Bölüm 2: Algoritma Karmaşıklığı

Algoritmalar

Zaman ve bellek | En iyi / en kötü / ortalama | Büyük O, Ω, Θ | Büyüme sınıfları

Döngü analizi | Yineleme bağıntıları | Ana Teorem | f1–f6 örnekleri

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/algorithms/slides/Bolum_02_Algoritma_Karmasikligi.html

GİRİŞ

Öğrenme Çıktıları

Kavramlar

Zaman ve alan karmaşıklığını, en iyi, en kötü ve ortalama durum analizini birbirinden ayırmak.

Gösterim

Büyük O, Ω ve Θ tanımlarını yazmak; bir fonksiyon için c ve n_0 sabitlerini bularak sınır kanıtlamak.

Döngü analizi

Tek, iç içe ve çarpımsal döngülerin adım sayısını toplam formülleriyle hesaplamak (f1, f2, f6).

Özyineleme

Özyinelemeli kod için bağıntı kurmak; açma, özyineleme ağacı ve Ana Teorem ile çözmek (f3, f4, f5).

Bölüm boyunca altı kısa örnek fonksiyon (`f1` – `f6`) üzerinde çalışacağız: her biri farklı bir büyüme sınıfını (n, n^3, 2^n, n, $n \log n$, $\log^* n$) temsil eder.

KARMAŞIKLIK KAVRAMI

Algoritma Karmaşıklığı

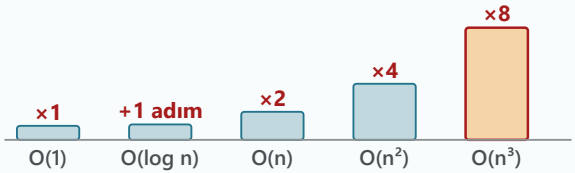
- **Karmaşıklık teorisi** (complexity theory), bir algoritmanın **kaynak kullanımını** ölçer: zaman ve bellek.
- Kaynak kullanımı, **girdi boyutu n**'nin bir fonksiyonu olarak yazılır: T(n), S(n).
- Amaç saniye ölçmek değil, girdi büyüdükçe kaynak ihtiyacının **nasıl büyüdüğünü** görmektir.

Zaman karmaşıklığı (time complexity)

Algoritmanın yaptığı **temel işlem sayısı**: atama, karşılaştırma, aritmetik işlem, dizi erişimi.

Alan karmaşıklığı (space complexity)

Algoritmanın girdiye ek olarak kullandığı bellek: değişkenler, yardımcı diziler, özyineleme yığını.



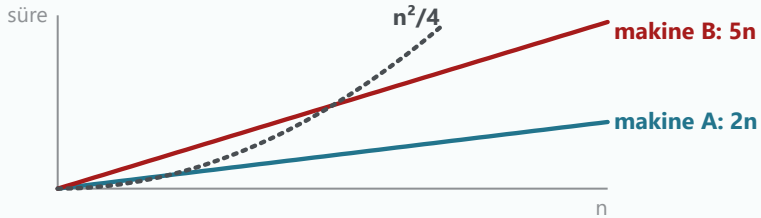
Temel soru: Girdi boyutu iki katına çıktığında çalışma süresi ne olur? Aynı mı kalır, iki katına mı, dört katına mı çıkar, yoksa karesi mi alınır?

KARMAŞIKLIK KAVRAMI

Süre Ölçümü Neden Yetmez?

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

- Bu kod 10^8 öğeli bir dizide doğrusal aramanın süresini ölçer.
- Sonuç **makineye**, işlemciye, JIT derleyicisine, önbelleğe ve o anki yüke göre değişir.
- Tek bir ölçüm, girdi büyüyünce ne olacağını **söylemez**.



Deneysel ölçüm

Gerçek süreyi verir ama makineye bağlıdır; kodun yazılmış olması gerekir.

Asimptotik analiz

Adım sayısını n'nin fonksiyonu olarak verir; makineden bağımsızdır, kod yazmadan yapılabilir.

Adım Sayma: `dogrusalArama`

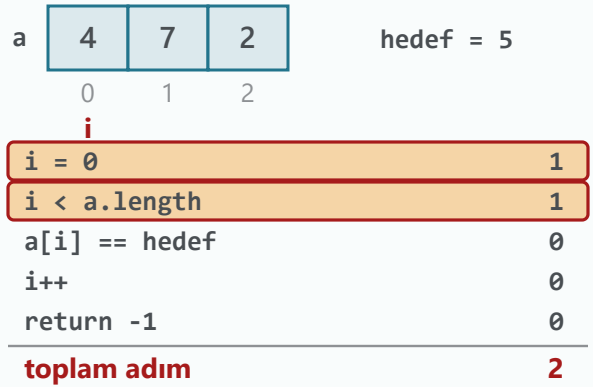
```
for (int i = 0; i < a.length; i++) {
  if (a[i] == hedef)
    return i;
}
return -1;
```

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

- Her temel işlem **1 birim** sayılır (RAM modeli). En kötü durum: hedef dizide yok.
- 3n + 3'teki **3** katsayısı ve **+3** makineye göre değişir; değişmeyen şey **n ile doğrusal** büyümedir → T(n) = Θ(n).

dogrusalArama: Adımları Sayarak İzle

```
▶ for (int i = 0; i < a.length; i++) {
2   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



a = [4, 7, 2], hedef = 5 (dizide yok: en kötü durum). `i = 0` atanır ve `0 < 3` denetlenir: 2 adım.

Adım 1 / 8

DEĞİŞKENLER

`i = 0` `hedef = 5` `adım = 2`

dogrusalArama: Adımları Sayarak İzle

```
1 for (int i = 0; i < a.length; i++) {
▶   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



hedef = 5

i

i = 0	1
i < a.length	1
a[i] == hedef	1
i++	0
return -1	0
toplam adım	**3**

a[0] == hedef? 4 ≠ 5. Karşılaştırma sayısı 1 oldu.

Adım 2 / 8

DEĞİŞKENLER

i = 0 | hedef = 5 | adım = 3

dogrusalArama: Adımları Sayarak İzle

```
▶ for (int i = 0; i < a.length; i++) {
2   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



hedef = 5

i = 0	1
i < a.length	2
a[i] == hedef	1
i++	1
return -1	0
toplam adım	**5**

`i++` → i = 1, sonra `1 < 3` denetlenir: döngü sürer.

Adım 3 / 8

DEĞİŞKENLER

`i = 1` `hedef = 5` `adım = 5`

dogrusalArama: Adımları Sayarak İzle

```
1 for (int i = 0; i < a.length; i++) {
▶   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



hedef = 5

i = 0	1
i < a.length	2
a[i] == hedef	2
i++	1
return -1	0
toplam adım	**6**

a[1] == hedef? 7 ≠ 5. Karşılaştırma sayısı 2 oldu.

Adım 4 / 8

DEĞİŞKENLER

i = 1 | hedef = 5 | adım = 6

dogrusalArama: Adımları Sayarak İzle

```
▶ for (int i = 0; i < a.length; i++) {
2   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



`i = 0`	1
`i < a.length`	3
`a[i] == hedef`	2
`i++`	2
`return -1`	0
toplam adım	**8**

`i++` → i = 2, sonra `2 < 3` denetlenir: döngü sürer.

Adım 5 / 8

DEĞİŞKENLER

`i = 2` `hedef = 5` `adım = 8`

dogrusalArama: Adımları Sayarak İzle

```
1 for (int i = 0; i < a.length; i++) {
▶   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



i = 0	1
i < a.length	3
a[i] == hedef	3
i++	2
return -1	0
toplam adım	**9**

a[2] == hedef? 2 ≠ 5. Karşılaştırma sayısı 3 oldu.

Adım 6 / 8

DEĞİŞKENLER

`i = 2` `hedef = 5` `adım = 9`

dogrusalArama: Adımları Sayarak İzle

```
▶ for (int i = 0; i < a.length; i++) {
2   if (a[i] == hedef)
3     return i;
4 }
5 return -1;
```



`i++` → i = 3. `3 < 3` yanlış: koşul n + 1 = **4. kez** denetlendi, döngü biter.

Adım 7 / 8

DEĞİŞKENLER

`i = 3` `hedef = 5` `adım = 11`

dogrusalArama: Adımları Sayarak İzle

```
1 for (int i = 0; i < a.length; i++) {
2   if (a[i] == hedef)
3     return i;
4 }
▶ return -1;
```



`return -1`. Toplam **12** adım = 3n + 3 (n = 3). Her öğe için sabit sayıda (3) işlem: Θ(n).

Adım 8 / 8

DEĞİŞKENLER

`i = 3` `hedef = 5` `adım = 12`

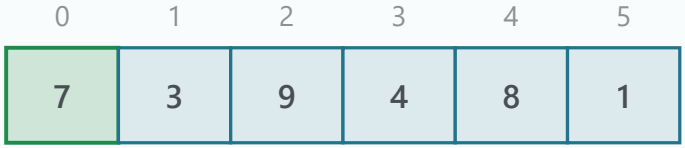
KARMAŞIKLIK KAVRAMI

En İyi, En Kötü ve Ortalama Durum

En iyi durum (best case)
En elverişli girdide en az adım. Bir alt sınır verir; tek başına pek bilgi taşımaz.

En kötü durum (worst case)
Her girdi için geçerli bir **garanti**: algoritma bundan uzun sürmez.

Ortalama durum (average case)
Girdilerin bir olasılık dağılımı üzerinden beklenen adım sayısı.



hedef 7 → 1 karşılaştırma

En iyi durum: hedef ilk öğedir, tek karşılaştırma yeterlidir → Θ(1).

En İyi, En Kötü ve Ortalama Durum

En iyi durum (best case)
En elverişli girdide en az adım. Bir alt sınır verir; tek başına pek bilgi taşımaz.

En kötü durum (worst case)
Her girdi için geçerli bir **garanti**: algoritma bundan uzun sürmez.

Ortalama durum (average case)
Girdilerin bir olasılık dağılımı üzerinden beklenen adım sayısı.



hedef 5 → 6 karşılaştırma, bulunamadı

En kötü durum: hedef dizide yok (ya da en sonda); n = 6 karşılaştırma → Θ(n).

KARMAŞIKLIK KAVRAMI

En İyi, En Kötü ve Ortalama Durum

En iyi durum (best case)
En elverişli girdide en az adım. Bir alt sınır verir; tek başına pek bilgi taşımaz.

En kötü durum (worst case)
Her girdi için geçerli bir **garanti**: algoritma bundan uzun sürmez.

Ortalama durum (average case)
Girdilerin bir olasılık dağılımı üzerinden beklenen adım sayısı.



hedef 4 → 4 karşılaştırma

Ortalama durum: hedef her konumda eşit olasılıkla ise ortalama (1 + 2 + … + n)/n = (n + 1)/2 = 3,5 karşılaştırma → yine Θ(n).

KARMAŞIKLIK KAVRAMI

Alan Karmaşıklığı

Alan karmaşıklığı genellikle **ek bellek** (auxiliary space) için verilir: girdinin kendisi sayılmaz. Özyinelemede her açık çağrı yığında bir **çerçeve** (stack frame) tutar.

Kod	Ek bellek	Bellek	Zaman
`f1` : tek döngü	yalnızca `x` ve `i`	$\Theta(1)$	$\Theta(n)$
`new int[n]` oluşturmak	n öğeli yardımcı dizi	$\Theta(n)$	$\Theta(n)$
`f3` : f3(n−1) + f3(n−1)	yığın derinliği n (aynı anda en çok n çerçeve)	$\Theta(n)$	$\Theta(2^n)$
`f4` : f4(n/2) + f4(n/2)	yığın derinliği $\approx \log_2 n$	$\Theta(\log n)$	$\Theta(n)$
Birleştirme sıralaması	birleştirme için yardımcı dizi	$\Theta(n)$	$\Theta(n \log n)$

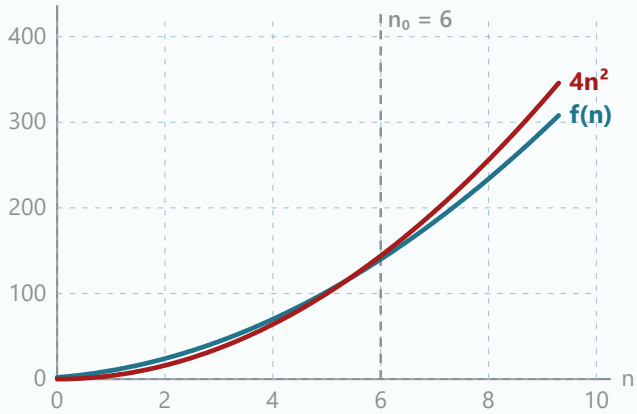
Zaman ve bellek farklı büyüyebilir. f3 üstel sayıda çağrı yapar, ama çağrılar sırayla bittiği için yığında aynı anda en çok n çerçeve bulunur.

ASİMPTOTİK GÖSTERİM

Büyük O (Big-O) Gösterimi

$$f(n) = O(g(n)) \Leftrightarrow \exists\ c > 0, n_0 :$$
$$0 \leq f(n) \leq c \cdot g(n), \text{her } n \geq n_0 \text{ için}$$

- Büyük n'ler için f, g'nin bir sabit katını **aşmaz**: g bir **üst sınırdır**.
- Küçük n'ler ($n < n_0$) önemsizdir; sabit çarpanlar c'ye gömülür.
- Örnek: $3n^2 + 5n + 2 \leq 4n^2$, her $n \geq 6$ için → **$O(n^2)$**.



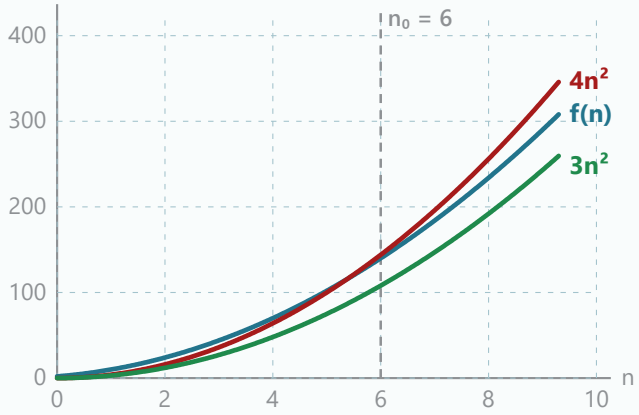
"Büyük O en kötü durumu temsil eder" ifadesi tam doğru değildir. Büyük O bir **üst sınır** türüdür; en kötü, en iyi ya da ortalama durumun her biri için kullanılabilir. Genellikle en kötü durum süresi için verildiğinden bu ikisi karıştırılır.

ASİMPTOTİK GÖSTERİM

Büyük Omega (Ω) ve Büyük Teta (Θ)

Gösterim	Anlamı	Tanım ($\exists$ c, $n_0 > 0$, her $n \geq n_0$ için)	Benzetme
f = O(g)	üst sınır	$f(n) \leq c \cdot g(n)$	f ≤ g
f = Ω(g)	alt sınır	$f(n) \geq c \cdot g(n)$	f ≥ g
f = Θ(g)	sıkı (kesin) sınır	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	f = g

- $f = \Theta(g) \Leftrightarrow f = O(g)$ **ve** $f = \Omega(g)$.
- $3n^2 + 5n + 2$ hem $O(n^2)$ hem $O(n^3)$'tür; ama yalnızca n^2 için **Θ**'dır.
- Doğrusal arama: en kötü durum Θ(n), en iyi durum Θ(1); tüm girdiler için O(n) ve Ω(1).
- **Durum** (hangi girdi) ile **sınır** (O, Ω, Θ) ayrı kavramlardır. Eklemeli sıralama en kötü $\Theta(n^2)$, en iyi Θ(n): tüm girdiler için tek bir Θ yazılamaz. Birleştirme sıralaması ise her girdide Θ(n log n)'dir.

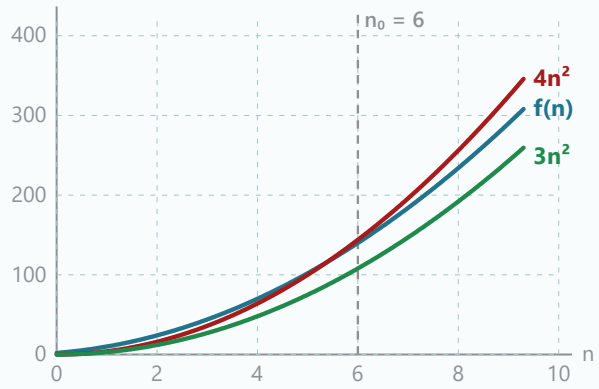


Örnek: $3n^2 + 5n + 2 = \Theta(n^2)$

Üst sınır: $3n^2 + 5n + 2 \leq c_2\, n^2$

$c_2 = 4$ deneyelim: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0$ $\Leftrightarrow n \geq 5{,}37$. Demek ki $\mathbf{n_0 = 6}$.

Adım 1 / 4



Örnek: $3n^2 + 5n + 2 = \Theta(n^2)$

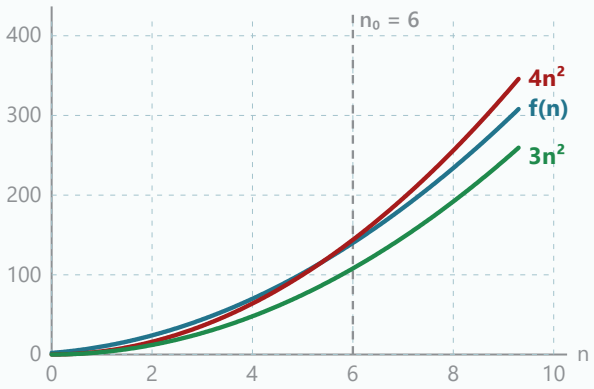
Üst sınır: $3n^2 + 5n + 2 \leq c_2\, n^2$

$c_2 = 4$ deneyelim: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0$ $\Leftrightarrow n \geq 5{,}37$. Demek ki **$n_0 = 6$**.

Kontrol: $n = 5 \rightarrow 102 > 100$ ✗

$n = 6 \rightarrow 140 \leq 144$ ✓

n = 5'te eşitsizlik bozulur, n = 6'dan itibaren sağlanır (sol taraf 6n + 8 artar, sağ taraf 8n + 4 artar).



Örnek: $3n^2 + 5n + 2 = \Theta(n^2)$

Üst sınır: $3n^2 + 5n + 2 \leq c_2\, n^2$

$c_2 = 4$ deneyelim: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0$ $\Leftrightarrow n \geq 5{,}37$. Demek ki $\mathbf{n_0 = 6}$.

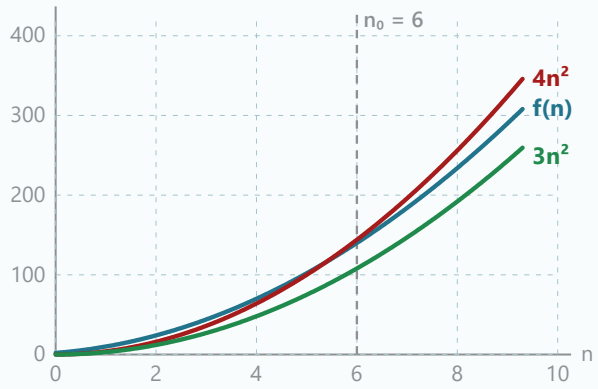
Kontrol: $n = 5 \rightarrow 102 > 100$ ✗

$n = 6 \rightarrow 140 \leq 144$ ✓

n = 5'te eşitsizlik bozulur, n = 6'dan itibaren sağlanır (sol taraf 6n + 8 artar, sağ taraf 8n + 4 artar).

Alt sınır: $3n^2 + 5n + 2 \geq 3n^2$

$c_1 = 3$ seçilirse eşitsizlik her $n \geq 1$ için doğrudur, çünkü $5n + 2 > 0$.



Örnek: $3n^2 + 5n + 2 = \Theta(n^2)$

$$\text{Üst sınır: } 3n^2 + 5n + 2 \leq c_2\, n^2$$

$c_2 = 4$ deneyelim: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0$ $\Leftrightarrow n \geq 5{,}37$. Demek ki $\mathbf{n_0 = 6}$.

$$\text{Kontrol: } n = 5 \rightarrow 102 > 100\ \text{✗}$$
$$n = 6 \rightarrow 140 \leq 144\ \text{✓}$$

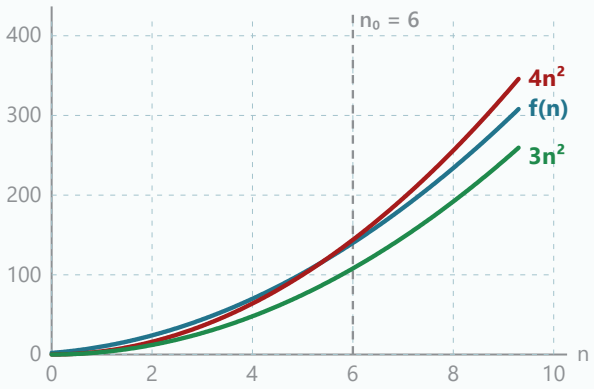
n = 5'te eşitsizlik bozulur, n = 6'dan itibaren sağlanır (sol taraf 6n + 8 artar, sağ taraf 8n + 4 artar).

$$\text{Alt sınır: } 3n^2 + 5n + 2 \geq 3n^2$$

$c_1 = 3$ seçilirse eşitsizlik her $n \geq 1$ için doğrudur, çünkü $5n + 2 > 0$.

$$3n^2 \leq f(n) \leq 4n^2,\ n \geq 6$$

$c_1 = 3$, $c_2 = 4$, $n_0 = 6$ ile tanım sağlanır: $\mathbf{f(n) = \Theta(n^2)}$.
Sabitler tek değildir; $c_2 = 10$, $n_0 = 1$ de çalışır.



Asimptotik Hesap Kuralları

Sabit çarpanları at

$c \cdot f(n) = \Theta(f(n))$: $5n = \Theta(n)$, $n^2/3 = \Theta(n^2)$.

Düşük dereceli terimleri at

Polinomda en büyük derece kalır: $4n^3 + 100n^2 + 7 = \Theta(n^3)$.

Toplam kuralı (ardışık kod)

$O(f) + O(g) = O(\max(f, g))$: $\Theta(n)$ döngü + $\Theta(n^2)$ döngü = $\Theta(n^2)$.

Çarpım kuralı (iç içe kod)

n kez çalışan döngü içinde $O(g)$ iş: $O(n \cdot g)$. İç işin her turda aynı olup olmadığına bakılır.

Logaritma tabanı önemsiz

$\log_a n = \log_b n / \log_b a$, sabit çarpan farkı: $O(\log_2 n) = O(\log_{10} n) = O(\log n)$.

Üslü ifadelerde taban önemli

$3^n = O(2^n)$ **yanlıştır**, çünkü $3^n/2^n = 1{,}5^n \to \infty$. Logaritmanın aksine üstel fonksiyonda taban değişikliği sabit bir çarpanla karşılanamaz.

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

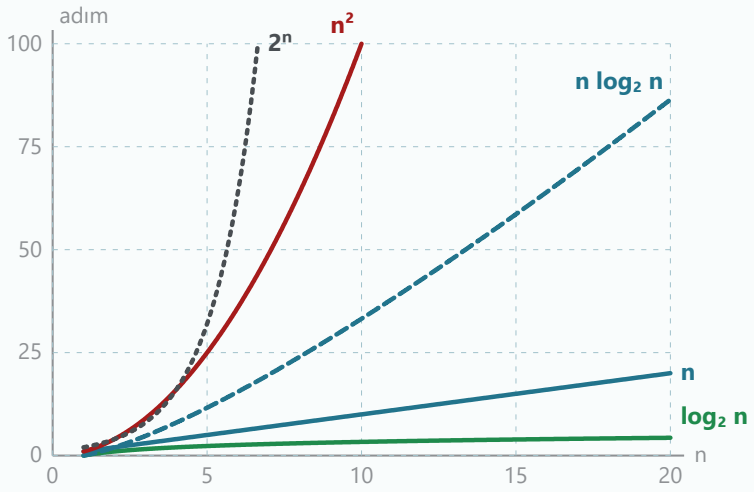
ASİMPTOTİK GÖSTERİM

Yaygın Büyüme Sınıfları

Sınıf	Adı	Tipik örnek	n iki katına çıkınca	Bu bölümde
$O(1)$	Sabit	ilk öğeyi yazdırma, dizi erişimi	değişmez	—
$O(\log n)$	Logaritmik	ikili arama, hızlı üs alma, Öklid	+1 adım	f6 (üst sınır)
$O(\sqrt{n})$	Karekök	asallık testi (deneme bölmesi)	≈ 1,41 katı	—
$O(n)$	Doğrusal	doğrusal arama, en büyük öğe, dizinin toplamı	2 katı	f1, f4
$O(n \log n)$	Log-doğrusal	birleştirme sıralaması, heap sort, hızlı sıralama (ortalama)	2 katından biraz fazla	f5
$O(n^2)$	Karesel	kabarcık, seçimli, eklemeli sıralama	4 katı	—
$O(n^3)$	Kübik	klasik matris çarpımı	8 katı	f2
$O(2^n)$	Üstel	tüm alt kümeler, saf özyinelemeli Fibonacci	karesi alınır	f3
$O(n!)$	Faktöriyel	tüm permütasyonlar	çok daha hızlı patlar	—

Sınıflar yukarıdan aşağı yavaşlar. Polinom sınıflar (n^k) genelde "verimli", üstel olanlar "verimsiz" sayılır. f6'nın gerçek sınıfı $\log n$'den de yavaş büyüyen $\log^* n$'dir.

Büyüme Eğrileri



- Küçük n'de eğriler birbirine yakındır; n = 4'te n^2 ile 2^n eşittir (16).
- **2^n** n ≈ 6,6'da, **n^2** n = 10'dan sonra 100 adımı aşar.
- **n log n** doğrusala yakın kalır; **log n** neredeyse düzdür.
- Uçlar daha da ayrılır: n = 10'da n! 3,6 milyonu bulur; √n ise n = 100'de yalnızca 10'dur.
- Asimptotik analiz, eğrilerin **büyük n**'deki sıralamasıyla ilgilenir.

ASİMPTOTİK GÖSTERİM

Sayılarla Büyüme: Hesaplayıcı

n = 1000 | 10 | 100 | 10^4 | 10^6 | n × 2

f(n)	işlem sayısı	10^9 işlem/s ile süre	bağıl
1	1	1 ns	
$\log_2 n$	10	10 ns	
$\sqrt{n}$	31,6	31,6 ns	
n	1.000	1 µs	
$n \log_2 n$	9.965,8	10 µs	
n^2	1.000.000	1 ms	
n^3	$1 \cdot 10^9$	1 s	
2^n	$1{,}1 \cdot 10^{301}$	$\approx 10^{284}$ yıl	
n!	$4 \cdot 10^{2567}$	$\approx 10^{2551}$ yıl	

n = **1.000** · çubuk: işlem sayısının basamak sayısı (log ölçek) · kırmızı: 1 saatten uzun

"n × 2" ile n sütunu 2, n^2 4, n^3 8 katına çıkar; $\log_2 n$ yalnızca 1 artar, 2^n karesini alır. Daha hızlı bilgisayar sabit bir çarpan kazandırır, daha iyi algoritma büyüme sınıfını değiştirir.

DÖNGÜ ANALİZİ

f1: Tek Döngü → O(n)

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

- **İlklendirme:** `int x = 0;` x'e 0 atar; sabit zamanlı işlem, O(1).
- **Döngü:** `for (int i = 0; i < n; i++)` i, 0'dan n − 1'e kadar ilerler: **n kez** çalışır.
- **Artırma:** `x++;` her döngü adımında bir kez çalışır; O(1).

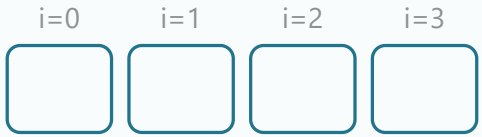
$$T(n) = O(1) + n \cdot O(1) = O(n)$$

Döngü n kez çalışır ve gövdesi sabit zamanlıdır; toplam zaman karmaşıklığı **Θ(n)** olur. Fonksiyon `x = n` döndürür, yani dönüş değeri adım sayısının kendisidir: f1(16) = 16, f1(32) = 32.

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
▶   int x = 0;
3   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 0

`x` sıfırlanır. Henüz döngü başlamadı.

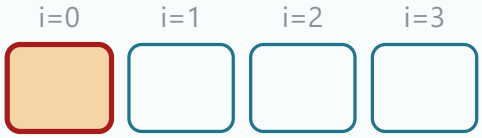
Adım 1 / 11

DEĞİŞKENLER

`n = 4` `x = 0` `i = –`

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 0

i = 0: koşul 0 < 4 doğru → gövdeye girilir.

Adım 2 / 11

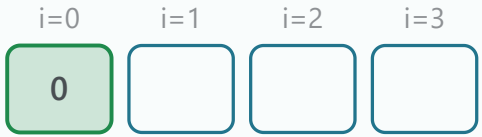
DEĞİŞKENLER

`n = 4` `x = 0` `i = 0`

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 1

`x++` → x = 1. Bu, 1. döngü adımı.

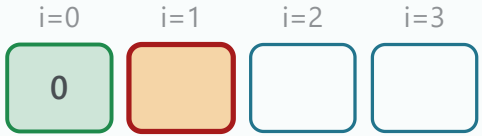
Adım 3 / 11

DEĞİŞKENLER

`n = 4` `x = 1` `i = 0`

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 1

i = 1: koşul 1 < 4 doğru → gövdeye girilir.

Adım 4 / 11

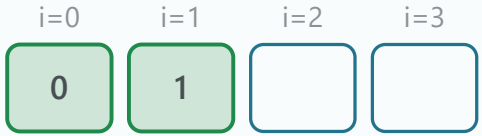
DEĞİŞKENLER

`n = 4` `x = 1` `i = 1`

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 2

`x++` → x = 2. Bu, 2. döngü adımı.

Adım 5 / 11

DEĞİŞKENLER

`n = 4` `x = 2` `i = 1`

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 2

i = 2: koşul 2 < 4 doğru → gövdeye girilir.

Adım 6 / 11

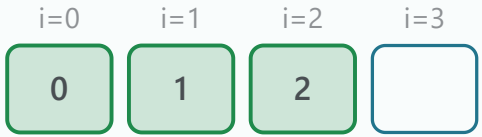
DEĞİŞKENLER

n = 4 | x = 2 | i = 2

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 3

`x++` → x = 3. Bu, 3. döngü adımı.

Adım 7 / 11

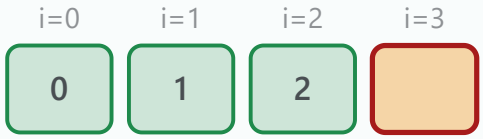
DEĞİŞKENLER

`n = 4` `x = 3` `i = 2`

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 3

i = 3: koşul 3 < 4 doğru → gövdeye girilir.

Adım 8 / 11

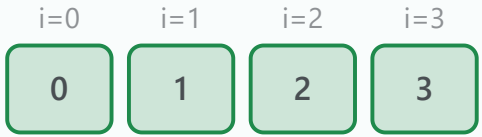
DEĞİŞKENLER

n = 4 | x = 3 | i = 3

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 4

`x++` → x = 4. Bu, 4. döngü adımı.

Adım 9 / 11

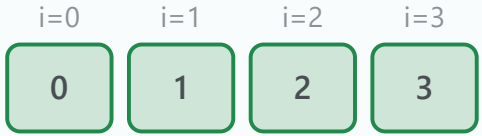
DEĞİŞKENLER

`n = 4` `x = 4` `i = 3`

DÖNGÜ ANALİZİ

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ sayısı: 4

i = 4: koşul 4 < 4 yanlış → döngü biter. Koşul n + 1 = 5 kez denetlendi.

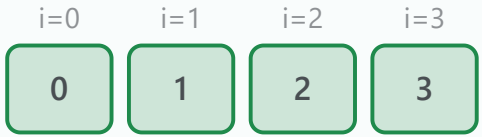
Adım 10 / 11

DEĞİŞKENLER

n = 4 | x = 4 | i = 4

f1(4): Kodu Adım Adım İzle

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
4     x++;
5   }
▶   return x;
7 }
```



x++ sayısı: 4

`return x` → 4. Gövde tam n = 4 kez çalıştı: Θ(n).

Adım 11 / 11

DEĞİŞKENLER

n = 4 | x = 4 | i = –

DÖNGÜ ANALİZİ

f2: İç İçe Döngü → $O(n^3)$

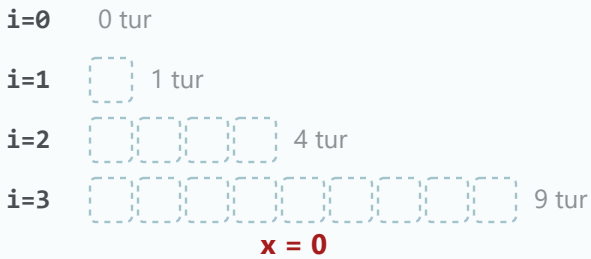
```
public int f2(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
      x++;
    }
  }
  return x;
}
```

- **İlklendirme:** `int x = 0;` sabit zamanlı, O(1).
- **Dış döngü:** i, 0'dan n − 1'e kadar: **n kez** çalışır.
- **İç döngü:** j, 0'dan i · i − 1'e kadar: **i^2 kez** çalışır.
- İç döngünün tur sayısı **i'nin güncel değerine bağlıdır**; bu yüzden basitçe "n · n" denemez, tur sayılarını **toplamak** gerekir.

Tuzak: İç içe iki döngü görünce otomatik olarak $O(n^2)$ demek yanlıştır. Burada iç döngünün sınırı i^2 olduğundan sonuç **n^3** mertebesindedir.

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
▶   int x = 0;
3   for (int i = 0; i < n; i++) {
4     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
8   return x;
9 }
```



`f2(4)` çağrıldı, `x = 0`. Gri kutular iç döngünün yapacağı turlardır: i. satırda i^2 kutu.

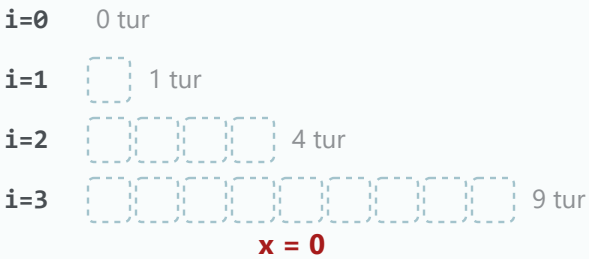
Adım 1 / 9

DEĞİŞKENLER

`n = 4` `i = –` `j = –` `x = 0`

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
8   return x;
9 }
```



i = 0: iç döngü koşulu `0 < 0 · 0` hemen yanlış; iç döngü **hiç** çalışmaz.

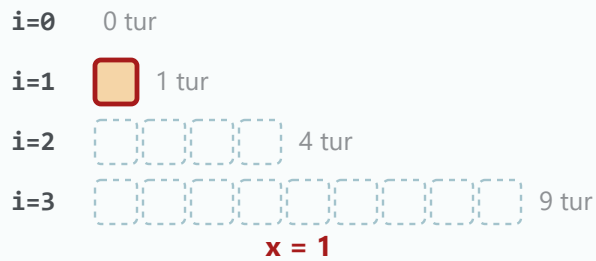
Adım 2 / 9

DEĞİŞKENLER

`n = 4` `i = 0` `j = 0` `x = 0`

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
▶       x++;
6     }
7   }
8   return x;
9 }
```



i = 1: iç döngü j < 1·1 = 1 olduğu sürece döner. İlk tur (j = 0): x = 1.

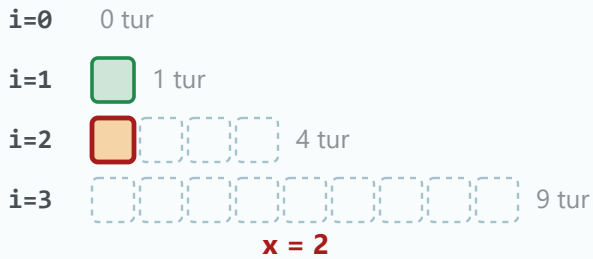
Adım 3 / 9

DEĞİŞKENLER

n = 4 | i = 1 | j = 0 | x = 1

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶       for (int j = 0; j < i * i; j++) {
▶         x++;
6       }
7   }
8   return x;
9 }
```



i = 2: iç döngü j < 2·2 = 4 olduğu sürece döner. İlk tur (j = 0): x = 2.

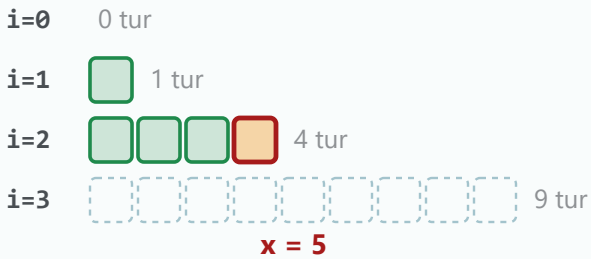
Adım 4 / 9

DEĞİŞKENLER

n = 4 | i = 2 | j = 0 | x = 2

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶       for (int j = 0; j < i * i; j++) {
▶         x++;
6       }
7   }
8   return x;
9 }
```



j = 3 son tur: i = 2 satırı 4 turda bitti, x = **5**.

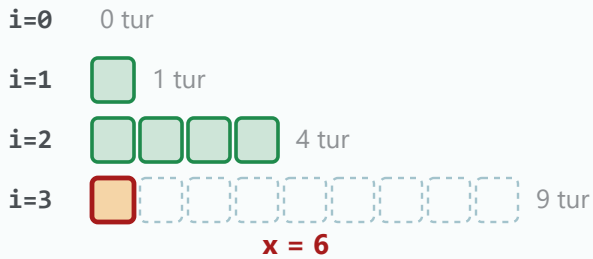
Adım 5 / 9

DEĞİŞKENLER

`n = 4` `i = 2` `j = 3` `x = 5`

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶       for (int j = 0; j < i * i; j++) {
▶         x++;
6       }
7   }
8   return x;
9 }
```



i = 3: iç döngü j < 3·3 = 9 olduğu sürece döner. İlk tur (j = 0): x = 6.

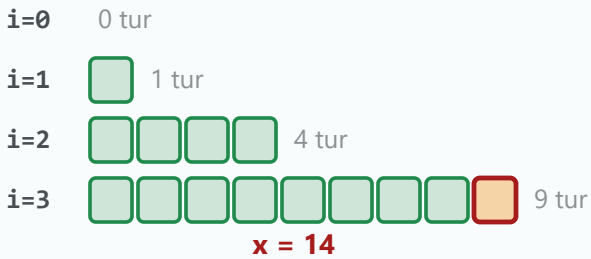
Adım 6 / 9

DEĞİŞKENLER

n = 4 | i = 3 | j = 0 | x = 6

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶       for (int j = 0; j < i * i; j++) {
▶         x++;
6       }
7   }
8   return x;
9 }
```



j = 8 son tur: i = 3 satırı 9 turda bitti, x = **14**.

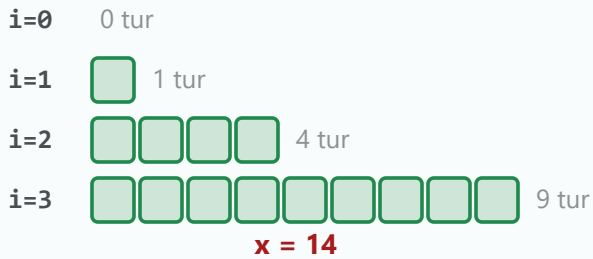
Adım 7 / 9

DEĞİŞKENLER

`n = 4` `i = 3` `j = 8` `x = 14`

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
8   return x;
9 }
```



i = 4: `4 < 4` yanlış, dış döngü biter. Turlar 0 + 1 + 4 + 9.

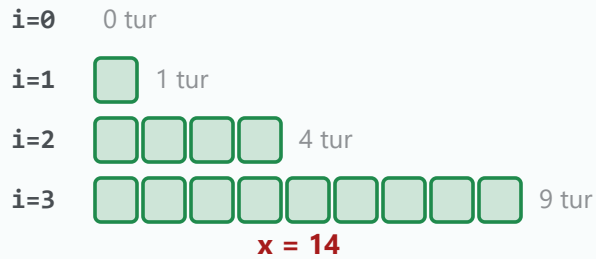
Adım 8 / 9

DEĞİŞKENLER

`n = 4` `i = 4` `j = –` `x = 14`

f2(4): Kodu Adım Adım İzle

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
4     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
▶   return x;
9 }
```



`return 14` = $0^2 + 1^2 + 2^2 + 3^2 = \Sigma\, i^2$. Satırlar i^2 uzadığı için toplam ≈ $n^3/3$ büyür: **$\Theta(n^3)$**.

Adım 9 / 9

DEĞİŞKENLER

n = **4** | i = **–** | j = **–** | x = **14**

DÖNGÜ ANALİZİ

f2(5): İç Döngü Kaç Kez Çalışır?

Her çubuk, dış döngünün bir turunda (sabit i) iç döngünün kaç kez çalıştığını gösterir. Çubukların toplam alanı f2'nin yaptığı `x++` sayısıdır.



toplam x++ = 0

i = 0 iken iç döngü **0** kez çalışır (0 × 0 = 0).

Adım 1 / 6

DÖNGÜ ANALİZİ

f2(5): İç Döngü Kaç Kez Çalışır?

Her çubuk, dış döngünün bir turunda (sabit i) iç döngünün kaç kez çalıştığını gösterir. Çubukların toplam alanı f2'nin yaptığı `x++` sayısıdır.



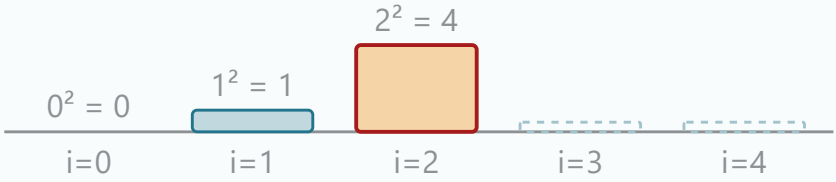
toplam x++ = 1

i = 1 iken iç döngü **1** kez çalışır (1 × 1 = 1).

DÖNGÜ ANALİZİ

f2(5): İç Döngü Kaç Kez Çalışır?

Her çubuk, dış döngünün bir turunda (sabit i) iç döngünün kaç kez çalıştığını gösterir. Çubukların toplam alanı f2'nin yaptığı `x++` sayısıdır.



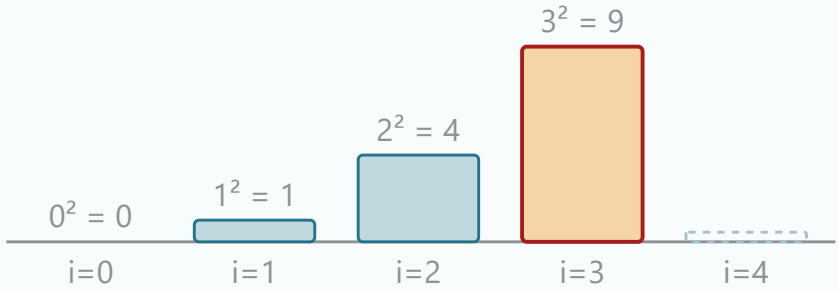
toplam x++ = 5

i = 2 iken iç döngü **4** kez çalışır (2 × 2 = 4).

DÖNGÜ ANALİZİ

f2(5): İç Döngü Kaç Kez Çalışır?

Her çubuk, dış döngünün bir turunda (sabit i) iç döngünün kaç kez çalıştığını gösterir. Çubukların toplam alanı f2'nin yaptığı `x++` sayısıdır.



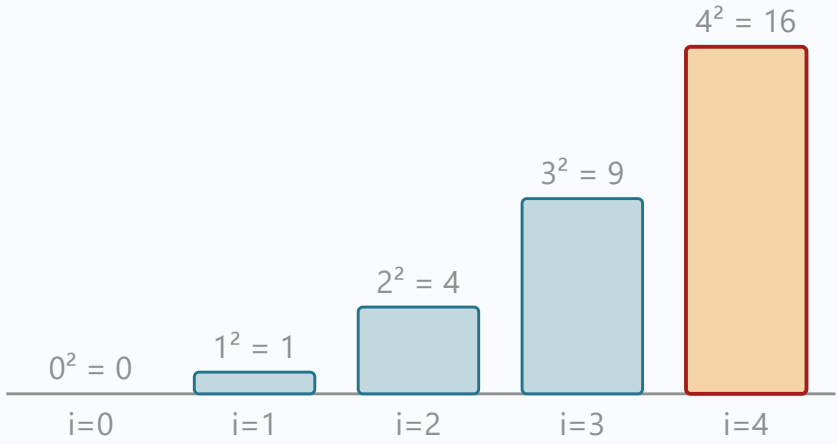
toplam x++ = 14

i = 3 iken iç döngü **9** kez çalışır (3 × 3 = 9).

DÖNGÜ ANALİZİ

f2(5): İç Döngü Kaç Kez Çalışır?

Her çubuk, dış döngünün bir turunda (sabit i) iç döngünün kaç kez çalıştığını gösterir. Çubukların toplam alanı f2'nin yaptığı `x++` sayısıdır.

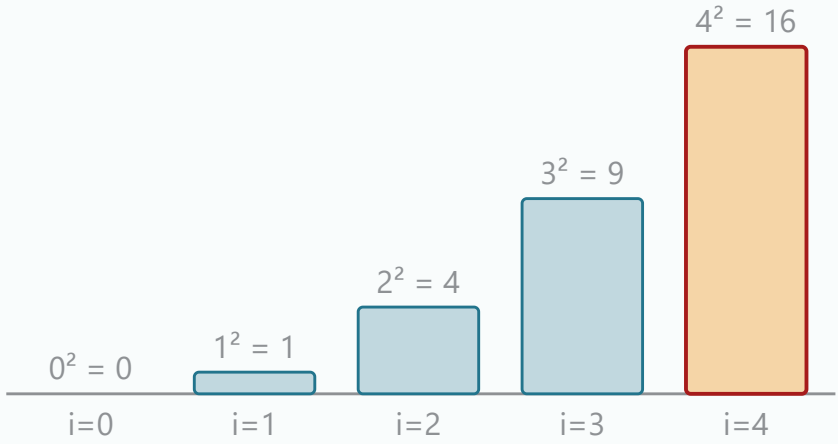


toplam x++ = 30

i = 4 iken iç döngü **16** kez çalışır. Genel olarak her i için iç döngü **i^2** kez çalışır.

f2(5): İç Döngü Kaç Kez Çalışır?

Her çubuk, dış döngünün bir turunda (sabit i) iç döngünün kaç kez çalıştığını gösterir. Çubukların toplam alanı f2'nin yaptığı `x++` sayısıdır.



toplam x++ = 30

Toplam: 0 + 1 + 4 + 9 + 16 = **30** = (n − 1) · n · (2n − 1) / 6 = 4 · 5 · 9 / 6. f2(5) = 30 döner.

f2: Toplamı Hesaplamak

İç döngünün toplam tur sayısı (i = 0'dan n − 1'e):

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

$$= (2n^3 - 3n^2 + n) / 6 \approx n^3 / 3$$

Baskın terim $n^3/3$'tür; sabit 1/3 ve düşük dereceli terimler atılır: **$T(n) = \Theta(n^3)$**.

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

Kontrol: n iki katına çıkınca (16 → 32) adım sayısı 10.416 / 1.240 ≈ **8,4** katına çıkar; $2^3 = 8$'e yakındır. Kübik büyümenin işareti budur.

DÖNGÜ ANALİZİ

Döngü Kalıpları ve Toplam Formülleri

Kalıp	Tur sayısı	Sonuç
`for (i = 0; i < n; i++)`	$\Sigma\ 1 = n$	$\Theta(n)$
`for i < n` içinde `for (j = 0; j < n; j++)`	$n \cdot n$	$\Theta(n^2)$
`for i < n` içinde `for (j = 0; j < i; j++)`	$\Sigma\ i = n(n-1)/2$	$\Theta(n^2)$
`for i < n` içinde `for (j = 0; j < i * i; j++)`	$\Sigma\ i^2 = (n-1)n(2n-1)/6$	$\Theta(n^3)$
`for (i = 1; i < n; i *= 2)`	$\lfloor \log_2 n \rfloor$	$\Theta(\log n)$
`for (i = n; i > 0; i /= 2)` içinde `for j < n`	$n \cdot (\lfloor \log_2 n \rfloor + 1)$	$\Theta(n \log n)$
`for (i = 1; i < n; i *= 2)` içinde `for j < i`	$1 + 2 + 4 + \ldots < 2n$	$\Theta(n)$
`for (i = 0; i < n; i = 1 << i)` (f6)	$\log^* n + 1$ kadar	$\Theta(\log^* n)$

Yöntem: her dış tur için iç döngünün tur sayısı yazılır, sonra bunlar toplanır. Toplam bilinen bir seriye (aritmetik, kareler, geometrik) indirgenir.

f6: Kaydırmalı Döngü

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

- **İlklendirme:** `int i = 0;` i'ye 0 atar; sabit zamanlı, O(1).
- **Koşul:** `i < n` her turda i'nin n'den küçük olduğunu denetler.
- **Güncelleme:** `i = 1 << i` 1'i i bit sola kaydırır; sonuç **2^i**'dir ve sabit zamanda hesaplanır.
- Güncelleme `i = 2 * i` değil, `i = 2^i` 'dir. i'nin **kendisi üs** olur; bu da i'yi çok daha hızlı büyütür.

DÖNGÜ ANALİZİ

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
▶   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



`x` sıfırlanır.

Adım 1 / 13

DEĞİŞKENLER

`n = 20` `x = 0` `i = –`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 0 < 20 → doğru

i = 0 ile başlanır; koşul 0 < 20 doğru.

DEĞİŞKENLER

n = 20 | x = 0 | i = 0

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
▶     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 0 < 20 → doğru

`x++` → x = 1.

Adım 3 / 13

DEĞİŞKENLER

`n = 20` `x = 1` `i = 0`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 1 < 20 → doğru

i = 1 << 0 = 1; koşul 1 < 20 doğru.

DEĞİŞKENLER

n = 20 | x = 1 | i = 1

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
▶     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 1 < 20 → doğru

`x++` → x = 2.

Adım 5 / 13

DEĞİŞKENLER

`n = 20` `x = 2` `i = 1`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 2 < 20 → doğru

i = 1 << 1 = 2; koşul 2 < 20 doğru.

Adım 6 / 13

DEĞİŞKENLER

`n = 20` `x = 2` `i = 2`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
▶     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 2 < 20 → doğru

`x++` → x = 3.

Adım 7 / 13

DEĞİŞKENLER

`n = 20` `x = 3` `i = 2`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 4 < 20 → doğru

i = 1 << 2 = 4; koşul 4 < 20 doğru.

Adım 8 / 13

DEĞİŞKENLER

n = 20 | x = 3 | i = 4

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
▶     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 4 < 20 → doğru

`x++` → x = 4.

Adım 9 / 13

DEĞİŞKENLER

`n = 20` `x = 4` `i = 4`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 16 < 20 → doğru

i = 1 << 4 = 16; koşul 16 < 20 doğru.

Adım 10 / 13

DEĞİŞKENLER

`n = 20` `x = 4` `i = 16`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
▶     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 16 < 20 → doğru

`x++` → x = 5.

Adım 11 / 13

DEĞİŞKENLER

`n = 20` `x = 5` `i = 16`

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 65536 < 20 → yanlış, döngü biter

i = 1 << 16 = 65536; koşul 65536 < 20 yanlış → döngü biter.

Adım 12 / 13

DEĞİŞKENLER

n = 20 | x = 5 | i = 65536

f6(20): Kodu Adım Adım İzle

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i == 2^i
4   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
▶   return x;
8 }
```

i değerleri (n = 20); i ← 1 << i



koşul: 65536 < 20 → yanlış, döngü biter

`return x` → 5. i değerleri 0, 1, 2, 4, 16: yalnızca 5 tur.

Adım 13 / 13

DEĞİŞKENLER

`n = 20` `x = 5` `i = –`

f6: Gerçekte Kaç Tur Döner?

i değerleri: $0 \to 2^0 = 1 \to 2^1 = 2 \to 2^2 = 4 \to 2^4 = 16 \to 2^{16} = 65536$. Her tur i'yi **üs** yapar (2'nin kuvvet kulesi).

Sık görülen akıl yürütme: "k adım sonra $2^k \geq n$, öyleyse $k \geq \log_2 n$" — bu, `i = 2 * i` döngüsü için doğrudur. f6'da ise k tur sonra i, k − 1 katlı bir kule olur; tur sayısı **log* n** (yinelenmiş logaritma) kadardır.

O(log n) bir **üst sınır** olarak doğrudur, ama sıkı değildir: **Θ(log* n)**.

n	f6(n)	$\lceil \log_2 n \rceil$
4	3	2
16	4	4
32	5	5
1000	5	10
65536	5	16

Java'da tuzak: `int` kaydırma miktarı 32'ye göre mod alınır: `1 << 65536` = 1. Bu yüzden n > 65536 için i 1, 2, 4, 16, 65536 döngüsüne girer ve f6 **hiç bitmez**. "Taşmayı yok say" varsayımı burada önemlidir.

ÖZYİNELEMELİ ANALİZ

Yineleme Bağıntısı (Recurrence)

Özyinelemeli bir fonksiyonun süresi, **kendi daha küçük girdilerdeki süresi** cinsinden yazılır:

$$T(n) = (\text{alt çağrıların süreleri}) + (\text{çağrı dışında yapılan iş}), \quad T(\text{taban}) = \Theta(1)$$

Açma (yerine koyma)

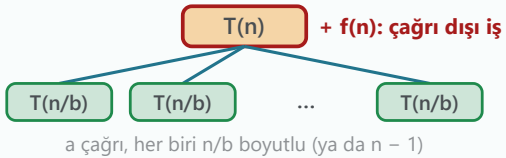
Bağıntıyı kendi içine tekrar tekrar koyup bir örüntü bulmak; taban duruma ulaşınca toplamak. (f3)

Özyineleme ağacı

Her çağrıyı bir düğüm olarak çizip her düzeydeki işi toplamak. (f3, f4, f5)

Ana Teorem

T(n) = aT(n/b) + f(n) biçimindeki böl-ve-fethet bağıntılarını doğrudan çözmek. (f4, f5)



Bağıntı kurarken sorular: Kaç özyinelemeli çağrı var? Her çağrının girdisi ne kadar küçülüyor (n − 1 mi, n/2 mi)? Çağrılar dışında ne kadar iş yapılıyor?

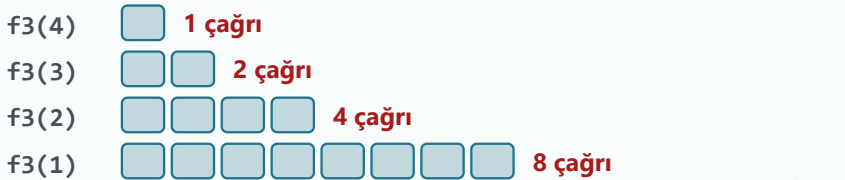
ÖZYİNELEMELİ ANALİZ

f3: İki Çağrı, n − 1 → $O(2^n)$

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

- **Taban durum (base case):** n ≤ 1 iken 1 döner; sabit zamanlı, O(1).
- **Özyinelemeli durum (recursive case):** n > 1 iken iki kez `f3(n - 1)` çağrılır ve sonuçlar toplanır (sabit iş c).

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

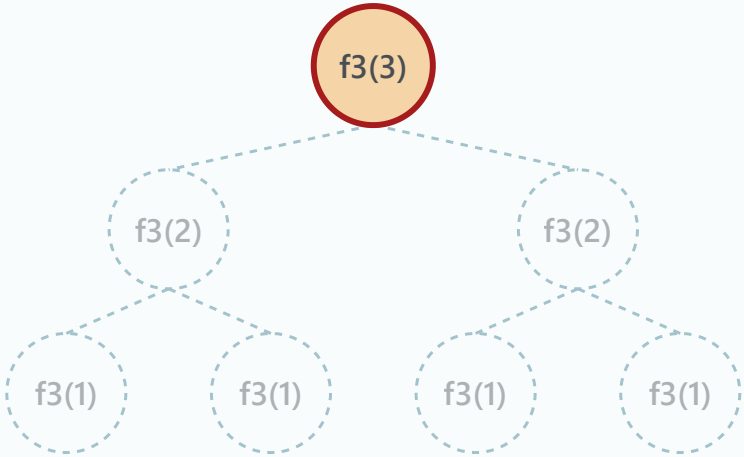


girdi 1 azalır, çağrı sayısı 2 katına çıkar: 1 + 2 + 4 + 8 = 15 = $2^4 - 1$

Girdi her çağrıda yalnızca **1 azalır** ama çağrı sayısı her düzeyde **ikiye katlanır**. Bu bağıntı T(n/b) biçiminde olmadığı için Ana Teorem uygulanamaz; açma yöntemiyle çözülür.

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
▶   if (n <= 1) {
3     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(3)` çağrıldı: 3 > 1, taban durum değil.

Adım 1 / 10

DEĞİŞKENLER

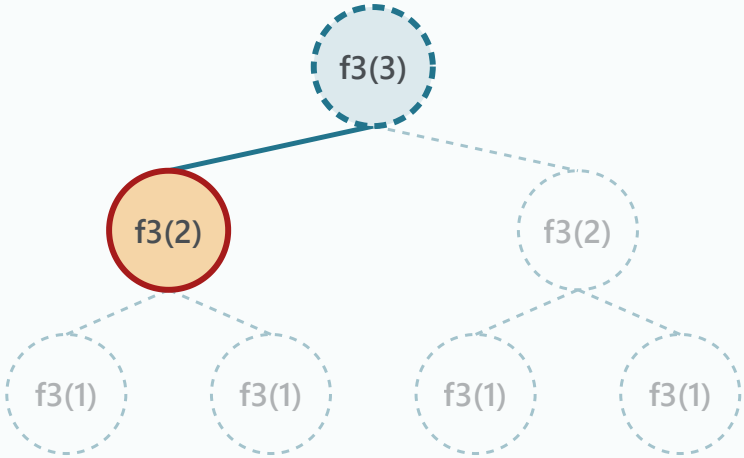
`n = 3` `sonuç = –` `çağrı = 1`

ÇAĞRI YIĞINI

`f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



Satır 5: ilk `f3(2)` çağrısı. f3(3) yığında bekliyor.

Adım 2 / 10

DEĞİŞKENLER

`n = 2` `sonuç = –` `çağrı = 2`

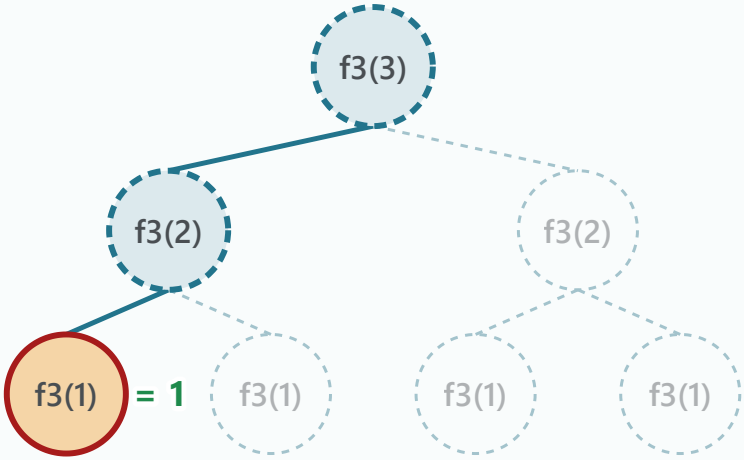
ÇAĞRI YIĞINI

`f3(2)`

`f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(1)` : n ≤ 1 → taban durum, 1 döner.

Adım 3 / 10

DEĞİŞKENLER

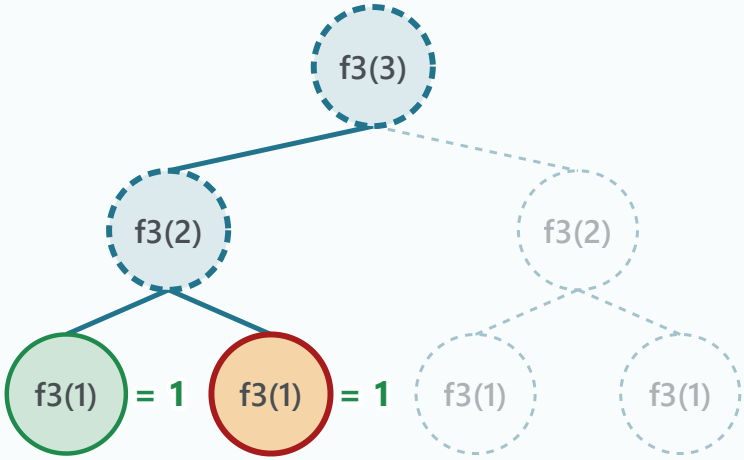
`n = 1` `sonuç = 1` `çağrı = 3`

ÇAĞRI YIĞINI

- `f3(1)`
- `f3(2)`
- `f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



f3(2) ikinci çağrısını yapar: yine `f3(1)` = 1.

Adım 4 / 10

DEĞİŞKENLER

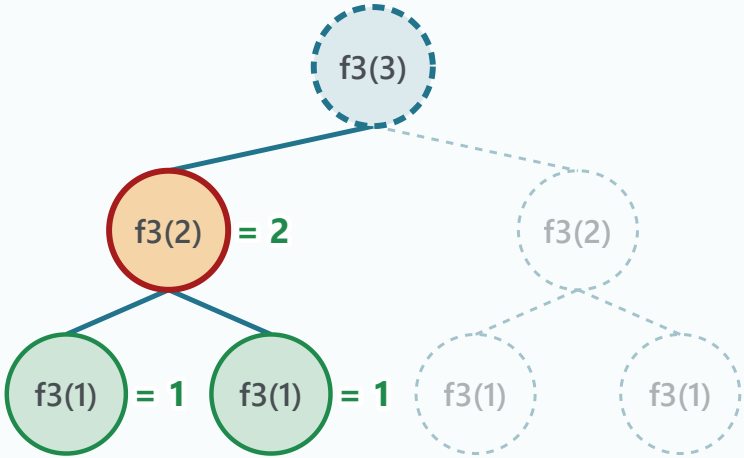
`n = 1` `sonuç = 1` `çağrı = 4`

ÇAĞRI YIĞINI

- `f3(1)`
- `f3(2)`
- `f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



f3(2) = 1 + 1 = 2 döner. Sol alt ağaç bitti.

Adım 5 / 10

DEĞİŞKENLER

`n = 2` `sonuç = 2` `çağrı = 4`

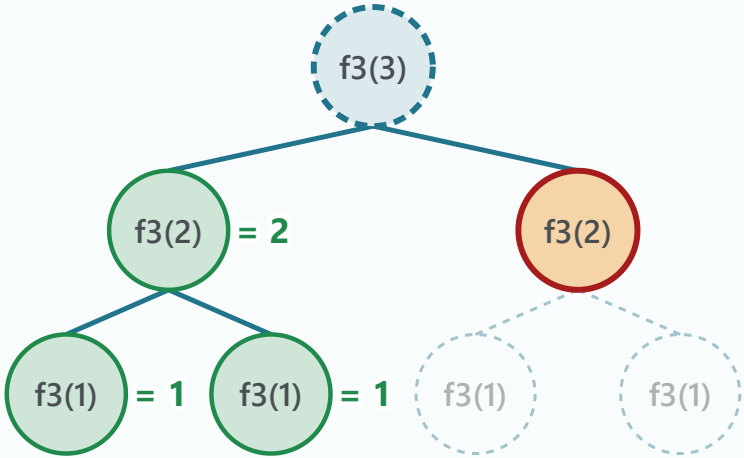
ÇAĞRI YIĞINI

`f3(2)`

`f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



f3(3), ikinci `f3(2)` 'yi çağırır: aynı iş **baştan** yapılır.

Adım 6 / 10

DEĞİŞKENLER

`n = 2` `sonuç = –` `çağrı = 5`

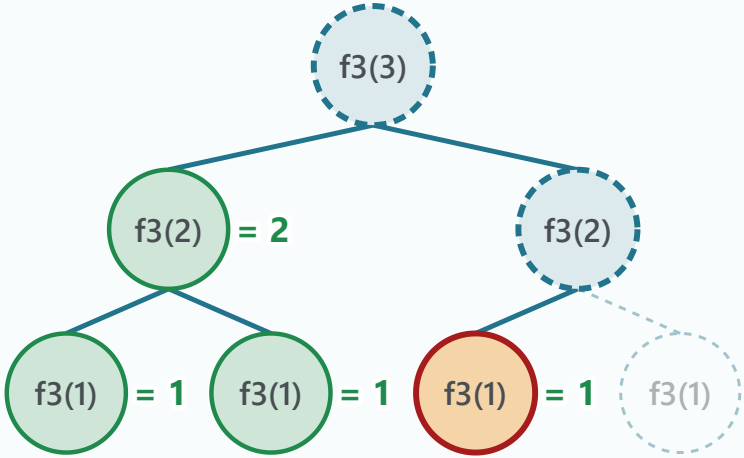
ÇAĞRI YIĞINI

`f3(2)`

`f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(1)` = 1.

Adım 7 / 10

DEĞİŞKENLER

`n = 1` `sonuç = 1`

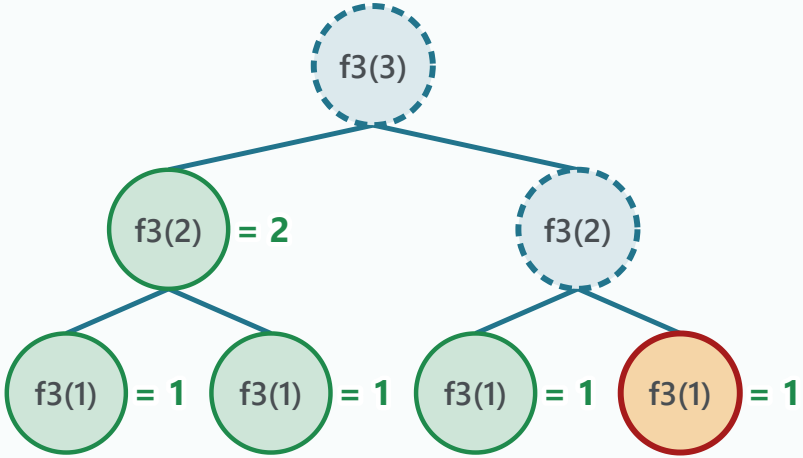
`çağrı = 6`

ÇAĞRI YIĞINI

- `f3(1)`
- `f3(2)`
- `f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(1)` = 1.

Adım 8 / 10

DEĞİŞKENLER

`n = 1` `sonuç = 1`

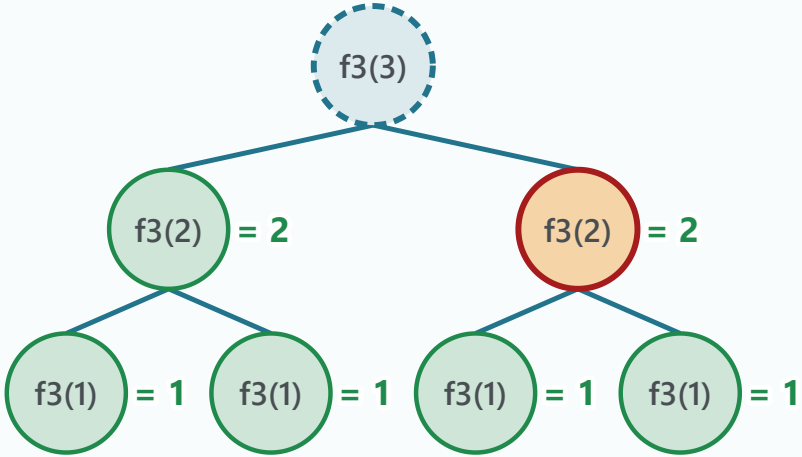
`çağrı = 7`

ÇAĞRI YIĞINI

- `f3(1)`
- `f3(2)`
- `f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



İkinci f3(2) = 2 döner.

Adım 9 / 10

DEĞİŞKENLER

`n = 2` `sonuç = 2` `çağrı = 7`

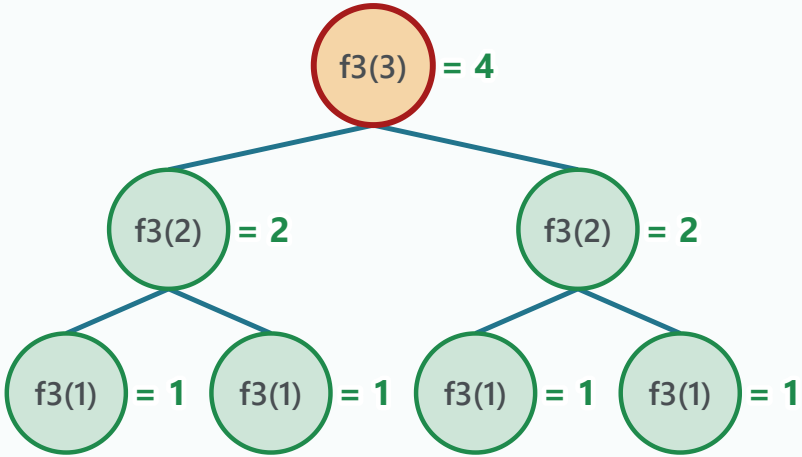
ÇAĞRI YIĞINI

`f3(2)`

`f3(3)`

f3(3): Çağrıları Adım Adım İzle

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



f3(3) = 2 + 2 = 4 döner. Toplam **7 = 2^3 − 1** çağrı; yığın derinliği en çok 3.

Adım 10 / 10

DEĞİŞKENLER

`n = 3` `sonuç = 4` `çağrı = 7`

ÇAĞRI YIĞINI

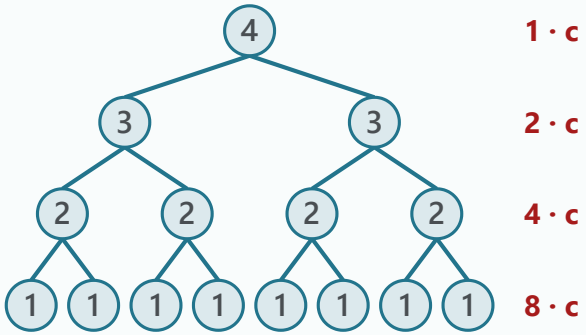
`f3(3)`

f3: Bağıntıyı Açma

$$T(n) = 2T(n-1) + c$$

Bağıntının kendisi: iki alt çağrı, sabit ek iş.

Adım 1 / 5



Java ile sayıldığında: f3(16) 65.535, f3(32) 4.294.967.295 çağrı yapar ($2^n - 1$).

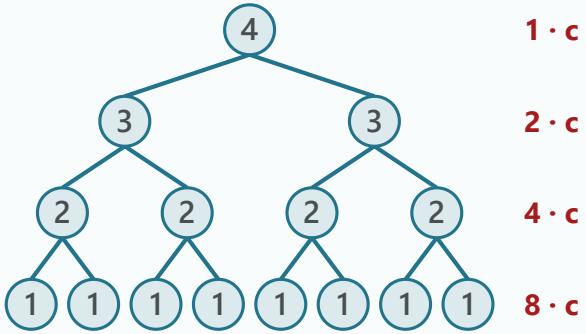
f3: Bağıntıyı Açma

$$T(n) = 2T(n-1) + c$$

Bağıntının kendisi: iki alt çağrı, sabit ek iş.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

T(n − 1) yerine kendi tanımını koyarız.



Java ile sayıldığında: f3(16) 65.535, f3(32) 4.294.967.295 çağrı yapar ($2^n - 1$).

f3: Bağıntıyı Açma

$$T(n) = 2T(n-1) + c$$

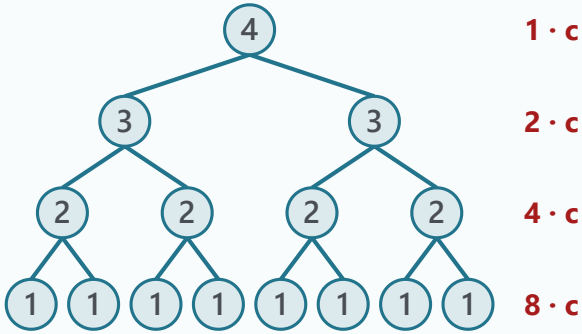
Bağıntının kendisi: iki alt çağrı, sabit ek iş.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

T(n − 1) yerine kendi tanımını koyarız.

$$= 8T(n-3) + 7c$$

Bir kez daha açınca örüntü belirir: katsayılar 2, 4, 8 …



Java ile sayıldığında: f3(16) 65.535, f3(32) 4.294.967.295 çağrı yapar ($2^n - 1$).

f3: Bağıntıyı Açma

$$T(n) = 2T(n-1) + c$$

Bağıntının kendisi: iki alt çağrı, sabit ek iş.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

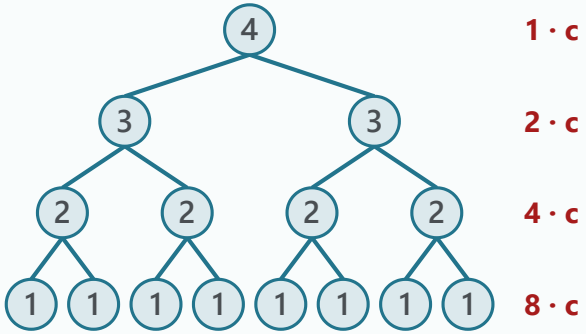
T(n − 1) yerine kendi tanımını koyarız.

$$= 8T(n-3) + 7c$$

Bir kez daha açınca örüntü belirir: katsayılar 2, 4, 8 …

$$= 2^k\, T(n-k) + (2^k - 1)c$$

k adım sonra genel biçim. Ek iş (c) ihmal edilirse *T(n) = 2ᵏ T(n − k)* yazılır; büyüme sınıfı değişmez.



Java ile sayıldığında: f3(16) 65.535, f3(32) 4.294.967.295 çağrı yapar ($2^n - 1$).

f3: Bağıntıyı Açma

$$T(n) = 2T(n-1) + c$$

Bağıntının kendisi: iki alt çağrı, sabit ek iş.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

T(n − 1) yerine kendi tanımını koyarız.

$$= 8T(n-3) + 7c$$

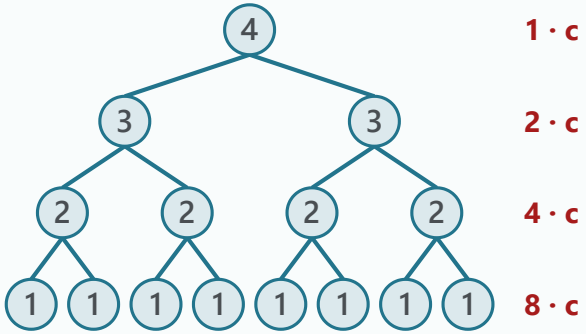
Bir kez daha açınca örüntü belirir: katsayılar 2, 4, 8 …

$$= 2^k\, T(n-k) + (2^k - 1)c$$

k adım sonra genel biçim. Ek iş (c) ihmal edilirse *T(n) = 2ᵏ T(n − k)* yazılır; büyüme sınıfı değişmez.

$$k = n-1 : T(n) = 2^{n-1}c + (2^{n-1} - 1)c = (2^n - 1)c$$

Taban durum T(1) = c'ye n − k = 1 iken ulaşılır. Sonuç: **T(n) = Θ(2ⁿ)**.



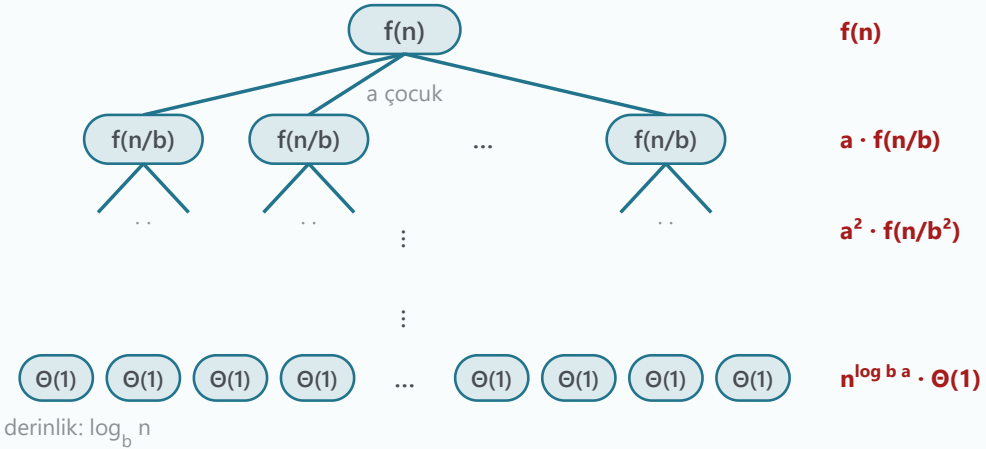
Java ile sayıldığında: f3(16) 65.535, f3(32) 4.294.967.295 çağrı yapar ($2^n - 1$).

Ana Teorem: Genel Biçim

Böl ve fethet (divide and conquer) algoritmalarının bağıntılarını doğrudan çözmek için kullanılır:

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

a: alt problem (çağrı) sayısı · **b:** küçülme oranı, alt problem boyutu **n/b** · **f(n):** bölme ve birleştirme işi. Derinlik $\log_b n$, yaprak sayısı $a^{\log_b n} = n^{\log_b a}$.



Ana Teorem: Üç Durum

f(n), yaprakların toplam işi olan $\mathbf{n^{\log_b a}}$ ile karşılaştırılır. c = f(n)'nin polinom derecesi:

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

Durum 2 ve 3'te yalnızca $f(n) = O(n^c)$ yazmak yetmez. Örneğin $f(n) = 1$ de $O(n^2)$'dir, ama $T(n) = T(n/2) + 1$'in çözümü $\Theta(n^2)$ olamaz. Bu yüzden Durum 2'de $f = \mathbf{\Theta}(n^c)$, Durum 3'te $f = \mathbf{\Omega}(n^c)$ ve **düzenlilik koşulu** istenir; sonuçlar Θ ile sıkıdır.

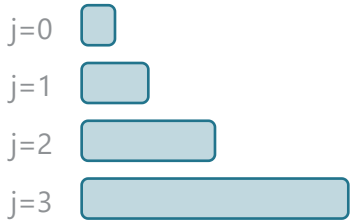
Düzenlilik koşulu (regularity), $f(n) = n^c$ gibi polinomlar için kendiliğinden sağlanır: $a(n/b)^c = (a/b^c)\, n^c$ ve $a/b^c < 1$.

ANA TEOREM

Neden Üç Durum? Düzey Toplamları

$f(n) = n^c$ ise ağacın j. düzeyinde a^j düğüm, her biri $(n/b^j)^c$ iş yapar: **düzey toplamı = $n^c \cdot (a/b^c)^j$**. Toplam, oranı $r = a/b^c$ olan bir geometrik seridir.

$4T(n/2) + n$



$r = 2 > 1$: artar

Durum 1: son düzey (yapraklar) baskın → $\Theta(n^2)$

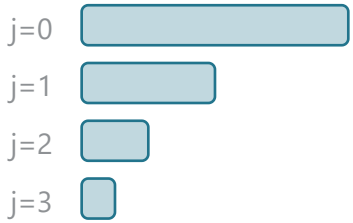
$2T(n/2) + n$



$r = 1$: sabit

Durum 2: log n düzey × n iş → $\Theta(n \log n)$

$2T(n/2) + n^2$



$r = 1/2 < 1$: azalır

Durum 3: kök baskın → $\Theta(n^2)$

$r > 1 \Leftrightarrow b^c < a \Leftrightarrow c < \log_b a$. Yani Ana Teoremin üç durumu, geometrik serinin **artan**, **sabit** ya da **azalan** olmasına karşılık gelir.

ANA TEOREM

Örnek: T(n) = 2T(n/2) + Θ(n)

$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$

Parametreleri okuyalım: iki alt problem, her biri n/2 boyutunda, doğrusal birleştirme işi.

Adım 1 / 5

Bu bağıntı nerede çıkar?

- **Birleştirme sıralaması** (merge sort): diziyi ikiye böl, iki yarıyı sırala, Θ(n)'de birleştir.
- Bölümdeki `f5` fonksiyonu: `f1(n)` + iki `f5(n/2)` çağrısı.
- Hızlı sıralamanın (quicksort) dengeli bölme durumu.

Örnek: T(n) = 2T(n/2) + Θ(n)

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$$

Parametreleri okuyalım: iki alt problem, her biri n/2 boyutunda, doğrusal birleştirme işi.

$$\log_b a = \log_2 2 = 1$$

Yaprak işinin üssü 1'dir: $n^{\log_2 2} = n$.

Bu bağıntı nerede çıkar?

- **Birleştirme sıralaması** (merge sort): diziyi ikiye böl, iki yarıyı sırala, Θ(n)'de birleştir.
- Bölümdeki `f5` fonksiyonu: `f1(n)` + iki `f5(n/2)` çağrısı.
- Hızlı sıralamanın (quicksort) dengeli bölme durumu.

ANA TEOREM

Örnek: T(n) = 2T(n/2) + Θ(n)

$$a = 2,\quad b = 2,\quad f(n) = \Theta(n)$$

Parametreleri okuyalım: iki alt problem, her biri n/2 boyutunda, doğrusal birleştirme işi.

$$\log_b a = \log_2 2 = 1$$

Yaprak işinin üssü 1'dir: $n^{\log_2 2} = n$.

$$f(n) = \Theta(n^1) \rightarrow c = 1$$

f(n)'nin derecesi c = 1.

Bu bağıntı nerede çıkar?

- **Birleştirme sıralaması** (merge sort): diziyi ikiye böl, iki yarıyı sırala, Θ(n)'de birleştir.
- Bölümdeki `f5` fonksiyonu: `f1(n)` + iki `f5(n/2)` çağrısı.
- Hızlı sıralamanın (quicksort) dengeli bölme durumu.

Örnek: T(n) = 2T(n/2) + Θ(n)

$$a = 2,\quad b = 2,\quad f(n) = \Theta(n)$$

Parametreleri okuyalım: iki alt problem, her biri n/2 boyutunda, doğrusal birleştirme işi.

$$\log_b a = \log_2 2 = 1$$

Yaprak işinin üssü 1'dir: $n^{\log_2 2} = n$.

$$f(n) = \Theta(n^1) \rightarrow c = 1$$

f(n)'nin derecesi c = 1.

$$c = \log_b a \rightarrow \text{Durum } 2$$

Üsler eşit: her düzey aynı (n) işi yapar.

Bu bağıntı nerede çıkar?

- **Birleştirme sıralaması** (merge sort): diziyi ikiye böl, iki yarıyı sırala, Θ(n)'de birleştir.
- Bölümdeki `f5` fonksiyonu: `f1(n)` + iki `f5(n/2)` çağrısı.
- Hızlı sıralamanın (quicksort) dengeli bölme durumu.

Örnek: T(n) = 2T(n/2) + Θ(n)

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$$

Parametreleri okuyalım: iki alt problem, her biri n/2 boyutunda, doğrusal birleştirme işi.

$$\log_b a = \log_2 2 = 1$$

Yaprak işinin üssü 1'dir: $n^{\log_2 2} = n$.

$$f(n) = \Theta(n^1) \rightarrow c = 1$$

f(n)'nin derecesi c = 1.

$$c = \log_b a \rightarrow \text{Durum 2}$$

Üsler eşit: her düzey aynı (n) işi yapar.

$$T(n) = \Theta(n \log n)$$

Durum 2: $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \mathbf{\Theta(n \log n)}$.

Bu bağıntı nerede çıkar?

- **Birleştirme sıralaması** (merge sort): diziyi ikiye böl, iki yarıyı sırala, Θ(n)'de birleştir.
- Bölümdeki `f5` fonksiyonu: `f1(n)` + iki `f5(n/2)` çağrısı.
- Hızlı sıralamanın (quicksort) dengeli bölme durumu.

Ana Teorem: Örnekler

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

Durum 3 satırlarında düzenlilik: $2(n/2)^2 = n^2/2 \le ½\, n^2$ ve $1 \cdot (n/2) \le ½\, n$, $k = ½$ ile sağlanır.

ANA TEOREM

Ana Teorem Ne Zaman Uygulanamaz?

Çıkarmalı bağıntılar

$T(n) = 2T(n - 1) + c$ (f3) ya da $T(n) = T(n - 1) + n$: alt problem n/b değil, $n - 1$. Açma yöntemi kullanılır.

Polinom olmayan fark

$T(n) = 2T(n/2) + n / \log n$: f(n), n'den yalnızca log çarpanı kadar küçük; hiçbir duruma girmez.

Eşit olmayan bölme

$T(n) = T(n/3) + T(2n/3) + n$: alt problemler farklı boyutta. Özyineleme ağacı (ya da Akra–Bazzi) ile $\Theta(n \log n)$.

Sabit olmayan a ya da a < 1

$T(n) = 2^n T(n/2) + n$ ya da $T(n) = ½ T(n/2) + n$: teoremin $a \geq 1$ sabit varsayımı bozulur.

Temel biçimin bir genişlemesi: $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ ise $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$. Örneğin $T(n) = 2T(n/2) + n \log n = \Theta(n \log^2 n)$.

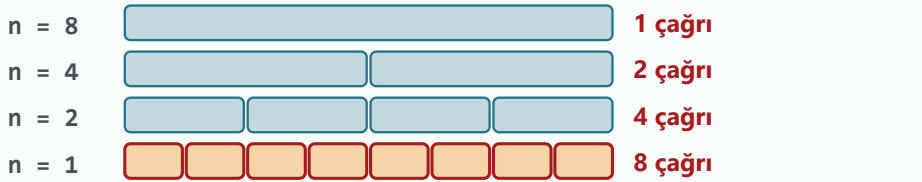
UYGULAMA: F4 VE F5

f4: İki Çağrı, n/2 → O(n)

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

- **Taban durum:** n ≤ 1 iken 1 döner; O(1).
- **Özyinelemeli durum:** n > 1 iken iki kez `f4(n/2)` çağrılır, sonuçlar toplanır (sabit iş).

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

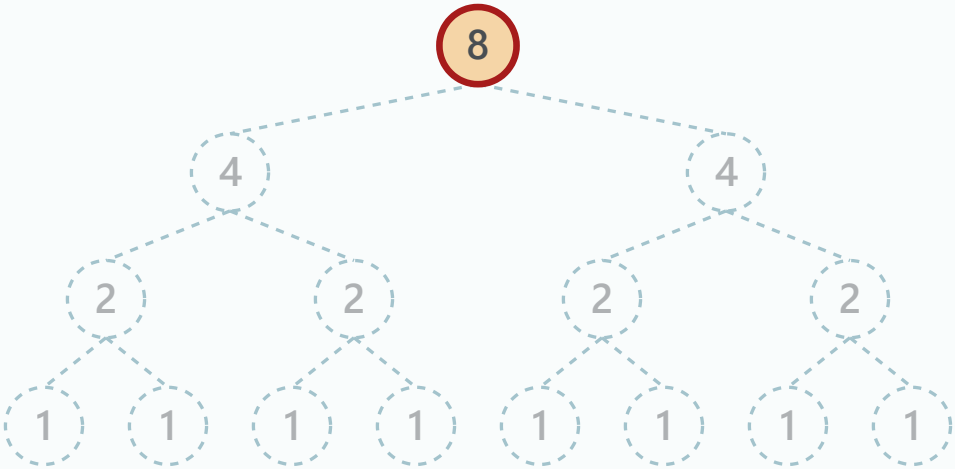


girdi yarıya iner: yalnızca $\log_2 8 + 1 = 4$ düzey, $1 + 2 + 4 + 8 = 15 = 2n - 1$ çağrı

f3 ile tek fark girdinin **n − 1** yerine **n/2** olmasıdır; ama bu, sonucu $\Theta(2^n)$'den **$\Theta(n)$**'ye indirir. İki çağrı her zaman üstel süre demek değildir: girdinin ne kadar küçüldüğü belirleyicidir.

UYGULAMA: F4 VE F5

f4(8): Özyineleme Ağacı



1 · c

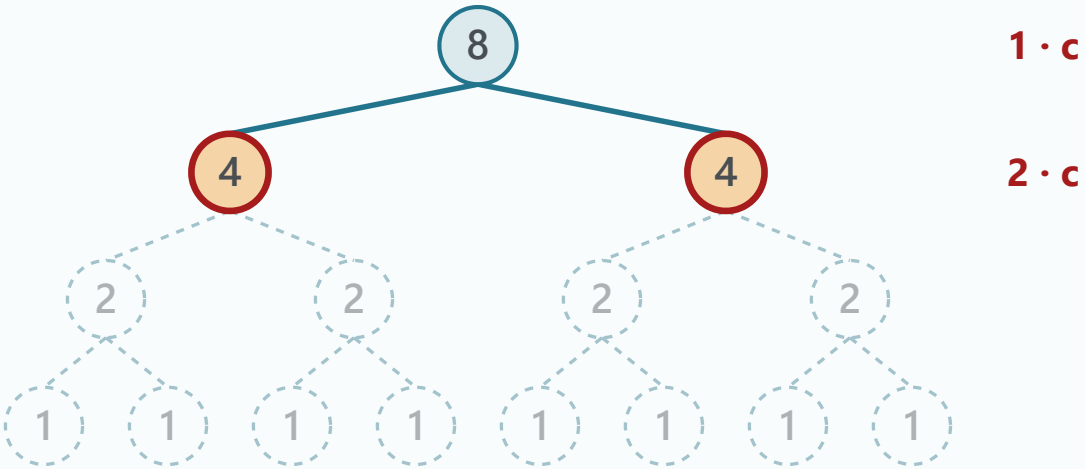
Kök: `f4(8)` yalnızca sabit iş (c) yapar ve iki çağrı başlatır.

Adım 1 / 4

Düzey toplamları geometrik olarak artar; toplamı son terim (yapraklar) belirler. Java ile: f4(16) 31, f4(32) 63 çağrı yapar (2n − 1).

UYGULAMA: F4 VE F5

f4(8): Özyineleme Ağacı



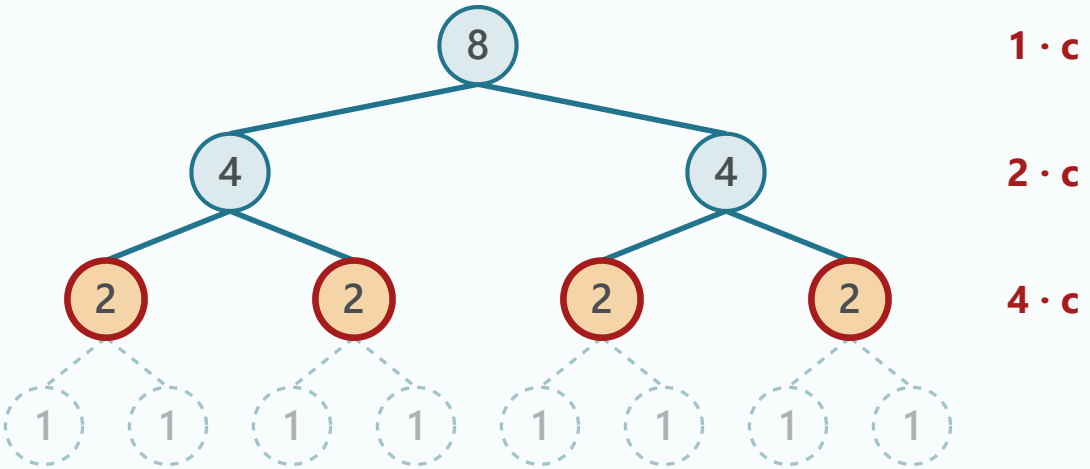
Düzey 1: iki `f4(4)` çağrısı, toplam 2c.

Adım 2 / 4

Düzey toplamları geometrik olarak artar; toplamı son terim (yapraklar) belirler. Java ile: f4(16) 31, f4(32) 63 çağrı yapar (2n − 1).

UYGULAMA: F4 VE F5

f4(8): Özyineleme Ağacı



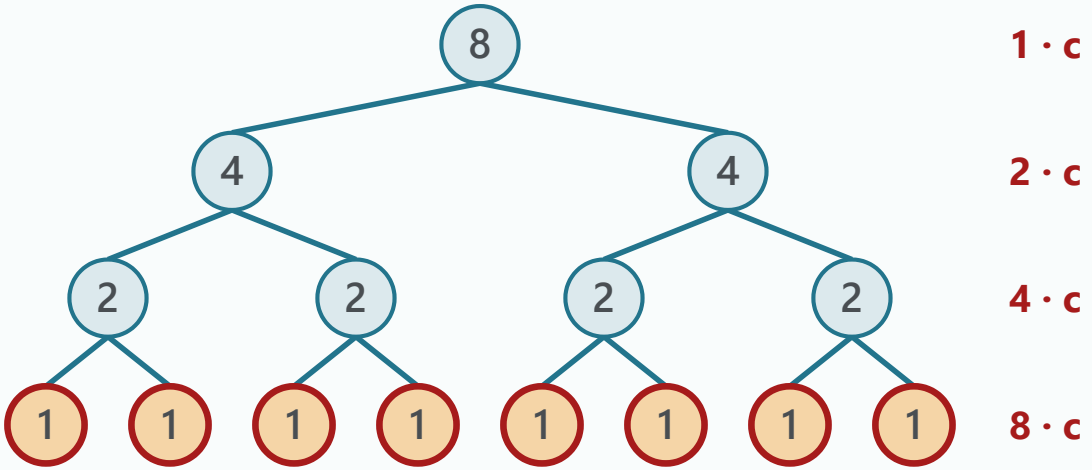
Düzey 2: dört `f4(2)` çağrısı, toplam 4c. Düzey işi her adımda **ikiye katlanıyor**.

Adım 3 / 4

Düzey toplamları geometrik olarak artar; toplamı son terim (yapraklar) belirler. Java ile: f4(16) 31, f4(32) 63 çağrı yapar (2n − 1).

UYGULAMA: F4 VE F5

f4(8): Özyineleme Ağacı



Yapraklar: sekiz `f4(1)`, toplam 8c = n · c. Toplam c(1 + 2 + 4 + 8) = 15c = (2n − 1)c → **Θ(n)**.

Adım 4 / 4

Düzey toplamları geometrik olarak artar; toplamı son terim (yapraklar) belirler. Java ile: f4(16) 31, f4(32) 63 çağrı yapar (2n − 1).

UYGULAMA: F4 VE F5

f4: Ana Teorem ile Çözüm

$$T(n) = 2T(n/2) + O(1)$$

Genel biçim T(n) = aT(n/b) + f(n) ile eşleştirelim.

Adım 1 / 5

Ağaç ile aynı sonuç

Özyineleme ağacında 2n − 1 düğüm vardı. Ana Teorem bu toplamı hesaplamadan aynı sonucu verir: yapraklar baskın.

Not: f4 için "O(log n)" demek sık yapılan bir hatadır; doğrusu Θ(n)'dir. log n, yalnızca **yığın derinliğidir** (bellek).

f4: Ana Teorem ile Çözüm

$$T(n) = 2T(n/2) + O(1)$$

Genel biçim T(n) = aT(n/b) + f(n) ile eşleştirelim.

$$a = 2, b = 2, f(n) = O(1)$$

Örnekte a = 2, b = 2, f(n) = O(1).

Ağaç ile aynı sonuç

Özyineleme ağacında 2n − 1 düğüm vardı. Ana Teorem bu toplamı hesaplamadan aynı sonucu verir: yapraklar baskın.

Not: f4 için "O(log n)" demek sık yapılan bir hatadır; doğrusu Θ(n)'dir. log n, yalnızca **yığın derinliğidir** (bellek).

f4: Ana Teorem ile Çözüm

$$T(n) = 2T(n/2) + O(1)$$

Genel biçim T(n) = aT(n/b) + f(n) ile eşleştirelim.

$$a = 2, b = 2, f(n) = O(1)$$

Örnekte a = 2, b = 2, f(n) = O(1).

$$\log_b a = \log_2 2 = 1$$

Yaprak işi $n^{\log_2 2} = n^1 = n$.

Ağaç ile aynı sonuç

Özyineleme ağacında 2n − 1 düğüm vardı. Ana Teorem bu toplamı hesaplamadan aynı sonucu verir: yapraklar baskın.

Not: f4 için "O(log n)" demek sık yapılan bir hatadır; doğrusu Θ(n)'dir. log n, yalnızca **yığın derinliğidir** (bellek).

f4: Ana Teorem ile Çözüm

$$T(n) = 2T(n/2) + O(1)$$

Genel biçim T(n) = aT(n/b) + f(n) ile eşleştirelim.

$$a = 2, b = 2, f(n) = O(1)$$

Örnekte a = 2, b = 2, f(n) = O(1).

$$\log_b a = \log_2 2 = 1$$

Yaprak işi $n^{\log_2 2} = n^1 = n$.

$$f(n) = O(n^0) \rightarrow c = 0 < 1$$

f(n) = O(1), c = 0'a karşılık gelir; c, $\log_b a$ = 1'den küçüktür.

Ağaç ile aynı sonuç

Özyineleme ağacında 2n − 1 düğüm vardı. Ana Teorem bu toplamı hesaplamadan aynı sonucu verir: yapraklar baskın.

Not: f4 için "O(log n)" demek sık yapılan bir hatadır; doğrusu Θ(n)'dir. log n, yalnızca **yığın derinliğidir** (bellek).

f4: Ana Teorem ile Çözüm

$$T(n) = 2T(n/2) + O(1)$$

Genel biçim T(n) = aT(n/b) + f(n) ile eşleştirelim.

$$a = 2, b = 2, f(n) = O(1)$$

Örnekte a = 2, b = 2, f(n) = O(1).

$$\log_b a = \log_2 2 = 1$$

Yaprak işi $n^{\log_2 2} = n^1 = n$.

$$f(n) = O(n^0) \to c = 0 < 1$$

f(n) = O(1), c = 0'a karşılık gelir; c, $\log_b a = 1$'den küçüktür.

$$\text{Durum 1: } T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Ana Teoreme göre $f(n) = O(n^c)$, $c < \log_b a$ ise $T(n) = \Theta(n^{\log_b a})$. Sonuç: **Θ(n)**.

Adım 5 / 5

Ağaç ile aynı sonuç

Özyineleme ağacında 2n − 1 düğüm vardı. Ana Teorem bu toplamı hesaplamadan aynı sonucu verir: yapraklar baskın.

Not: f4 için "O(log n)" demek sık yapılan bir hatadır; doğrusu Θ(n)'dir. log n, yalnızca **yığın derinliğidir** (bellek).

UYGULAMA: F4 VE F5

f5: Doğrusal İş + İki Çağrı → O(n log n)

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

- **Taban durum:** n ≤ 1 iken 1 döner; O(1).
- **Özyinelemeli durum:** n > 1 iken önce `f1(n)` çağrılır (Θ(n) iş), sonra iki kez `f5(n/2)`.

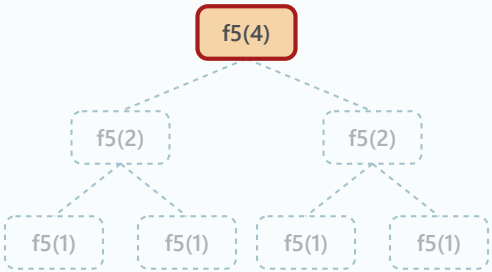
$$T(n) = 2T(n/2) + f1(n) = 2T(n/2) + \Theta(n)$$

f4'ten tek farkı her çağrıda yapılan **doğrusal iş**tir. Bu, birleştirme sıralamasının bağıntısıyla aynıdır. Sonuç "O(n + log n)" değil, **Θ(n log n)**'dir.

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
▶   if (n <= 1) {
3     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```

düzey işi



`f5(4)` çağrıldı: 4 > 1, taban durum değil.

Adım 1 / 13

DEĞİŞKENLER

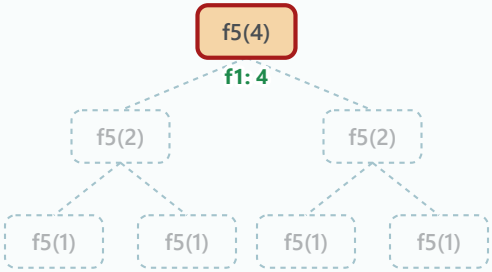
- `n = 4`
- `f1(n) = –`
- `sonuç = –`

ÇAĞRI YIĞINI

`f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



düzey işi
4

Satır 5 önce `f1(4)` 'ü hesaplar: 4 turluk döngü, **4** birim iş. Sonra iki `f5(2)` çağrısı.

Adım 2 / 13

DEĞİŞKENLER

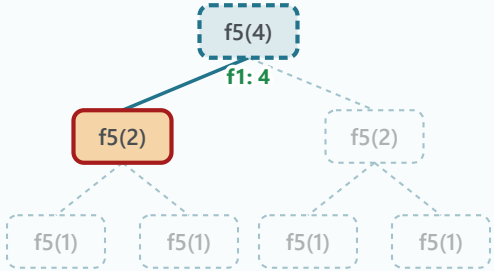
- `n = 4`
- `f1(n) = 4`
- `sonuç = –`

ÇAĞRI YIĞINI

`f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
▶   if (n <= 1) {
3     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



düzey işi
4

`f5(2)` çağrıldı: 2 > 1, taban durum değil.

Adım 3 / 13

DEĞİŞKENLER

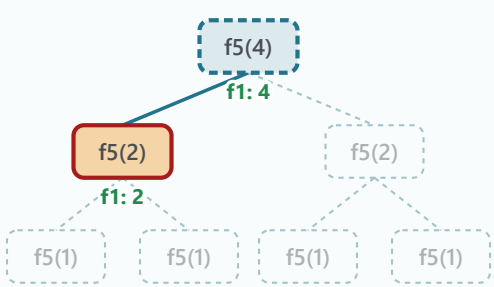
- `n = 2`
- `f1(n) = –`
- `sonuç = –`

ÇAĞRI YIĞINI

- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



Satır 5 önce `f1(2)`'ü hesaplar: 2 turluk döngü, **2** birim iş. Sonra iki `f5(1)` çağrısı.

Adım 4 / 13

DEĞİŞKENLER

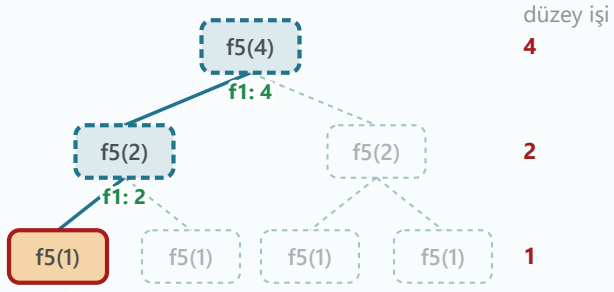
- `n = 2`
- `f1(n) = 2`
- `sonuç = –`

ÇAĞRI YIĞINI

- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → taban durum, 1 döner.

Adım 5 / 13

DEĞİŞKENLER

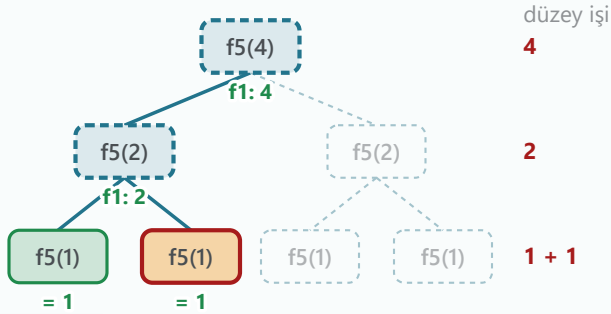
- `n = 1`
- `f1(n) = –`
- `sonuç = 1`

ÇAĞRI YIĞINI

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → taban durum, 1 döner.

Adım 6 / 13

DEĞİŞKENLER

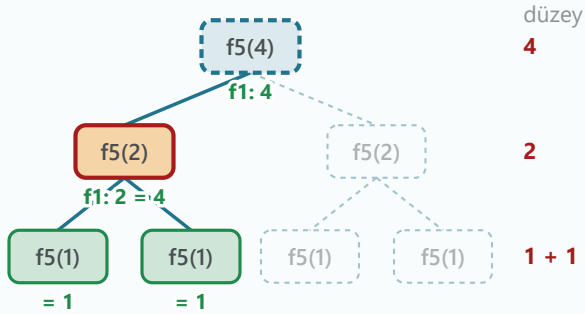
- `n = 1`
- `f1(n) = –`
- `sonuç = 1`

ÇAĞRI YIĞINI

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



f5(2) = f1(2) + f5(1) + f5(1) = 2 + 1 + 1 = **4** döner.

Adım 7 / 13

DEĞİŞKENLER

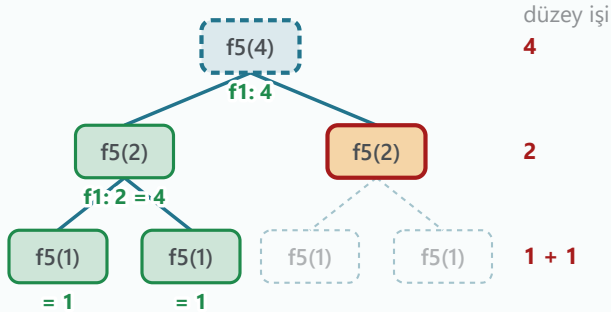
- `n = 2`
- `f1(n) = 2`
- `sonuç = 4`

ÇAĞRI YIĞINI

- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
▶   if (n <= 1) {
3     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



İkinci `f5(2)` çağrısı: ilkiyle aynı işi baştan yapar.

Adım 8 / 13

DEĞİŞKENLER

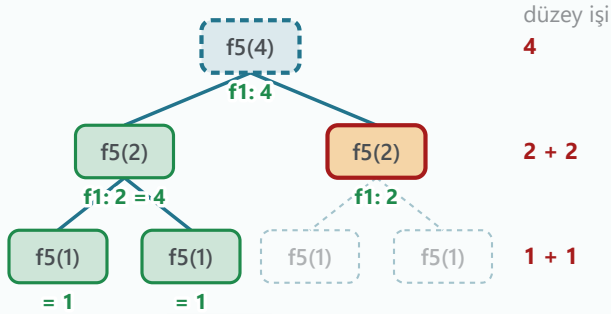
- `n = 2`
- `f1(n) = –`
- `sonuç = –`

ÇAĞRI YIĞINI

- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



Satır 5 önce `f1(2)` 'ü hesaplar: 2 turluk döngü, **2** birim iş. Sonra iki `f5(1)` çağrısı.

Adım 9 / 13

DEĞİŞKENLER

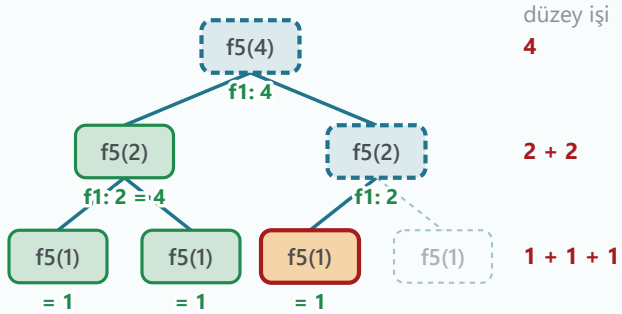
- `n = 2`
- `f1(n) = 2`
- `sonuç = –`

ÇAĞRI YIĞINI

- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → taban durum, 1 döner.

Adım 10 / 13

DEĞİŞKENLER

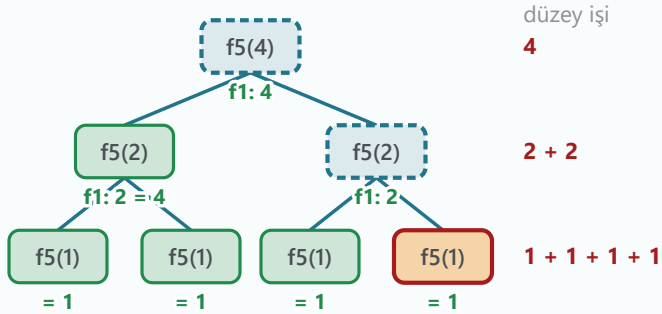
- `n = 1`
- `f1(n) = –`
- `sonuç = 1`

ÇAĞRI YIĞINI

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
  if (n <= 1) {
    return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → taban durum, 1 döner.

Adım 11 / 13

DEĞİŞKENLER

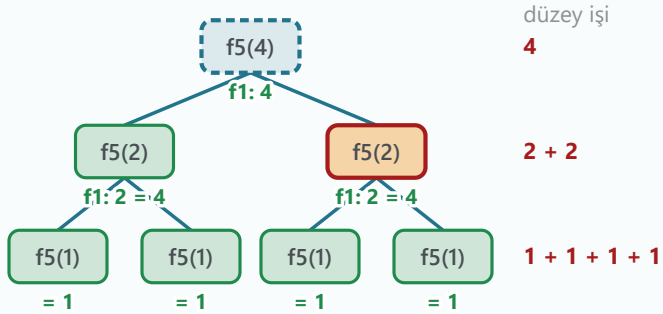
- `n = 1`
- `f1(n) = –`
- `sonuç = 1`

ÇAĞRI YIĞINI

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



f5(2) = f1(2) + f5(1) + f5(1) = 2 + 1 + 1 = **4** döner.

Adım 12 / 13

DEĞİŞKENLER

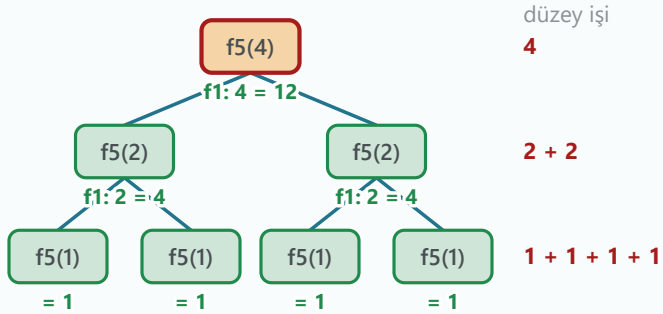
- `n = 2`
- `f1(n) = 2`
- `sonuç = 4`

ÇAĞRI YIĞINI

- `f5(2)`
- `f5(4)`

f5(4): Çağrıları Adım Adım İzle

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



f5(4) = 4 + 4 + 4 = **12**. Her düzeyin işi n = 4: $\log_2 4 = 2$ düzey · 4 + 4 yaprak → $n \log_2 n + n$. Bağıntı $2T(n/2) + n$: **Θ(n log n)**.

Adım 13 / 13

DEĞİŞKENLER

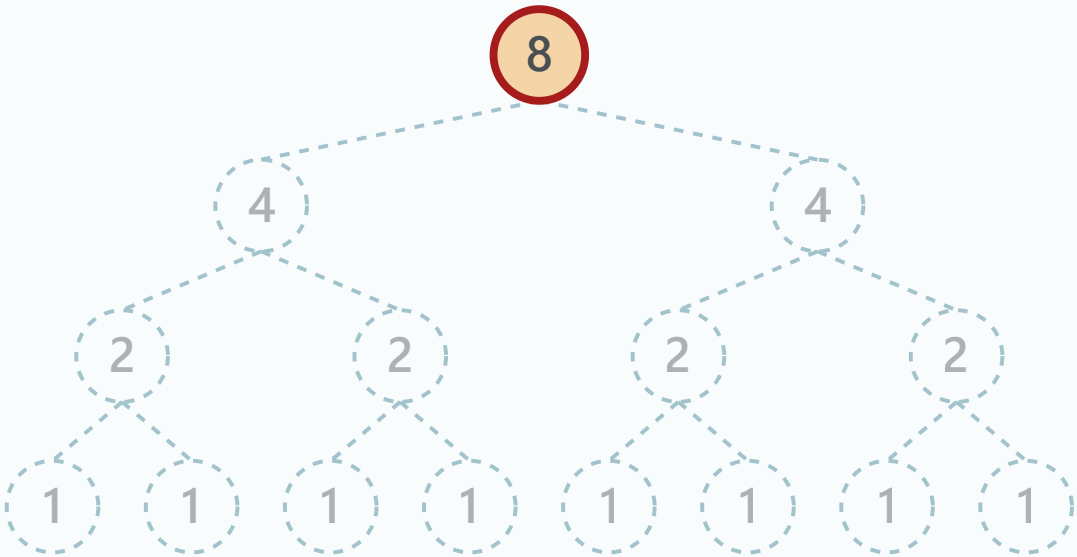
- `n = 4`
- `f1(n) = 4`
- `sonuç = 12`

ÇAĞRI YIĞINI

`f5(4)`

UYGULAMA: F4 VE F5

f5(8): Özyineleme Ağacı



8

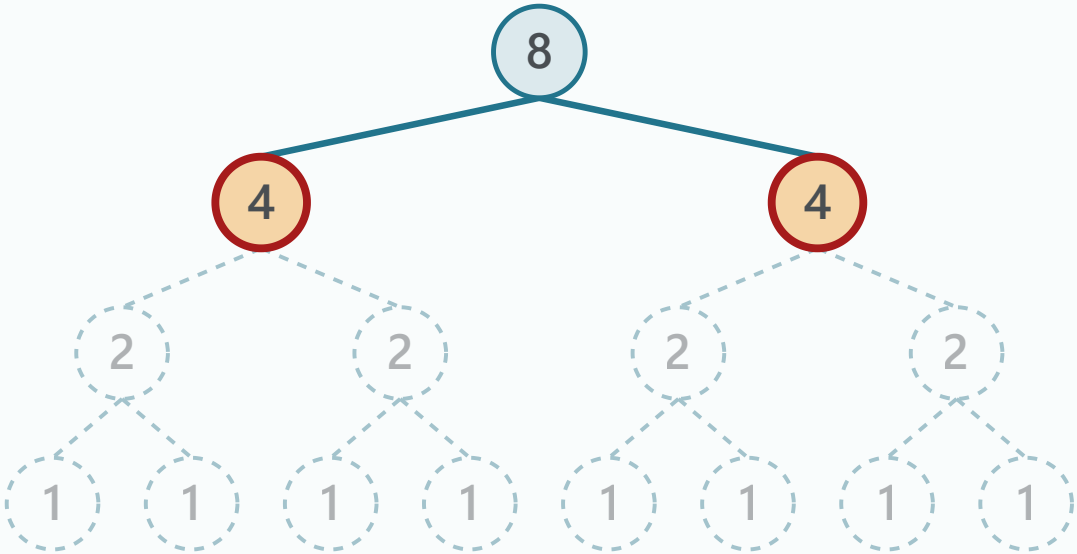
Kök: `f5(8)` içinde `f1(8)` 8 adım yapar.

Adım 1 / 4

$\log_2$ n düzeyin her biri n iş yapar. Java ile: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

UYGULAMA: F4 VE F5

f5(8): Özyineleme Ağacı



8

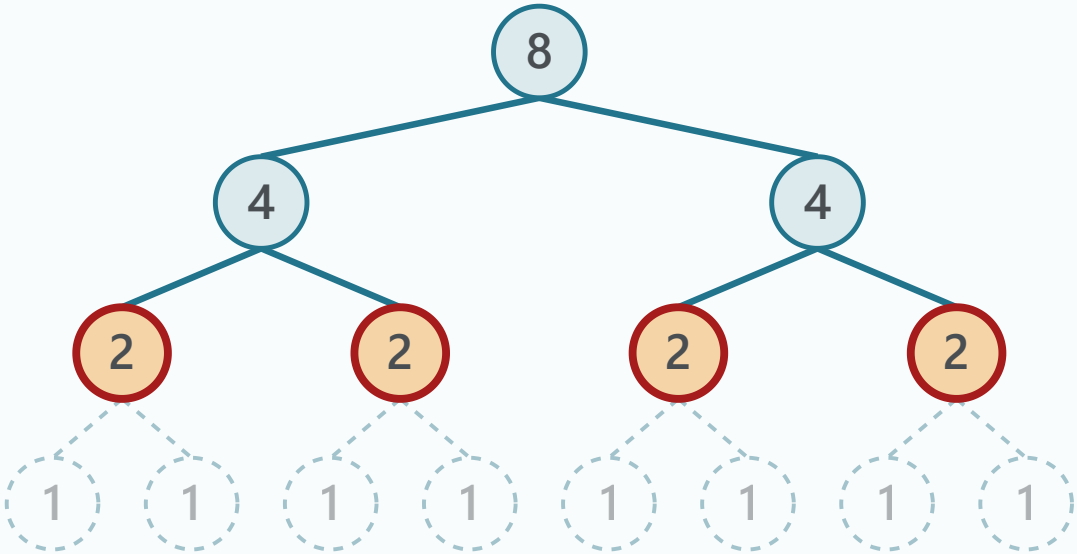
4 + 4 = 8

Düzey 1: iki `f5(4)`, her biri f1(4) = 4 adım → toplam 8.

Adım 2 / 4

$\log_2 n$ düzeyin her biri n iş yapar. Java ile: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5(8): Özyineleme Ağacı



8

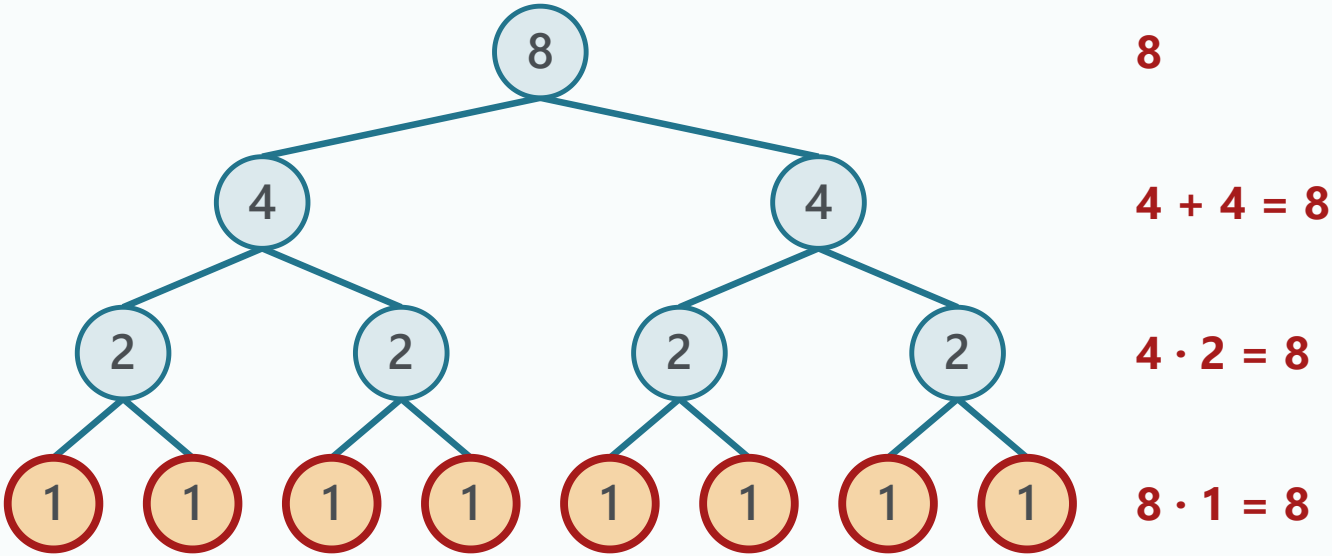
4 + 4 = 8

4 · 2 = 8

Düzey 2: dört `f5(2)`, her biri 2 adım → yine 8. Her düzey **n** iş yapar.

$\log_2$ n düzeyin her biri n iş yapar. Java ile: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5(8): Özyineleme Ağacı



Yapraklar 8 × 1. Toplam $3 \cdot 8 + 8 = 32 = n \log_2 n + n \rightarrow$ **Θ(n log n)**. Java: f5(8) = 32.

$\log_2 n$ düzeyin her biri n iş yapar. Java ile: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5: Ana Teorem ile Çözüm

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

T(n) = aT(n/b) + f(n) biçiminde.

Adım 1 / 4

	f4	f5
Çağrı	2 × n/2	2 × n/2
Ek iş f(n)	O(1)	Θ(n)
c vs $\log_b a$	0 < 1	1 = 1
Durum	1	2
Sonuç	Θ(n)	Θ(n log n)

UYGULAMA: F4 VE F5

f5: Ana Teorem ile Çözüm

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

T(n) = aT(n/b) + f(n) biçiminde.

$$a = 2, b = 2, f(n) = \Theta(n)$$

Parametreler f4 ile aynı, yalnızca f(n) değişti.

Adım 2 / 4

	f4	f5
Çağrı	2 × n/2	2 × n/2
Ek iş f(n)	O(1)	Θ(n)
c vs $\log_b a$	0 < 1	1 = 1
Durum	1	2
Sonuç	Θ(n)	Θ(n log n)

f5: Ana Teorem ile Çözüm

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

T(n) = aT(n/b) + f(n) biçiminde.

$$a = 2, b = 2, f(n) = \Theta(n)$$

Parametreler f4 ile aynı, yalnızca f(n) değişti.

$$\log_b a = \log_2 2 = 1, \quad c = 1$$

f(n) = Θ(n) → c = 1 = $\log_b a$.

	f4	f5
Çağrı	2 × n/2	2 × n/2
Ek iş f(n)	O(1)	Θ(n)
c vs $\log_b a$	0 < 1	1 = 1
Durum	1	2
Sonuç	Θ(n)	Θ(n log n)

f5: Ana Teorem ile Çözüm

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

T(n) = aT(n/b) + f(n) biçiminde.

$$a = 2, b = 2, f(n) = \Theta(n)$$

Parametreler f4 ile aynı, yalnızca f(n) değişti.

$$\log_b a = \log_2 2 = 1, \quad c = 1$$

f(n) = Θ(n) → c = 1 = $\log_b a$.

$$\text{Durum 2: } T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

Üsler eşit: **Θ(n log n)**. Bölümün başındaki örnekle aynı bağıntı.

	f4	f5
Çağrı	2 × n/2	2 × n/2
Ek iş f(n)	O(1)	Θ(n)
c vs $\log_b a$	0 < 1	1 = 1
Durum	1	2
Sonuç	Θ(n)	Θ(n log n)

KARŞILAŞTIRMA

f1–f6: Java ile Ölçülen Adım Sayıları

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

n iki katına çıkınca oran, büyüme sınıfını ele verir: $\Theta(n) \rightarrow \times 2$, $\Theta(n \log n) \rightarrow \times 2$'den biraz fazla, $\Theta(n^3) \rightarrow \times 8$, $\Theta(2^n) \rightarrow \times 2^n$.

KARŞILAŞTIRMA

Etkileşimli f1–f6 Adım Sayacı

n = 16 n / 2 n × 2

Fonksiyon	Adım sayısı	Göreli büyüklük (log ölçek)
f1 Θ(n)	16	
f2 Θ(n^3)	1.240	
f3 Θ(2^n)	65.535	
f4 Θ(n)	31	
f5 Θ(n log n)	95	
f6 Θ(log* n)	4	

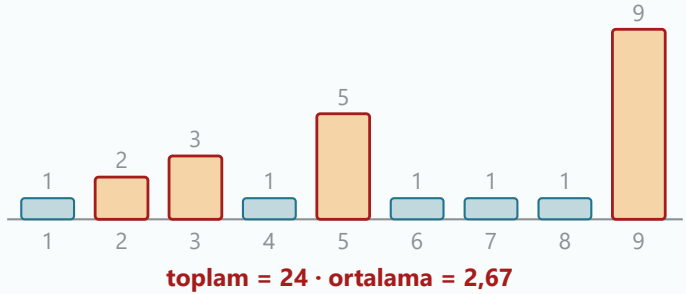
n = **16**: f3 tek başına **65.535** çağrı yapar.

Adım: f1, f2, f6 için `x++` sayısı; f3, f4 için çağrı sayısı; f5 için çağrı + f1 döngü adımları. Değerler kodun mantığıyla hesaplanır (f3 için BigInt).

KARŞILAŞTIRMA

Kısa Bakış: Amortize Analiz

- **Amortize analiz** (amortized analysis), bir **işlem dizisinin** toplam maliyetini işlem sayısına böler.
- Dinamik dizi: dolunca kapasite **ikiye katlanır** ve tüm öğeler kopyalanır (pahalı); diğer eklemeler 1 birimdir.
- n eklemede kopyalar 1 + 2 + 4 + … < 2n → toplam < 3n → ekleme başına **amortize Θ(1)**.
- Ortalama durumdan farklıdır: olasılık yoktur, **her** işlem dizisi için garanti verir.



Kapasite 1'den başlayarak 9 ekleme; turuncu çubuklar kapasite artırma (kopyalama) içerir.

Tek bir eklemenin en kötü durumu Θ(n) olsa da, dizinin tamamı için 24 < 3 · 9 = 27 birim harcanır. Ayrıntı: toplama, hesap (accounting) ve potansiyel yöntemleri.

ÖZET

Özet

Konu	Hatırlanacaklar
Karmaşıklık	zaman = temel işlem sayısı T(n); bellek = ek bellek (yığın dahil); en iyi / en kötü / ortalama
Gösterim	O: üst, Ω: alt, Θ: sıkı sınır; c ve n_0 ile kanıtlanır; sabitler ve düşük terimler atılır
Döngüler	tur sayılarını topla: f1 $\Sigma 1 = n \rightarrow \Theta(n)$; f2 $\Sigma i^2 \rightarrow \Theta(n^3)$; f6 $i \leftarrow 2^i \rightarrow \Theta(\log^* n)$ (O(log n) sıkı değil)
Özyineleme	bağıntı kur: çağrı sayısı, küçülme biçimi, ek iş; açma veya özyineleme ağacı ile çöz
f3	$2T(n-1) + c = (2^n - 1)c \rightarrow \Theta(2^n)$; Ana Teorem uygulanamaz
Ana Teorem	$aT(n/b) + f(n)$: f(n)'yi $n^{\log_b a}$ ile karşılaştır; yapraklar / dengeli / kök baskın
f4, f5	$2T(n/2) + 1 \rightarrow$ Durum 1, $\Theta(n)$; $2T(n/2) + n \rightarrow$ Durum 2, $\Theta(n \log n)$

ÖZET

Etkileşimli Simülatörler

Asimptotik Büyüme Gezgini

log n, n, n log n, n^2, n^3 ve 2^n'nin aynı n'de karşılaştırması; iki fonksiyonun kesişme (crossover) noktası.

Özyineleme ve Ana Teorem

Verilen a, b ve $f(n) = n^k$ ile $T(n) = aT(n/b) + f(n)$ için özyineleme ağacının düzeyleri ve Ana Teorem durumu.

Büyüme Hesaplayıcı

Verilen bir n için 1'den n!'e kadar her fonksiyonun işlem sayısı ve saniyede 10^9 işlem yapan bir bilgisayardaki süresi; "n × 2" ile girdiyi ikiye katlama.

ÖZET

Sık Yapılan Hatalar

Büyük O = en kötü durum sanmak

O bir üst sınırdır; en kötü durum ise bir girdi türüdür. "En iyi durum O(1)" da geçerli bir cümledir.

Sabitleri ve küçük terimleri yazmak

$O(3n + 3)$, $O(n^2/3 + n)$ yazılmaz: $O(n)$ ve $O(n^2)$ yeterlidir.

"İç içe döngü = n^2"

Belirleyici olan iç döngünün sınırıdır: f2'de i^2 olduğu için $\Theta(n^3)$; `i *= 2` döngüsü ise log n tur döner.

"İki çağrı = üstel"

f3 $(n - 1)$ $\Theta(2^n)$, f4 $(n/2)$ $\Theta(n)$'dir. Küçülme biçimi her şeyi değiştirir.

Ana Teoremi her bağıntıya uygulamak

$T(n - 1)$ içeren, eşit bölmeyen ya da $f(n) = n/\log n$ olan bağıntılarda teorem geçerli değildir.

Durum 3'te düzenliliği unutmak

$c > \log_b a$ yetmez; $a \cdot f(n/b) \leq k \cdot f(n)$ $(k < 1)$ de sağlanmalıdır.

ÖZET

Örnek Problemler

1. $2n + 10 = O(n)$ olduğunu c ve n_0 bularak gösterin.
2. $T(n) = 3T(n/3) + n$ bağıntısını çözün.
3. $T(n) = 9T(n/3) + n$ bağıntısını çözün.
4. $T(n) = T(n/2) + n$ bağıntısını çözün (düzenliliği kontrol edin).
5. `for (i = 1; i < n; i *= 2) for (j = 0; j < i; j++) x++;` kaç kez `x++` yapar?
6. `for (i = n; i > 0; i /= 2) for (j = 0; j < n; j++) x++;` karmaşıklığı nedir?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. f2'de iç döngü koşulu `j < i` olsaydı karmaşıklık ne olurdu? f2(5) ne döndürürdü?
2. `return g(n/2) + g(n/2) + g(n/2) + g(n/2);` (taban $n \le 1$) için bağıntıyı yazıp çözün.
3. f3'ün son satırı `return 2 * f3(n - 1);` olsaydı dönüş değeri ve süresi nasıl değişirdi?
4. $T(n) = 8T(n/2) + n^2$ ve $T(n) = 7T(n/2) + n^2$ bağıntılarını çözün; hangi algoritmalara karşılık gelirler?
5. $n^2 \neq O(n)$ olduğunu çelişki ile kanıtlayın.
6. f6'yı `for (i = 1; i < n; i *= 2)` yaparsak n = 1000 için kaç tur döner? f6(1000) ile karşılaştırın.

▶ **Cevaplar / ipuçları**

ÖZET

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 3 ve 4).
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 1.4: Analysis of Algorithms).
- Kleinberg ve Tardos, *Algorithm Design*, Pearson, 2005 (Bölüm 2 ve 5).
- D. E. Knuth, "Big Omicron and Big Omega and Big Theta", *SIGACT News* 8(2), 1976.
- J. L. Bentley, D. Haken ve J. B. Saxe, "A General Method for Solving Divide-and-Conquer Recurrences", *SIGACT News* 12(3), 1980.

Son

Bölüm 2: Algoritma Karmaşıklığı

Sonraki bölüm: Bölüm 3: Sıralama Algoritmaları · Terim sözlüğü