Bölüm 2: Algoritma Analizi

Veri Yapıları

Algoritma ve verimlilik | Deneysel analiz | İlkel işlemleri sayma | O, Ω, Θ

Döngü kalıpları | Doğrusal ve ikili arama | Alan karmaşıklığı | Doğruluk ve sağlamlık

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Ölçme ve sayma

Bir algoritmayı deneysel olarak ölçmek ve ilkel işlemlerini sayarak çözümlemek.

Asimptotik gösterim

Zaman ve alan karmaşıklığını O, Ω ve Θ gösterimleriyle ifade etmek.

Koddan karmaşıklığa

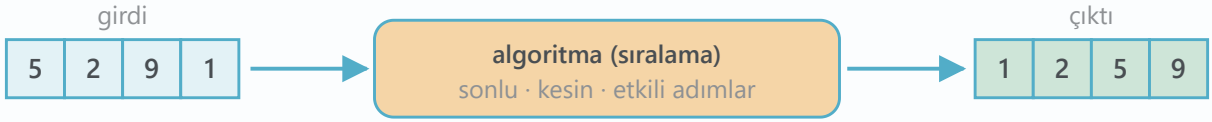
Döngü kalıplarından en iyi, ortalama ve en kötü durum maliyetini çıkarmak.

Karşılaştırma

Aynı problemi çözen algoritmaları maliyet, doğruluk ve sağlamlık açısından karşılaştırmak.

Algoritmalar

- Bir algoritma, bir problemi çözmek için adım adım bir prosedürdür: **girdi** → **adımlar** → **çıktı**.
- Aynı problemi çözmek için farklı algoritmalar bulunabilir; bazıları diğerlerine göre daha verimli çalışır.
- Doğru algoritma seçimi kritiktir: iyi bir algoritma, işi daha az kaynak kullanarak yapar.



Bazı yaygın algoritmalar:

Sıralama

Verileri belirli bir sıraya göre düzenler.

`kabarcık, birleştirme sıralaması`

Arama

Belirli bir öğeyi veri kümesinde bulur.

`doğrusal arama, ikili arama`

Çizge

Çizge teorisi problemlerini çözer.

`en kısa yol, ağ akışı`

İyi Bir Algoritma

Bir problemi etkili bir şekilde çözen algoritmadır. Dört ölçüte bakarız:

Zaman Karmaşıklığı (Time Complexity)

Çalışma süresi veya işlem sayısı gibi faktörler zaman karmaşıklığını belirler.

Alan Karmaşıklığı (Space Complexity)

Algoritmanın kullandığı bellek miktarı alan karmaşıklığını belirler.

Doğruluk (Correctness)

Her geçerli girdi için doğru sonucu üretmelidir. Yanlış sonuç veren bir algoritma kullanışsızdır.

Sağlamlık (Robustness)

Çeşitli durumlar ve girdilerle başa çıkabilmeli, farklı senaryolara uyum sağlayabilmelidir.

Deneysel Analiz: `System.nanoTime()`

```
for (int n = 1000; n <= 64000; n *= 2) {
  int[] a = new int[n];
  for (int i = 0; i < n; i++) a[i] = i; // tekrarsız
  long bas = System.nanoTime();
  tekrarVar(a);                         // ölçülen iş
  long sure = System.nanoTime() - bas;  // ns
  System.out.println(n + " " + sure / 1e6 + " ms");
}
```

Ölç

`System.nanoTime()` iki okuma arasındaki farkı ns cinsinden verir (duvar saati değil).

Tekrarla

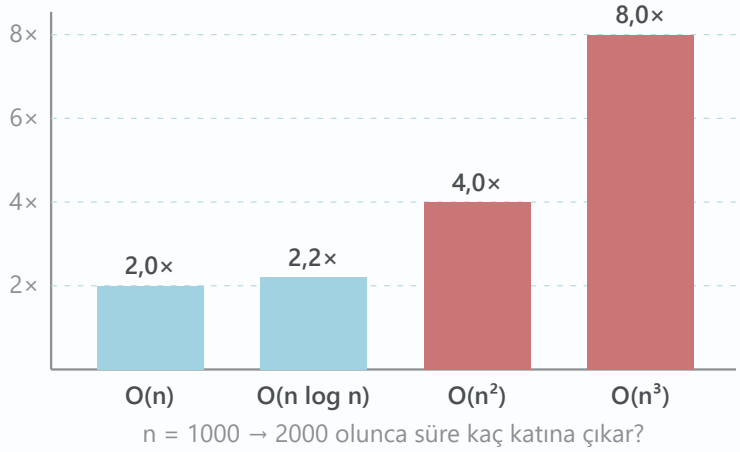
JIT ısınması, çöp toplayıcı ve diğer süreçler gürültü yapar: birkaç kez çalıştır, medyanı al.

Çiz

Süreyi n'ye karşı çiz; n'yi **ikiye katlayınca** sürenin kaç katına çıktığına bak.

Dikkat: `new int[n]` sıfırlarla dolar → `tekrarVar` (tekrar arayan $O(n^2)$ metot) ilk karşılaştırmada `true` döner. En kötü durumu ölçmek için tekrarsız veri gerekir.

Katlama Deneyi ve Deneysel Analizin Sınırları



Deneysel analizin sınırları (Goodrich §4.1):

- Sonuçlar donanıma, JVM'e ve yüke bağlıdır; iki algoritma **aynı ortamda** ölçülmelidir.
- Yalnız sınırlı sayıda girdi denenebilir; önemli bir girdi kaçabilir.
- Ölçmek için algoritmanın **tamamen kodlanması** gerekir.

Katlama deneyi: $T(2n)/T(n) \approx 2^k$ ise $T(n) \approx c \cdot n^k$. Sınırları aşmak için: **ilkel işlemleri sayarak** analiz — donanımdan bağımsız, tüm girdiler için, kod çalıştırmadan.

İlkel İşlemleri Sayma: `maksimum()`

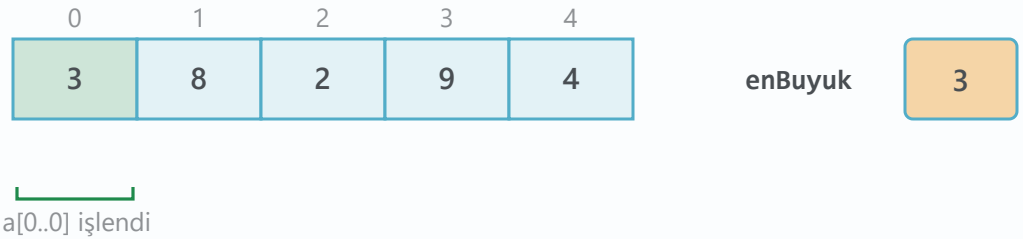
```
int maksimum(int[] a) {
  int enBuyuk = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > enBuyuk)
      enBuyuk = a[i];
  }
  return enBuyuk;
}
```

- **İlkel işlem**: atama, karşılaştırma, aritmetik, dizi indisleme, metot çağrısı / dönüş.
- Her birinin süresi sabit kabul edilir → yalnız **kaç kez** çalıştıklarını sayarız.

Satır	İfade	Kaç kez?
2	`enBuyuk = a[0]`	1
3	`i = 1`	1
3	`i < a.length`	n
4	`a[i] > enBuyuk`	n − 1
5	`enBuyuk = a[i]`	0 … n − 1
3	`i++`	n − 1
7	`return`	1

En iyi: **3n + 1** (a[0] en büyük, hiç güncelleme yok) · En kötü: **4n** (dizi artan sıralı) → her iki durumda **Θ(n)**. Sabitler donanıma bağlı olduğundan atılır.

Kod İzleme: `maksimum(a)`



enBuyuk = a[0] = 3: ilk öğe geçici en büyük.

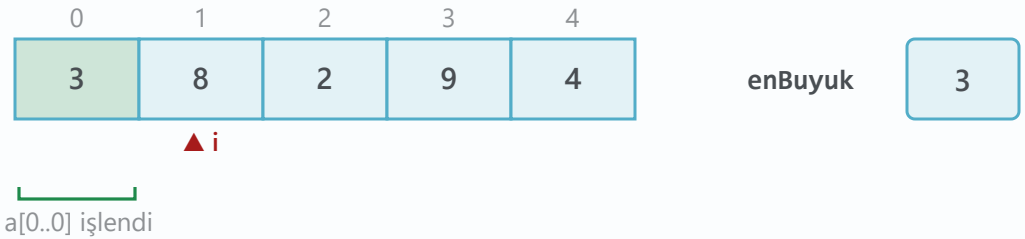
Adım 1 / 13

```
1 int maksimum(int[] a) {
▶   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 3`

Kod İzleme: `maksimum(a)`



i = 1, 1 < 5 → döngü sürer. Değişmez: enBuyuk = a[0..0]'in en büyüğü.

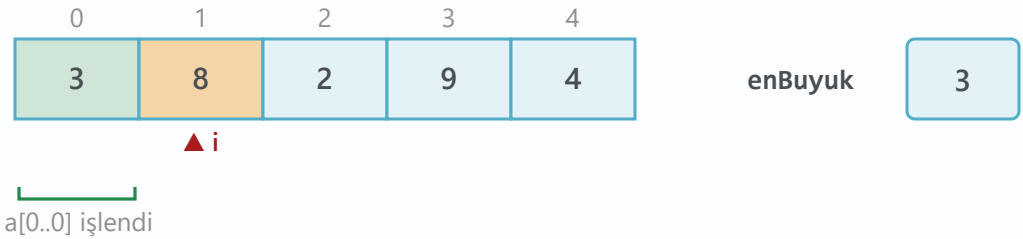
Adım 2 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 3` `i = 1` `karşılaştırma = 0` `güncelleme = 0`

Kod İzleme: `maksimum(a)`



a[1] = 8 > 3 → **doğru**.

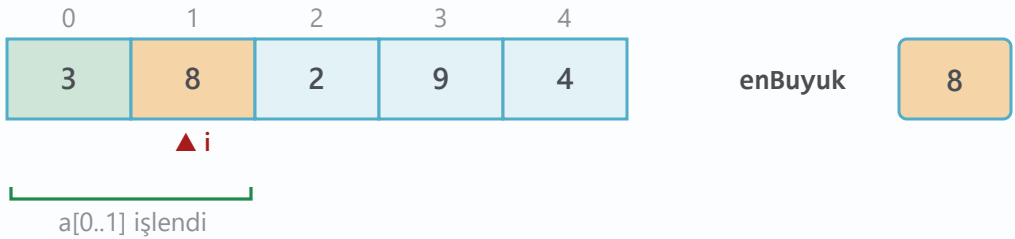
Adım 3 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 3` `i = 1` `karşılaştırma = 1` `güncelleme = 0`

Kod İzleme: `maksimum(a)`



enBuyuk = 8: yeni en büyük bulundu.

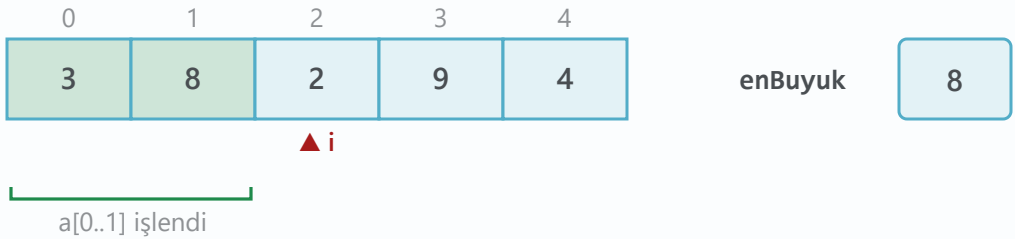
Adım 4 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
▶       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 8` `i = 1` `karşılaştırma = 1` `güncelleme = 1`

Kod İzleme: `maksimum(a)`



i = 2, 2 < 5 → döngü sürer. Değişmez: enBuyuk = a[0..1]'in en büyüğü.

Adım 5 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

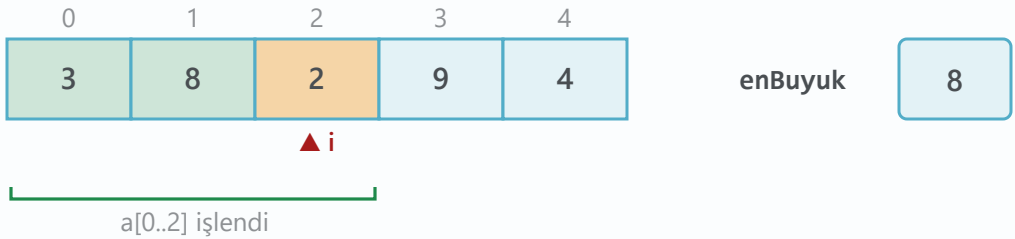
DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 8`

`i = 2` `karşılaştırma = 1`

`güncelleme = 1`

Kod İzleme: `maksimum(a)`



a[2] = 2 > 8 → **yanlış**; enBuyuk değişmez.

Adım 6 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

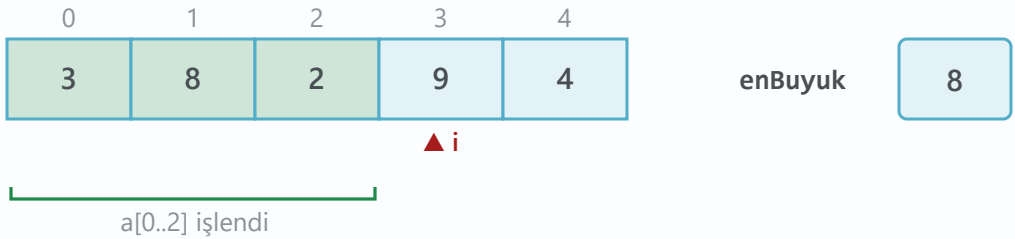
DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 8`

`i = 2` `karşılaştırma = 2`

`güncelleme = 1`

Kod İzleme: `maksimum(a)`



i = 3, 3 < 5 → döngü sürer. Değişmez: enBuyuk = a[0..2]'in en büyüğü.

Adım 7 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 8`

`i = 3` `karşılaştırma = 2`

`güncelleme = 1`

Kod İzleme: `maksimum(a)`



a[3] = 9 > 8 → **doğru**.

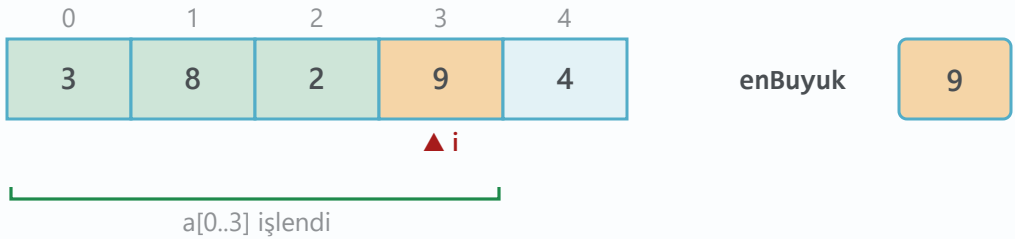
Adım 8 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 8` `i = 3` `karşılaştırma = 3` `güncelleme = 1`

Kod İzleme: `maksimum(a)`



enBuyuk = 9: yeni en büyük bulundu.

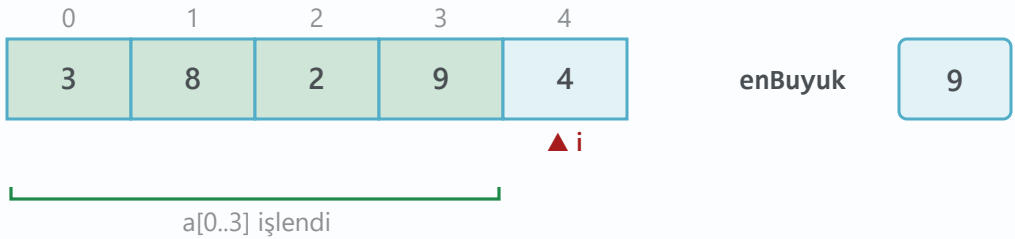
Adım 9 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
▶       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 9` `i = 3` `karşılaştırma = 3` `güncelleme = 2`

Kod İzleme: `maksimum(a)`



i = 4, 4 < 5 → döngü sürer. Değişmez: enBuyuk = a[0..3]'in en büyüğü.

Adım 10 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

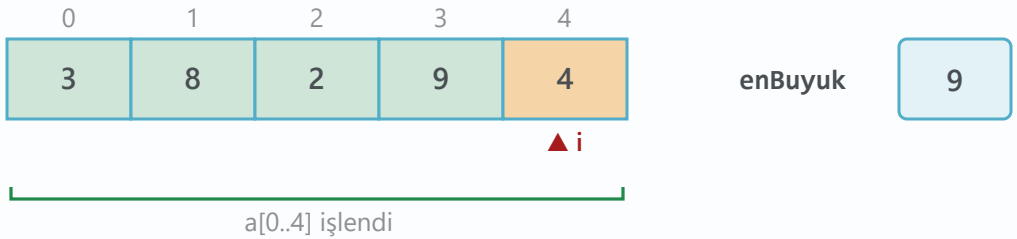
DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 9`

`i = 4` `karşılaştırma = 3`

`güncelleme = 2`

Kod İzleme: `maksimum(a)`



a[4] = 4 > 9 → **yanlış**; enBuyuk değişmez.

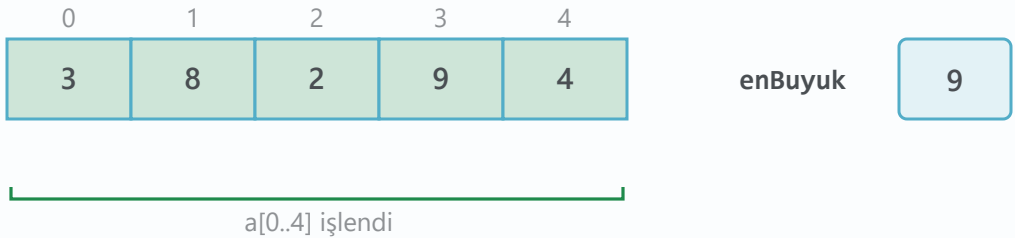
Adım 11 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 9` `i = 4` `karşılaştırma = 4` `güncelleme = 2`

Kod İzleme: `maksimum(a)`



i = 5: 5 < 5 yanlış → döngü biter. Tüm dizi işlendi.

Adım 12 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
7   return enBuyuk;
8 }
```

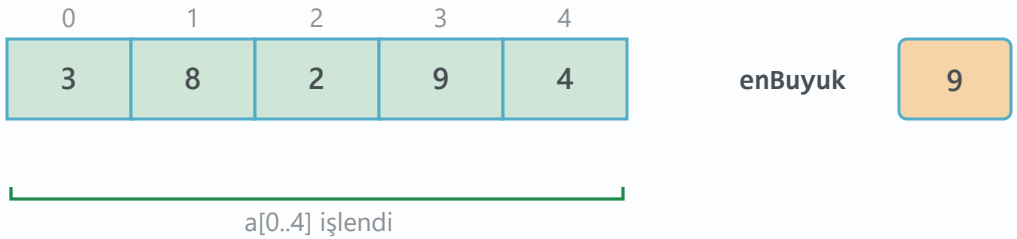
DEĞİŞKENLER

`a = [3, 8, 2, 9, 4]` `enBuyuk = 9`

`i = 5` `karşılaştırma = 4`

`güncelleme = 2`

Kod İzleme: `maksimum(a)`



return 9. n = 5 için n − 1 = 4 karşılaştırma yapıldı → **Θ(n)**.

Adım 13 / 13

```
1 int maksimum(int[] a) {
2   int enBuyuk = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > enBuyuk)
5       enBuyuk = a[i];
6   }
▶   return enBuyuk;
8 }
```

DEĞİŞKENLER

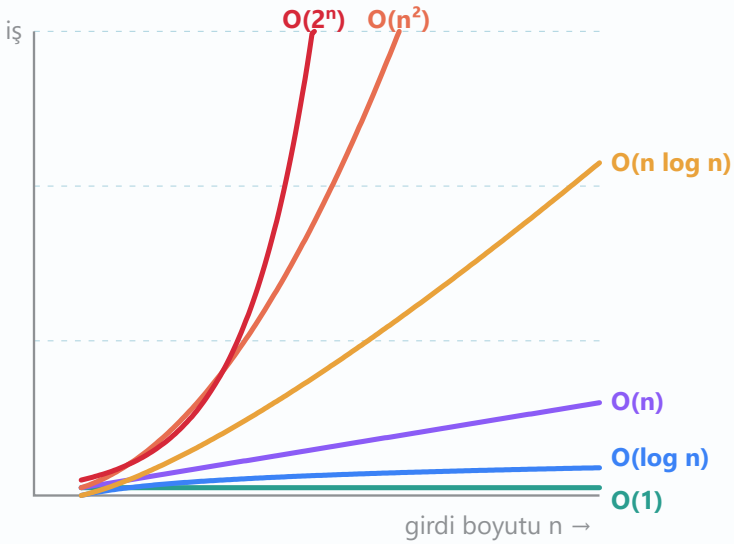
`a = [3, 8, 2, 9, 4]` `enBuyuk = 9`

`karşılaştırma = 4` `güncelleme = 2`

ASİMPTOTİK GÖSTERİM

Zaman Karmaşıklığı ve Büyük O (Big O)

- Algoritmanın ne kadar sürede çalıştığını, **girdi boyutu n'ye bağlı** olarak ifade eder.
- Büyük O gösterimi, çalışma süresi için bir **üst sınır** belirtir; genellikle en kötü durum için kullanılır.
- **O(1)**: işlem sayısı girdi boyutuyla büyümez; farklı girdilerde süre birebir aynı olmak zorunda değildir.
- **O(n)**: girdi büyüdükçe süre doğrusal olarak artar.

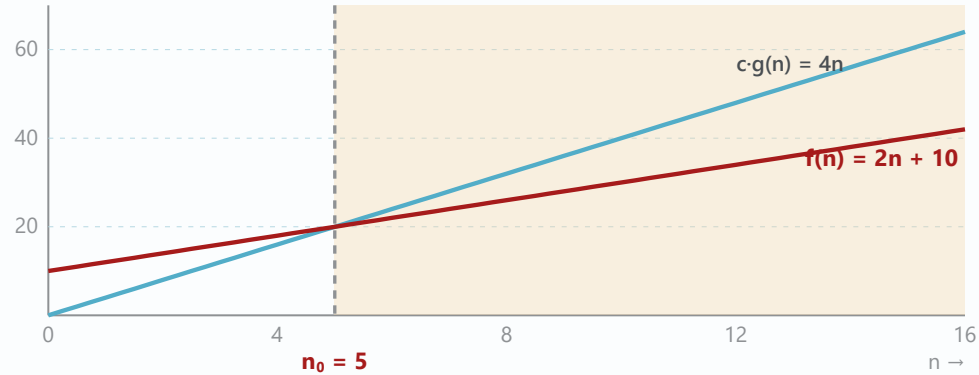


n arttıkça işlem sayısı

Büyük O'da yalnızca en hızlı büyüyen terim önemlidir; sabitler ve küçük terimler atılır:
$3n^2 + 5n + 2 \rightarrow O(n^2)$ · $4n + 100 \rightarrow O(n)$ · $7 \rightarrow O(1)$

O, Ω ve Θ: Tanım ve Grafik

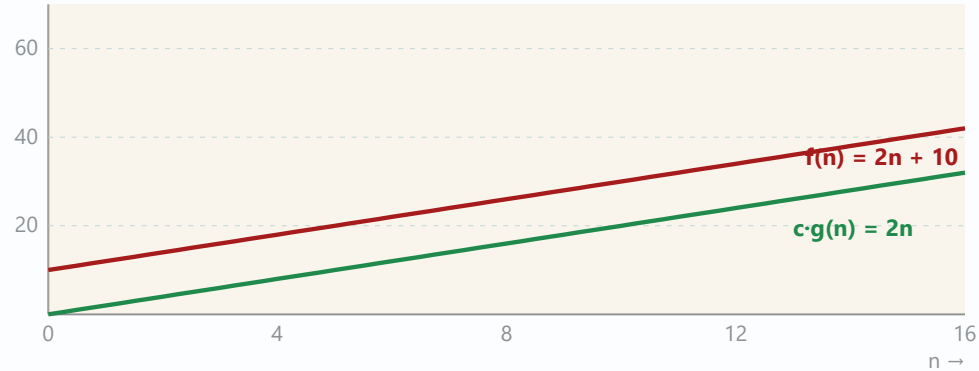
Tanım: f(n) = O(g(n)) ⇔ öyle c > 0 ve $n_0 \geq 1$ vardır ki **her $n \geq n_0$ için $f(n) \leq c \cdot g(n)$**. Ω için ≥, Θ için ikisi birden.



Büyük O: $n \geq n_0 = 5$ için $2n + 10 \leq \mathbf{4} \cdot n \rightarrow f(n) = \mathbf{O(n)}$ (c = 4). Küçük n'de (n < 5) f(n) daha büyük; önemli değil.

O, Ω ve Θ: Tanım ve Grafik

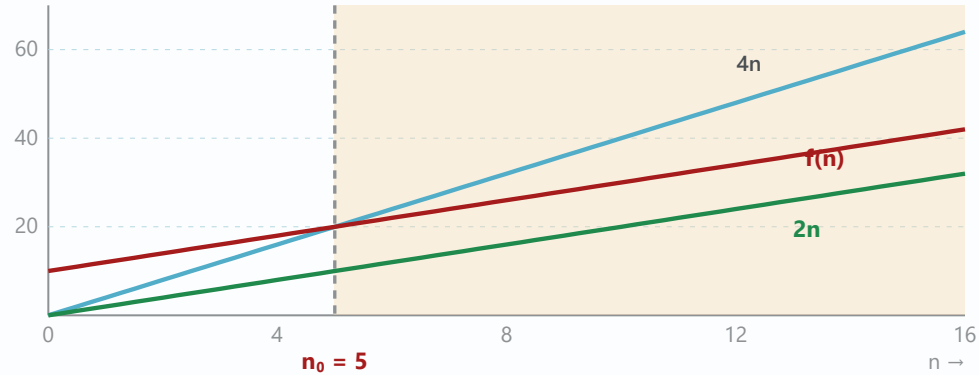
Tanım: f(n) = O(g(n)) ⇔ öyle c > 0 ve $n_0 \geq 1$ vardır ki **her $n \geq n_0$ için $f(n) \leq c \cdot g(n)$**. Ω için ≥, Θ için ikisi birden.



Büyük Omega: her n ≥ 1 için 2n + 10 ≥ **2**·n → f(n) = **Ω(n)**: f, n'den daha yavaş büyümez (alt sınır).

O, Ω ve Θ: Tanım ve Grafik

Tanım: f(n) = O(g(n)) ⇔ öyle c > 0 ve $n_0 \geq 1$ vardır ki **her $n \geq n_0$ için $f(n) \leq c \cdot g(n)$**. Ω için ≥, Θ için ikisi birden.



Büyük Teta: n ≥ 5 için 2n ≤ f(n) ≤ 4n → f(n) = **Θ(n)**: iki doğru arasına sıkışır, büyüme hızı tam olarak n.

ASİMPTOTİK GÖSTERİM

Asimptotik Sınırlar ve Durumlar

O(g(n)) · Üst sınır
Çalışma miktarı, yeterince büyük n için g(n)'den daha hızlı büyümez.

Ω(g(n)) · Alt sınır
Çalışma miktarı, yeterince büyük n için g(n)'den daha yavaş büyümez.

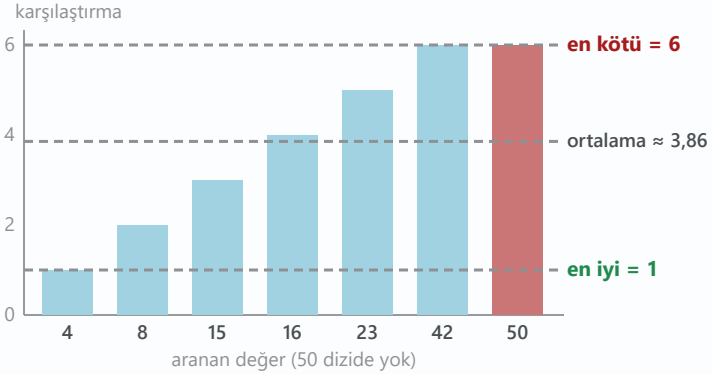
Θ(g(n)) · Sıkı sınır
Üst ve alt sınır aynıdır; büyüme hızı g(n) ile aynı sınıftadır.

Analiz	Ne sorar?	Doğrusal arama örneği
En iyi durum	En elverişli geçerli girdide kaç işlem gerekir?	İlk öğe hedefse Θ(1)
Ortalama durum	Bir girdi dağılımı varsayımı altında beklenen maliyet nedir?	Hedefin konumu eşit olasılıklıysa Θ(n)
En kötü durum	Her geçerli girdi için garanti edilen üst maliyet nedir?	Hedef yoksa veya sondaysa Θ(n)

Amortize analiz: Tek bir işlem pahalı olabilir; uzun bir işlem dizisinin toplam maliyeti işlem sayısına bölünür. Örneğin büyüyen dizide yeniden boyutlandırma O(n), sona ekleme ise amortize O(1)'dir.

En İyi, Ortalama ve En Kötü Durum

Dizi [4, 8, 15, 16, 23, 42] üzerinde doğrusal arama; her olası hedef için karşılaştırma sayısı:



- Aynı n için girdiye göre maliyet değişir.
- **En kötü durum** bir **garanti** verir ve hesaplaması kolaydır → genellikle bu raporlanır.
- **Ortalama** bir olasılık dağılımı varsayar (burada dizide olmayan 50 dahil 7 hedef eşit olasılıklı: 27/7 ≈ 3,86).
- **En iyi durum** tek başına yanıltıcıdır.

Büyük O Gösterimi Örnekleri

Gösterim	Adı	Örnek	Neden?
O(1)	Sabit	Dizinin ilk öğesine erişme	Dizinin boyutu ne olursa olsun erişim süresi sabit
O(log n)	Logaritmik	Sıralı dizide ikili arama	Her adımda arama alanı yarıya iner
O(n)	Doğrusal	Diziyi baştan sona tarama	Her öğe bir kez işlenir
O(n log n)	Doğrusal logaritmik	Birleştirme sıralaması (Merge Sort)	Böl ve fethet: log n düzey, her düzeyde n iş
O(n^2)	Kare	Tüm öğe çiftlerini karşılaştırma	İç içe iki döngü
O(2^n)	Üstel	Tüm alt kümeleri bulma	n öğeli kümenin 2^n alt kümesi var

ASİMPTOTİK GÖSTERİM

n Büyüdükçe Ne Olur?

Karmaşıklık	n = 10	n = 100	n = 1.000
$\log_2 n$	3,3	6,6	10
n	10	100	1.000
$n \log_2 n$	33	664	9.966
n^2	100	10.000	1.000.000
2^n	1.024	$\approx 1{,}3 \cdot 10^{30}$	$\approx 1{,}1 \cdot 10^{301}$

Saniyede 10^9 işlem yapan bir bilgisayar, n = 100 için 2^n adımı yaklaşık **$4 \cdot 10^{13}$ yılda** bitirir. Evrenin yaşı $\approx 1{,}4 \cdot 10^{10}$ yıl.

KODDAN KARMAŞIKLIĞA · ADIM SAYMA

Döngü Kalıplarından Karmaşıklığa

Kalıp	Gövde kaç kez?	n = 16	Sınıf
`for (i = 0; i < n; i++)`	n	16	O(n)
`for (i = 0; i < n; i += 2)`	n / 2	8	O(n)
`for (i = 1; i < n; i *= 2)`	$\lfloor \log_2 n \rfloor$	4	O(log n)
iç içe: `i < n`, `j < n`	n^2	256	$O(n^2)$
iç içe: `i < n`, `j < i`	$n(n-1)/2$	120	$O(n^2)$
ardışık iki `i < n` döngüsü	2n	32	O(n)

Ardışık bloklar

maliyetler **toplanır**, en büyüğü kalır: O(n) + $O(n^2)$ = $O(n^2)$

İç içe döngüler

maliyetler **çarpılır**: dış × iç (iç döngü dışa bağlıysa topla)

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



adimSay(16): kaç kez ikiye katlayarak n'ye ulaşırız?

Adım 1 / 12

```
▶ int adimSay(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

DEĞİŞKENLER

n = **16**

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



Sayaç **k = 0**.

Adım 2 / 12

```
1 int adimSay(int n) {
▶   int k = 0;
3   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

DEĞİŞKENLER

`n = 16` `k = 0`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



i = 1 < 16 → gövde çalışır.

Adım 3 / 12

```
1  int adimSay(int n) {
2    int k = 0;
▶    for (int i = 1; i < n; i *= 2)
4      k++;
5    return k;
6  }
```

DEĞİŞKENLER

`n = 16` `k = 0` `i = 1`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



k++ → k = 1.

Adım 4 / 12

```
1 int adimSay(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
      k++;
5   return k;
6 }
```

DEĞİŞKENLER

n = **16** | k = **1** | i = **1**

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



i = 2 < 16 → gövde çalışır.

Adım 5 / 12

```
1 int adimSay(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

DEĞİŞKENLER

`n = 16` `k = 1` `i = 2`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



k++ → k = 2.

Adım 6 / 12

```
1 int adimSay(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
▶     k++;
5   return k;
6 }
```

DEĞİŞKENLER

`n = 16` `k = 2` `i = 2`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



i = 4 < 16 → gövde çalışır.

Adım 7 / 12

```
1  int adimSay(int n) {
2    int k = 0;
▶    for (int i = 1; i < n; i *= 2)
4      k++;
5    return k;
6  }
```

DEĞİŞKENLER

`n = 16` `k = 2` `i = 4`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



k++ → k = 3.

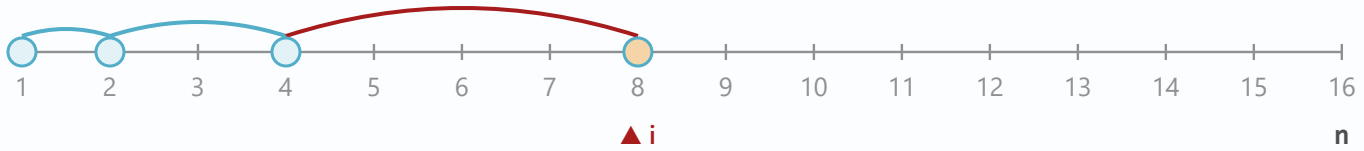
Adım 8 / 12

```
1 int adimSay(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
      k++;
5   return k;
6 }
```

DEĞİŞKENLER

n = **16** | k = **3** | i = **4**

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



i = 8 < 16 → gövde çalışır.

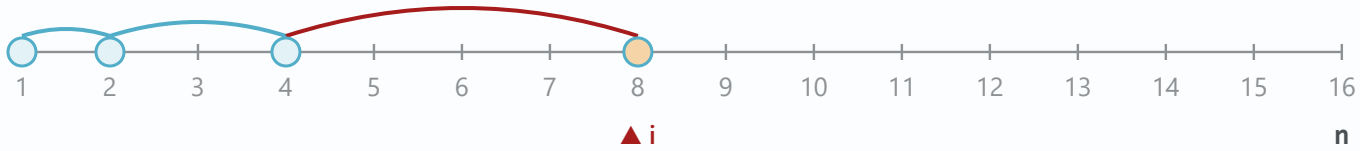
Adım 9 / 12

```
1 int adimSay(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

DEĞİŞKENLER

`n = 16` `k = 3` `i = 8`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



k++ → k = 4.

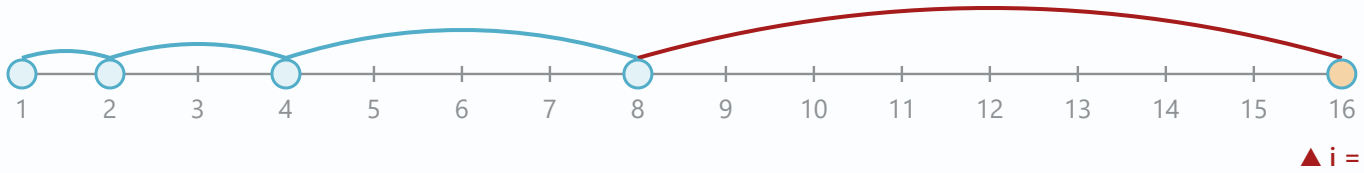
Adım 10 / 12

```
1 int adimSay(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
▶     k++;
5   return k;
6 }
```

DEĞİŞKENLER

`n = 16` `k = 4` `i = 8`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



i *= 2 → **i = 16**; 16 < 16 yanlış → döngü biter.

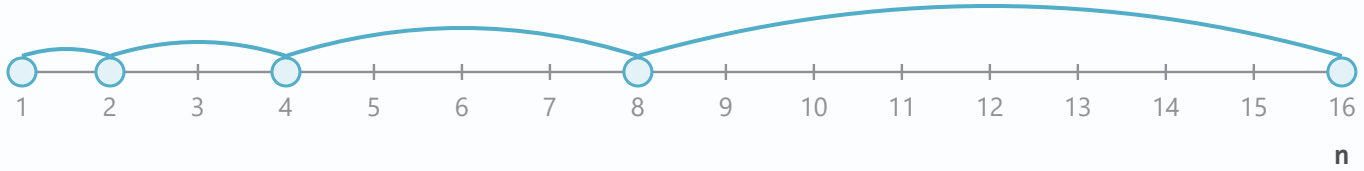
Adım 11 / 12

```
1 int adimSay(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

DEĞİŞKENLER

`n = 16` `k = 4` `i = 16`

Kod İzleme: `adimSay(16)` — i her turda ikiye katlanır



return 4 = $\log_2 16$. n'yi 1000 kat büyütmek yalnız ~10 tur ekler → **O(log n)**.

Adım 12 / 12

```
1 int adimSay(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
4     k++;
▶   return k;
6 }
```

DEĞİŞKENLER

n = **16** k = **4**

Koddan Karmaşıklığa

O(1)

```
int ilk(int[] a) {
    return a[0];
}
```

O(n)

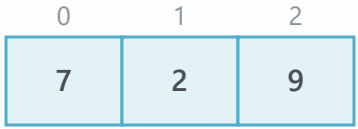
```
int toplam(int[] a) {
    int s = 0;
    for (int x : a)  // n kez
        s += x;
    return s;
}
```

O(n^2)

```
boolean tekrarVar(int[] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i + 1; j < a.length; j++)
            if (a[i] == a[j])
                return true;
    return false;
}
// dış döngü n, iç döngü ≈ n/2
// → n(n-1)/2 karşılaştırma
```

Kural: sabitler ve küçük terimler atılır. $n(n-1)/2 = n^2/2 - n/2 \rightarrow O(n^2)$

Kod İzleme: `toplam()`



main, `toplam(a)` 'yı çağırır. Yeni bir çağrı çerçevesi açılır.

Adım 1 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
3      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
▶  System.out.println(toplam(a));
```

DEĞİŞKENLER

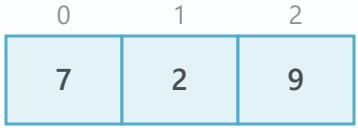
`a = [7, 2, 9]`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: `toplam()`



Metoda girildi; `a` aynı diziyi gösterir (kopyalanmaz).

Adım 2 / 12

```
▶ int toplam(int[] a) {
2     int s = 0;
3     for (int x : a) {
4         s += x;
5     }
6     return s;
7 }
8
9 // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]`

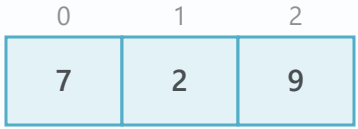
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



Toplam değişkeni **s = 0** ile başlar.

Adım 3 / 12

```
1  int toplam(int[] a) {
▶      int s = 0;
3      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

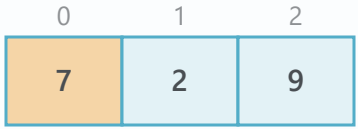
`a = [7, 2, 9]` `s = 0`

ÇAĞRI YIĞINI

- `toplam(a)`
- `main()`

ÇIKTI

Kod İzleme: `toplam()`



Döngü 1. tur: **x = a[0] = 7**.

Adım 4 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 0` `x = 7`

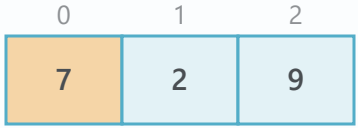
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



s += x → s = 0 + 7 = **7**.

Adım 5 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
3      for (int x : a) {
▶          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 7` `x = 7`

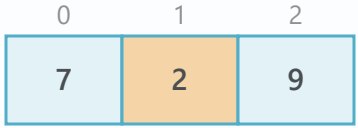
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



Döngü 2. tur: **x = a[1] = 2**.

Adım 6 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 7` `x = 2`

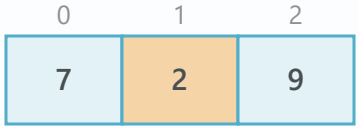
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



s += x → s = 7 + 2 = **9**.

Adım 7 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
3      for (int x : a) {
▶          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 9` `x = 2`

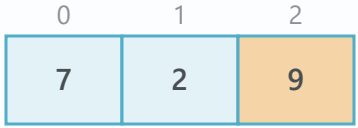
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



Döngü 3. tur: **x = a[2] = 9**.

Adım 8 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 9` `x = 9`

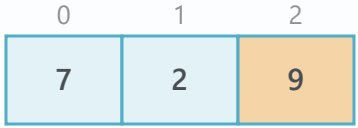
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



s += x → s = 9 + 9 = **18**.

Adım 9 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
3      for (int x : a) {
▶          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 18` `x = 9`

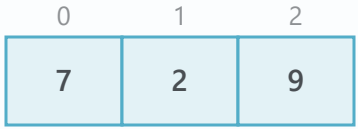
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



Dizide öğe kalmadı; döngüden çıkılır.

Adım 10 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 18`

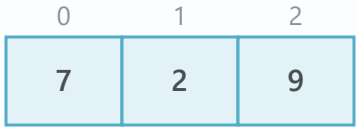
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



return 18: çerçeve kapanır, değer çağırana döner.

Adım 11 / 12

```
1  int toplam(int[] a) {
2      int s = 0;
3      for (int x : a) {
4          s += x;
5      }
▶      return s;
7  }
8
9  // main içinde:
10 System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]` `s = 18`

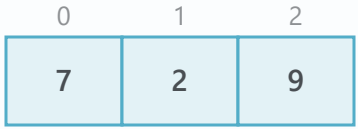
ÇAĞRI YIĞINI

`toplam(a)`

`main()`

ÇIKTI

Kod İzleme: `toplam()`



`println` 18 yazar. Döngü gövdesi **n = 3** kez çalıştı → **O(n)**.

Adım 12 / 12

```
1 int toplam(int[] a) {
2     int s = 0;
3     for (int x : a) {
4         s += x;
5     }
6     return s;
7 }
8
9 // main içinde:
▶ System.out.println(toplam(a));
```

DEĞİŞKENLER

`a = [7, 2, 9]`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
18
```

Kod İzleme: `dogrusalArama(a, 23)`

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

Sıralı bir dizide **23** aranıyor; doğrusal arama sıralı olmayı **kullanmaz**, baştan tek tek bakar.

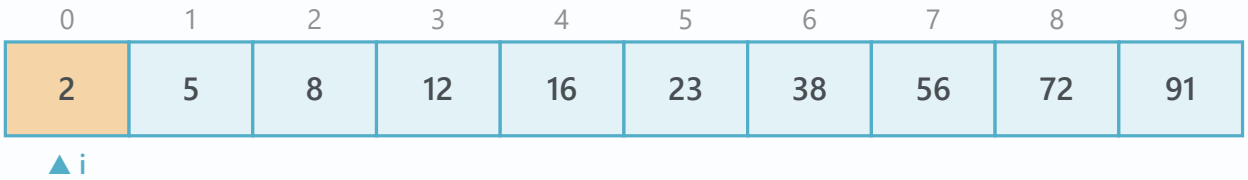
Adım 1 / 7

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

DEĞİŞKENLER

`hedef = 23`

Kod İzleme: `dogrusalArama(a, 23)`



a[0] = 2 ≠ 23 → i ilerler. Her adımda yalnızca **bir** öğe elenir.

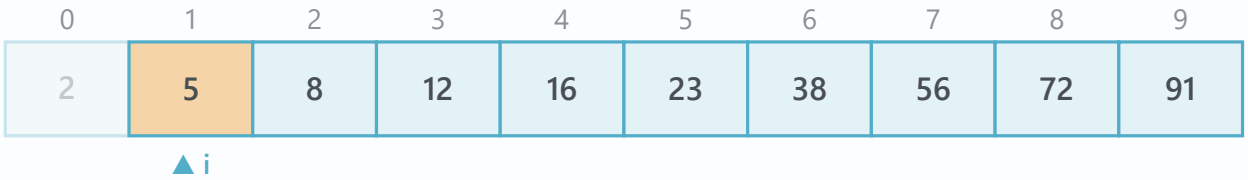
Adım 2 / 7

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

DEĞİŞKENLER

`hedef = 23` `i = 0` `a[i] = 2` `karşılaştırma = 1`

Kod İzleme: `dogrusalArama(a, 23)`



a[1] = 5 ≠ 23 → i ilerler.

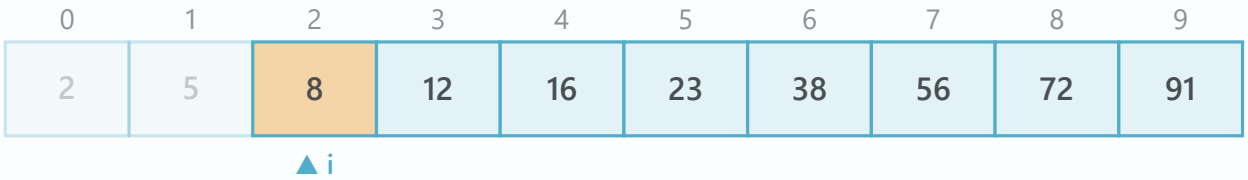
Adım 3 / 7

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

DEĞİŞKENLER

`hedef = 23` `i = 1` `a[i] = 5` `karşılaştırma = 2`

Kod İzleme: `dogrusalArama(a, 23)`



a[2] = 8 ≠ 23 → i ilerler.

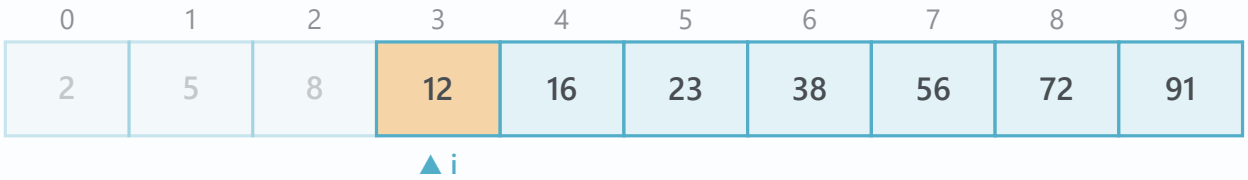
Adım 4 / 7

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

DEĞİŞKENLER

`hedef = 23` `i = 2` `a[i] = 8` `karşılaştırma = 3`

Kod İzleme: `dogrusalArama(a, 23)`



a[3] = 12 ≠ 23 → i ilerler.

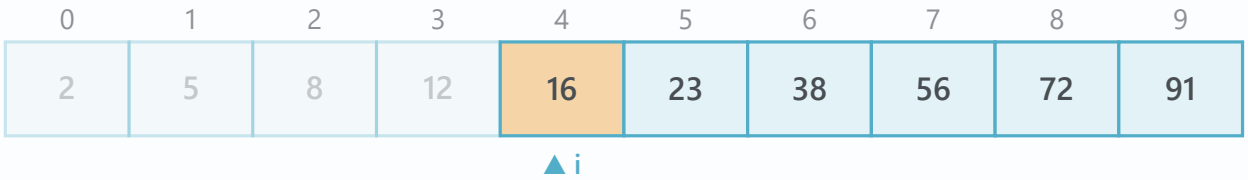
Adım 5 / 7

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

DEĞİŞKENLER

`hedef = 23` `i = 3` `a[i] = 12` `karşılaştırma = 4`

Kod İzleme: `dogrusalArama(a, 23)`



a[4] = 16 ≠ 23 → i ilerler.

Adım 6 / 7

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

DEĞİŞKENLER

`hedef = 23` `i = 4` `a[i] = 16` `karşılaştırma = 5`

Kod İzleme: `dogrusalArama(a, 23)`



a[5] = 23 == 23 → **return 5**. 6 karşılaştırma; hedef sonda ya da yoksa n = 10 karşılaştırma → **O(n)**.

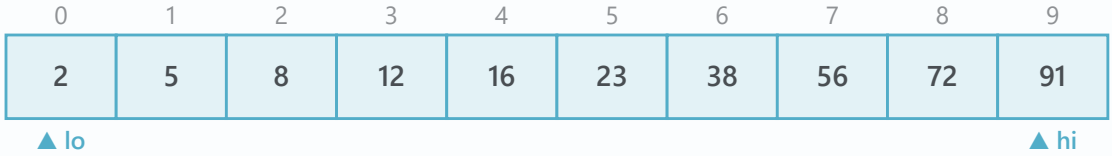
Adım 7 / 7

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

DEĞİŞKENLER

`hedef = 23` `i = 5` `a[i] = 23` `karşılaştırma = 6`

Kod İzleme: `ikiliArama(a, 23)`



Arama aralığı tüm dizi: **lo = 0**, **hi = 9**.

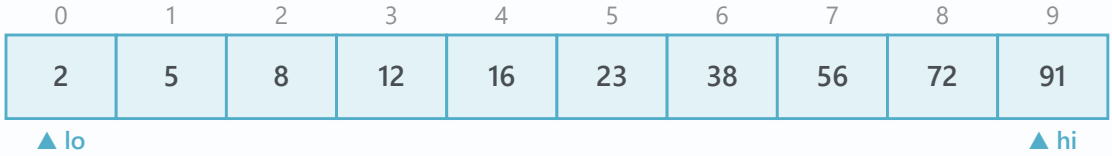
Adım 1 / 10

```
1  int ikiliArama(int[] a, int hedef) {
▶      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 0` `hi = 9`

Kod İzleme: `ikiliArama(a, 23)`



0 ≤ 9 → aralık boş değil, 1. tur başlar.

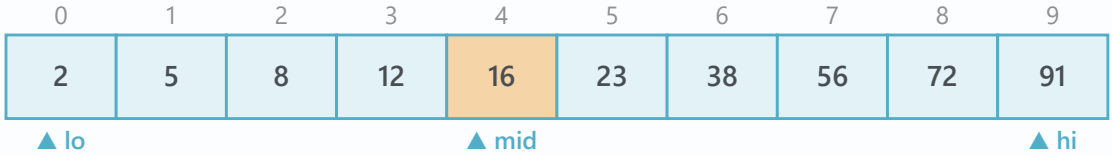
Adım 2 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 0` `hi = 9` `tur = 1`

Kod İzleme: `ikiliArama(a, 23)`



mid = (0 + 9) / 2 = 4, a[4] = 16.

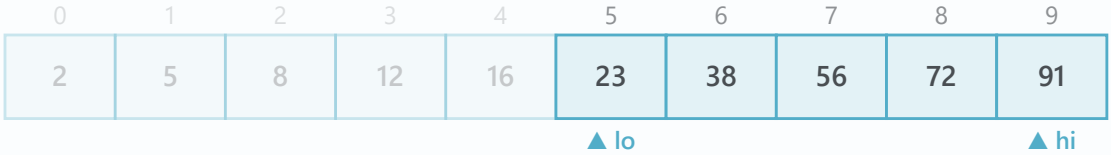
Adım 3 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 0` `hi = 9` `mid = 4` `a[mid] = 16` `tur = 1`

Kod İzleme: `ikiliArama(a, 23)`



16 < 23 → hedef sağ yarıda: **lo = mid + 1 = 5**. Sol yarı elendi.

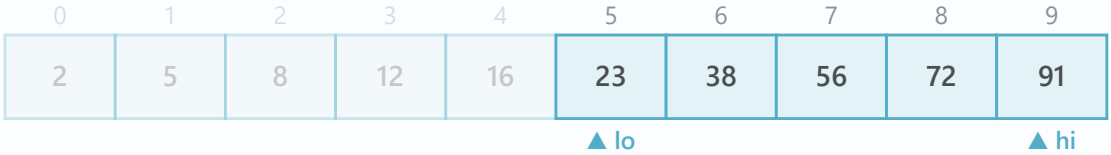
Adım 4 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
▶          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 9` `mid = 4` `a[mid] = 16` `tur = 1`

Kod İzleme: `ikiliArama(a, 23)`



5 ≤ 9 → aralık boş değil, 2. tur başlar.

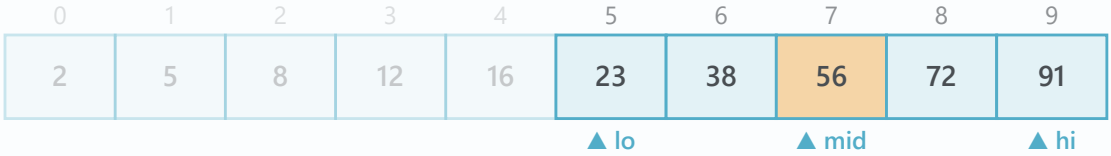
Adım 5 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 9` `tur = 2`

Kod İzleme: `ikiliArama(a, 23)`



mid = (5 + 9) / 2 = 7, a[7] = 56.

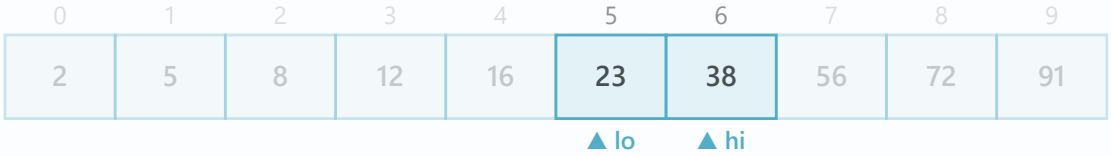
Adım 6 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 9` `mid = 7` `a[mid] = 56` `tur = 2`

Kod İzleme: `ikiliArama(a, 23)`



56 > 23 → hedef sol yarıda: **hi = mid − 1 = 6**. Sağ yarı elendi.

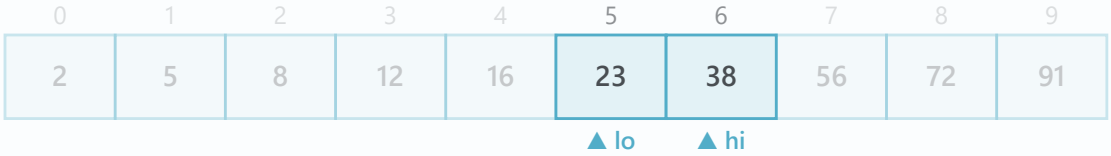
Adım 7 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
▶          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 6` `mid = 7`
`a[mid] = 56` `tur = 2`

Kod İzleme: `ikiliArama(a, 23)`



5 ≤ 6 → aralık boş değil, 3. tur başlar.

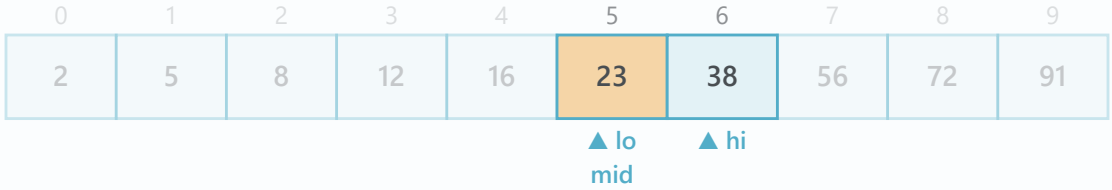
Adım 8 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 6` `tur = 3`

Kod İzleme: `ikiliArama(a, 23)`



mid = (5 + 6) / 2 = 5, a[5] = 23.

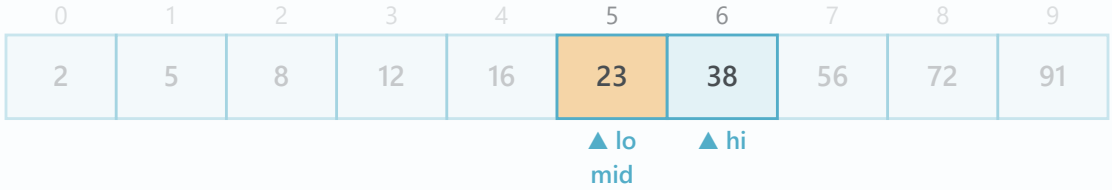
Adım 9 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 6` `mid = 5`
`a[mid] = 23` `tur = 3`

Kod İzleme: `ikiliArama(a, 23)`



23 == 23 → bulundu, **return 5**. Yalnızca **3 tur**; doğrusal arama 6 karşılaştırma yapardı. Her tur aralığı yarıya indirir → **O(log n)**.

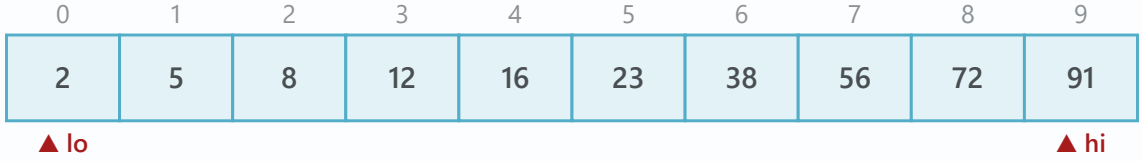
Adım 10 / 10

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
▶          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 23` `lo = 5` `hi = 6` `mid = 5`
`a[mid] = 23` `tur = 3`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



Bu kez **40** dizide **yok**: en kötü durum. Aralık tüm dizi.

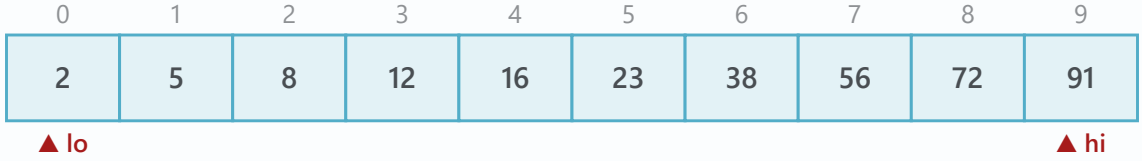
Adım 1 / 15

```
1  int ikiliArama(int[] a, int hedef) {
▶      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 0` `hi = 9`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



0 ≤ 9 → aralık boş değil, 1. tur.

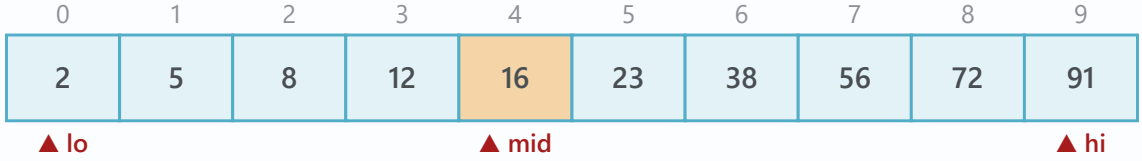
Adım 2 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 0` `hi = 9` `tur = 1`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



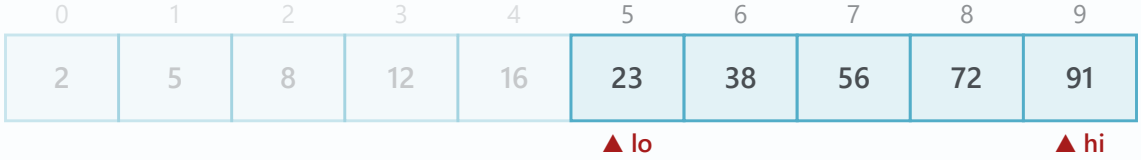
1. tur: **mid = (0 + 9) / 2 = 4**, a[4] = 16.


```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 0` `hi = 9` `mid = 4`
`a[mid] = 16` `tur = 1`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



16 < 40 → **lo = 5**; sol taraf elendi.

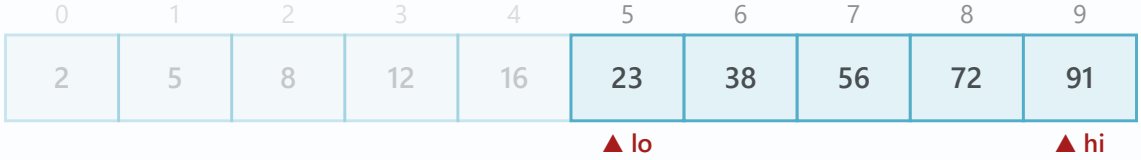
Adım 4 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
▶          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 5` `hi = 9` `mid = 4` `a[mid] = 16` `tur = 1`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



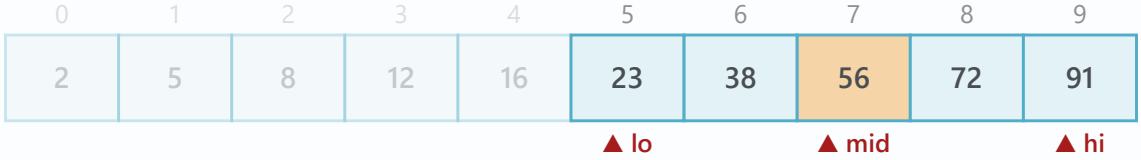
5 ≤ 9 → aralık boş değil, 2. tur.


```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 5` `hi = 9` `tur = 2`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



2. tur: **mid = (5 + 9) / 2 = 7**, a[7] = 56.

Adım 6 / 15

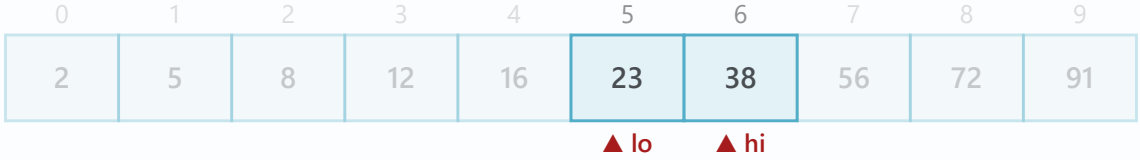
```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 5` `hi = 9` `mid = 7`

`a[mid] = 56` `tur = 2`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



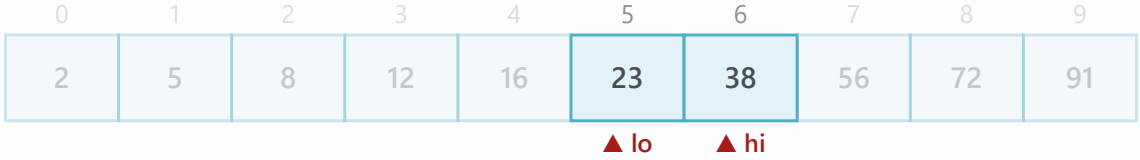
56 > 40 → **hi = 6**; sağ taraf elendi.


```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
▶          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 5` `hi = 6` `mid = 7`
`a[mid] = 56` `tur = 2`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



5 ≤ 6 → aralık boş değil, 3. tur.

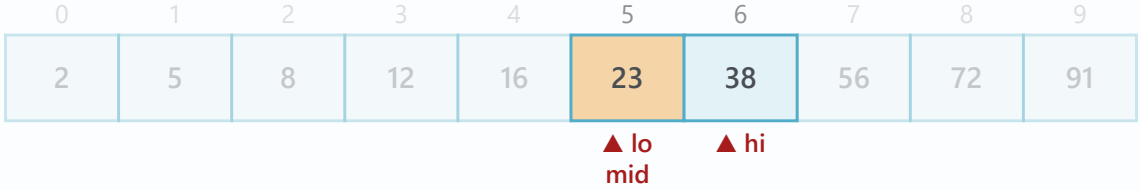
Adım 8 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 5` `hi = 6` `tur = 3`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



3. tur: **mid = (5 + 6) / 2 = 5**, a[5] = 23.

Adım 9 / 15

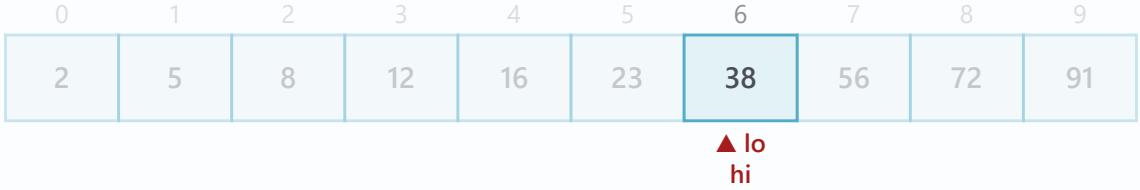
```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 5` `hi = 6` `mid = 5`

`a[mid] = 23` `tur = 3`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



23 < 40 → **lo = 6**; sol taraf elendi.

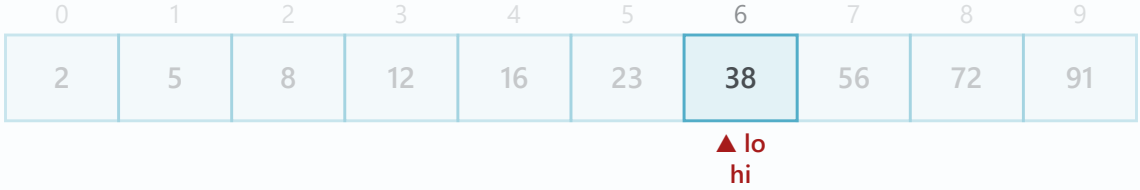
Adım 10 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
▶          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 6` `hi = 6` `mid = 5` `a[mid] = 23` `tur = 3`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



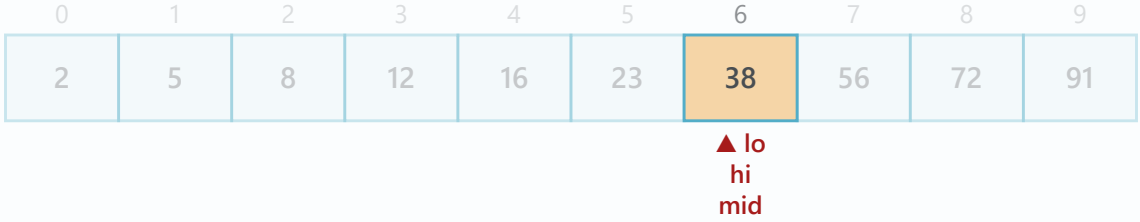
6 ≤ 6 → aralık boş değil, 4. tur.


```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 6` `hi = 6` `tur = 4`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



4. tur: **mid = (6 + 6) / 2 = 6**, a[6] = 38.

Adım 12 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 6` `hi = 6` `mid = 6`

`a[mid] = 38` `tur = 4`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



38 < 40 → **lo = 7**; sol taraf elendi.

Adım 13 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
▶          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 7` `hi = 6` `mid = 6` `a[mid] = 38` `tur = 4`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



7 ≤ 6 yanlış → aralık boş, döngü biter.

Adım 14 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 7` `hi = 6` `tur = 4`

Kod İzleme: `ikiliArama(a, 40)` — hedef yok



return -1. En kötü durumda bile yalnız **4 tur** = $\lfloor \log_2 10 \rfloor + 1$; doğrusal arama 10 karşılaştırma yapardı.

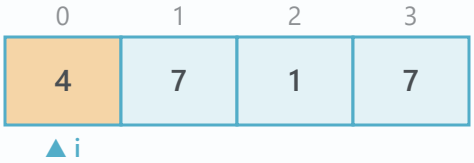
Adım 15 / 15

```
1  int ikiliArama(int[] a, int hedef) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == hedef) return mid;
6          if (a[mid] < hedef) lo = mid + 1;
7          else hi = mid - 1;
8      }
▶      return -1;
10 }
```

DEĞİŞKENLER

`hedef = 40` `lo = 7` `hi = 6` `tur = 4`

Kod İzleme: `tekrarVar()`



Dış döngü: **i = 0** (a[i] = 4). j, i+1'den başlar.

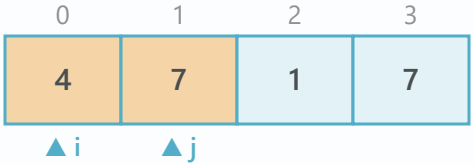
Adım 1 / 7

```
1 boolean tekrarVar(int[] a) {
▶     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
4             if (a[i] == a[j])
5                 return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 0` `karşılaştırma = 0`

Kod İzleme: `tekrarVar()`



a[0] = 4 ile a[1] = 7 karşılaştırılır → farklı, j ilerler.

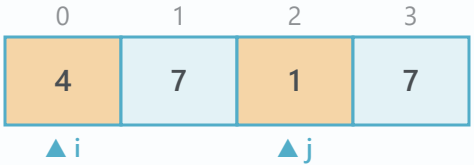
Adım 2 / 7

```
1 boolean tekrarVar(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 0` `j = 1` `a[i] = 4` `a[j] = 7` `karşılaştırma = 1`

Kod İzleme: `tekrarVar()`



a[0] = 4 ile a[2] = 1 karşılaştırılır → farklı, j ilerler.

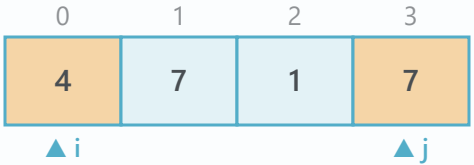
Adım 3 / 7

```
1 boolean tekrarVar(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 0` `j = 2` `a[i] = 4` `a[j] = 1` `karşılaştırma = 2`

Kod İzleme: `tekrarVar()`



a[0] = 4 ile a[3] = 7 karşılaştırılır → farklı, j ilerler.

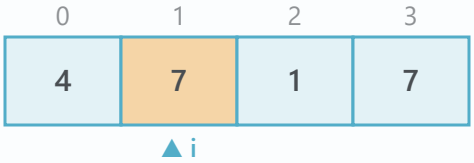
Adım 4 / 7

```
1 boolean tekrarVar(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 0` `j = 3` `a[i] = 4` `a[j] = 7` `karşılaştırma = 3`

Kod İzleme: `tekrarVar()`



Dış döngü: **i = 1** (a[i] = 7). j, i+1'den başlar.

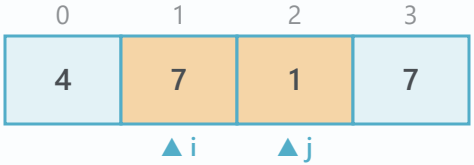
Adım 5 / 7

```
1 boolean tekrarVar(int[] a) {
▶     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
4             if (a[i] == a[j])
5                 return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 1` `karşılaştırma = 3`

Kod İzleme: `tekrarVar()`



a[1] = 7 ile a[2] = 1 karşılaştırılır → farklı, j ilerler.

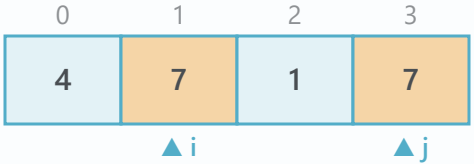
Adım 6 / 7

```
1 boolean tekrarVar(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶             if (a[i] == a[j])
5                 return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 1` `j = 2` `a[i] = 7` `a[j] = 1` `karşılaştırma = 4`

Kod İzleme: `tekrarVar()`



a[1] = 7 ve a[3] = 7 **eşit** → **return true**. En kötü durumda (tekrar yoksa) n(n−1)/2 = 6 karşılaştırma yapılırdı → **O(n²)**.

Adım 7 / 7

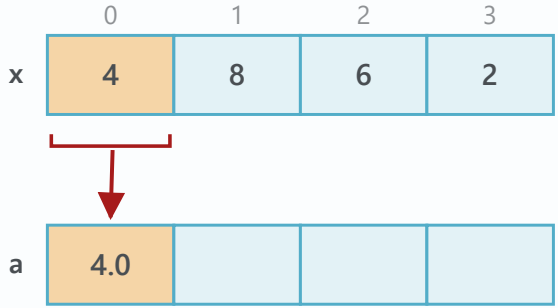
```
1 boolean tekrarVar(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
▶                   return true;
6     return false;
7 }
```

DEĞİŞKENLER

`i = 1` `j = 3` `a[i] = 7` `a[j] = 7` `karşılaştırma = 5`

Örnek Problem: Önek Ortalamaları

Girdi x[0..n−1]; çıktı **a[j] = (x[0] + … + x[j]) / (j + 1)**. Kullanım: hareketli ortalama, finansal eğilim analizi (Goodrich §4.3).



a[0] = (4) / 1 = 4.0

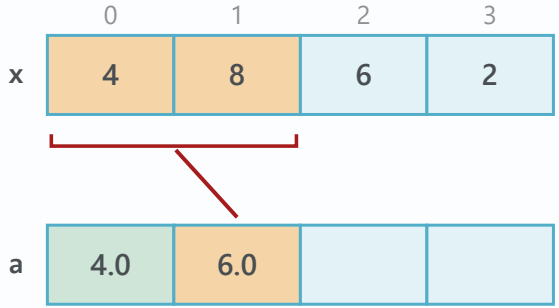
a[0] = x[0] … x[0] öğelerinin ortalaması = **4.0**.

Adım 1 / 4

İki çözüm: her j için baştan topla (O(n^2)) · birikimli toplamı taşı (O(n)).

Örnek Problem: Önek Ortalamaları

Girdi x[0..n−1]; çıktı **a[j] = (x[0] + … + x[j]) / (j + 1)**. Kullanım: hareketli ortalama, finansal eğilim analizi (Goodrich §4.3).



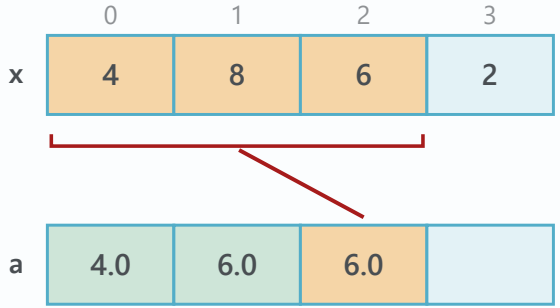
a[1] = (4 + 8) / 2 = 6.0

a[1] = x[0] … x[1] öğelerinin ortalaması = **6.0**.

İki çözüm: her j için baştan topla ($O(n^2)$) · birikimli toplamı taşı ($O(n)$).

Örnek Problem: Önek Ortalamaları

Girdi x[0..n−1]; çıktı **a[j] = (x[0] + … + x[j]) / (j + 1)**. Kullanım: hareketli ortalama, finansal eğilim analizi (Goodrich §4.3).



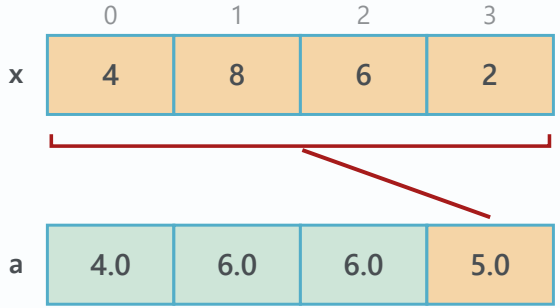
a[2] = (4 + 8 + 6) / 3 = 6.0

a[2] = x[0] … x[2] öğelerinin ortalaması = **6.0**.

İki çözüm: her j için baştan topla (O(n^2)) · birikimli toplamı taşı (O(n)).

Örnek Problem: Önek Ortalamaları

Girdi x[0..n−1]; çıktı **a[j] = (x[0] + … + x[j]) / (j + 1)**. Kullanım: hareketli ortalama, finansal eğilim analizi (Goodrich §4.3).



a[3] = (4 + 8 + 6 + 2) / 4 = 5.0

a[3] = x[0] … x[3] öğelerinin ortalaması = **5.0**.

İki çözüm: her j için baştan topla (O(n^2)) · birikimli toplamı taşı (O(n)).

Kod İzleme: `onekOrt1` — her j için baştan topla



n = 4; sonuç dizisi **a** oluşturulur (0.0 ile dolu).

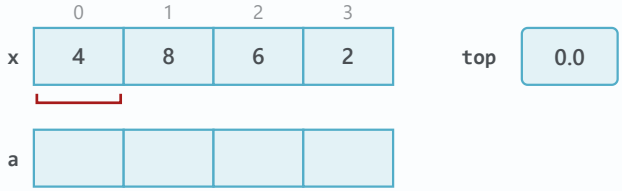
Adım 1 / 20

```
1  double[] onekOrt1(double[] x) {
▶    int n = x.length;
▶    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`n = 4` `a = [0.0, 0.0, 0.0, 0.0]`

Kod İzleme: `onekOrt1` — her j için baştan topla



j = 0: top = 0 — her j için toplam **baştan** başlar.

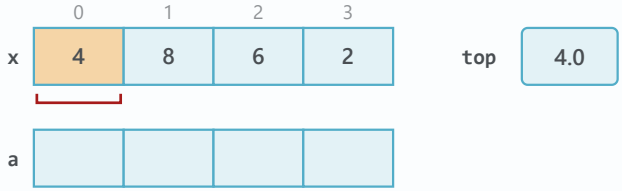
Adım 2 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 0` `top = 0.0` `toplama = 0`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[0] = 4 → top = 4.0.

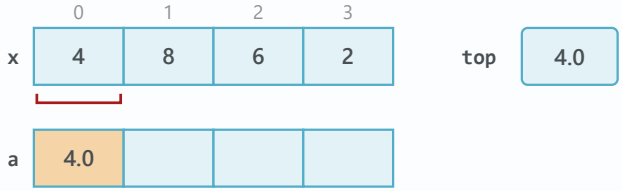
Adım 3 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 0` `i = 0` `top = 4.0` `toplama = 1`

Kod İzleme: `onekOrt1` — her j için baştan topla



a[0] = 4.0 / 1 = **4.0**.

Adım 4 / 20

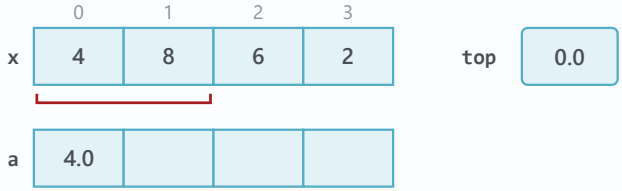
```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
▶      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 0` `top = 4.0`

`a = [4.0, 0.0, 0.0, 0.0]` `toplama = 1`

Kod İzleme: `onekOrt1` — her j için baştan topla



j = 1: top = 0 — her j için toplam **baştan** başlar.

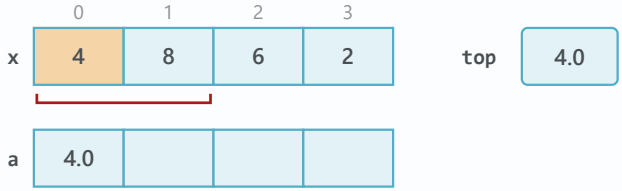
Adım 5 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 1` `top = 0.0` `toplama = 1`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[0] = 4 → top = 4.0.

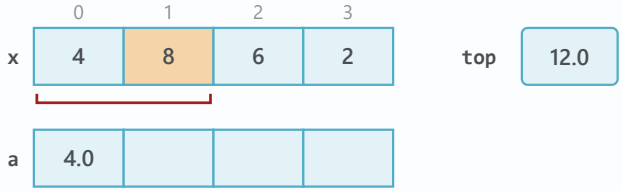
Adım 6 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 1` `i = 0` `top = 4.0` `toplama = 2`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[1] = 8 → top = 12.0.

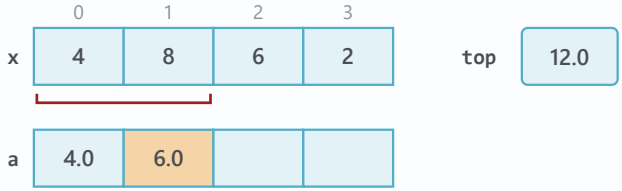
Adım 7 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 1` `i = 1` `top = 12.0` `toplama = 3`

Kod İzleme: `onekOrt1` — her j için baştan topla



a[1] = 12.0 / 2 = **6.0**.

Adım 8 / 20

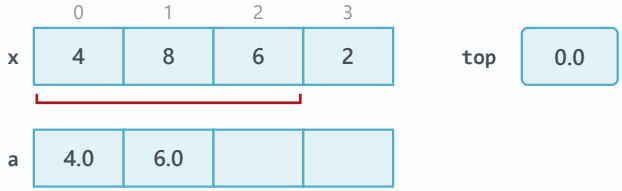
```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
▶      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 1` `top = 12.0`

`a = [4.0, 6.0, 0.0, 0.0]` `toplama = 3`

Kod İzleme: `onekOrt1` — her j için baştan topla



j = 2: top = 0 — her j için toplam **baştan** başlar.

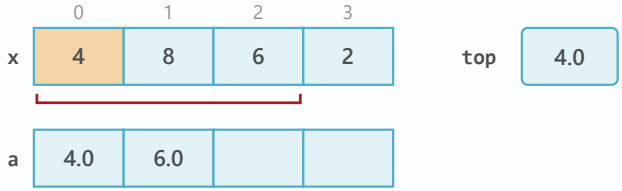
Adım 9 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 2` `top = 0.0` `toplama = 3`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[0] = 4 → top = 4.0.

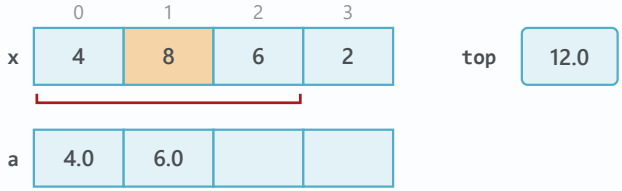
Adım 10 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 2` `i = 0` `top = 4.0` `toplama = 4`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[1] = 8 → top = 12.0.

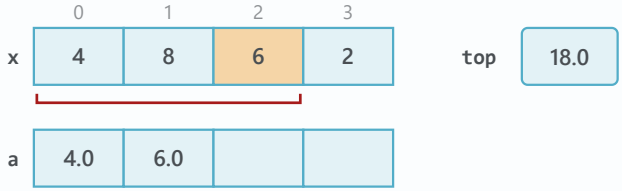
Adım 11 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
       top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 2` `i = 1` `top = 12.0` `toplama = 5`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[2] = 6 → top = 18.0.

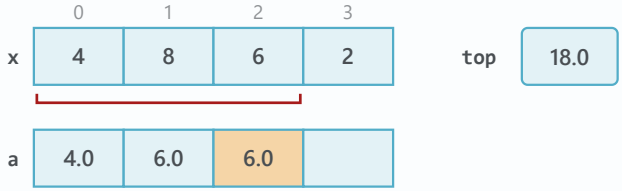
Adım 12 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 2` `i = 2` `top = 18.0` `toplama = 6`

Kod İzleme: `onekOrt1` — her j için baştan topla



a[2] = 18.0 / 3 = **6.0**.

Adım 13 / 20

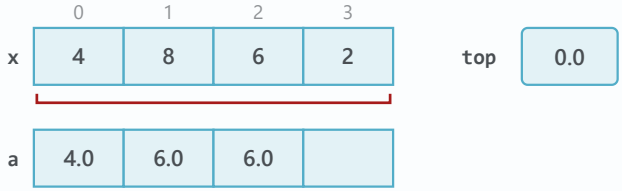
```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
▶      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 2` `top = 18.0`

`a = [4.0, 6.0, 6.0, 0.0]` `toplama = 6`

Kod İzleme: `onekOrt1` — her j için baştan topla



j = 3: top = 0 — her j için toplam **baştan** başlar.

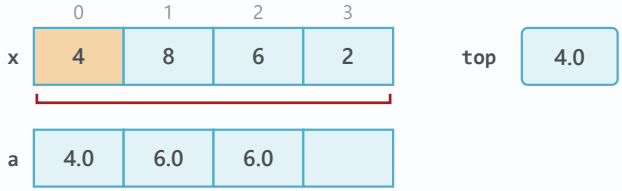
Adım 14 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 3` `top = 0.0` `toplama = 6`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[0] = 4 → top = 4.0.

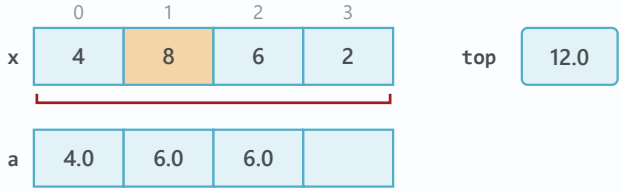
Adım 15 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 3` `i = 0` `top = 4.0` `toplama = 7`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[1] = 8 → top = 12.0.

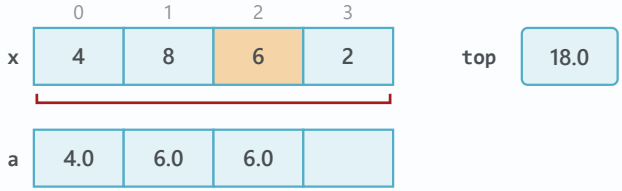
Adım 16 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 3` `i = 1` `top = 12.0` `toplama = 8`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[2] = 6 → top = 18.0.

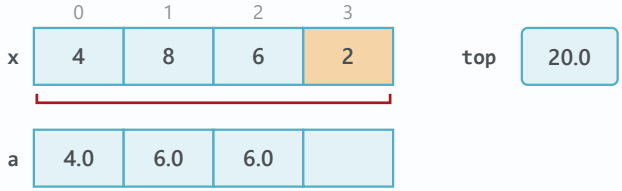
Adım 17 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 3` `i = 2` `top = 18.0` `toplama = 9`

Kod İzleme: `onekOrt1` — her j için baştan topla



top += x[3] = 2 → top = 20.0.

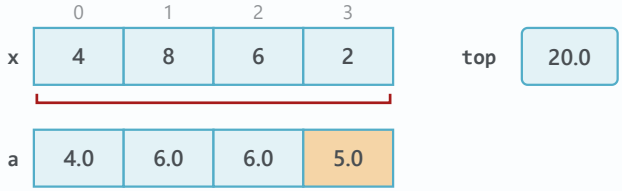
Adım 18 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
▶        top += x[i];
8      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 3` `i = 3` `top = 20.0` `toplama = 10`

Kod İzleme: `onekOrt1` — her j için baştan topla



a[3] = 20.0 / 4 = **5.0**.

Adım 19 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
▶      a[j] = top / (j + 1);
9    }
10   return a;
11 }
```

DEĞİŞKENLER

`j = 3` `top = 20.0`

`a = [4.0, 6.0, 6.0, 5.0]` `toplama = 10`

Kod İzleme: `onekOrt1` — her j için baştan topla

	0	1	2	3
x	4	8	6	2

a	4.0	6.0	6.0	5.0

return a. Toplama sayısı 1 + 2 + 3 + 4 = **10** = n(n+1)/2 → **O(n²)**.

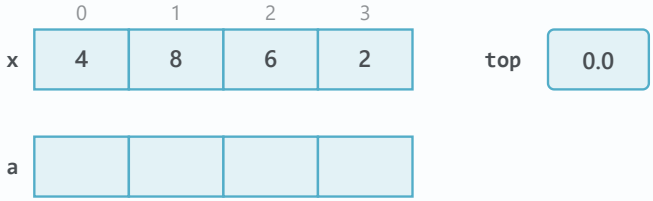
Adım 20 / 20

```
1  double[] onekOrt1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double top = 0;
6      for (int i = 0; i <= j; i++)
7        top += x[i];
8      a[j] = top / (j + 1);
9    }
▶    return a;
11 }
```

DEĞİŞKENLER

`a = [4.0, 6.0, 6.0, 5.0]` `toplama = 10`

Kod İzleme: `onekOrt2` — birikimli toplam



top döngüden **önce** bir kez sıfırlanır ve turlar boyunca korunur.

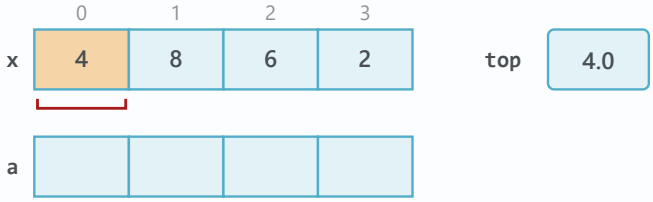
Adım 1 / 10

```
1  double[] onekOrt2(double[] x) {
▶    int n = x.length;
▶    double[] a = new double[n];
▶    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
7      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

DEĞİŞKENLER

`n = 4` `a = [0.0, 0.0, 0.0, 0.0]` `top = 0.0`

Kod İzleme: `onekOrt2` — birikimli toplam



j = 0: top += x[0] = 4 → top = 4.0 (önceki toplam yeniden kullanıldı).

Adım 2 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
▶      top += x[j];
7      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

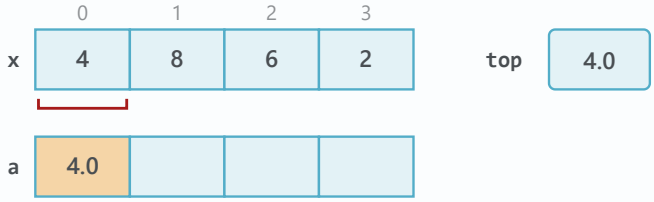
DEĞİŞKENLER

`j = 0` `top = 4.0`

`a = [0.0, 0.0, 0.0, 0.0]`

`toplama = 1`

Kod İzleme: `onekOrt2` — birikimli toplam



a[0] = 4.0 / 1 = **4.0**.

Adım 3 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
▶      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

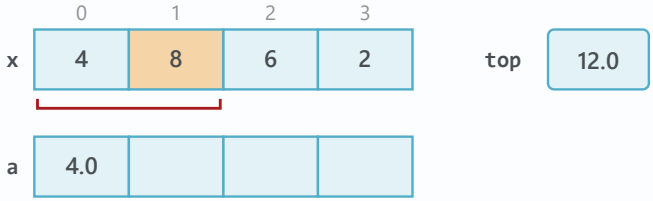
DEĞİŞKENLER

`j = 0` `top = 4.0`

`a = [4.0, 0.0, 0.0, 0.0]`

`toplama = 1`

Kod İzleme: `onekOrt2` — birikimli toplam



j = 1: top += x[1] = 8 → top = 12.0 (önceki toplam yeniden kullanıldı).

Adım 4 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
▶      top += x[j];
7      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

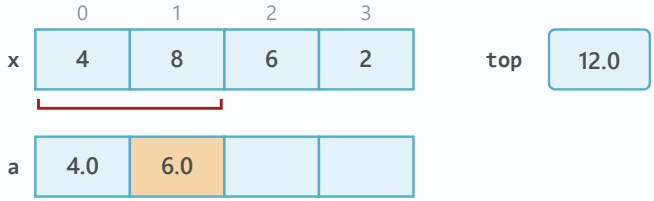
DEĞİŞKENLER

`j = 1` `top = 12.0`

`a = [4.0, 0.0, 0.0, 0.0]`

`toplama = 2`

Kod İzleme: `onekOrt2` — birikimli toplam



a[1] = 12.0 / 2 = **6.0**.

Adım 5 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
▶      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

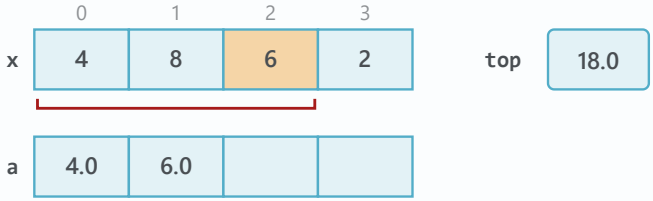
DEĞİŞKENLER

`j = 1` `top = 12.0`

`a = [4.0, 6.0, 0.0, 0.0]`

`toplama = 2`

Kod İzleme: `onekOrt2` — birikimli toplam



j = 2: top += x[2] = 6 → top = 18.0 (önceki toplam yeniden kullanıldı).

Adım 6 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
7      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

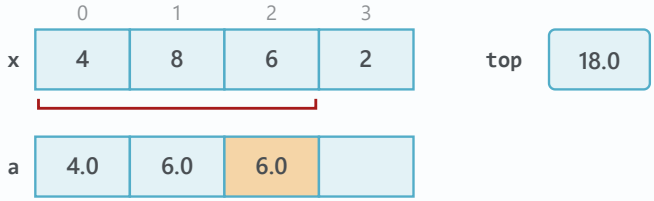
DEĞİŞKENLER

`j = 2` `top = 18.0`

`a = [4.0, 6.0, 0.0, 0.0]`

`toplama = 3`

Kod İzleme: `onekOrt2` — birikimli toplam



a[2] = 18.0 / 3 = **6.0**.

Adım 7 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
▶      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

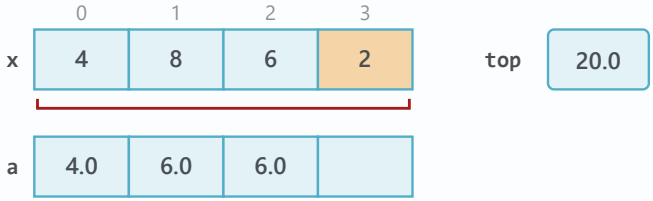
DEĞİŞKENLER

`j = 2` `top = 18.0`

`a = [4.0, 6.0, 6.0, 0.0]`

`toplama = 3`

Kod İzleme: `onekOrt2` — birikimli toplam



j = 3: top += x[3] = 2 → top = 20.0 (önceki toplam yeniden kullanıldı).

Adım 8 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
▶      top += x[j];
7      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

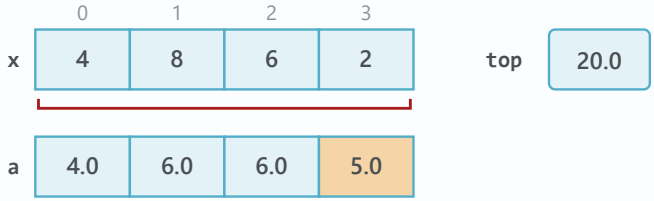
DEĞİŞKENLER

`j = 3` `top = 20.0`

`a = [4.0, 6.0, 6.0, 0.0]`

`toplama = 4`

Kod İzleme: `onekOrt2` — birikimli toplam



a[3] = 20.0 / 4 = **5.0**.

Adım 9 / 10

```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
▶      a[j] = top / (j + 1);
8    }
9    return a;
10 }
```

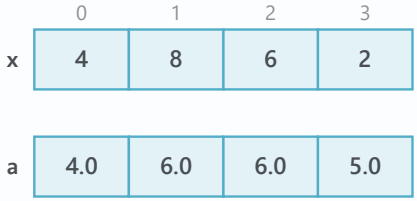
DEĞİŞKENLER

`j = 3` `top = 20.0`

`a = [4.0, 6.0, 6.0, 5.0]`

`toplama = 4`

Kod İzleme: `onekOrt2` — birikimli toplam



Aynı sonuç, yalnız **4** toplama (onekOrt1: 10). Tek döngü → **O(n)**.

Adım 10 / 10

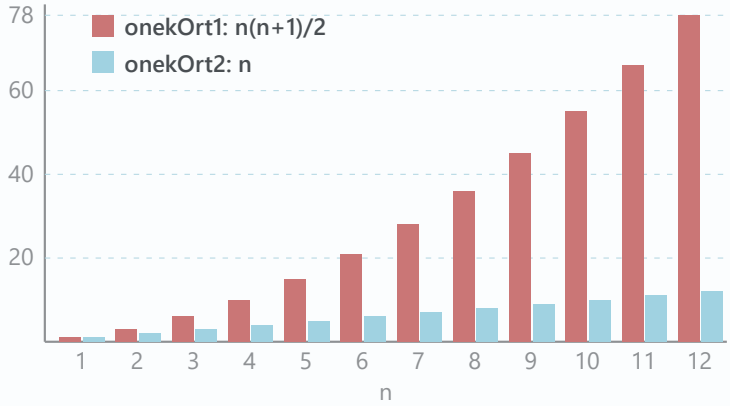
```
1  double[] onekOrt2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double top = 0;
5    for (int j = 0; j < n; j++) {
6      top += x[j];
7      a[j] = top / (j + 1);
8    }
▶    return a;
10 }
```

DEĞİŞKENLER

`a = [4.0, 6.0, 6.0, 5.0]`

`toplama = 4`

İki Çözümün Karşılaştırması



n	onekOrt1	onekOrt2
10	55	10
100	5.050	100
1.000	500.500	1.000
1.000.000	500.000.500.000	1.000.000

Fark, **tekrarlanan işi** ortadan kaldırmaktan geliyor: onekOrt2 önceki toplamı hatırlar. n = 10^6 için ~$5 \cdot 10^{11}$ yerine 10^6 toplama.

Alan Karmaşıklığı

- Algoritmanın kullandığı bellek miktarını ölçer; genellikle Büyük O gösterimiyle ifade edilir.
- Bellek kullanımının girdi boyutuna bağlı artışını gösterir (girdinin kendisi çoğu zaman sayılmaz; ek alan).
- Sınırlı bellek kaynaklarına sahip sistemlerde bellek verimliliği önemlidir.

O(1) · Sabit

Örnek: bir değişken oluşturma. Bellek kullanımı girdi boyutuyla değişmez.



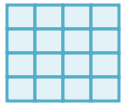
O(n) · Doğrusal

Örnek: bir dizinin tüm öğelerini saklama; onekOrt'un a dizisi.



O(n^2) · Kare

Örnek: n×n boyutunda bir matrisi saklama. Bellek, n'nin karesiyle artar.



Doğruluk (Correctness)

- Doğru bir algoritma, her geçerli girdi için sonlu sürede beklenen çıktıyı üretir.
- Hızlı çalışan ama yanlış sonuç veren bir algoritmanın değeri yoktur.

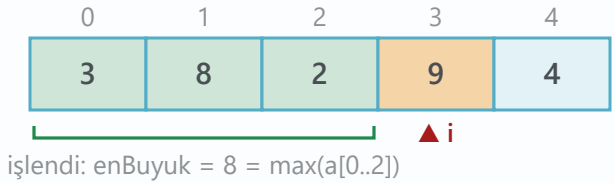
Doğruluk nasıl gösterilir?

Test Senaryoları
Çıktılar, bilinen doğru sonuçlarla karşılaştırılır.

Uç Durumlar
Boş girdi, tek öğe, en büyük/en küçük değerler.

Döngü Değişmezi
Her adımda doğru kalan bir özellikle ispat.

Karşı Örnek
Tek bir yanlış sonuç, algoritmanın doğru olmadığını gösterir.



işlendi: enBuyuk = 8 = max(a[0..2])

Değişmez (invariant):
her turun başında enBuyuk, a[0..i−1]'in en büyüğüdür.

Döngü bitince i = n → doğru sonuç.

ALAN VE DOĞRULUK

Sağlamlık (Robustness)

- Algoritmanın beklenmeyen durumlarla başa çıkma yeteneğidir.
- Sağlam bir algoritma, normal aralığın dışındaki girdileri bile çökmeden veya yanlış sonuç üretmeden işleyebilir.

Sağlamlık nasıl sınanır?

Geçersiz Girdiler
Uygun olmayan veya beklenmeyen girdilerle başa çıkma.
`null, negatif boyut`

Hatalı Veriler
Bozuk veya eksik verilerle başa çıkma.
`eksik satır, yanlış biçim`

Büyük Veri Kümeleri
Çok büyük girdileri işleyebilme.
`10⁸ kayıt`

Aşırı Yüklenme
Algoritmanın aşırı yük altında nasıl davrandığı.
`aynı anda binlerce istek`

ÖZET

Sık Yapılan Hatalar

“O = en kötü durum”

O bir **üst sınır** gösterimidir; en iyi / en kötü durum ayrı kavramdır. Her durum için O, Ω, Θ yazılabilir.

“O(1) her zaman hızlıdır”

Sabit büyük olabilir: her çağrıda 1000 adım atan O(1) işlem, $n < 1000$ iken n adımlık O(n) işlemden yavaştır.

“İç içe döngü = n^2”

İç döngü `j *= 2` ile ilerliyorsa n log n; sabit sayıda dönüyorsa O(n).

“log tabanı önemlidir”

$\log_2 n$ ile $\log_{10} n$ arasında yalnız sabit çarpan var: ikisi de O(log n).

`(lo + hi) / 2` taşması

Çok büyük dizilerde `int` taşar; `lo + (hi - lo) / 2` güvenlidir.

`a == b` ile dizi karşılaştırmak

`==` referansları karşılaştırır; içerik için `Arrays.equals(a, b)`.

Laboratuvar: İkili Arama ve Önek Ortalamaları

İkili arama: 23 aranıyor

Girdi (sıralı sayılar | aranan) 2, 5, 8, 12, 16, 23, 38, 56, 72, 91 | 23

KOD · **adım 1 / 13**

```
 1 int ikiliArama(int[] a, int hedef) {
 2     int lo = 0, hi = a.length - 1;
 3     while (lo <= hi) {
 4         int mid = (lo + hi) / 2;
 5         if (a[mid] == hedef) return mid;
 6         if (a[mid] < hedef) lo = mid + …
 7         else hi = mid - 1;
 8     }
 9     return -1;
10 }
```

a

2	5	8	12	16	23	38	56	72	91

hedef = **23** tur = **0**

DÖNÜŞ DEĞERİ –

ikiliArama, 10 öğeli sıralı a dizisi ve hedef = 23 ile çağrılır.

Özet

Konu	Hatırlanacaklar
Deneysel analiz	süre donanıma ve girdiye bağlıdır; katlama deneyi: $T(2n)/T(n) \approx 2^k \rightarrow T(n) \approx c \cdot n^k$
İlkel işlem sayma	atama, karşılaştırma, aritmetik, dizi erişimi; donanımdan bağımsız, tüm girdiler için
O, Ω, Θ	üst, alt ve sıkı sınır; sabit çarpanlar ve küçük terimler atılır
Durumlar	en iyi, ortalama, en kötü; garanti için en kötü durum, uzun işlem dizisi için amortize
Döngü kalıpları	tek döngü $O(n)$ · iç içe $O(n^2)$ · yarıya inen $O(\log n)$ · dış n, iç $\log n \rightarrow O(n \log n)$
Algoritma seçimi	önek ortalamaları $O(n^2) \rightarrow O(n)$; sıralı dizide ikili arama $O(\log n)$
Alan, doğruluk	ek bellek de Büyük O ile ölçülür; doğru ve sağlam olmayan hızlı algoritmanın değeri yoktur

ÖZET

Temel Veri Yapılarında İşlem Maliyetleri

Veri yapısı	i. öğeye erişim	Arama	Ekleme	Silme
Dizi	O(1)	O(n)	O(n)	O(n)
Bağlı liste	O(n)	O(n)	O(1)†	O(1)†
Yığın	yalnızca üst	O(n)	O(1) push	O(1) pop
Kuyruk	yalnızca ön	O(n)	O(1) enqueue	O(1) dequeue
Eşleme / Küme (hash)	—	O(1)*	O(1)*	O(1)*

† Konum (önceki düğüm) biliniyorsa. * Ortalama durum; en kötü durumda O(n).

ÖZET

Örnek Problemler

1. Tüm öğe çiftlerini karşılaştıran algoritmanın zaman karmaşıklığı nedir?
2. $f(n) = 5n^3 + 2n^2 + 100$ hangi Θ sınıfındadır?
3. $n = 1024$ öğeli sıralı dizide ikili arama en kötü durumda kaç tur döner?
4. `for (int i = 1; i < n; i *= 2)` döngüsü kaç kez döner?
5. Doğrusal aramanın en iyi ve en kötü durumu nedir?
6. A algoritması $100n$, B algoritması n^2 adım atıyor. Hangi n değerleri için A daha hızlıdır?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. Kod parçasının Büyük O'su nedir?

```
for (int i = n; i > 1; i /= 2)
  for (int j = 0; j < n; j++) s++;
```

2. f(n) = $3n^2 + 5n + 2 = O(n^2)$ olduğunu **c** ve $\mathbf{n_0}$ vererek gösterin.
3. Katlama deneyinde n iki katına çıkınca süre ≈ 8 katına çıkıyor. T(n) hangi sınıfta?
4. n = 5 için onekOrt1 ve onekOrt2 kaçar toplama yapar?
5. `for (i = 0; i < n; i++) for (j = i; j < n; j++) s++;` sonunda s kaçtır?
6. $O(n^2)$ bir algoritma n = 1000 için 2 saniye sürüyor. n = 4000 için süre yaklaşık kaçtır?

▶ **Cevaplar / ipuçları**

Kaynaklar

- Goodrich, Tamassia ve Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, 2014.
- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022.
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, 2011.

Son

Bölüm 2: Algoritma Analizi

Sonraki sunum: Bölüm 3: Diziler · Çalışma soruları · Terim sözlüğü