Şekil 8.23: 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.

8.7 Artımlı öğrenme

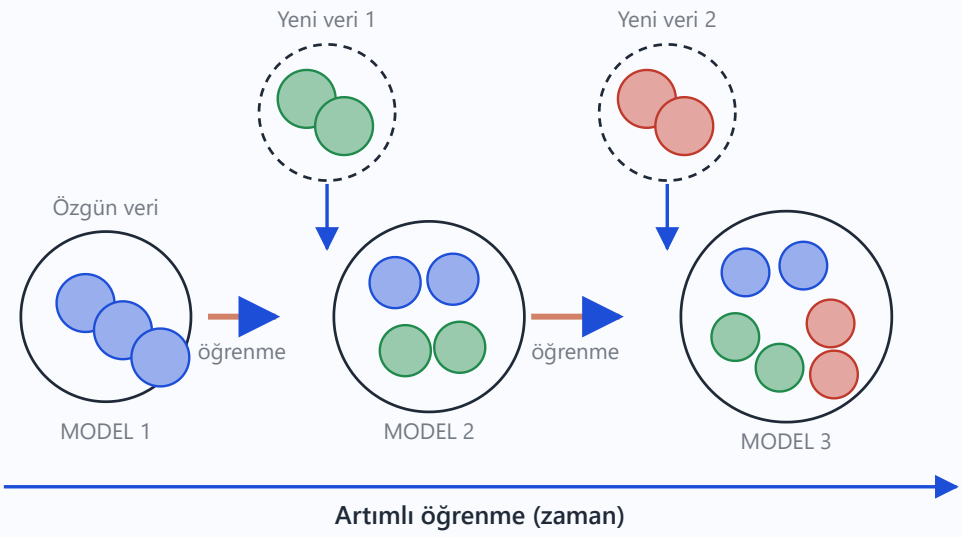
Makine öğrenmesinde model çoğunlukla bütün veriyle bir kez eğitilir. Veri sürekli geliyorsa bu yaklaşım pahalıdır. Her yeni veri parçasında modeli baştan eğitmek, o ana kadar gelen bütün veriyi yeniden işlemek demektir. Çevrim içi bakış burada da işe yarar: model yalnız yeni veriyle güncellenir.

8.7.1 Modeli yeni veriyle güncellemek

Tanım 8.11

Artımlı öğrenme (*incremental learning*), modelin her yeni veriyle, geçmiş veriyi yeniden işlemeden güncellenmesidir. Güncellenmiş model hem eski hem yeni verideki bilgiyi temsil eder.

Şekil 8.24 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 8.24: 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.

8.7.2 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 8.10

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.

8.7.3 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.

8.8 Problemler ve sonuçlar

Tablo 8.24 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 8.24: 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.

8.9 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.

8.10 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.

8.11 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*

8.12 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 $1/5 \cdot 5/6 \cdot 6/7 \cdot 7/8 = 1/8 = k/n$ 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.

8.13 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.

8.14 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 III · BÖLÜM 9

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/advanced-algorithms-java/Bolum09/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.

9.1 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.

9.1.1 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.

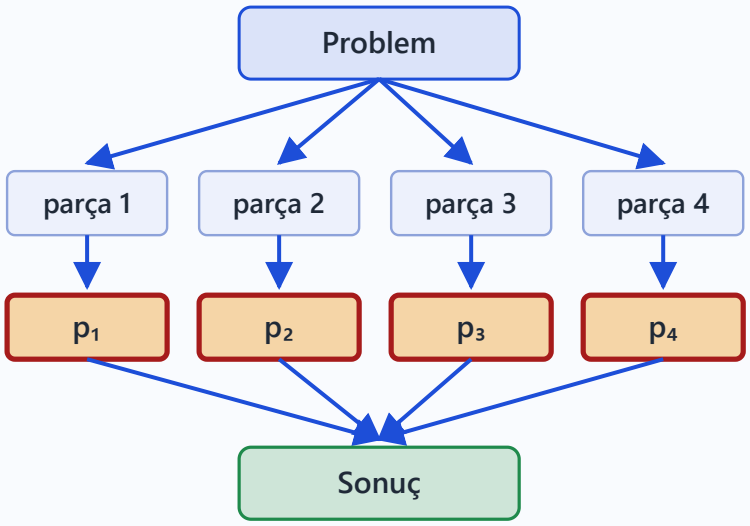
Tanım 9.1 (Paralel algoritma)

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.

9.1.2 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 9.1 bu akışı dört işlemci için gösterir.



Şekil 9.1: 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.

9.1.3 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.

9.1.4 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.

9.2 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.

9.2.1 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 9.1 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 9.1: 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.

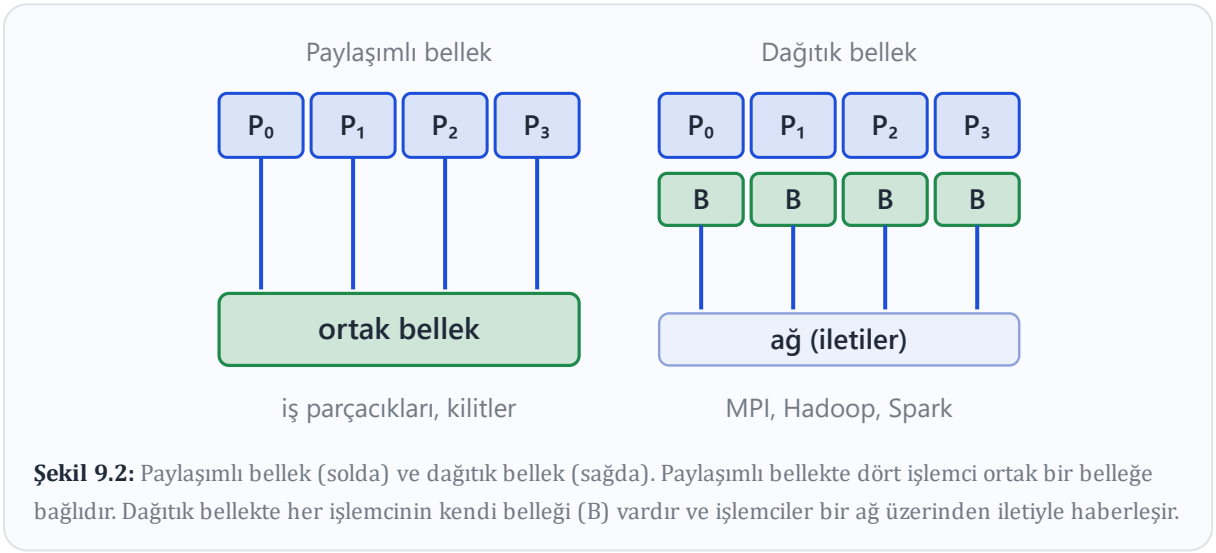
MISD uygulamada nadirdir. Aynı veri farklı komutlarla işlenir. Uçak denetim sistemleri gibi hata toleranslı sistemlerde aynı girdi birkaç ayrı birimde bağımsız olarak hesaplanır ve sonuçlar karşılaştırılır. Bu düzen MISD sınıfına örnek gösterilir.

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.

9.2.2 Bellek mimarileri

MIMD makineler belleği düzenleme biçimlerine göre ikiye ayrılır. Şekil 9.2 iki düzeni yan yana gösterir.



Şekil 9.2: 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.

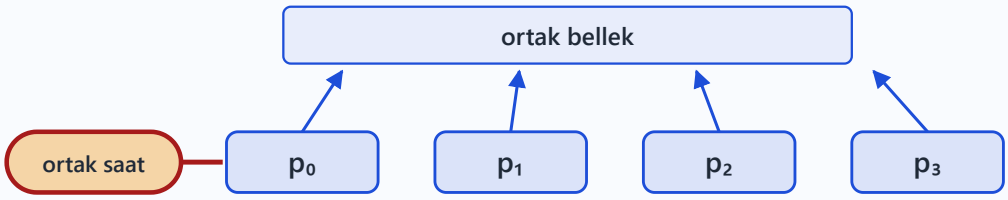
Bu notta Java örnekleri paylaşımlı bellekte çalışır. Boru hattı ve boru hatlı ağaç örnekleri ise işlemcilerin yalnız komşularıyla ileti alışverişi yaptığı dağıtık bir düzeni varsayar. MapReduce de aslen dağıtık bellek için tasarlanmıştır; buradaki Java örneği aynı fikri tek makinede gösterir.

9.2.3 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 9.2 (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 9.3: 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 9.2 üç yaygın türü gösterir.

Tür	Okuma	Yazma
EREW	dışlayıcı	dışlayıcı
CREW	eşzamanlı	dışlayıcı
CRCW	eşzamanlı	eşzamanlı

Tablo 9.2: 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 9.1

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.

9.3 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.

9.3.1 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 9.3 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*)	$C = p \cdot T_p$

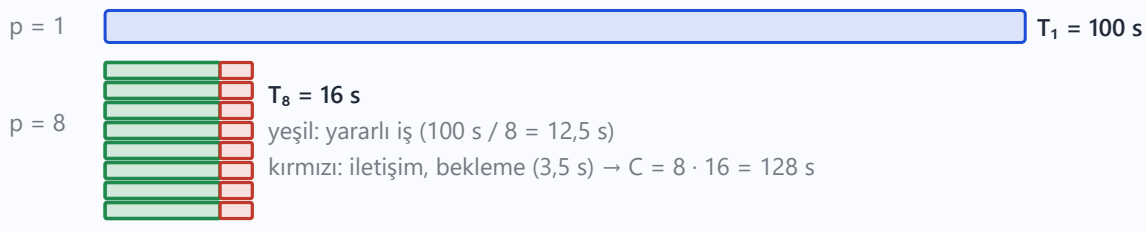
Tablo 9.3: Paralel başarım ölçütleri.

Tanım 9.3 (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 9.4: 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 9.2

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 9.4 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.

9.3.2 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.

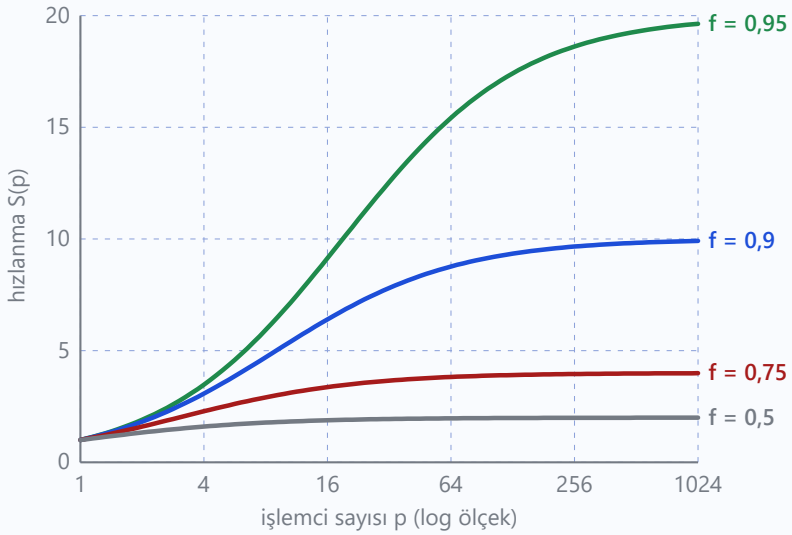
Teorem 9.1 (Amdahl yasası)

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 9.5: 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 9.3

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 9.5'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

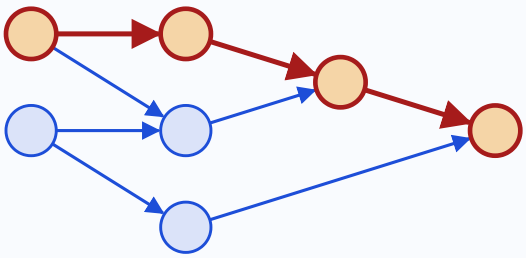
Amdahl hesabında sıralı kısmı unutmak. $f = 0{,}9$ ve $p = 8$ için “paralel kısım 8 kat hızlanır, öyleyse program da yaklaşık 8 kat hızlanır” denemez. Doğru hesapta sıralı %10 olduğu gibi kalır ve hızlanma 4,71'dir. Programın %10'u sıralıysa en iyi durumda bile hızlanma 10 ile sınırlıdır.

9.3.3 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 9.4 (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 9.6: Yedi işlemli bir bağımlılık çizgesi. En uzun zincir dört düğümdür (kırmızı yol). $W = 7$, $D = 4$, paralellik $W / D = 1{,}75$.

Şekil 9.6'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 9.2 (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 9.4

$n = 1024$ 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.

9.4 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.

9.4.1 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.

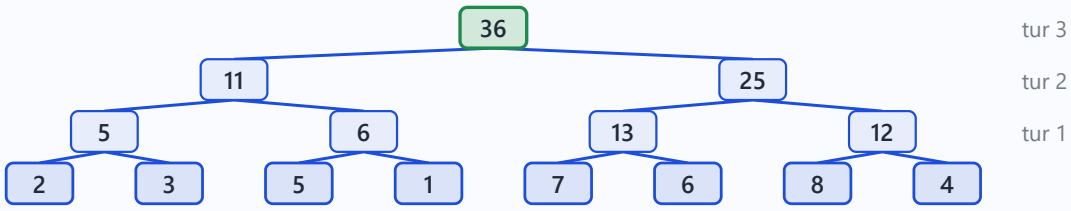
Zinciri kıran şey toplamanın **birleşmeli** (*associative*) olmasıdır: $(a + b) + c = a + (b + c)$. Birleşmeli bir işlemde parantezler istenildiği gibi yerleştirilebilir. Sekiz sayının toplamı sıralı döngüde

> *((((((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 9.7 bu ağacı gösterir.



Şekil 9.7: 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.

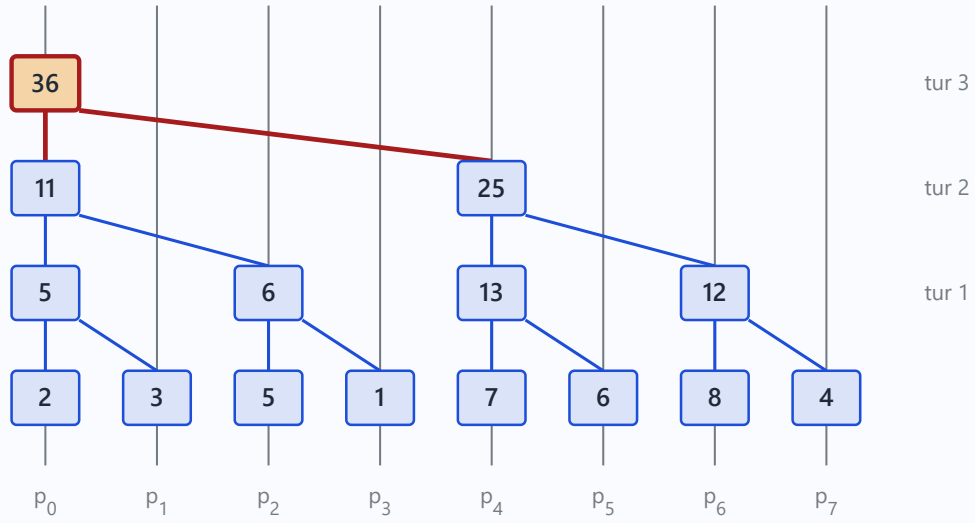
> **Sık yapılan hata**
>
> Birleşmeli olmayan bir işlemi ağaçla indirgemek. Çıkarma birleşmeli değildir: *(8 − 4) − 2 = 2* ama *8 − (4 − 2) = 6*. Ağaç sırası, sıralı sıradan farklı parantezleme kullandığı için sonuç değişir. Gerçel sayılarda toplama birleşmeli olsa da kayan noktalı toplama bilgisayarda yuvarlama yüzünden tam birleşmeli değildir. Paralel toplam sıralı toplamdan son basamaklarda farklı çıkabilir.

9.4.2 Ağaç toplamı adım adım

Örnek dizi 2, 3, 5, 1, 7, 6, 8, 4'tür; toplamı 36'dır. Sekiz işlemci $p_0 \dots p_7$ 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 9.8: 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.

9.4.3 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
```

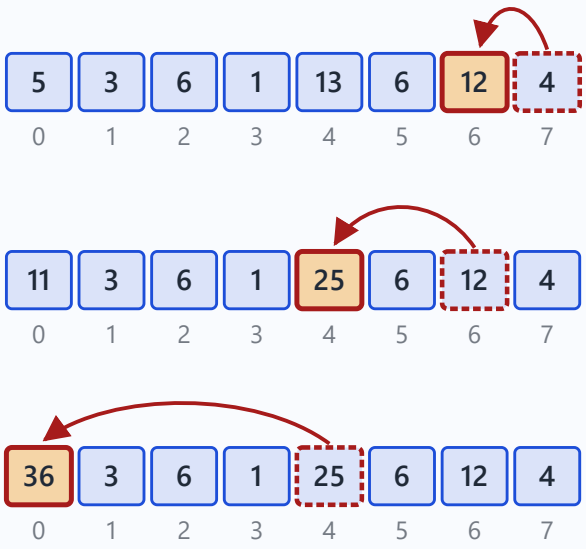
Kodun parçaları şöyle çalışır:

- `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 9.4 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 9.4: `a = {2, 3, 5, 1, 7, 6, 8, 4}` için ağaç toplamı izlemesi.



Şekil 9.9: 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 9.5

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.

9.4.4 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 9.5 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 9.5: 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.

9.5 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.

9.5.1 Problem

Tanım 9.5 (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.

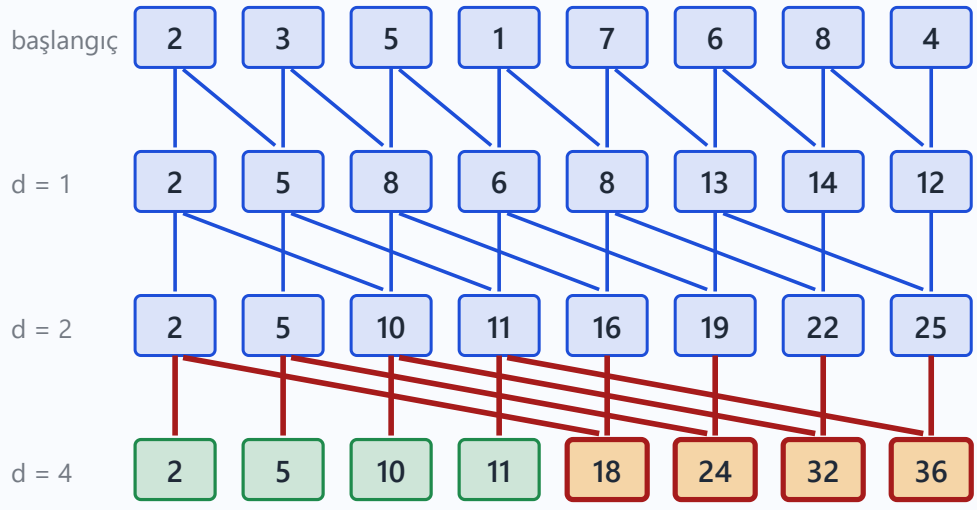
9.5.2 Hillis–Steele algoritması

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 9.10: 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 9.6 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 9.6: 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.

9.5.3 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 9.7 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 9.7: 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 9.11: 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 9.6

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*.

9.5.4 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 9.8: 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.

9.6 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.

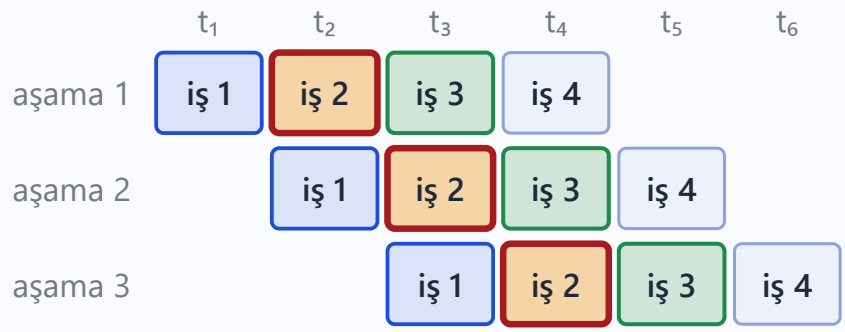
9.6.1 Boru hattı fikri

Tanım 9.6 (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 9.12: 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 9.12 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.

9.6.2 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.

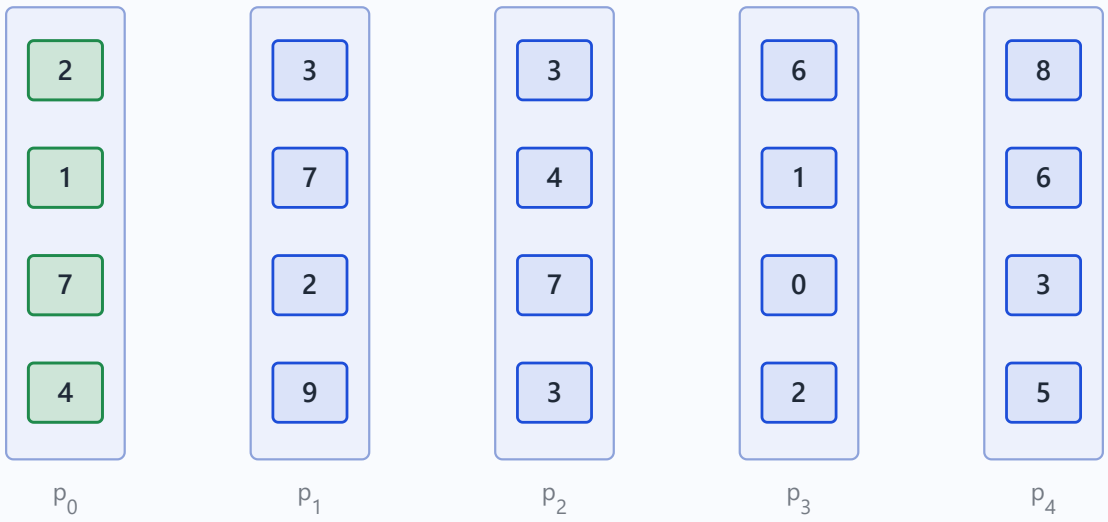
Amaç şudur: her satır r için p_j'deki değer, p_0 … p_j'nin o satırdaki değerlerinin toplamı olsun. Bu, her satırda ayrı bir önek toplamıdır. Sonunda p_4 her satırın toplamını tutar. Bir adımda her işlemci sağ komşusuna bir değer gönderebilir; alıcı onu kendi değerine ekler. p_0'ın değerleri zaten sonuçtur, 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.

9.6.3 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 9.9'da verilmiştir.

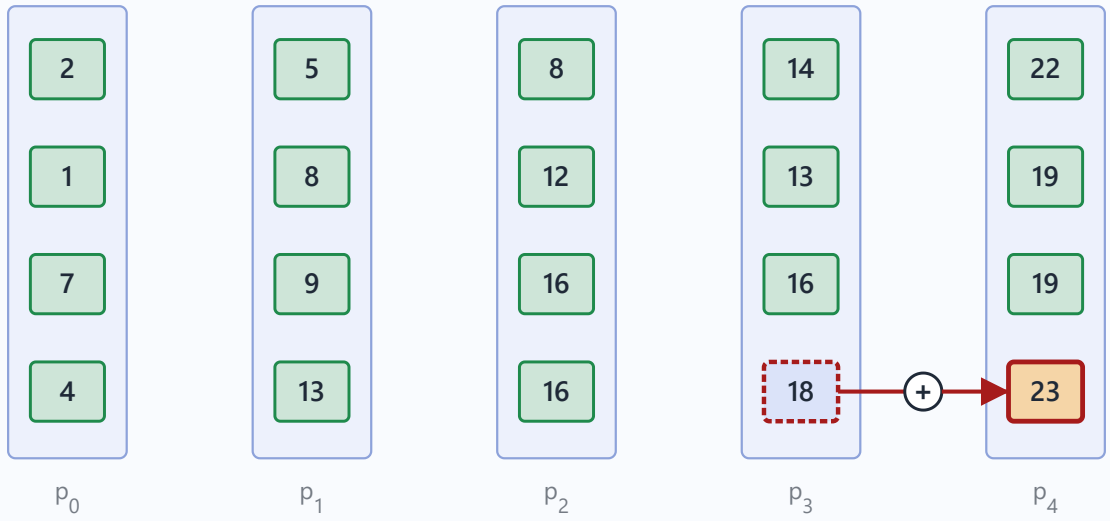


Şekil 9.13: Boru hattının başlangıcı. Beş işlemcinin her biri 4 satırlık bir sütun tutar. p_0'ın değerleri zaten sonuçtur.

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 9.9: 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 9.14: 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.

9.6.4 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 9.10 kodun her adımda yaptığı toplamaları ve p_4'ün sütununu verir. Değerler kodun çalıştırılmasıyla doğrulanmıştır; Tablo 9.9 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 9.10: 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.

9.7 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.

9.7.1 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.

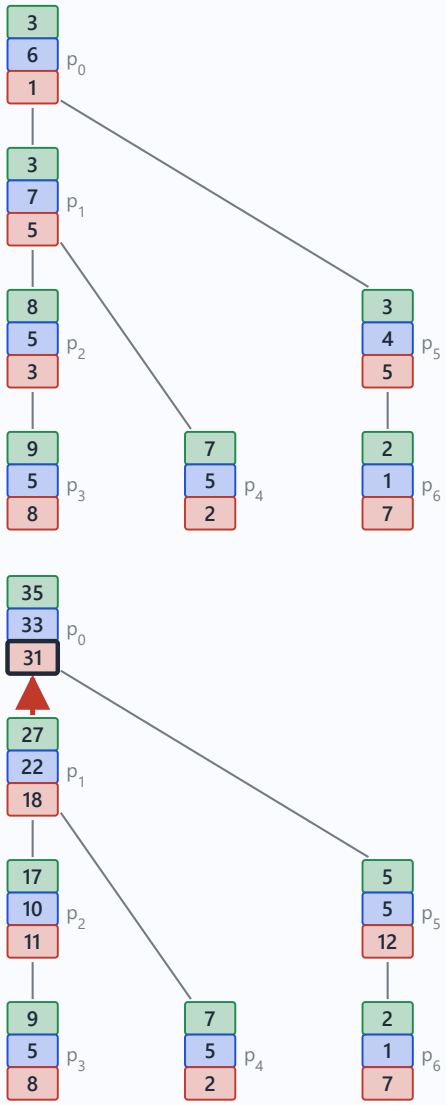
Saf yöntem bileşenleri sırayla indirger. Önce yeşil bileşen köke çıkarılır, sonra mavi, sonra kırmızı. Her bileşen $h = 3$ turda köke çıkar; saf yöntem toplam $k \cdot h = 9$ tur ister. **Boru hatlı ağaç** (*pipelined tree*) yönteminde bir bileşen yukarı çıkarken bir sonraki bileşen yapraklardan yola çıkar.

9.7.2 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 9.11 turları verir.



Şekil 9.15: 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 9.11: 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 $p_0, p_1, \ldots, p_6$'dır.

9.7.3 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.

9.8 Paralel algoritma türleri ve Java

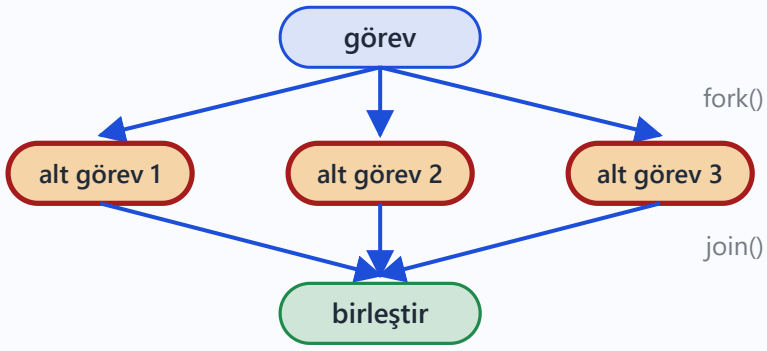
Önceki kesimlerdeki algoritmalar PRAM ya da ileti alışverişi yapan işlemciler gibi soyut modellerde çalıştı. Bu kesim aynı fikirleri gerçek bir dilde, Java'da kurar. Java programları paylaşımlı bellekli bir makinede çalışır. İş parçacıkları ortak nesnelere erişebilir. İş parçacıklarını doğrudan yönetmek yerine çoğunlukla görev tabanlı araçlar kullanılır.

9.8.1 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 9.16 yapının fikrini gösterir. Bir görev alt görevlere **çatallanır** (*fork*). Alt görevler paralel çalışır. Ana görev alt görevlerin bitmesini **bekler** (*join*) ve sonuçlarını birleştirir.



Şekil 9.16: Fork/Join yapısı. Bir görev üç alt göreve çatallanır (fork), alt görevler paralel çalışır, sonuçlar join ile birleştirilir.

9.8.2 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.

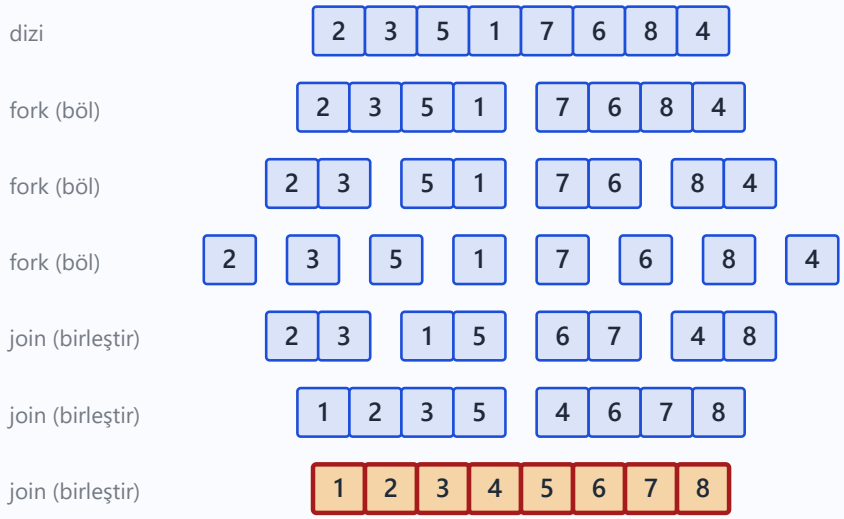
`ForkJoinPool`, bir iş parçacığı havuzudur. Havuzdaki her iş parçacığının kendi görev kuyruğu vardır. Bir iş parçacığı `fork()` ettiği görevi kendi kuyruğuna koyar. Kuyruğu boşalan iş parçacığı boş beklemez; başka bir iş parçacığının kuyruğunun öbür ucundan bir görev alır. Bu düzene **iş çalma** (*work stealing*) denir. İş çalma, yük dengelemeyi kendiliğinden sağlar: hangi görevin ne kadar süreceği önceden bilinmese de boşta kalan çekirdek iş bulur.

Tablo 9.12 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 9.12: Paralel birleştirme sıralamasında görev düzeyleri.



Şekil 9.17: 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.

9.8.3 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 9.13: 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.

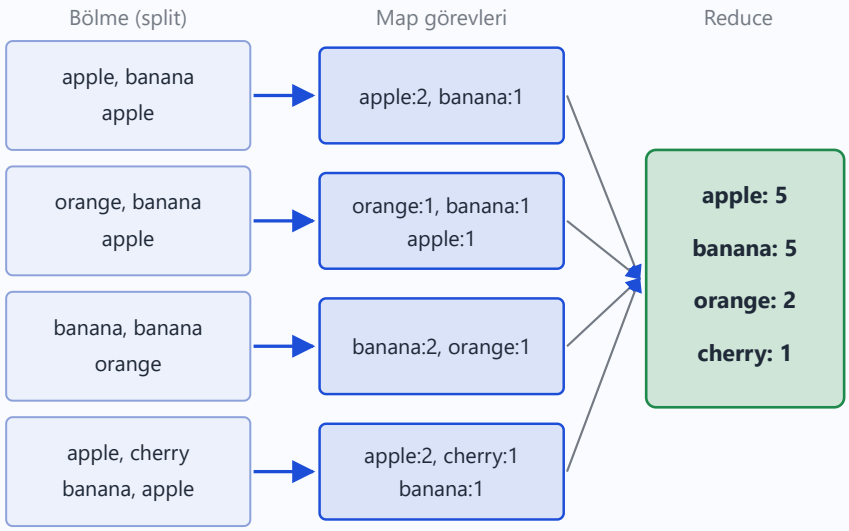
Darboğazı kaldırmak için birleştirmenin kendisi de paralelleştirilir. Fikir şöyledir: büyük yarının ortanca öğesi seçilir, öteki yarıda bu öğenin yeri ikili aramayla bulunur. Böylece iki yarı ikişer parçaya ayrılır ve parçalar birbirinden bağımsız olarak, yine paralel birleştirilir. Bu sürümün derinliği $\Theta(\log^3 n)$'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.

9.8.4 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 9.18 akışı gösterir.



Şekil 9.18: 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 9.14 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 9.14: 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.

9.8.5 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.

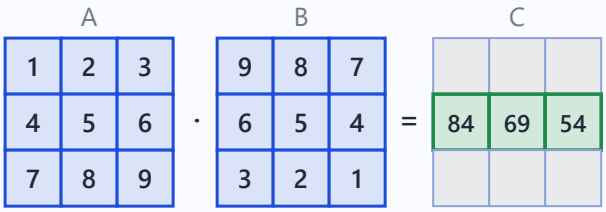
```
// her satır ayrı bir görev
class SatirGorevi extends RecursiveAction {
  protected void compute() {
    for (int j = 0; j < n; j++) {
      C[satir][j] = 0;
      for (int k = 0; k < n; k++)
        C[satir][j] += A[satir][k] * B[k][j];
    }
  }
}
try (var havuz = new ForkJoinPool()) {
  for (int 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 9.15, 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 9.15: 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 9.19: 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.

9.9 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.

9.9.1 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 9.16 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 9.16: 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 9.7 (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 9.7

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.

9.9.2 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.

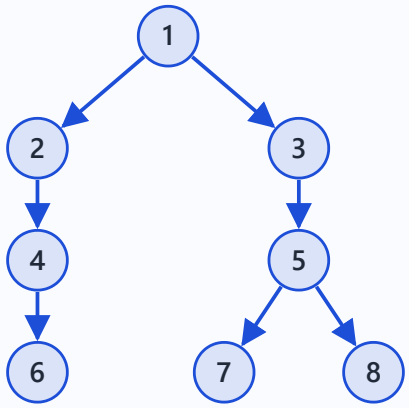
Paylaşılan durumdan kaçınmak. En iyi çözüm, paylaşılan değişkeni hiç kullanmamaktır. Her görev kendi kısmi sonucunu üretir; sonuçlar görevler bittikten sonra birleştirilir. Ağaç toplamı, MapReduce ve satır görevleri bu ilkeyle çalışır. Sayaç örneğinde her iş parçacığı kendi yerel sayacını artırır ve `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.

9.9.3 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 9.20'deki çizgede 1'den başlayan gezinme 2 ve 3 için iki görev açar; bu görevler de kendi komşuları için yeni görevler açar.



Şekil 9.20: 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 9.17: 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.

9.9.4 Java'da paralellik araçları

Java, farklı düzeylerde paralellik araçları sunar. Tablo 9.18 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 9.18: 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.

9.10 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.

9.11 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.

9.12 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*

9.13 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.

9.14 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.

9.15 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 2
Çakışmayan en çok etkinliği seçme; en erken biten önce.

adaptive adversary → uyarlanır düşman 8
Algoritmanın rastgele kararlarını görüp sonraki isteği buna göre seçen düşman; buna karşı rastgeleliğin faydası büyük ölçüde kaybolur.

adversary → düşman 8
Algoritmayı en kötü duruma zorlayan girdiyi seçen hayalî rakip.

Amdahl's law → Amdahl yasası 9
Sıralı kalan kısmın, elde edilebilecek en büyük hızlanmayı sınırlaması.

amplification → tekrarla güçlendirme 7
Rastgele bir algoritmayı bağımsız olarak tekrarlayıp hata olasılığını küçültme tekniği.

ant colony optimization → karınca kolonisi optimizasyonu 6
Karıncalardan esinlenen yöntem: kısa tur kuran yapay karıncalar kenarlara daha çok feromon bırakır, feromon zamanla buharlaşır.

approximation algorithm → yaklaşım algoritması 6
NP-zor problemler için polinom sürede, optimuma kanıtlı bir oranla yakın çözüm üreten algoritma.

approximation ratio → yaklaşım oranı 6
Algoritmanın çözümü ile optimum arasındaki en kötü oran (ρ).

approximation scheme (PTAS / FPTAS) → yaklaşım şeması 6
Her ε > 0 için (1 + ε) oranlı çözüm veren algoritma ailesi.

assignment problem → atama problemi 5
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)) 4
Kısmi çözümü adım adım genişleten, çıkmazda son seçimi geri alıp başka seçeneği deneyen arama.

barrier → bariyer 9
Bütün işlemciler ona ulaşmadan hiçbirinin geçemediği eşgüdüm noktası.

Belady's anomaly → Belady anomalisi 8
FIFO'da çerçeve sayısı artınca sayfa hatasının artabilmesi.

best-first search → en iyi önce arama 5
Sınırı en umut verici olan düğümü önce açan arama stratejisi.

bin packing → kutulara yerleştirme 2
Boyutları verilen nesneleri sabit kapasiteli en az sayıda kutuya yerleştirme problemi; NP-zordur.

bound → sınır 5
Bir daldan elde edilebilecek en iyi değerin iyimser tahmini.

branch and bound → dal sınır 5
Durum uzayını dallandırıp her dal için sınır hesaplayan, sınırı bilinen en iyi çözümden kötü olan dalları budayan yöntem.

branching → dallanma 5
Bir alt problemi daha küçük alt problemlere ayırma.

C

cache → önbellek 8
Sık kullanılan verileri tutan küçük, hızlı bellek.

canonical coin system → kanonik para sistemi 2
Açgözlü para üstü algoritmasının her tutar için en az madeni parayı verdiği para sistemi.

Carmichael number → Carmichael sayısı 7
Kendisiyle aralarında asal olan her taban için Fermat testini geçen bileşik sayı.

chromatic number → renk sayısı 4
Bir çizgeyi, kenarla bağlı iki düğüm aynı rengi almayacak biçimde boyamaya yeten en küçük renk sayısı.

closest pair of points → en yakın nokta çifti 1
Düzlemdeki noktalar arasında birbirine en yakın iki noktayı bulma.

coin change problem → para üstü problemi 2
Bir tutarı en az sayıda madeni parayla oluşturma.

competitive analysis → rekabetçi analiz 8
Çevrim içi algoritmayı en iyi çevrim dışı algoritmayla (OPT) karşılaştırma.

competitive ratio → rekabet oranı 8
Her girdi için ALG ≤ c · OPT (+ sabit) sağlayan en küçük c.

compulsory miss (cold miss) → zorunlu hata (soğuk hata) 8
Önbellek başlangıçta boş olduğu için ilk farklı sayfalarda hiçbir politikanın önleyemediği sayfa hatası.

concept drift → kavram kayması 8
Verinin dağılımının zamanla değişmesi; artımlı öğrenmede eski veri bu durumda yanıltıcı olur.

constraint satisfaction problem → kısıt sağlama problemi 4
Değişkenlere tüm kısıtları sağlayan değerler atama problemi.

convergence → yakınsama (sayısal yakınsama; approximation algorithm ise yaklaşım algoritması) 6
Yinelemelerin gerçek değere giderek yaklaşması.

convex hull → dışbükey örtü 1
Nokta kümesini içeren en küçük dışbükey çokgen.

D

data stream → veri akışı 8
Tek geçişte, az bellekle işlenmesi gereken sürekli gelen veri.

dead node → ölü düğüm 5
Genişletilmiş, olursuz bulunmuş ya da sınırı yetersiz olduğu için budanmış düğüm; bir daha ele alınmaz.

decision problem → karar problemi 4
Cevabı evet ya da hayır olan problem.

depth-first branch and bound → derinlik öncelikli dal sınır 5
Canlı düğümleri yığınla (derinlik öncelikli) gezen, belleği derinlikle sınırlı dal sınır.

distributed memory → dağıtık bellek 9
Her düğümün kendi belleği olduğu, verinin ağ üzerinden iletiyle taşındığı düzen; iyi ölçeklenir ama iletişim pahalıdır.

divide and conquer → böl ve fethet 1
Problemi küçük alt problemlere bölüp özyinelemeli çözen ve sonuçları birleştiren yaklaşım.

dynamic programming (DP) → dinamik programlama 3
Örtüşen alt problemleri bir kez çözüp sonuçlarını saklayarak kullanan tasarım yöntemi.

E

E-node (expanding node) → E-düğüm 5
O anda çocukları, yani alt problemleri üretilen düğüm.

edge contraction → kenar büzme 7
Bir kenarın iki ucunu tek düğümde birleştirme işlemi (Karger algoritması).

efficiency → verimlilik 9
İşlemci başına hızlanma: hızlanma / p.

entropy → entropi 2
Bir kaynağın sembol başına ortalama bilgi miktarı; önek-özgür kodun ortalama uzunluğu için alt sınır.

eviction policy → çıkarma politikası 8
Önbellek dolunca hangi sayfanın çıkarılacağını belirleyen kural (FIFO, LRU, OPT).

exchange argument → değiştirme argümanı 2
Optimal çözümü açgözlü seçimi içerecek biçimde değiştirerek doğruluğu kanıtlama.

expected running time → beklenen çalışma süresi 7
Rastgele algoritmanın verilen bir girdi için çalışma süresinin beklenen değeri.

expected value → beklenen değer 7
Bir rastgele değişkenin olasılıklarla ağırlıklı ortalaması.

F

feasible → olurlu 5
Tüm kısıtları sağlayan (çözüm).

Fisher–Yates shuffle → Fisher–Yates karıştırma 7
Her permütasyonu eşit olasılıkla üreten O(n) karıştırma.

forward checking → ileri denetim 4
Bir değişkene değer atanınca bağlı değişkenlerin olası değerlerinden çelişenleri hemen çıkarma.

fractional knapsack → kesirli sırt çantası 2
Eşyaların kesirli alınabildiği, değer/ağırlık oranına göre açgözlü çözülen problem.

G

genetic algorithm → genetik algoritma 6
Bir çözüm popülasyonuyla çalışan yöntem: iyi çözümler daha sık seçilir, çaprazlamayla birleştirilir, mutasyon çeşitliliği korur.

geometric distribution → geometrik dağılım 7
Başarı olasılığı p olan denemelerde ilk başarıya kadar geçen deneme sayısının dağılımı.

global optimum → genel optimum 6
Bütün çözümlerin en iyisi; yerel arama bir yerel optimumda durur ve buna ulaşamayabilir.

graph coloring → çizge boyama 4
Komşu düğümlere farklı renk verecek biçimde düğümleri boyama.

greedy algorithm → açgözlü algoritma 2
Her adımda o an en iyi görünen seçimi yapıp geri dönmeyen algoritma.

greedy-choice property → açgözlü seçim özelliği 2
Yerel en iyi seçimin bir optimal çözüme götürebilmesi.

H

Hamiltonian cycle → Hamilton çevrimi 6
Her düğümden tam bir kez geçip başlangıca dönen çevrim.

Hamiltonian path → Hamilton yolu 4
Her düğümden tam bir kez geçen yol.

hit (cache hit) → isabet 8
İstenen sayfanın önbellekte bulunması; maliyeti 0'dır. Bulunmazsa sayfa hatası oluşur.

Huffman coding → Huffman kodlama 2
Sık karakterlere kısa kod veren, önek içermeyen en iyi kodlama.

Hungarian method → Macar yöntemi 5
Atama problemini $O(n^3)$ zamanda çözen algoritma.

I

incremental learning → artımlı öğrenme 8
Modeli her yeni veriyle, geçmişi yeniden işlemeden güncelleme.

incumbent → enİyi 5
Dal sınırda şimdiye kadar bulunan en iyi tam çözümün değeri.

indicator random variable → gösterge değişkeni 7
Bir olay gerçekleşince 1, gerçekleşmeyince 0 değerini alan rastgele değişken.

integer linear programming (ILP) → tam sayılı doğrusal programlama 5
Değişkenlerin tam sayı olması gereken doğrusal programlama.

J

job sequencing with deadlines → son tarihli iş sıralama 2
Son tarihleri olan işlerden kârı en büyük çizelgeyi seçme.

K

Karatsuba multiplication → Karatsuba çarpımı 1
Büyük sayıları üç alt çarpımla O(n^1,585) sürede çarpan yöntem.

#

0/1 knapsack → 0/1 sırt çantası 3
Her eşyanın ya bütünüyle alındığı ya hiç alınmadığı, kapasiteyi aşmadan değeri en büyükleyen problem.

L

Las Vegas algorithm → Las Vegas algoritması 7
Her zaman doğru yanıt veren, süresi rastgele olan algoritma.

least recently used (LRU) → LRU (en uzun süredir kullanılmayan) 8
Önbellek dolunca en uzun süredir kullanılmayan sayfayı çıkaran politika; rekabet oranı k'dir.

least-cost (LC) search → en düşük maliyetli (LC) arama 5
Canlı düğümlerden tahmini maliyeti $\hat{c} = f + \hat{g}$ en küçük olanı genişleten arama.

linear congruential generator (LCG) → doğrusal eşlik üreteci 7
Bir sonraki sayıyı $(a \cdot x + c) \bmod m$ formülüyle üreten, en eski ve en basit sözde rastgele sayı üreteci; düşük bitleri kısa dönemlidir.

list scheduling → liste çizelgeleme 6
Her işi o an en erken boşalan makineye veren açgözlü çizelgeleme.

list update problem → liste güncelleme problemi 8
Erişilen öğeye göre listeyi yeniden düzenleyerek erişim maliyetini azaltma problemi.

live node → canlı düğüm 5
Dal sınırda üretilmiş ama henüz genişletilmemiş, hâlâ umut vaat eden düğüm; canlı düğümler bir kuyruk yapısında bekler.

local optimum → yerel optimum 6
Komşu çözümlerin hiçbirinin daha iyi olmadığı çözüm.

local search → yerel arama 6
Bir çözümden başlayıp küçük değişikliklerle iyileştiren arama (ör. 2-opt).

locality of reference → yerellik 8
Programların belleğe rastgele değil, yakın geçmişte kullandıkları yerlere yeniden erişme eğilimi; LRU'yu FIFO'dan iyi yapan budur.

longest common subsequence (LCS) → en uzun ortak alt dizi 3
İki dizide aynı göreli sırayla geçen (ardışık olması gerekmeyen) en uzun karakter dizisi.

longest common substring → en uzun ortak alt dizgi 3
İki dizgide ardışık olarak ortak geçen en uzun parça.

longest increasing subsequence (LIS) → en uzun artan alt dizi 3
Dizideki öğelerden sırası korunarak seçilebilen en uzun artan dizi.

longest processing time first (LPT) → en uzun iş önce 6
İşleri büyükten küçüğe sıralayıp her birini en az yüklü makineye veren çizelgeleme kuralı.

lost update → kayıp güncelleme 9
İki iş parçacığının aynı değeri okuyup yazması sonucunda bir yazmanın ötekini ezmesi.

LP relaxation → LP gevşetmesi 5
Tam sayı koşulunu kaldırıp doğrusal programı çözerek sınır elde etme.

M

makespan → en büyük yük 6
Çizelgede son işin bittiği an (C_{max}).

marking algorithm → işaretleme algoritması 8
Evrelere bölünen ve her evrede yalnız işaretlenmemiş sayfaları çıkaran çevrim içi önbellek politikası.

Markov's inequality → Markov eşitsizliği 7
Negatif olmayan X için $P(X \geq a) \leq E[X] / a$ eşitsizliği.

matching → eşleştirme 6
Hiçbir ikisi ortak uç paylaşmayan kenar kümesi.

matrix chain multiplication → matris zinciri çarpımı 3
Bir matris dizisini en az skaler çarpımla çarpmak için parantezleme sırasını bulma problemi.

matroid → matroid 2
Açgözlü algoritmanın en iyi çözümü bulmasını güvence altına alan küme yapısı.

maximum clique → maksimum klik 5
Her iki düğümü komşu olan en büyük düğüm kümesi.

maximum coverage → en büyük kapsama 2
En fazla k küme seçerek olabildiğince çok öğeyi kapsama problemi; açgözlü seçim en iyinin en az $1 - 1/e$ katına ulaşır.

median → ortanca 8
Sıralı verinin ortasındaki değer.

memoization → bellekleme 3
Yukarıdan aşağı özyinelemede hesaplanan sonuçları saklayıp yeniden kullanma.

message passing → ileti 9
Dağıtık bellekte işlemcilerin veri alışverişi için gönderdiği bilgi.

metaheuristic → metasezgisel 6
Yerel optimumdan kaçmak için genel arama stratejisi (tavlama, genetik algoritma).

min-conflicts heuristic → en az çakışma sezgiseli 4
Çakışan bir değişkene en az çakışma yaratan değeri veren yerel arama.

minimum cut → minimum kesim 7
Çizgeyi iki parçaya ayıran en az kenarlı (ya da en küçük kapasiteli) kesim.

minimum remaining values (MRV) → MRV (en az kalan değer) 4
Kısıt problemlerinde, en az olası değeri kalan değişkeni önce seçme sezgiseli.

Monte Carlo algorithm → Monte Carlo algoritması 7
Süresi sınırlı, yanıtı küçük bir olasılıkla yanlış olabilen algoritma.

move-to-front (MTF) → öne taşı 8
Erişilen öğeyi listenin başına taşıyan kural; 2-rekabetçidir.

moving average → hareketli ortalama 8
Son k değerin ortalamasını her yeni değerde güncelleme.

N

N-queens problem → n-vezir problemi 4
n × n tahtaya birbirini tehdit etmeyen n vezir yerleştirme.

nearest neighbor heuristic → en yakın komşu 6
Her adımda ziyaret edilmemiş en yakın şehre giden TSP sezgiseli.

NP-complete → NP-tam 4
Hem NP'de hem NP-zor olan problem.

NP-hard → NP-zor 4
NP'deki her problemin polinom sürede indirgenebildiği (en az NP kadar zor) problem.

O

oblivious adversary → habersiz düşman 8
Rastgele algoritmanın kodunu bilen ama yazı-tura sonuçlarını göremeyen, bu yüzden istek dizisini baştan seçen düşman.

offline algorithm → çevrim dışı algoritma 8
Tüm girdiyi baştan bilen algoritma.

online algorithm → çevrim içi algoritma 8
Girdiyi parça parça alıp geleceği bilmeden her adımda karar veren algoritma.

optimal → optimal 6
Amaç fonksiyonuna göre en iyi (çözüm).

optimal substructure → optimal alt yapı 3
Optimal çözümün, alt problemlerin optimal çözümlerinden kurulabilmesi.

optimization problem → en iyileme problemi 4
Cevabı "evet" ya da "hayır" değil, en iyi değer olan problem; çoğu zaman bir karar problemine çevrilebilir.

overlapping subproblems → örtüşen alt problemler 3
Aynı alt problemin özyinelemede tekrar tekrar ortaya çıkması.

P

page fault → sayfa hatası 8
İstenen sayfanın önbellekte olmaması.

paging → sayfalama 8
Sınırlı önbellekte hangi sayfanın tutulacağına karar verme problemi.

parallel algorithm → paralel algoritma 9
İşi aynı anda çalışan birden çok işlemciye bölen algoritma.

partial solution → kısmi çözüm 4
Geri izlemede adım adım kurulan, henüz tamamlanmamış çözüm.

pipelining → boru hattı 9
İşi aşamalara bölüp farklı verileri aşamalarda üst üste işleme.

pivot → pivot 7
Bölümlemede diziyi ikiye ayırmak için seçilen öğe.

polynomial time → polinom zaman 4
Bir k sabiti için $O(n^k)$ süre.

power iteration → kuvvet yinelemesi 6
Bir vektörü matrisle tekrar tekrar çarpıp en büyük özdeğer ve özvektöre yakınsatan yöntem.

PRAM model → PRAM modeli 9
Ortak belleği paylaşan, eş zamanlı adımlarla çalışan işlemcilerden oluşan soyut model.

prefix sum (scan) → önek toplamı 9
Her konumda o konuma kadarki öğelerin toplamı; paralel olarak O(log n) derinlikte.

prefix-free code → önek-özgür kod 2
Hiçbir kod sözcüğünün bir başka kod sözcüğünün öneki olmadığı kod.

primality test → asallık testi 7
Bir sayının asal olup olmadığını sınama (Miller–Rabin).

pruning → budama 4
Çözüme götüremeyeceği anlaşılan dalı aramadan çıkarma.

pseudo-polynomial → sözde-polinom 3
Girdi uzunluğuna değil girdideki sayısal değere (ör. W) göre polinom olan süre.

pseudo-random number → sözde rastgele sayı 7
Deterministik bir üreteçle elde edilen, rastgele görünen sayı.

Q

quickselect → rastgele seçme (QuickSelect) 7
Rastgele pivotla k. en küçük öğeyi beklenen O(n) sürede bulma.

R

race condition → yarış durumu 9
Sonucun iş parçacıklarının çalışma sırasına bağlı olması.

randomized algorithm → rastgele algoritma 7
Adımlarında rastgele seçim yapan algoritma.

reduced cost matrix → indirgenmiş maliyet matrisi 5
Satır ve sütun en küçükleri çıkarılmış maliyet matrisi; çıkarılan toplam TSP için alt sınırdır.

reduction → indirgeme 6
Bir problemin örneklerini başka bir problemin örneklerine dönüştürme.

reservoir sampling → rezervuar örnekleme 8
Uzunluğu bilinmeyen akıştan k öğelik düzgün rastgele örneklem tutma.

running median → çevrim içi ortanca 8
Akıştaki değerlerin ortancasını iki heap ile güncel tutma.

S

satisfiability (SAT) → mantıksal doyurulabilirlik 4
Mantıksal değişkenlerden kurulmuş bir formülü doğru yapan bir değer ataması olup olmadığını soran problem; ilk NP-tam problemdir.

scheduling → çizelgeleme 6
İşleri makinelere ya da zaman dilimlerine atama problemi.

seed → tohum 7
Sözde rastgele sayı üretecinin başlangıç değeri.

self-organizing list → kendini düzenleyen liste 8
Öne taşı ya da transpoz gibi bir kuralla, erişilen öğeyi öne doğru taşıyarak kendini kullanıma uyduran liste.

shared memory → paylaşımlı bellek 9
Bütün işlemcilerin aynı adres uzayını gördüğü düzen; iletişim ucuzdur ama eşzamanlı erişim denetlenmelidir.

simulated annealing → benzetimli tavlama 6
Kötüleşen adımları azalan bir olasılıkla kabul eden metasezgisel.

ski rental problem → kayak kiralama problemi 8
Kiralama ile satın alma arasında geleceği bilmeden seçim yapma problemi.

sliding window → kayan pencere 8
Toplamı bir değişkende tutup yeni öğeyi ekleyerek, en eski öğeyi çıkararak son k öğe üzerindeki değeri her adımda güncelleyen yöntem.

speedup → hızlanma 9
Sıralı sürenin paralel süreye oranı: T_1 / T_p.

stack algorithm → yığın algoritması 8
Daha büyük bellekte sayfa kümesi küçük bellektekini kapsayan sayfa değiştirme politikası; Belady anomalisi göstermez.

state-space tree → durum uzayı ağacı 4
Kısmi çözümlerin düğüm, seçimlerin kenar olduğu arama ağacı.

Strassen's algorithm → Strassen algoritması 1
Matrisleri sekiz yerine yedi alt çarpımla O(n^2,81) sürede çarpan yöntem.

strong liar → güçlü yalancı 7
Bileşik bir sayı için tanık olmayan, onu asal gibi gösteren taban değeri.

subproblem → alt problem 3
Asıl problemin daha küçük bir örneği.

subset sum → alt küme toplamı 3
Toplamı verilen değere eşit bir alt küme olup olmadığı problemi.

T

tabulation → tablolama 3
Alt problemleri küçükten büyüğe (aşağıdan yukarı) bir tabloda doldurma.

throughput → verim 9
Birim zamanda tamamlanan iş sayısı; boru hattında aşama sayısı kadar artar.

top-down / bottom-up → yukarıdan aşağı / aşağıdan yukarı 3
DP'nin iki biçimi: özyineleme + bellekleme ya da döngüyle tablo doldurma.

traceback (solution reconstruction) → geri okuma (backtracking ise geri izleme) 3
Doldurulmuş DP tablosundan geriye doğru giderek çözümün kendisini bulma.

transpose (heuristic) → transpoz 8
Erişilen öğeyi bir önündekiyle takas eden liste düzenleme kuralı.

travelling salesman problem (TSP) → gezgin satıcı problemi 6
Her şehre bir kez uğrayıp başlangıca dönen en kısa turu bulma.

triangle inequality → üçgen eşitsizliği 6
Her üç nokta için d(a, c) ≤ d(a, b) + d(b, c) koşulu.

U

unbounded knapsack → sınırsız sırt çantası 3
Her eşyadan istenen sayıda alınabilen sırt çantası problemi.

up-sweep / down-sweep → yukarı süpürme / aşağı süpürme 9
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 6
Her kenarın en az bir ucunu içeren düğüm kümesi.

W

witness → tanık 7
Bir sayının bileşik olduğunu kanıtlayan taban değeri (Miller–Rabin).

work / span → iş / derinlik 9
Toplam işlem sayısı (T_1) ve en uzun bağımlılık zinciri (T_∞).

work stealing → iş çalma 9
Boşta kalan iş parçacığının başka birinin kuyruğundan iş alması.

work-efficient → iş-verimli 9
İşi, aynı problemin en iyi sıralı algoritmasının süresiyle aynı mertebede olan paralel algoritma.

EK B

İleri Algoritmalar İçin Ön Bilgiler

İLERİ ALGORİTMALAR · ÖN BİLGİLER

İleri Algoritmalar dersi Algoritmalar dersinin devamıdır; karmaşıklık analizi, sıralama, arama, çizge ve dizgi algoritmaları bilindiği varsayılır. Ders Java ile programlamayı, temel veri yapılarını, ayrık matematiği, ispat tekniklerini, toplamları ve logaritmaları, olasılığı ve saymayı bir araya getirir. Bu sayfadaki 20 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 (4) **Temel veri yapıları (3)** **Ayrık matematik ve ispat (2)**

Analiz araçları (2) **Olasılık ve sayma (3)** **Diğer matematik ve kavramlar (6)**

Java ve programlama

Sunumlardaki kodları okumak, izlemek ve çözümleri Java ile yazmak için gereken bilgiler.

[ ] 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 3 · Tablo Doldurma İzleme**

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
```

↻ 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 1 · Hızlı Üs Alma: İzleme** **Bölüm 4 · 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 2 · 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 3 · 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
```

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 8 · 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 + " ");
```

▲

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 · Huffman: Adımlar** **Bölüm 8 · 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)
```

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 · 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
```

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 5 · 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.
```

∴ 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 2 · Para Üstü Problemi** **Bölüm 2 · Neden En Erken Biten?**

Bölüm 6 · 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.
```

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 7 · 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 1 · 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
```

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 6 · Kaba Kuvvetle Karşılaştırma**

Bölüm 4 · 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 7 · Olasılık Hatırlatması**

Bölüm 7 · Hata Olasılığını Küçültme **Bölüm 7 · 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 7 · Beklenen Karşılaştırma** **Bölüm 7 · QuickSort Analizi**

Bölüm 8 · 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 · Matris Çarpımı** **Bölüm 6 · Power Iteration**

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 1 · Yönelim Testi** **Bölüm 1 · En Yakın Nokta Çifti**

Bölüm 6 · 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 7 · 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 4 · O(1) Güvenlik Testi**

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 3 · 0/1 Sırt Çantası Problemi** **Bölüm 4 · P, NP ve NP-Zor** **Bölüm 5 · 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 9 · Yarış Durumu** **Bölüm 9 · Fork/Join Adım Adım** **Bölüm 9 · 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

İleri Algoritmalarda Önemli İsimler

İLERİ ALGORİTMALAR · İSİMLER

Dersin dokuz bölümündeki algoritmaların ve fikirlerin arkasındaki 43 isim: Karatsuba çarpımından ve Huffman kodlamadan Bellman'ın dinamik programlamasına, Cook'un NP-tamlık teoremine, dal sınır yöntemine, rekabetçi analize ve paralel hesaplamaya 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.

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 3 · Fibonacci: Düz Özyineleme** **Bölüm 3 · 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 7 · Miller–Rabin Asallık Testi** **Bölüm 7 · 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 6 · 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 · Büyük Sayı Çarpımı**

aritmetik seri döngü sayma

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 4 · 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 7 · Örnek: Karıştırma** **Bölüm 7 · 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 1 · Birleştirme Sıralaması** **Bölüm 7 · 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 7 · Monte Carlo ile π** **Bölüm 7 · 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 5 · Tam Sayılı Programlama** **Bölüm 5 · 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 2 · 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 7 · Sözde Rastgele Sayılar** **Bölüm 4 · Geri Dönüp Dene**

sözde rastgele sayılar geri izleme

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 2 · Huffman Kodlama** **Bölüm 2 · Huffman Neden En İyi?**

Huffman kodlama önek kodu açgözlü seçim

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 3 · Dinamik Programlama Nedir?** **Bölüm 3 · İki Temel Özellik**

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 9 · Amdahl Yasası**

doğrusal yoklama hızlanma Amdahl yasası

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 7 · Miller–Rabin Asallık Testi**

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 7 · Rastgele Seçme (QuickSelect)**

hızlı sıralama bölümleme QuickSelect

KA

Anatoly Karatsuba

1937–2008 · Sovyet · Moskova

Karatsuba çarpımı · 1960/1962

1960'ta Kolmogorov'un seminerinde iki n basamaklı sayıyı çarpmanın Ω(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 1 · Büyük Sayı Çarpımı** **Bölüm 1 · 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 5 · Dal Sınır Fikri** **Bölüm 5 · Tam Sayılı Programlama**

dal sınır LP gevşetmesi

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 3 · Diğer Klasik DP Problemleri** **Bölüm 6 · 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 5 · İndirgenmiş Maliyet Matrisi** **Bölüm 5 · TSP: 6 Şehirli Örnek**

dal sınır indirgenmiş maliyet matrisi gezgin satıcı

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 2 · Ne Zaman Doğru Çalışır?**

Edmonds-Karp polinom zaman matroid

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 polio minoları tanımlaması, bulmaca ve döşeme problemlerinin bilgisayarla aranmasına esin kaynağı oldu. Golomb kodlaması da onun katkısıdır.

DERSTE **Bölüm 4 · Durum Uzayı Ağacı** **Bölüm 4 · 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 kuralının 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 6 · Liste Çizelgeleme** **Bölüm 6 · LPT Kuralı** **Bölüm 1 · 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 8 · OPT Adım Adım** **Bölüm 8 · 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 9 · 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 3 · Diğer Klasik DP Problemleri** **Bölüm 3 · DP Tasarım Adımları**

dinamik programlama en olası yol

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 1 · Strassen'in 7 Çarpımı** **Bölüm 1 · Strassen Analizi**

Strassen algoritması matris çarpımı

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 4 · P, NP ve NP-Zor**

NP-tam SAT P = NP?

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 4 · P, NP ve NP-Zor** **Bölüm 6 · 3-SAT → Düğüm Kapsama**

NP-tam indirgeme Rabin-Karp

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 4 · P, NP ve NP-Zor** **Bölüm 6 · 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. $T_p \ge W/p$ ve $T_p \ge D$ alt sınırlarıyla birlikte okunduğunda, p işlemcili süre en iyinin iki katını geçmez. Brent sayısal kök bulmadaki Brent yöntemiyle (1973) de tanınır.

DERSTE **Bölüm 9 · İş ve Derinlik**

Brent teoremi iş derinlik

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 6 · Metrik TSP** **Bölüm 6 · MST Tabanlı 2-Yaklaşım**

gezgin satıcı problemi yaklaşım oranı

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 3 · Diğer Klasik DP Problemleri**

Kadane algoritması en büyük toplamlı alt dizi O(n)

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 6 · Neden Yaklaşım Algoritmaları?** **Bölüm 4 · 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 9 · PRAM Modeli** **Bölüm 7 · Rastgele Algoritmalar**

#P paralel model rastgele indirgeme

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 8 · Rekabet Oranı** **Bölüm 8 · MTF 2-Rekabetçidir**

Bölüm 8 · 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 8 · Rezervuar Örnekleme** **Bölüm 8 · 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 9 · Paralel Önek Toplamı** **Bölüm 9 · 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 9 · Amdahl Yasası**

Gustafson yasası ölçeklenmiş hızlanma Amdahl yasası

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 9 · İş ve Derinlik** **Bölüm 9 · 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 9 · Paralel Önek Toplamı** **Bölüm 9 · 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 7 · Karger Minimum Kesim** **Bölüm 7 · Hata Olasılığını Küçültme**

minimum kesim Monte Carlo algoritması

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 6 · Yaklaşım Oranı** **Bölüm 6 · 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.