Bölüm 12: Rastgele Algoritmalar

Algoritmalar

Rastgelelik ve beklenen süre | Las Vegas ve Monte Carlo | Fisher–Yates karıştırma | BogoSort | Rastgele QuickSort | Monte Carlo ile π | Miller–Rabin | Karger min-kesim

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavram

Rastgele algoritmanın tanımını, beklenen çalışma süresini ve Las Vegas / Monte Carlo ayrımını açıklamak.

Karıştırma ve BogoSort

Fisher–Yates karıştırmasını yazmak, saf karıştırmanın neden yanlı olduğunu göstermek; BogoSort'un beklenen $\Theta(n \cdot n!)$ süresini hesaplamak.

Rastgele QuickSort

Rastgele pivotlu bölümlemeyi adım adım izlemek; beklenen $O(n \log n)$ ve en kötü $O(n^2)$ süreyi gerekçelendirmek.

Monte Carlo

π tahmini, Miller–Rabin asallık testi ve Karger min-kesim algoritmasında hata olasılığını hesaplamak ve tekrarla küçültmek.

GİRİŞ

Şanslı Algoritmalar

- Bazen problem çözmede **şans oyunlarına** benzer teknikler kullanılır.
- Algoritma, karar verme sürecinde **rastgelelikten** (şans) yararlanır: yazı-tura atar, zar atar, rastgele bir öğe seçer.
- Örnek oyun: iki zar atılır; toplam **7 ya da 11** ise oyuncu kazanır.



kazanma: 7 ya da 11

P(7) = 6/36, P(11) = 2/36 → P(kazanma) = 8/36 = **2/9 ≈ 0,22**. Kazanana kadar beklenen atış sayısı = 1 / (2/9) = **4,5**.

Rastgele Algoritmalar



- **Rastgele algoritma** (randomized algorithm): girdiye ek olarak **rastgele sayılar** kullanarak çalışır.
- Aynı girdiyle iki çalıştırma **farklı adımlar** izleyebilir: çalışma süresi (ve bazen çıktı) her çalıştırmada değişebilir.
- Süre artık bir **rastgele değişkendir**: **beklenen süre** E[T(n)] incelenir. Beklenen değer algoritmanın *kendi* yazı-turaları üzerinden alınır, girdi üzerinden değil.

GİRİŞ

Neden Rastgelelik?

Kötü girdiye karşı koruma

Deterministik QuickSort sıralı dizide $O(n^2)$ olur. Rastgele pivotta **hiçbir girdi** sürekli kötü değildir: kötü durum yalnızca şanssız yazı-turalarla oluşur.

Basitlik

Rastgele algoritma çoğu zaman en iyi deterministik algoritmadan çok daha kısa ve anlaşılırdır (Karger min-kesim, rastgele QuickSelect).

Hız

Miller–Rabin, yüzlerce basamaklı sayıların asallığını milisaniyeler içinde, çok küçük bir hata olasılığıyla belirler.

Örnekleme ve simetri kırma

Monte Carlo benzetimleri, rastgele örneklem ve dağıtık sistemlerde lider seçimi gibi işler rastgelelik olmadan zordur.

Bedel: süre ya da doğruluk artık **olasılıksal** bir garantidir; iyi bir rastgele sayı üreteci gerekir.

GİRİŞ

Olasılık Hatırlatması

Kavram	Tanım / kural	Örnek
Beklenen değer	$E[X] = \Sigma\, x \cdot P(X = x)$	bir zarın beklenen değeri 3,5
Doğrusallık	$E[X + Y] = E[X] + E[Y]$ (bağımsızlık gerekmez)	iki zarın toplamı: 7
Gösterge değişkeni	$X_A = 1$ (A olursa), 0 → $E[X_A] = P(A)$	QuickSort'ta "i ile j karşılaştırıldı mı?"
Geometrik dağılım	başarı olasılığı p olan denemeler: ilk başarıya kadar beklenen deneme **1/p**	BogoSort: $p = 1/n!$ → $n!$ karıştırma
Bağımsız tekrar	her biri p ile başarısız k bağımsız deneme: hepsi başarısız $\mathbf{p^k}$	Miller–Rabin: $(1/4)^k$
Markov eşitsizliği	$X \geq 0$ ise $P(X \geq c \cdot E[X]) \leq 1/c$	Las Vegas → Monte Carlo

ALGORİTMA TÜRLERİ

Las Vegas ve Monte Carlo

	Las Vegas	Monte Carlo
Doğruluk	**Her zaman doğru** sonuç	Küçük bir olasılıkla **yanlış** sonuç
Süre	Rastgele; beklenen süre analiz edilir	Sınırlı (deterministik üst sınır)
Rastgele olan	ne kadar süreceği	sonucun doğru olup olmadığı
Örnekler	Rastgele QuickSort, QuickSelect, BogoSort, Fisher–Yates	Miller–Rabin, Karger min-kesim, π tahmini

Tek yönlü hata

Miller–Rabin "bileşik" derse **kesin** doğrudur; yalnızca "asal" yanıtı yanlış olabilir.

İki yönlü hata

π tahmini gibi sayısal benzetimler her iki yönde de sapabilir; hata örnek sayısıyla küçülür.

Hata Olasılığını Küçültme

Tekrarla güçlendirme (amplification)

Tek turda yanılma olasılığı en çok q ise, k **bağımsız** tur sonra yanılma ≤ q^k.

- Miller–Rabin: q ≤ 1/4 → k = 20 turda ≤ $4^{-20} \approx 9 \cdot 10^{-13}$.
- Karger: başarı ≥ 2/(n(n−1)); n(n−1)/2 · ln n tekrarla başarısızlık ≤ 1/n.

Türler arası dönüşüm

- **Las Vegas → Monte Carlo:** algoritma 2 · E[T] adım sonra durdurulur. Markov: P(T ≥ 2 E[T]) ≤ 1/2 → en çok 1/2 olasılıkla "yanıt yok".
- **Monte Carlo → Las Vegas:** yanıt hızlıca **doğrulanabiliyorsa** (ör. bulunan çarpanlar çarpılır), doğru çıkana kadar tekrarlanır. Beklenen tekrar = 1/p.



Miller–Rabin hata sınırı 4^{-k} (logaritmik ölçek): her tur sınırı 4'e böler. 10^{-6}'nın altı için k = 10 tur yeter.

Örnek: Karıştırma

- Sıralı bir listenin öğelerini **karıştırmak** (shuffle) için rastgelelik kullanılır: kart destesi, rastgele test verisi, şarkı listesi.
- **Doğruluk:** her öğenin her konumda **eşit olasılıkla** bulunması; daha güçlüsü, n! permütasyonun her birinin olasılığı **1/n!** olmalı.
- **Çalışma zamanı:** Fisher–Yates her çalıştırmada tam n − 1 takas yapar: **Θ(n)**. Değişen, sürenin değil *sonucun* kendisidir.
- Java'da hazır: `Collections.shuffle(liste)` (Fisher–Yates).

n	n! permütasyon
3	6
4	24
10	3 628 800
52	$\approx 8{,}07 \cdot 10^{67}$

KARIŞTIRMA · KODU ADIM ADIM ÇALIŞTIR

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



Başlangıç: `a = [1, 2, 3, 4, 5]`. Sondan başlayarak her konuma, henüz yerleşmemiş öğelerden biri rastgele seçilir.

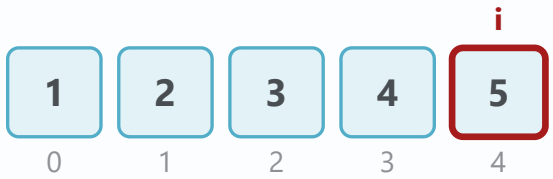
Adım 1 / 14

DEĞİŞKENLER

`a = [1, 2, 3, 4, 5]` `i = –` `j = –`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
▶   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`i = 4` : a[0..4] arasındaki 5 öğeden biri a[4] konumuna gelecek.

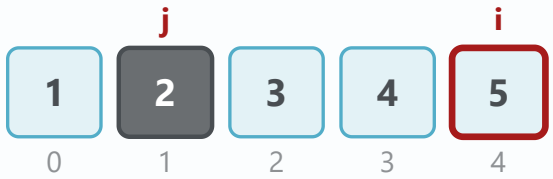
Adım 2 / 14

DEĞİŞKENLER

`a = [1, 2, 3, 4, 5]` `i = 4` `j = –`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`zar.nextInt(5)` → `j = 1` (olasılık 1/5).

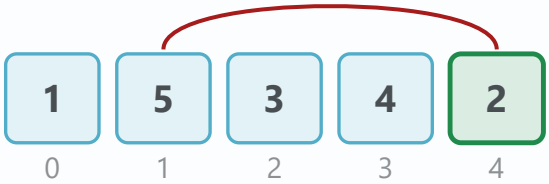
Adım 3 / 14

DEĞİŞKENLER

`a = [1, 2, 3, 4, 5]` `i = 4` `j = 1`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



a[4] ↔ a[1] takas → `[1, 5, 3, 4, 2]`. a[4] = 2 artık **kesinleşti** (yeşil).

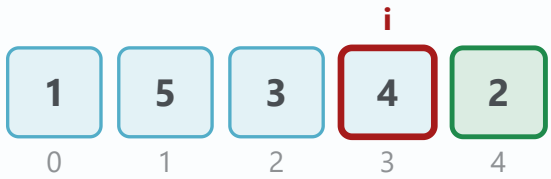
Adım 4 / 14

DEĞİŞKENLER

`a = [1, 5, 3, 4, 2]` `i = 4` `j = 1`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
▶   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`i = 3` : a[0..3] arasındaki 4 öğeden biri a[3] konumuna gelecek.

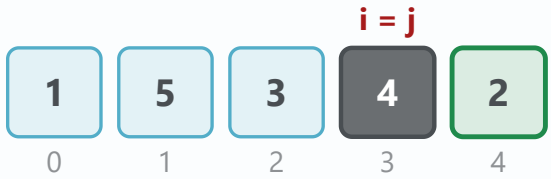
Adım 5 / 14

DEĞİŞKENLER

`a = [1, 5, 3, 4, 2]` `i = 3` `j = –`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`zar.nextInt(4)` → `j = 3` (j = i: öğe yerinde kalır).

Adım 6 / 14

DEĞİŞKENLER

`a = [1, 5, 3, 4, 2]` `i = 3` `j = 3`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



Kendisiyle takas: dizi değişmez; a[3] = 4 kesinleşti.

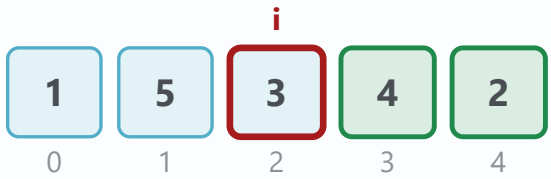
Adım 7 / 14

DEĞİŞKENLER

`a = [1, 5, 3, 4, 2]` `i = 3` `j = 3`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
▶   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`i = 2` : a[0..2] arasındaki 3 öğeden biri a[2] konumuna gelecek.

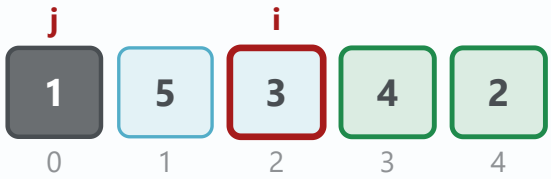
Adım 8 / 14

DEĞİŞKENLER

`a = [1, 5, 3, 4, 2]` `i = 2` `j = –`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`zar.nextInt(3)` → `j = 0` (olasılık 1/3).

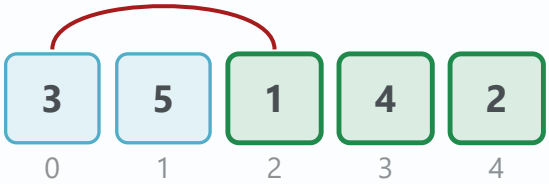
Adım 9 / 14

DEĞİŞKENLER

`a = [1, 5, 3, 4, 2]` `i = 2` `j = 0`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



a[2] ↔ a[0] takas → `[3, 5, 1, 4, 2]`. a[2] = 1 artık **kesinleşti** (yeşil).

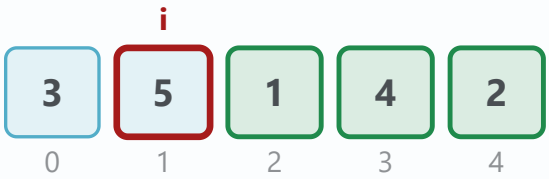
Adım 10 / 14

DEĞİŞKENLER

`a = [3, 5, 1, 4, 2]` `i = 2` `j = 0`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
▶   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`i = 1` : a[0..1] arasındaki 2 öğeden biri a[1] konumuna gelecek.

Adım 11 / 14

DEĞİŞKENLER

`a = [3, 5, 1, 4, 2]` `i = 1` `j = –`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



`zar.nextInt(2)` → `j = 1` (j = i: öğe yerinde kalır).

Adım 12 / 14

DEĞİŞKENLER

`a = [3, 5, 1, 4, 2]` `i = 1` `j = 1`

KARIŞTIRMA · KODU ADIM ADIM ÇALIŞTIR

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```



Kendisiyle takas: dizi değişmez; a[1] = 5 kesinleşti.

Adım 13 / 14

DEĞİŞKENLER

`a = [3, 5, 1, 4, 2]` `i = 1` `j = 1`

Fisher–Yates (Knuth) Karıştırması

```
1 static void karistir(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = zar.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
▶ }
```



Döngü bitti: `[3, 5, 1, 4, 2]`. Toplam olası çekiliş sayısı 5 · 4 · 3 · 2 = 120 = 5!: her permütasyon **tam bir kez** → düzgün dağılım.

Adım 14 / 14

DEĞİŞKENLER

`a = [3, 5, 1, 4, 2]` `i = 0` `j = –`

KARIŞTIRMA

Saf Karıştırma Yanlıdır

```
// YANLI: her i için 0..n-1 arasından seç
for (int i = 0; i < n; i++) {
  int j = zar.nextInt(n);
  takas(a, i, j);
}
```

- n çekilişin her biri n değer alır → **n^n** eşit olasılıklı yol.
- n = 3: 3^3 = 27 yol, 3! = 6 permütasyon. 27, 6'ya bölünmez → permütasyonlar **eşit olasılıklı olamaz**.
- Sık görülen bu kalıbın doğrusu Fisher–Yates'tir: `nextInt(i + 1)`.

sonuç	yol	olasılık	olması gereken
`123`	4/27	0,148	1/6 ≈ 0,167
`132`	5/27	0,185	1/6 ≈ 0,167
`213`	5/27	0,185	1/6 ≈ 0,167
`231`	5/27	0,185	1/6 ≈ 0,167
`312`	4/27	0,148	1/6 ≈ 0,167
`321`	4/27	0,148	1/6 ≈ 0,167

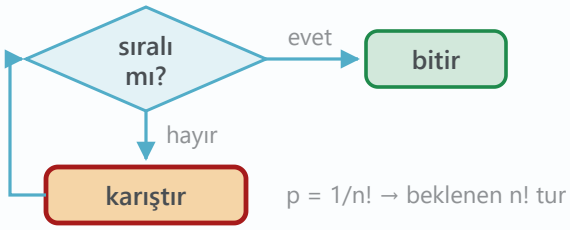
[1, 2, 3] dizisinin 27 çekiliş dizisi tek tek sayıldı (Fisher–Yates'te 3 · 2 = 6 yol, her permütasyon 1 kez).

BogoSort

- Sıralama algoritmaları arasında **en kötü performansa** sahip olanlardan biridir ("aptal sıralama", permutation sort).
- Temel mantık: liste sıralı olana kadar **rastgele karıştırılır**.
- Eğlenceli ve öğretici bir **Las Vegas** algoritmasıdır; gerçek dünyada kullanılmaz.

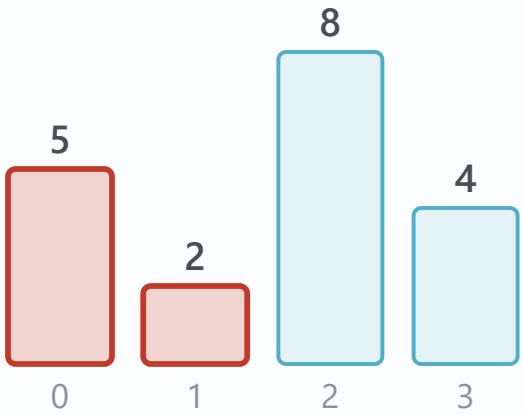
1. Liste sıralı mı? **Evet** → işlem biter.
2. **Hayır** → liste rastgele karıştırılır, adım 1'e dönülür.

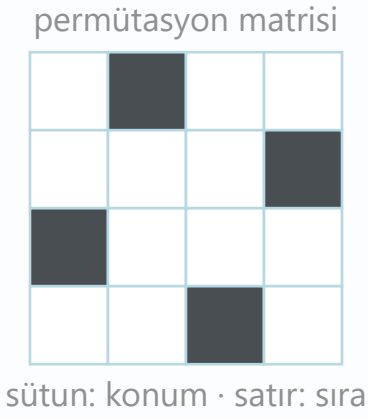
```
static void bogoSirala(int[] a) {
  while (!siraliMi(a))
    karistir(a);        // Fisher–Yates
}
static boolean siraliMi(int[] a) {
  for (int i = 0; i + 1 < a.length; i++)
    if (a[i] > a[i + 1]) return false;
  return true;
}
```



BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

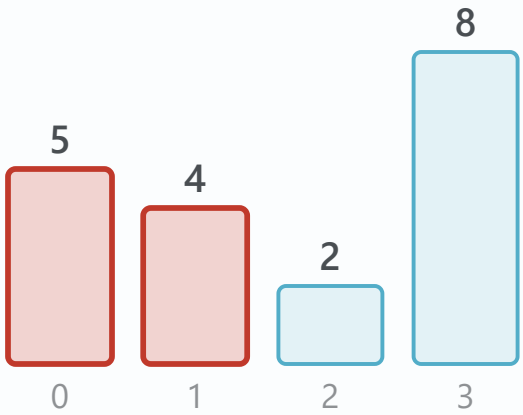


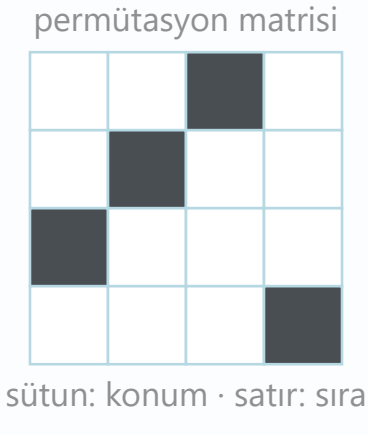


Başlangıç `[5, 2, 8, 4]` : a[0] = 5 > a[1] = 2 → **sıralı değil**, karıştırma başlar.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

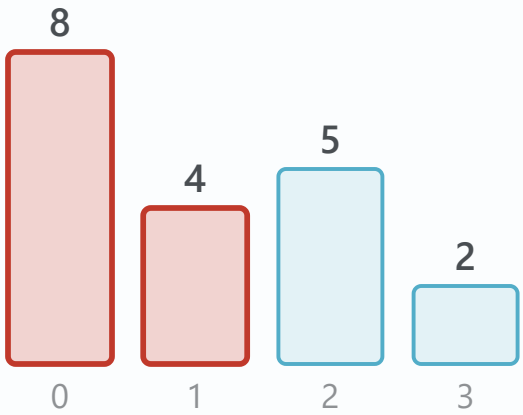




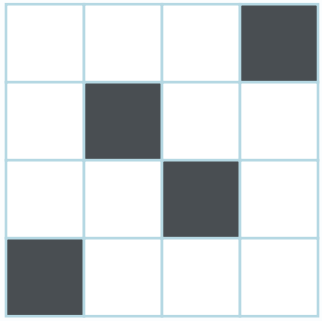
Karıştırma 1: `[5, 4, 2, 8]` → a[0] = 5 > a[1] = 4, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.



permütasyon matrisi

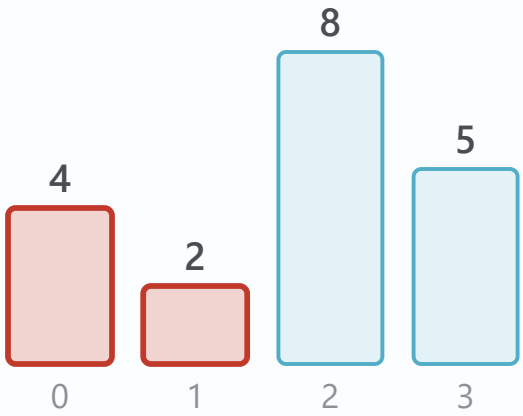


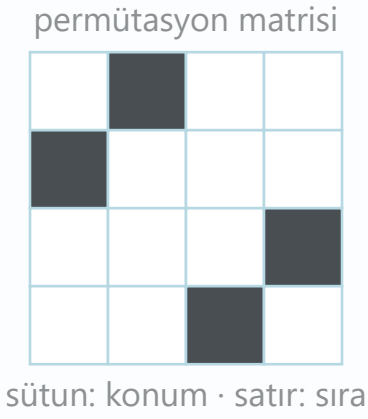
sütun: konum · satır: sıra

Karıştırma 2: `[8, 4, 5, 2]` → a[0] = 8 > a[1] = 4, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

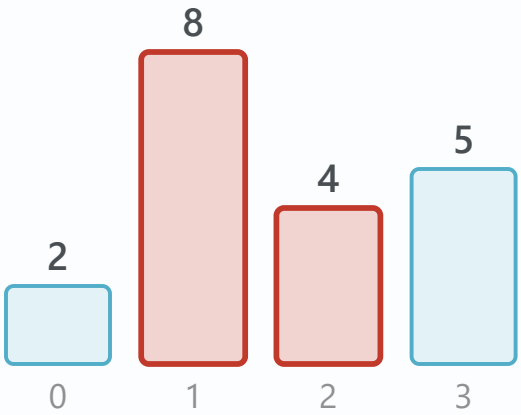


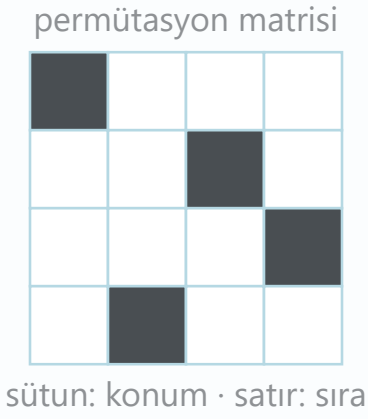


Karıştırma 3: `[4, 2, 8, 5]` → a[0] = 4 > a[1] = 2, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

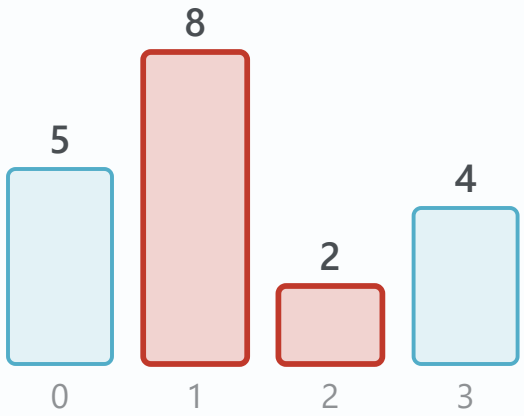


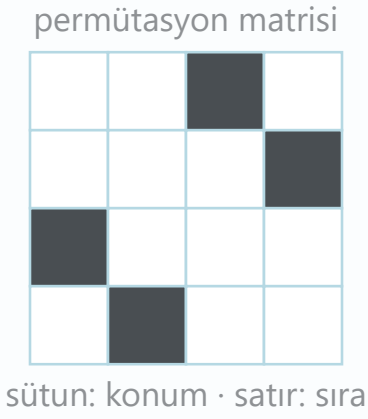


Karıştırma 4: `[2, 8, 4, 5]` → a[1] = 8 > a[2] = 4, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

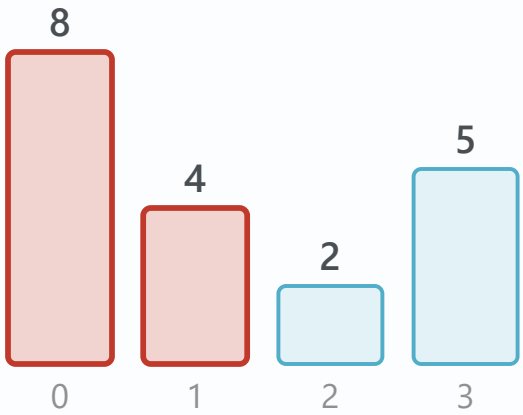


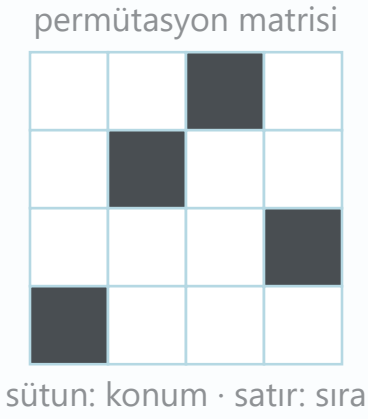


Karıştırma 5: `[5, 8, 2, 4]` → a[1] = 8 > a[2] = 2, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

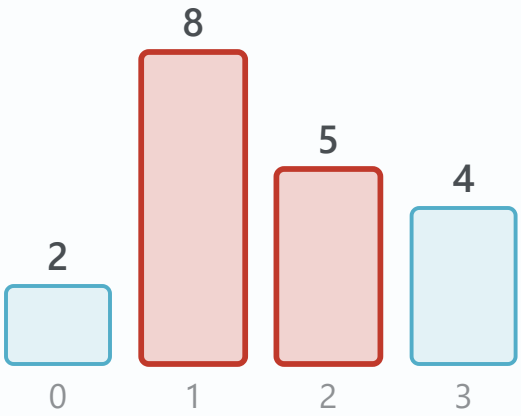


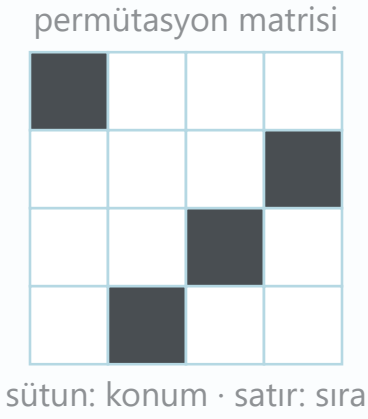


Karıştırma 6: `[8, 4, 2, 5]` → a[0] = 8 > a[1] = 4, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.

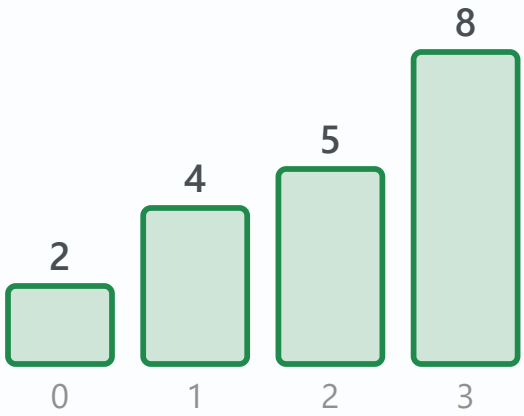


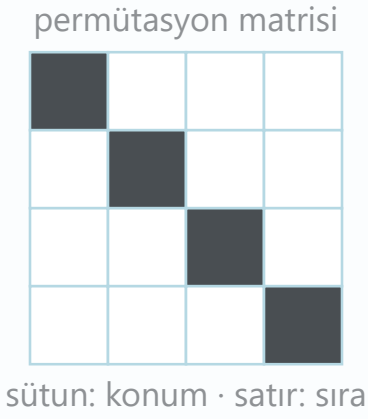


Karıştırma 7: `[2, 8, 5, 4]` → a[1] = 8 > a[2] = 5, sıralı değil.

BogoSort Örneği: [5, 2, 8, 4]

[5, 2, 8, 4] listesi; her karıştırma Fisher–Yates ile (tohumlu bir çalıştırma). Kırmızı: ilk ters sıralı komşu çift.





Karıştırma 8: `[2, 4, 5, 8]` → **sıralı!** Algoritma durur. Bu çalıştırmada 8 karıştırma gerekti; beklenen değer 4! = 24.

BogoSort Analizi

- n farklı öğe: rastgele bir karıştırmanın sıralı çıkma olasılığı **p = 1/n!**.
- Denemeler bağımsız → geometrik dağılım: beklenen karıştırma sayısı **1/p = n!**.
- Her tur: karıştırma O(n) + kontrol O(n) → **beklenen süre Θ(n · n!)**. Bu, O((n + 1)!) ile aynı büyüklüktedir, çünkü (n + 1)! = (n + 1) · n!.
- **En iyi durum**: dizi zaten sıralı → tek kontrol, O(n).
- **En kötü durum: sınırsız.** Şanssız karıştırmalar sonsuza dek sürebilir (olasılığı 0'a gider). O((n + 1)!) yalnızca *beklenen* süre için bir sınırdır, en kötü durum için değil.

n	beklenen karıştırma n!	süre (10^9 karıştırma/s)
5	120	0,12 µs
10	3 628 800	3,6 ms
12	479 001 600	0,48 s
13	6 227 020 800	6,2 s
15	1 307 674 368 000	≈ 22 dakika
20	2 432 902 008 176 640 000	≈ 77 yıl

Pivot Seçimi ve Etkisi

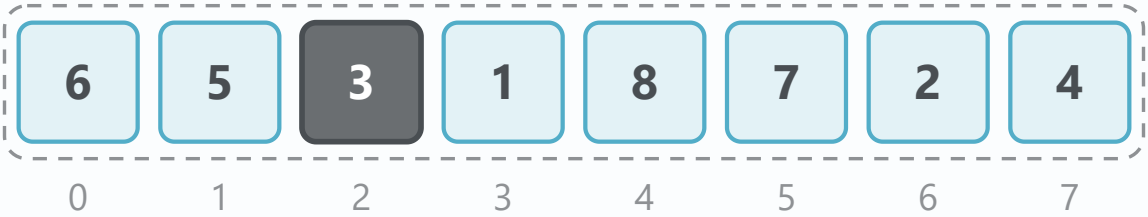
Pivot stratejisi	Kötü girdi	Beklenen / tipik	En kötü
İlk ya da son öğe	sıralı / ters sıralı dizi (çok yaygın!)	O(n log n)*	$O(n^2)$
Ortanca-üç (median-of-3)	özel hazırlanmış "killer" diziler	O(n log n)*	$O(n^2)$
Rastgele öğe	**yok**: kötülük yalnızca şanssızlıktan	O(n log n), **her girdi** için	$O(n^2)$
Gerçek ortanca (BFPRT)	yok	O(n log n)	O(n log n)

- * Deterministik stratejilerde O(n log n) yalnızca girdi rastgele sıralıysa geçerlidir (**ortalama durum**). Rastgele pivotta ise beklenti algoritmanın kendi seçimleri üzerindendir (**beklenen süre**).
- İyi pivot seçimi **beklenen** süreyi iyileştirir, en kötü durumu ortadan kaldırmaz: rastgele pivotta da $O(n^2)$ mümkündür, ama olasılığı çok küçüktür.
- BFPRT'nin sabitleri büyüktür; uygulamada rastgele ya da ortanca-üç pivot tercih edilir.

RASTGELE QUICKSORT

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.

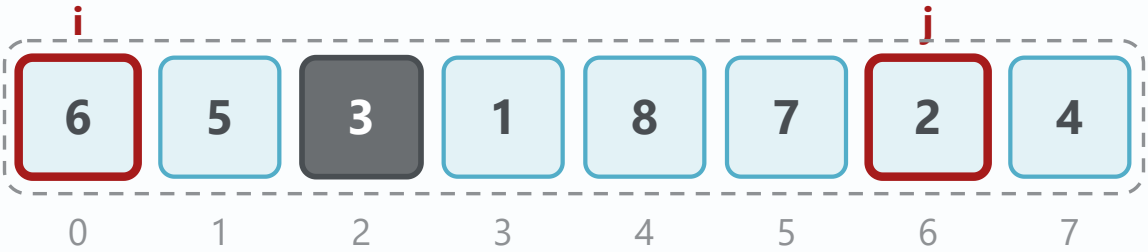


Alt dizi a[0..7] = `[6, 5, 3, 1, 8, 7, 2, 4]` : pivot **rastgele** seçildi → **3** (indis 2, siyah kutu).

Adım 1 / 25

Rastgele Pivotlu QuickSort Çalıştırması

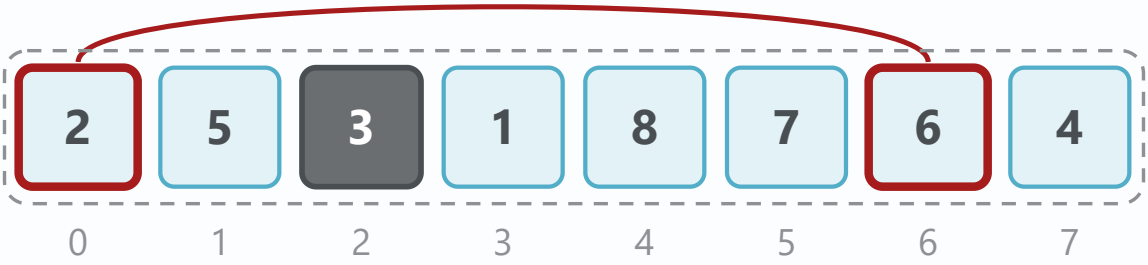
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



i soldan ilerler, 3'ten küçükleri atlar ve a[0] = 6'da durur; j sağdan ilerler, 3'ten büyükleri atlar ve a[6] = 2'de durur. i < j → takas edilecek.

Rastgele Pivotlu QuickSort Çalıştırması

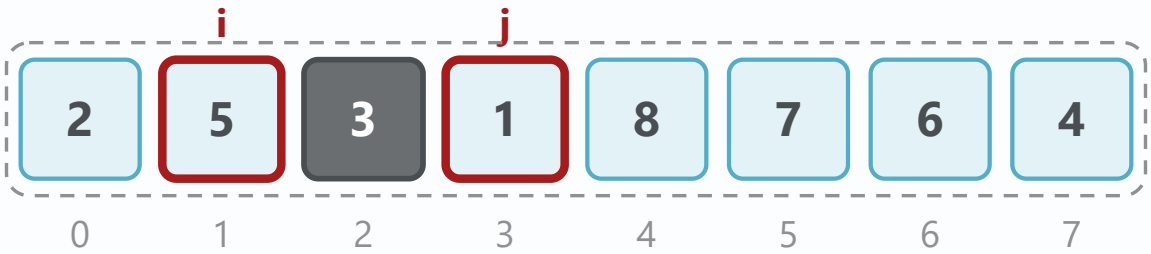
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



a[0] ↔ a[6] takas → `[2, 5, 3, 1, 8, 7, 6, 4]`.

Rastgele Pivotlu QuickSort Çalıştırması

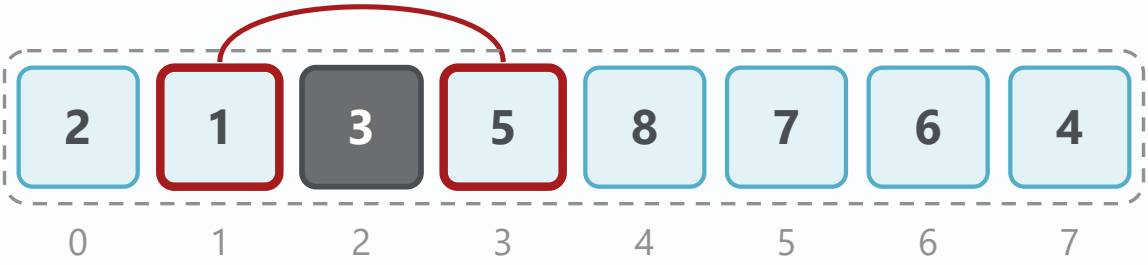
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



i soldan ilerler, 3'ten küçükleri atlar ve a[1] = 5'te durur; j sağdan ilerler, 3'ten büyükleri atlar ve a[3] = 1'de durur. i < j → takas edilecek.

Rastgele Pivotlu QuickSort Çalıştırması

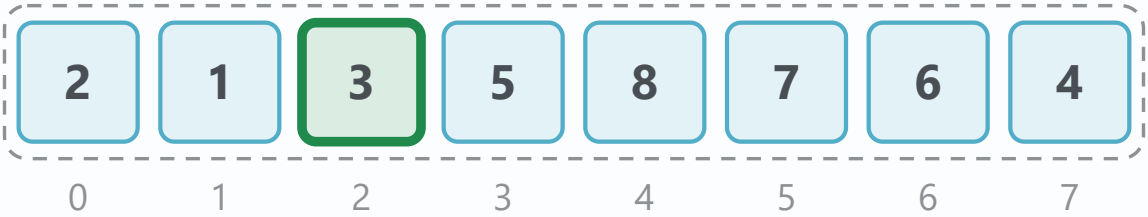
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



a[1] ↔ a[3] takas → `[2, 1, 3, 5, 8, 7, 6, 4]`.

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



İşaretçiler karşılaştı (i ≥ j): pivot **3** kalıcı yerinde (indis 2). Solunda yalnızca küçükler, sağında yalnızca büyükler var.

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Alt dizi a[0..1] = `[2, 1]` : pivot **rastgele** seçildi → **1** (indis 1, siyah kutu).

Rastgele Pivotlu QuickSort Çalıştırması

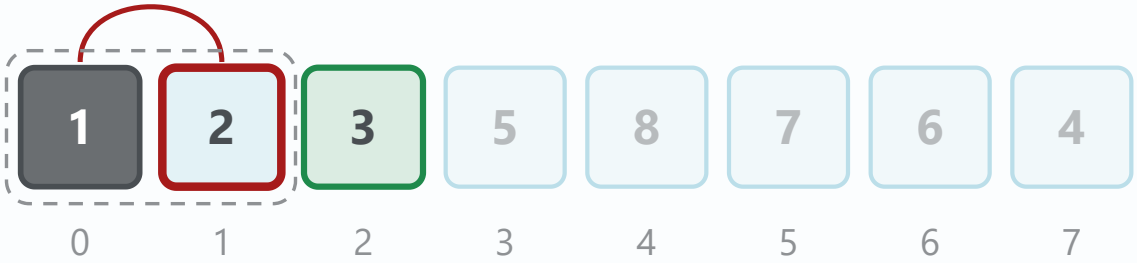
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



i soldan ilerler, 1'den küçükleri atlar ve a[0] = 2'de durur; j sağdan ilerler, 1'den büyükleri atlar ve a[1] = 1'de durur. i < j → takas edilecek.

Rastgele Pivotlu QuickSort Çalıştırması

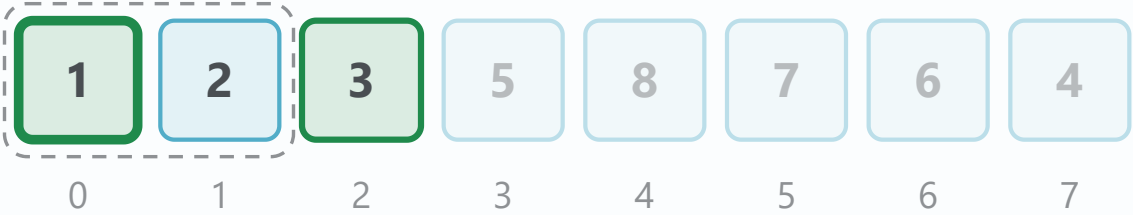
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



a[0] ↔ a[1] takas → `[1, 2, 3, 5, 8, 7, 6, 4]`. Pivotun kendisi de yer değiştirebilir; konumu izlenir.

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



İşaretçiler karşılaştı (i ≥ j): pivot **1** kalıcı yerinde (indis 0). Solunda yalnızca küçükler, sağında yalnızca büyükler var.

Rastgele Pivotlu QuickSort Çalıştırması

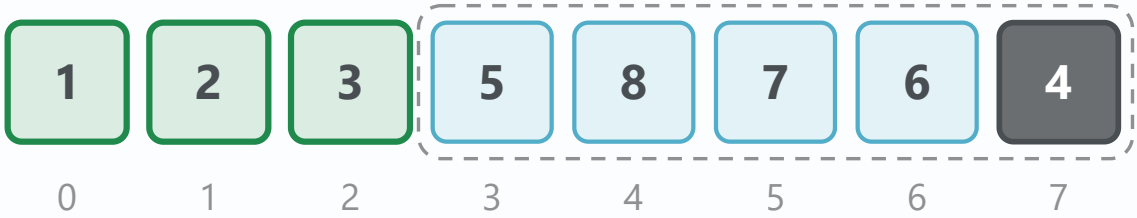
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Tek öğeli alt dizi a[1] = 2 zaten sıralı: kesinleşti.

Rastgele Pivotlu QuickSort Çalıştırması

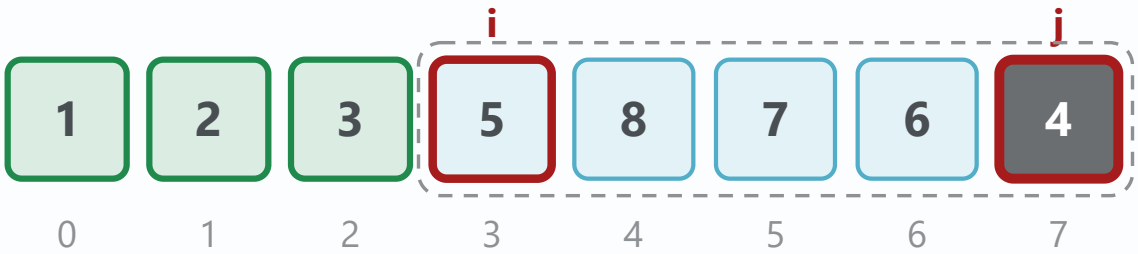
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Alt dizi a[3..7] = `[5, 8, 7, 6, 4]` : pivot **rastgele** seçildi → **4** (indis 7, siyah kutu).

Rastgele Pivotlu QuickSort Çalıştırması

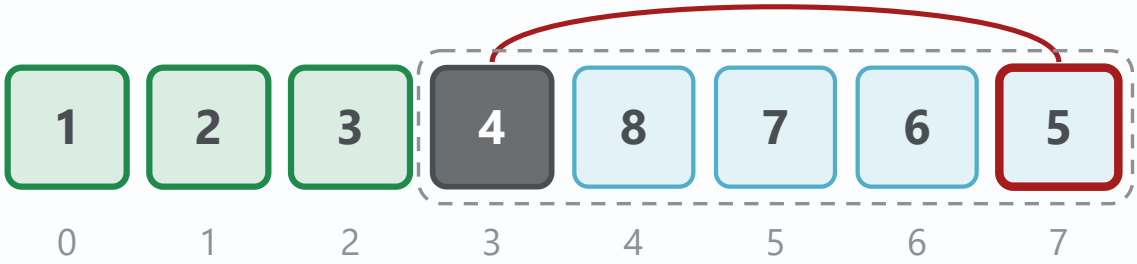
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



i soldan ilerler, 4'ten küçükleri atlar ve a[3] = 5'te durur; j sağdan ilerler, 4'ten büyükleri atlar ve a[7] = 4'te durur. i < j → takas edilecek.

Rastgele Pivotlu QuickSort Çalıştırması

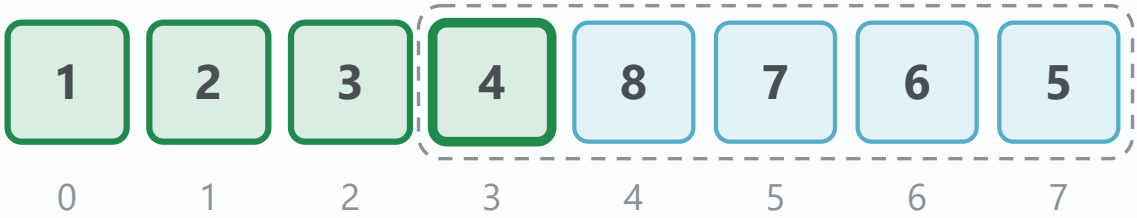
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



a[3] ↔ a[7] takas → `[1, 2, 3, 4, 8, 7, 6, 5]`. Pivotun kendisi de yer değiştirebilir; konumu izlenir.

Rastgele Pivotlu QuickSort Çalıştırması

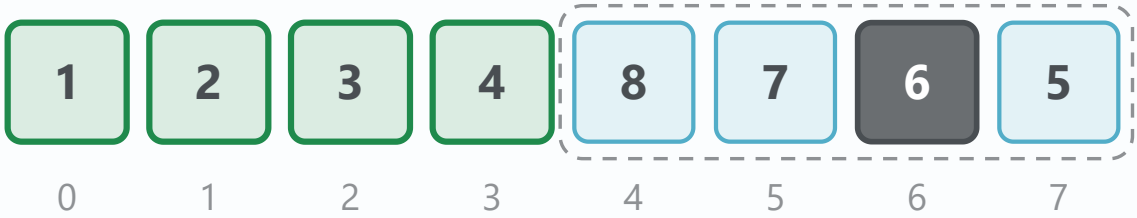
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



İşaretçiler karşılaştı (i ≥ j): pivot **4** kalıcı yerinde (indis 3). Solunda yalnızca küçükler, sağında yalnızca büyükler var.

Rastgele Pivotlu QuickSort Çalıştırması

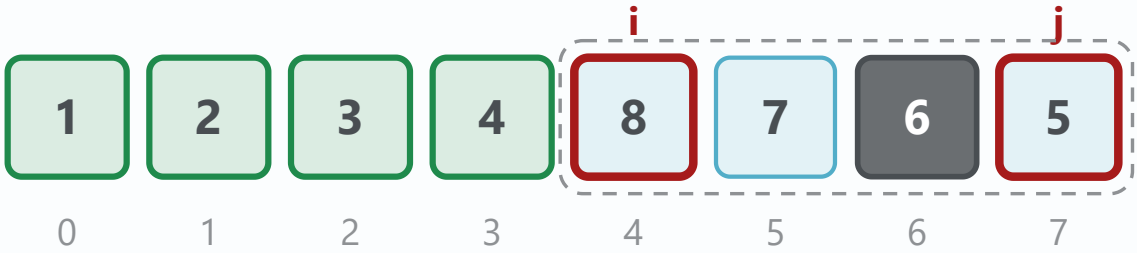
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Alt dizi a[4..7] = `[8, 7, 6, 5]` : pivot **rastgele** seçildi → **6** (indis 6, siyah kutu).

Rastgele Pivotlu QuickSort Çalıştırması

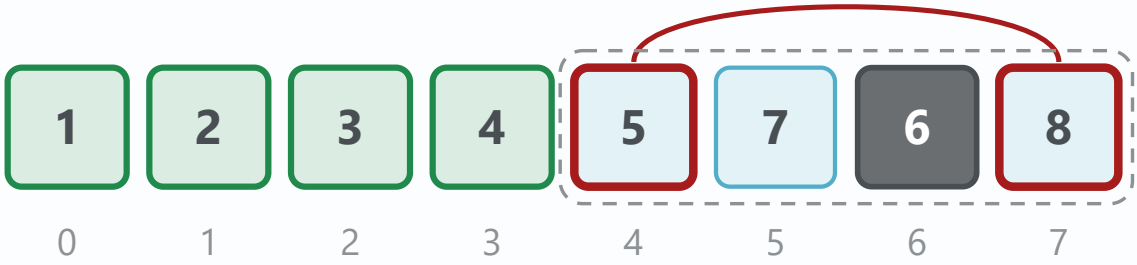
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



i soldan ilerler, 6'dan küçükleri atlar ve a[4] = 8'de durur; j sağdan ilerler, 6'dan büyükleri atlar ve a[7] = 5'te durur. i < j → takas edilecek.

Rastgele Pivotlu QuickSort Çalıştırması

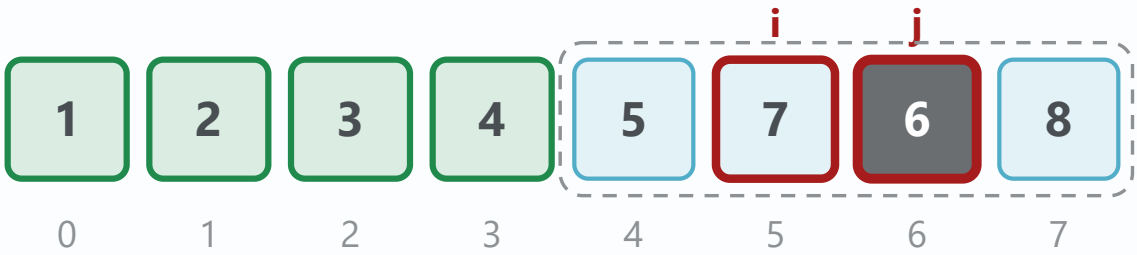
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



a[4] ↔ a[7] takas → `[1, 2, 3, 4, 5, 7, 6, 8]`.

Rastgele Pivotlu QuickSort Çalıştırması

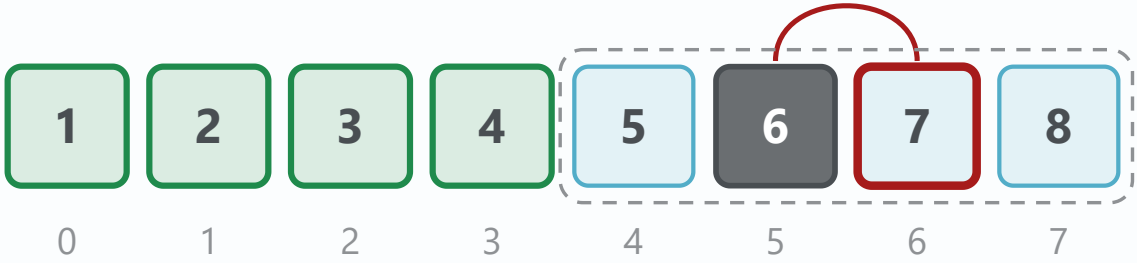
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



i soldan ilerler, 6'dan küçükleri atlar ve a[5] = 7'de durur; j sağdan ilerler, 6'dan büyükleri atlar ve a[6] = 6'da durur. i < j → takas edilecek.

Rastgele Pivotlu QuickSort Çalıştırması

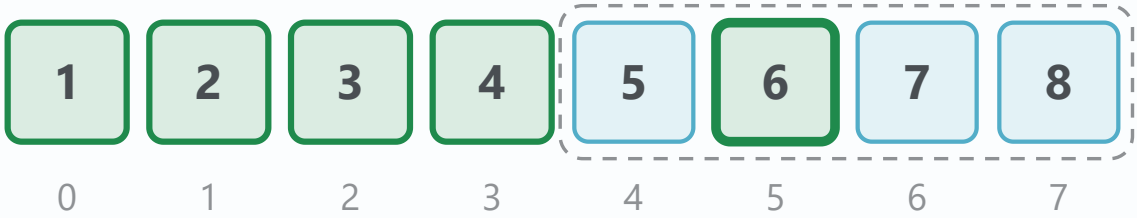
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



a[5] ↔ a[6] takas → `[1, 2, 3, 4, 5, 6, 7, 8]`. Pivotun kendisi de yer değiştirebilir; konumu izlenir.

Rastgele Pivotlu QuickSort Çalıştırması

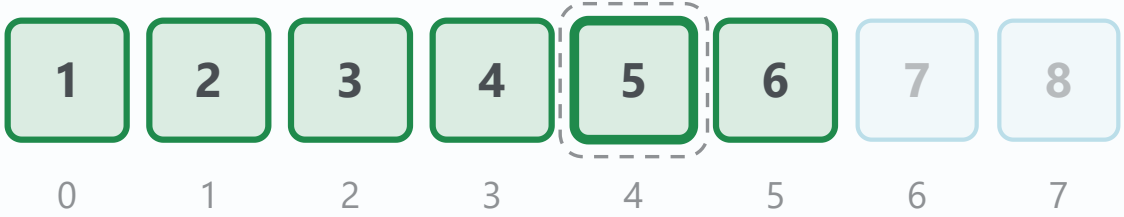
[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



İşaretçiler karşılaştı (i ≥ j): pivot **6** kalıcı yerinde (indis 5). Solunda yalnızca küçükler, sağında yalnızca büyükler var.

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Tek öğeli alt dizi a[4] = 5 zaten sıralı: kesinleşti.

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Alt dizi a[6..7] = `[7, 8]` : pivot **rastgele** seçildi → **8** (indis 7, siyah kutu).

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Pivot 8: i, 7'yi atlayıp 8'de durur; j zaten 8'de. i ≥ j → takas yok, **8** kalıcı yerinde (indis 7).

Rastgele Pivotlu QuickSort Çalıştırması

[6, 5, 3, 1, 8, 7, 2, 4]; pivotlar rastgele seçildi: 3, 1, 4, 6, 8. Siyah: pivot · kırmızı çerçeve: i ve j işaretçileri · yeşil: kesinleşmiş · soluk: etkin alt dizinin dışı.



Tek öğeli alt dizi a[6] = 7 zaten sıralı: kesinleşti. **Dizi sıralandı:** `[1, 2, 3, 4, 5, 6, 7, 8]`.

Rastgele QuickSort: Kod

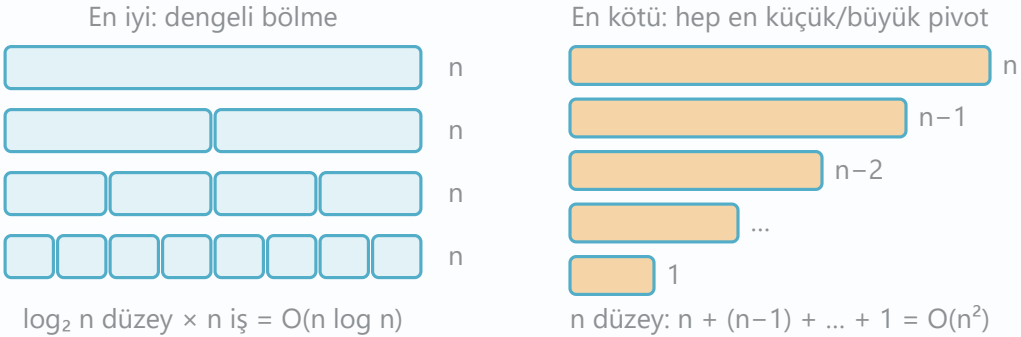
```
static Random zar = new Random();

static void hizliSirala(int[] a, int alt, int ust) {
  if (alt < ust) {
    int p = bol(a, alt, ust); // rastgele
    hizliSirala(a, alt, p - 1);
    hizliSirala(a, p + 1, ust);
  }
}

static int bol(int[] a, int alt, int ust) {
  int r = alt + zar.nextInt(ust - alt + 1);
  takas(a, r, ust);              // pivot sona
  int pivot = a[ust], k = alt;
  for (int i = alt; i < ust; i++)
    if (a[i] <= pivot)
      takas(a, i, k++);          // küçükleri sola
  takas(a, k, ust);              // pivot yerine
  return k;
}
```

- Pivot, alt dizinin öğeleri arasından **düzgün rastgele** seçilir. Sonuç her zaman doğrudur (**Las Vegas**); yalnızca süre pivot seçimlerine bağlıdır.
- `bol` rastgele indisi sona taşır ve Lomuto bölümlemesi uygular: [6, 5, 3, 1, 8, 7, 2, 4] için r = 2 (pivot 3) → [1, 2, 3, 6, 8, 7, 5, 4], dönüş 2.
- Ek bellek: özyineleme yığını; beklenen derinlik O(log n). Küçük parçaya önce inilip büyük parça döngüye çevrilirse en kötü durumda da O(log n).
- Tekrarlı anahtarlar çoksa üç yollu bölümleme (<, =, >) kullanılır.

QuickSort Analizi



Durum	Yineleme bağıntısı	Süre
En iyi	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$
Beklenen (rastgele pivot)	$T(n) = (1/n)\,\Sigma_q\,[T(q) + T(n-1-q)] + \Theta(n)$	$\Theta(n \log n)$
En kötü	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$

Rastgele pivotta en kötü durumun olasılığı: her adımda en küçük ya da en büyük seçilmeli → $2^{n-1}/n!$ ($n = 20$ için $\approx 2 \cdot 10^{-13}$).

Beklenen Karşılaştırma

- Öğeler sıralı hâlleriyle adlandırılır: $z_1 < z_2 < \ldots < z_n$. Gösterge $X_{ij} = 1 \Leftrightarrow z_i$ ile z_j karşılaştırılır.
- İki öğe yalnızca biri **pivotken** karşılaştırılır. $\{z_i, \ldots, z_j\}$ kümesinden **ilk** seçilen pivot arada bir öğe olursa z_i ile z_j ayrı parçalara düşer ve hiç karşılaşmaz.
- Bu kümeden ilk pivot düzgün rastgeledir → $P(X_{ij} = 1) =$ **$2 / (j - i + 1)$**.
- Doğrusallık: $E[X] = \Sigma_{i<j}\, 2/(j - i + 1) < 2n\, H_n \approx$ **$1{,}39\, n \log_2 n$**.
- Kesin değer: $E[X] = 2(n + 1)H_n - 4n$.

n	rastgele pivot (beklenen)	sıralı girdi, ilk öğe pivot
10	24,4	45
100	648	4 950
1 000	10 986	499 500
10^6	$2{,}48 \cdot 10^7$	$5{,}0 \cdot 10^{11}$



Pivot $z_4 \rightarrow z_2$ sola, z_6 sağa: hiç karşılaşmazlar.
Yalnızca z_2 ya da z_6 ilk pivotsa: P = 2/5.

Rastgele Seçme (QuickSelect)

```
static int sec(int[] a, int alt, int ust, int s) {
  if (alt == ust) return a[alt];
  int p = bol(a, alt, ust);
  if (s == p) return a[p];
  if (s < p) return sec(a, alt, p - 1, s);
  return sec(a, p + 1, ust, s);
}
```

- Amaç: dizinin **s. en küçük** öğesini (ör. ortanca) sıralamadan bulmak.
- Aynı rastgele bölümleme kullanılır, ama yalnızca **tek tarafa** inilir.
- Beklenen süre **O(n)** (ortanca için ≈ 3,39 n karşılaştırma); en kötü $O(n^2)$.
- Örnek: [6, 5, 3, 1, 8, 7, 2, 4] için s = 2 (0 tabanlı). Pivot 3 ile bölümleme p = 2 verir → ilk denemede yanıt **3**.

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
▶   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



Dizi [6, 5, 3, 1, 8, 7, 2, 4]; aranan s = 5 (6. en küçük). İlk çağrı tüm diziyi kapsar: alt = 0, ust = 7.

Adım 1 / 12

ÇAĞRI YIĞINI

- `sec(0, 7)`
- `main()`

DEĞİŞKENLER

`alt = 0` `ust = 7` `s = 5` `p = –`

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
▶   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



`bol` rastgele pivot 3 seçti; bölümleme sonrası pivot kalıcı yeri p = **2**: solu ≤ 3, sağı > 3.

Adım 2 / 12

ÇAĞRI YIĞINI

- **sec(0, 7)**
- main()

DEĞİŞKENLER

`alt = 0` `ust = 7` `s = 5` `p = 2`

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
▶   return sec(a, p + 1, ust, s);
7 }
```



s = 5 > p = 2 → aranan öğe sağda: yalnızca [3, 7] aralığına inilir; sol taraf bir daha incelenmez.

Adım 3 / 12

ÇAĞRI YIĞINI

- **sec(0, 7)**
- main()

DEĞİŞKENLER

alt = **0** | ust = **7** | s = **5** | p = **2**

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
▶   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



Yeni çağrı `sec(a, 3, 7, 5)` : aralıkta 5 öğe var, alt ≠ ust.

Adım 4 / 12

ÇAĞRI YIĞINI

- **sec(3, 7)**
- sec(0, 7)
- main()

DEĞİŞKENLER

alt = **3** | ust = **7** | s = **5** | p = **–**

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
▶   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



`bol` rastgele pivot 7 seçti; bölümleme sonrası pivot kalıcı yeri p = **6**: solu ≤ 7, sağı > 7.

Adım 5 / 12

ÇAĞRI YIĞINI

- **sec(3, 7)**
- sec(0, 7)
- main()

DEĞİŞKENLER

`alt = 3` `ust = 7` `s = 5` `p = 6`

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
▶   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



s = 5 < p = 6 → aranan öğe solda: yalnızca [3, 5] aralığına inilir.

ÇAĞRI YIĞINI

- **sec(3, 7)**
- sec(0, 7)
- main()

DEĞİŞKENLER

alt = **3** · ust = **7** · s = **5** · p = **6**

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
▶   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



Yeni çağrı `sec(a, 3, 5, 5)` : aralıkta 3 öğe var, alt ≠ ust.

Adım 7 / 12

ÇAĞRI YIĞINI

- **sec(3, 5)**
- sec(3, 7)
- sec(0, 7)
- main()

DEĞİŞKENLER

`alt = 3` `ust = 5` `s = 5` `p = –`

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
▶   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



`bol` rastgele pivot 5 seçti; bölümleme sonrası pivot kalıcı yeri p = **4**: solu ≤ 5, sağı > 5.

Adım 8 / 12

ÇAĞRI YIĞINI

- **sec(3, 5)**
- sec(3, 7)
- sec(0, 7)
- main()

DEĞİŞKENLER

`alt = 3` `ust = 5` `s = 5` `p = 4`

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
▶   return sec(a, p + 1, ust, s);
7 }
```



s = 5 > p = 4 → aranan öğe sağda: yalnızca [5, 5] aralığına inilir; sol taraf bir daha incelenmez.

ÇAĞRI YIĞINI

- **sec(3, 5)**
- sec(3, 7)
- sec(0, 7)
- main()

DEĞİŞKENLER

alt = **3** · ust = **5** · s = **5** · p = **4**

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
▶   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



Yeni çağrı `sec(a, 5, 5, 5)` : aralıkta 1 öğe var, alt = ust.

Adım 10 / 12

ÇAĞRI YIĞINI

- **sec(5, 5)**
- sec(3, 5)
- sec(3, 7)
- sec(0, 7)
- main()

DEĞİŞKENLER

`alt = 5` `ust = 5` `s = 5` `p = –`

QuickSelect: Kod İzleme

```
1 static int sec(int[] a, int alt, int ust, int s) {
▶   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



alt = ust = 5: tek öğe kaldı → `return a[5]` = **6**.

Adım 11 / 12

ÇAĞRI YIĞINI

- **sec(5, 5)**
- sec(3, 5)
- sec(3, 7)
- sec(0, 7)
- main()

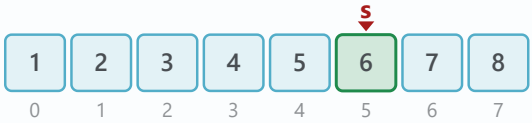
DEĞİŞKENLER

`alt = 5` `ust = 5` `s = 5`

`dönüş = 6`

QuickSelect: Kod İzleme

```
▶ static int sec(int[] a, int alt, int ust, int s) {
2   if (alt == ust) return a[alt];
3   int p = bol(a, alt, ust);
4   if (s == p) return a[p];
5   if (s < p) return sec(a, alt, p - 1, s);
6   return sec(a, p + 1, ust, s);
7 }
```



`sec(a, 0, 7, 5)` = **6**: sıralı dizide 5. konum (0 tabanlı). Dizinin tamamı sıralanmadı; yalnızca gereken parçalar bölündü.

Adım 12 / 12

ÇAĞRI YIĞINI

`main()`

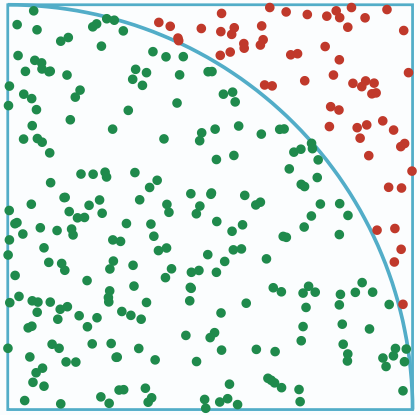
DEĞİŞKENLER

dönüş = **6**

Monte Carlo ile π

- Birim kareye düzgün rastgele noktalar atılır. Çeyrek dairenin alanı π/4, karenin alanı 1 → P(içeride) = π/4, yani **π ≈ 4 · iç / n**.
- Hata ≈ 1/√n ile azalır: bir basamak daha doğruluk için **100 kat** nokta gerekir.

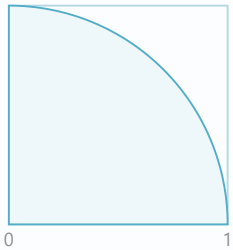
```
static double piTahmin(int n) {
  int ic = 0;
  for (int t = 0; t < n; t++) {
    double x = zar.nextDouble(), y = zar.nextDouble();
    if (x * x + y * y <= 1) ic++;
  }
  return 4.0 * ic / n;
}
```



300 nokta: 239 içeride → π ≈ 4 · 239/300 = **3,187**.

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
▶   int ic = 0;
3   for (int t = 0; t < n; t++) {
4     double x = zar.nextDouble(), y = zar.nextDouble();
5     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 0

içeride: 0

n = 8 nokta atılacak; `ic` çeyrek dairenin içine düşenleri sayar. (Tohumlu bir `Random` ile tekrarlanabilir çalıştırma.)

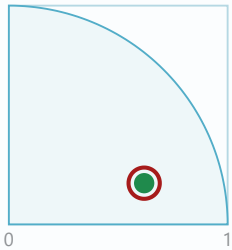
Adım 1 / 10

DEĞİŞKENLER

`n = 8` `ic = 0`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 1

içeride: 1

$x^2 + y^2 = 0{,}42$
$\leq 1 \rightarrow$ içeride

t = 0: (0,62; 0,19), $x^2 + y^2 = 0{,}42 \leq 1$ → **içeride**. ic = 1.

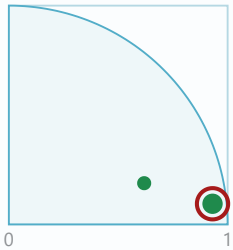
Adım 2 / 10

DEĞİŞKENLER

`t = 0` `x = 0.62` `y = 0.19` `x²+y² = 0.42` `ic = 1`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 2

içeride: 2

$x^2 + y^2 = 0{,}88$
$\leq 1 \rightarrow$ içeride

t = 1: (0,93; 0,09), $x^2 + y^2 = 0{,}88 \leq 1$ → **içeride**. ic = 2.

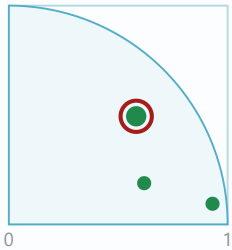
Adım 3 / 10

DEĞİŞKENLER

`t = 1` `x = 0.93` `y = 0.09` `x²+y² = 0.88` `ic = 2`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 3

içeride: 3

$x^2 + y^2 = 0{,}58$

$\leq 1 \rightarrow$ içeride

t = 2: (0,58; 0,49), $x^2 + y^2 = 0{,}58 \leq 1$ → **içeride**. ic = 3.

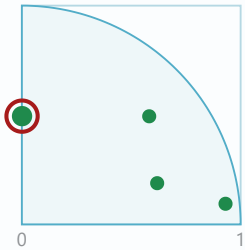
Adım 4 / 10

DEĞİŞKENLER

`t = 2` `x = 0.58` `y = 0.49` `x²+y² = 0.58` `ic = 3`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 4

içeride: 4

$x^2 + y^2 = 0{,}24$

$\leq 1 \rightarrow$ içeride

t = 3: (0,00; 0,49), $x^2 + y^2 = 0{,}24 \leq 1$ → **içeride**. ic = 4.

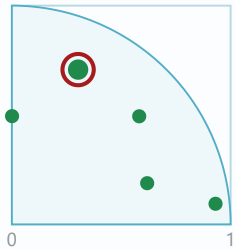
Adım 5 / 10

DEĞİŞKENLER

t = 3 | x = 0.00 | y = 0.49 | x²+y² = 0.24 | ic = 4

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 5

içeride: 5

$x^2 + y^2 = 0{,}59$

$\leq 1 \rightarrow$ içeride

t = 4: (0,30; 0,71), $x^2 + y^2 = 0{,}59 \leq 1$ → **içeride**. ic = 5.

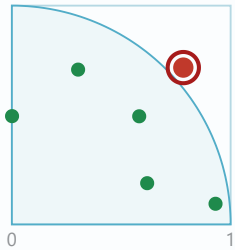
Adım 6 / 10

DEĞİŞKENLER

`t = 4` `x = 0.30` `y = 0.71` `x²+y² = 0.59` `ic = 5`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 6

içeride: 5

$x^2 + y^2 = 1{,}13$

$> 1 \rightarrow$ dışarıda

t = 5: (0,78; 0,72), $x^2 + y^2 = 1{,}13 > 1$ → dışarıda. ic = 5.

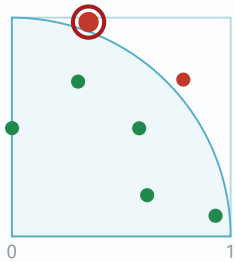
Adım 7 / 10

DEĞİŞKENLER

`t = 5` `x = 0.78` `y = 0.72` `x²+y² = 1.13` `ic = 5`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 7

içeride: 5

$x^2 + y^2 = 1{,}08$

$> 1 \rightarrow$ dışarıda

t = 6: (0,35; 0,98), $x^2 + y^2 = 1{,}08 > 1$ → dışarıda. ic = 5.

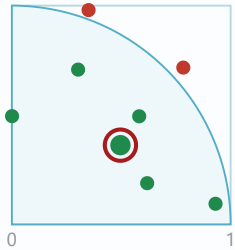
Adım 8 / 10

DEĞİŞKENLER

`t = 6` `x = 0.35` `y = 0.98` `x²+y² = 1.08` `ic = 5`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = zar.nextDouble(), y = zar.nextDouble();
▶     if (x * x + y * y <= 1) ic++;
6   }
7   return 4.0 * ic / n;
8 }
```



nokta: 8

içeride: 6

$x^2 + y^2 = 0{,}38$

$\leq 1 \rightarrow$ içeride

t = 7: (0,50; 0,36), $x^2 + y^2 = 0{,}38 \leq 1$ → **içeride**. ic = 6.

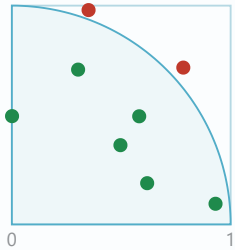
Adım 9 / 10

DEĞİŞKENLER

`t = 7` `x = 0.50` `y = 0.36` `x²+y² = 0.38` `ic = 6`

Monte Carlo π: Kod İzleme

```
1 static double piTahmin(int n) {
2   int ic = 0;
3   for (int t = 0; t < n; t++) {
4     double x = zar.nextDouble(), y = zar.nextDouble();
5     if (x * x + y * y <= 1) ic++;
6   }
▶   return 4.0 * ic / n;
8 }
```



nokta: 8

içeride: 6

π ≈ 4 · 6 / 8

= 3,0

`4.0 * 6 / 8` = **3,0**. 8 noktayla hata büyüktür (≈ 1,64/√8 ≈ 0,58); n büyüdükçe tahmin π'ye yaklaşır.

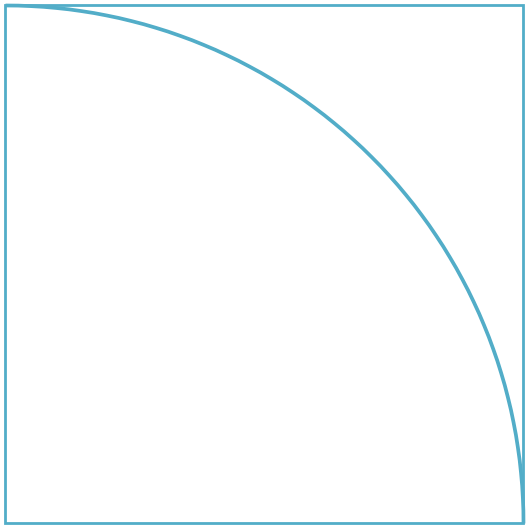
Adım 10 / 10

DEĞİŞKENLER

`n = 8` `ic = 6` `dönüş = 3.0`

Deneyin: Monte Carlo π

+10 nokta | +100 | +1000 | +10000 | sıfırla



nokta sayısı n	0
içeride	0
π tahmini	—
hata $\lvert\hat{\pi} - \pi\rvert$	—
standart sapma ≈ 1,64/√n	—

Nokta eklemek için bir düğmeye basın.

Yeşil: çeyrek dairenin içinde · kırmızı: dışında. Çizimde ilk 3000 nokta gösterilir, sayım tüm noktalarla sürer. $1{,}64 = 4 \cdot \sqrt{p(1-p)}$, $p = \pi/4$.

MONTE CARLO ALGORİTMALARI

Miller–Rabin Asallık Testi

- **Fermat'nın küçük teoremi:** n asal ve $1 < a < n$ ise $a^{n-1} \equiv 1 \pmod n$. Bu eşitlik bozulursa n kesin bileşiktir; ama bazı bileşikler (Carmichael sayıları) her taban için eşitliği sağlar.
- **Karekök özelliği:** n asal ise $x^2 \equiv 1 \pmod n$ yalnızca $x \equiv \pm 1$ için sağlanır.
- $n - 1 = 2^s \cdot d$ (d tek) yazılır; rastgele a için a^d, a^{2d}, …, $a^{2^{s-1}d} \pmod n$ dizisine bakılır.
- Dizi 1 ile başlamıyor ve $n - 1$ hiç görünmüyorsa a, n'nin **bileşik** olduğunun **tanığıdır** (witness). Aksi hâlde n bu turu "geçer".
- Bileşik n için tabanların en çok **1/4**'ü yalancıdır → k bağımsız turda hata ≤ $\mathbf{4^{-k}}$. Her tur O(log n) modüler çarpma: toplam $O(k \log^3 n)$ bit işlemi.



geçer: dizi 1 ile başlar ya da bir yerde n − 1 görünür

tanık: ilk öğe ≠ 1 ve n − 1 hiç görünmez → n kesin bileşik

Tek yönlü hata: "bileşik" yanıtı **kesindir**; "asal" yanıtı yalnızca **muhtemelen** doğrudur.

Miller–Rabin: Kod

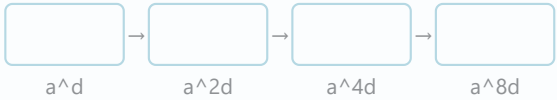
```
static boolean asalMi(long n, int k) {
  if (n < 4) return n == 2 || n == 3;
  if (n % 2 == 0) return false;
  long d = n - 1; int s = 0;      // n-1 = 2^s·d
  while (d % 2 == 0) { d /= 2; s++; }
  for (int tur = 0; tur < k; tur++) {
    long a = 2 + zar.nextLong(n - 3); // [2, n-2]
    long x = usMod(a, d, n);      // a^d mod n
    if (x == 1 || x == n - 1) continue;
    boolean tanik = true;
    for (int r = 1; r < s && tanik; r++) {
      x = x * x % n;
      if (x == n - 1) tanik = false;
    }
    if (tanik) return false;      // kesin bileşik
  }
  return true;                    // muhtemelen asal
}
```

- `usMod` : hızlı modüler üs alma (kare al-çarp), O(log d) çarpma.
- Taban a, [2, n − 2] aralığından düzgün seçilir.
- `long` ile `x * x` yalnızca n < $3 \cdot 10^9$ için taşmaz; daha büyük n için `BigInteger.modPow` ya da `isProbablePrime`.

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
▶    if (n < 4) return n == 2 || n == 3;
▶    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^0 \cdot 560$



n = 561 = 3 · 11 · 17 bir Carmichael sayısı: Fermat testini her tabanda geçer. n ≥ 4 ve tek → testler atlanmaz.

Adım 1 / 13

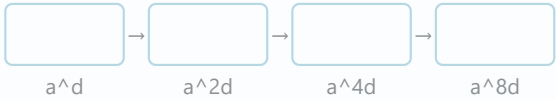
DEĞİŞKENLER

n = **561** k = **1**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
▶    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^0 \cdot 560$



d = n − 1 = 560, s = 0. Şimdi 560 içindeki 2 çarpanları ayrılacak.

Adım 2 / 13

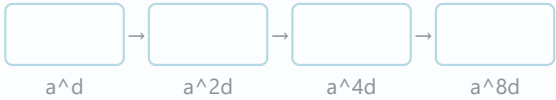
DEĞİŞKENLER

`n = 561` `d = 560` `s = 0`

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^1 \cdot 280$



d çift → d /= 2, s++: d = 280, s = 1.

Adım 3 / 13

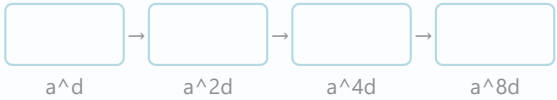
DEĞİŞKENLER

n = **561** | d = **280** | s = **1**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^2 \cdot 140$



d çift → d /= 2, s++: d = 140, s = 2.

Adım 4 / 13

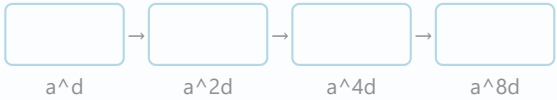
DEĞİŞKENLER

n = **561** | d = **140** | s = **2**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^3 \cdot 70$



d çift → d /= 2, s++: d = 70, s = 3.

Adım 5 / 13

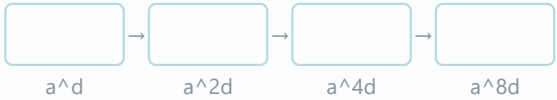
DEĞİŞKENLER

n = **561** | d = **70** | s = **3**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$



d çift → d /= 2, s++: d = 35, s = 4. d tek: $560 = 2^4 \cdot 35$.

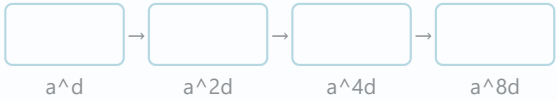
DEĞİŞKENLER

n = **561** d = **35** s = **4**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
▶    for (int tur = 0; tur < k; tur++) {
▶      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



Tur 0: rastgele taban çekildi, a = **2**.

Adım 7 / 13

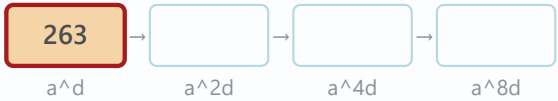
DEĞİŞKENLER

tur = 0 | d = 35 | s = 4 | a = 2

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
▶      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



$x = 2^{35} \bmod 561 =$ **263** (hızlı üs alma ile).

Adım 8 / 13

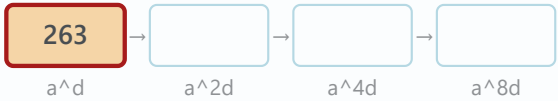
DEĞİŞKENLER

`tur = 0` `d = 35` `s = 4` `a = 2` `x = 263`

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
▶      if (x == 1 || x == n - 1) continue;
▶      boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



x = 263 ne 1 ne de n − 1 = 560 → tur bitmedi; `tanik = true` ile kareler denetlenecek.

Adım 9 / 13

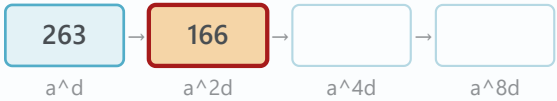
DEĞİŞKENLER

a = 2 | x = 263 | tanik = true

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
▶      for (int r = 1; r < s && tanik; r++) {
▶        x = x * x % n;
▶        if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



r = 1: $x = 263^2 \bmod 561 =$ **166**. 560 değil.

Adım 10 / 13

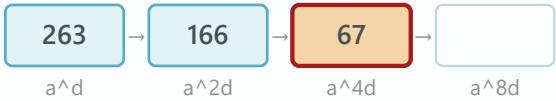
DEĞİŞKENLER

a = **2** r = **1** x = **166** tanik = **true**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
▶      for (int r = 1; r < s && tanik; r++) {
▶        x = x * x % n;
▶        if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



r = 2: $x = 166^2 \bmod 561 =$ **67**. 560 değil.

Adım 11 / 13

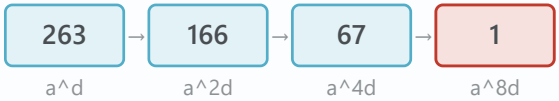
DEĞİŞKENLER

a = **2** r = **2** x = **67** tanik = **true**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
▶      for (int r = 1; r < s && tanik; r++) {
▶        x = x * x % n;
▶        if (x == n - 1) tanik = false;
14     }
15     if (tanik) return false;      // kesin bileşik
16   }
17   return true;                  // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



r = 3: $x = 67^2 \bmod 561 = \mathbf{1}$. 560 değil. 1'e n − 1 görmeden ulaşıldı.

Adım 12 / 13

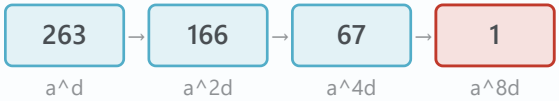
DEĞİŞKENLER

a = **2** r = **3** x = **1** tanik = **true**

Miller–Rabin: Kod İzleme (n = 561)

```
1  static boolean asalMi(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int tur = 0; tur < k; tur++) {
7      long a = 2 + zar.nextLong(n - 3); // [2, n-2]
8      long x = usMod(a, d, n);      // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean tanik = true;
11     for (int r = 1; r < s && tanik; r++) {
12       x = x * x % n;
13       if (x == n - 1) tanik = false;
14     }
▶      if (tanik) return false;      // kesin bileşik
16   }
17   return true;                    // muhtemelen asal
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2



$67^2 \equiv 1$ ama $67 \neq \pm 1 \rightarrow$ a = 2 tanık: 561 bileşik

Dizi 560'a hiç ulaşmadı: `tanik` true kaldı → `return false`. 561 **kesin bileşik**; 67, 1'in ±1 dışında bir karekökü.

Adım 13 / 13

DEĞİŞKENLER

`a = 2` `x = 1` `tanik = true` `dönüş = false`

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
▶    long sonuc = 1;
▶    a %= n;
4    while (d > 0) {
5      if ((d & 1) == 1)          // bit 1 ise
6        sonuc = sonuc * a % n;
7      a = a * a % n;             // kare al
8      d >>= 1;                   // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 1

a = 2

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

2^{35} mod 561 hesaplanacak. 35 = 32 + 2 + 1 olduğundan $2^{35} = 2^{32} \cdot 2^2 \cdot 2^1$. Başta sonuc = 1, a = 2.

Adım 1 / 8

DEĞİŞKENLER

a = **2** | d = **35** | sonuc = **1**

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)          // bit 1 ise
▶        sonuc = sonuc * a % n;
▶      a = a * a % n;             // kare al
▶      d >>= 1;                   // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 2 × a: 1 · 2 mod 561

a = 4 kare: 2^2 mod 561

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

Bit 0 = 1: sonuc = 1 · 2 mod 561 = **2** (2^1 katıldı).
Sonra a = 2^2 mod 561 = 4.

Adım 2 / 8

DEĞİŞKENLER

`a = 4` `d = 17` `sonuc = 2`

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit 1 ise
▶        sonuc = sonuc * a % n;
▶      a = a * a % n;               // kare al
▶      d >>= 1;                     // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 8 × a: 2 · 4 mod 561

a = 16 kare: 4^2 mod 561

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

Bit 1 = 1: sonuc = 2 · 4 mod 561 = **8** (2^2 katıldı). Sonra a = 4^2 mod 561 = 16.

Adım 3 / 8

DEĞİŞKENLER

`a = 16` `d = 8` `sonuc = 8`

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit 1 ise
6        sonuc = sonuc * a % n;
▶      a = a * a % n;               // kare al
▶      d >>= 1;                     // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 8 bit 0: çarpma yok

a = 256 kare: 16^2 mod 561

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

Bit 2 = 0: sonuc değişmez (8). Yalnızca kare alınır: a = 16^2 mod 561 = 256.

Adım 4 / 8

DEĞİŞKENLER

a = 256 | d = 4 | sonuc = 8

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit 1 ise
6        sonuc = sonuc * a % n;
▶      a = a * a % n;               // kare al
▶      d >>= 1;                     // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 8 bit 0: çarpma yok

a = 460 kare: 256^2 mod 561

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

Bit 3 = 0: sonuc değişmez (8). Yalnızca kare alınır: a = 256^2 mod 561 = 460.

Adım 5 / 8

DEĞİŞKENLER

`a = 460` `d = 2` `sonuc = 8`

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit 1 ise
6        sonuc = sonuc * a % n;
▶      a = a * a % n;               // kare al
▶      d >>= 1;                     // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 8 — bit 0: çarpma yok

a = 103 — kare: 460^2 mod 561

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

Bit 4 = 0: sonuc değişmez (8). Yalnızca kare alınır: a = 460^2 mod 561 = 103.

Adım 6 / 8

DEĞİŞKENLER

`a = 103` `d = 1` `sonuc = 8`

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit 1 ise
▶        sonuc = sonuc * a % n;
▶      a = a * a % n;               // kare al
▶      d >>= 1;                     // sonraki bit
9    }
10   return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 263 × a: 8 · 103 mod 561

a = 511 kare: 103^2 mod 561

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

Bit 5 = 1: sonuc = 8 · 103 mod 561 = **263** (2^{32} katıldı). Sonra a = 103^2 mod 561 = 511.

Adım 7 / 8

DEĞİŞKENLER

`a = 511` `d = 0` `sonuc = 263`

usMod: 2^{35} mod 561 Adım Adım

```
1  static long usMod(long a, long d, long n) {
2    long sonuc = 1;
3    a %= n;
4    while (d > 0) {
5      if ((d & 1) == 1)          // bit 1 ise
6        sonuc = sonuc * a % n;
7      a = a * a % n;             // kare al
8      d >>= 1;                   // sonraki bit
9    }
▶    return sonuc;
11 }
```

d = 35 = 100011_2

1	0	0	0	1	1
32	16	8	4	2	1

sonuc = 263

a = 511

a, sırayla 2^1, 2^2, 2^4, 2^8 ... (mod 561) olur

d = 0: döngü bitti. `return` **263** = 2^{35} mod 561; 34 yerine yalnızca 6 kare + 3 çarpma.

Adım 8 / 8

DEĞİŞKENLER

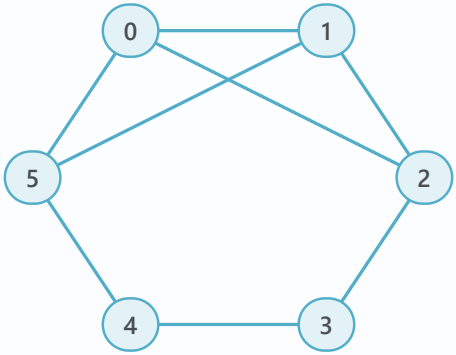
sonuc = **263** dönüş = **263**

Miller–Rabin Örnekleri

n	n − 1	a	a^d, a^{2d}, … (mod n)	sonuç
61	$2^2 \cdot 15$	2	`11, 60`	geçti (asal olabilir)
221	$2^2 \cdot 55$	174	`47, 220`	geçti (asal olabilir)
221	$2^2 \cdot 55$	137	`188, 205`	**bileşik** (a tanık)
561	$2^4 \cdot 35$	2	`263, 166, 67, 1`	**bileşik** (a tanık)

- **61** asaldır: dizi n − 1 = 60'a ulaşır.
- **221 = 13 · 17**: a = 174 bir **güçlü yalancıdır** (dizi 220'ye ulaşır); a = 137 ise tanıktır. Bu yüzden birden çok tur gerekir.
- **561 = 3 · 11 · 17** bir Carmichael sayısıdır: $2^{560} \equiv 1$, yani Fermat testi aldanır. Miller–Rabin ise $67^2 \equiv 1$ olduğunu görür: 67, 1'in ±1 dışında bir karekökü → **bileşik**.

Karger Min-Kesim Algoritması

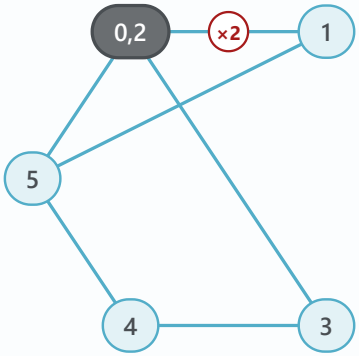


Başlangıç: 6 düğüm, 8 kenar. Minimum kesim 2'dir (ör. {3, 4} | geri kalanı). Koyu düğüm: son büzmede oluşan süper düğüm.

Adım 1 / 5

- **Problem:** çizgeyi iki parçaya ayıran en az kenarlı kesim (global min-cut).
- **Algoritma:** 2 düğüm kalana dek düzgün rastgele bir kenar büzülür; kalan kenar sayısı bir kesimdir.
- Bir çalıştırma minimum kesimi en az **$2/(n(n-1))$** olasılıkla bulur ($n = 6$: $\geq 1/15$). Bu çizgede 200 000 benzetimde gözlenen $\approx 0{,}46$.
- Çıktı yanlış (daha büyük kesim) olabilir → **Monte Carlo**. Ör. ilk kenarlar (1,2), (4,5), (3,4), (0,1) seçilirse {0,1,2} | {3,4,5} kesimi, 3 kenar.
- $n(n-1)/2 \cdot \ln n$ tekrar → başarısızlık $\leq 1/n$; toplam $O(n^4 \log n)$ (basit uygulama).

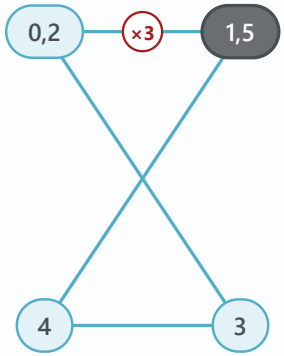
Karger Min-Kesim Algoritması



Rastgele kenar (0, 2) **büzüldü**: uçları tek süper düğümde birleşti, aradaki kenarlar (öz-döngüler) silindi, paralel kenarlar (×k) korunur. Kalan: 7 kenar.

- **Problem:** çizgeyi iki parçaya ayıran en az kenarlı kesim (global min-cut).
- **Algoritma:** 2 düğüm kalana dek düzgün rastgele bir kenar büzülür; kalan kenar sayısı bir kesimdir.
- Bir çalıştırma minimum kesimi en az **$2/(n(n-1))$** olasılıkla bulur ($n = 6$: $\geq 1/15$). Bu çizgede 200 000 benzetimde gözlenen $\approx 0{,}46$.
- Çıktı yanlış (daha büyük kesim) olabilir → **Monte Carlo**. Ör. ilk kenarlar (1,2), (4,5), (3,4), (0,1) seçilirse {0,1,2} | {3,4,5} kesimi, 3 kenar.
- $n(n-1)/2 \cdot \ln n$ tekrar → başarısızlık $\leq 1/n$; toplam $O(n^4 \log n)$ (basit uygulama).

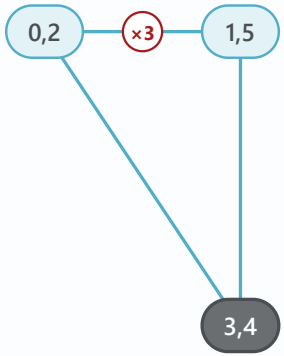
Karger Min-Kesim Algoritması



Rastgele kenar (1, 5) **büzüldü**: uçları tek süper düğümde birleşti, aradaki kenarlar (öz-döngüler) silindi, paralel kenarlar (×k) korunur. Kalan: 6 kenar.

- **Problem:** çizgeyi iki parçaya ayıran en az kenarlı kesim (global min-cut).
- **Algoritma:** 2 düğüm kalana dek düzgün rastgele bir kenar büzülür; kalan kenar sayısı bir kesimdir.
- Bir çalıştırma minimum kesimi en az **$2/(n(n-1))$** olasılıkla bulur ($n = 6$: $\geq 1/15$). Bu çizgede 200 000 benzetimde gözlenen ≈ 0,46.
- Çıktı yanlış (daha büyük kesim) olabilir → **Monte Carlo**. Ör. ilk kenarlar (1,2), (4,5), (3,4), (0,1) seçilirse {0,1,2} | {3,4,5} kesimi, 3 kenar.
- $n(n-1)/2 \cdot \ln n$ tekrar → başarısızlık $\leq 1/n$; toplam $O(n^4 \log n)$ (basit uygulama).

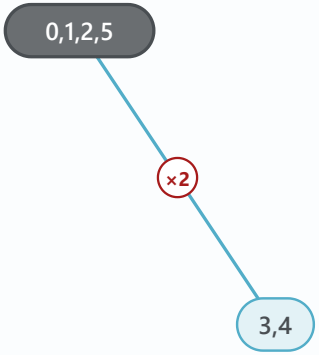
Karger Min-Kesim Algoritması



Rastgele kenar (3, 4) **büzüldü**: uçları tek süper düğümde birleşti, aradaki kenarlar (öz-döngüler) silindi, paralel kenarlar (×k) korunur. Kalan: 5 kenar.

- **Problem:** çizgeyi iki parçaya ayıran en az kenarlı kesim (global min-cut).
- **Algoritma:** 2 düğüm kalana dek düzgün rastgele bir kenar büzülür; kalan kenar sayısı bir kesimdir.
- Bir çalıştırma minimum kesimi en az **2/(n(n−1))** olasılıkla bulur (n = 6: ≥ 1/15). Bu çizgede 200 000 benzetimde gözlenen ≈ 0,46.
- Çıktı yanlış (daha büyük kesim) olabilir → **Monte Carlo**. Ör. ilk kenarlar (1,2), (4,5), (3,4), (0,1) seçilirse {0,1,2} | {3,4,5} kesimi, 3 kenar.
- $n(n-1)/2 \cdot \ln n$ tekrar → başarısızlık ≤ 1/n; toplam $O(n^4 \log n)$ (basit uygulama).

Karger Min-Kesim Algoritması



Rastgele kenar (0, 1) **büzüldü**: uçları tek süper düğümde birleşti, aradaki kenarlar (öz-döngüler) silindi, paralel kenarlar (×k) korunur. Kalan: 2 kenar. İki süper düğüm kaldı → bulunan kesim = **2** (minimum, başarılı çalıştırma).

- **Problem:** çizgeyi iki parçaya ayıran en az kenarlı kesim (global min-cut).
- **Algoritma:** 2 düğüm kalana dek düzgün rastgele bir kenar büzülür; kalan kenar sayısı bir kesimdir.
- Bir çalıştırma minimum kesimi en az **2/(n(n−1))** olasılıkla bulur (n = 6: ≥ 1/15). Bu çizgede 200 000 benzetimde gözlenen ≈ 0,46.
- Çıktı yanlış (daha büyük kesim) olabilir → **Monte Carlo**. Ör. ilk kenarlar (1,2), (4,5), (3,4), (0,1) seçilirse {0,1,2} | {3,4,5} kesimi, 3 kenar.
- $n(n-1)/2 \cdot \ln n$ tekrar → başarısızlık ≤ 1/n; toplam $O(n^4 \log n)$ (basit uygulama).

Diğer Uygulamalar

Çizge boyama
Farklı boyama düzenlerini rastgele kararlarla dener; komşu düğümlere aynı renk vermeden iyi boyamalar arar.

Rastgele DFS
Komşuları rastgele sırayla gezerek bağlantılılığı ve kapsayan ağaçları bulur (ör. labirent üretimi).

Eşleştirme
Rastgele artımlı algoritmalar kenarları rastgele sırayla ekleyerek maksimum eşleşmeye yaklaşır.

Yaklaşık sayma
Count-Min Sketch, rastgele hash fonksiyonlarıyla büyük veri akışında frekansları denetimli hatayla tahmin eder.

Matris işlemleri
Rastgele matris yaklaşımı düşük ranklı yaklaşımlar üretir: boyut indirgeme, veri sıkıştırma.

Hesaplamalı geometri
Rastgele artımlı dışbükey örtü, en yakın nokta çifti, Delaunay üçgenleme: beklenen O(n log n).

Rastgele zamanlama (iş atama) ile benzetimli tavlama ve genetik algoritma gibi metasezgiseller de rastgeleliği arama için kullanır; metasezgiseller optimuma yakınsamayı garanti etmez.

Sözde Rastgele Sayılar

- Bilgisayar gerçek rastgelelik yerine **sözde rastgele** (pseudo-random) sayı üretir: deterministik ama rastgele görünen bir dizi.
- **Doğrusal eşlik üreteci** (LCG): $X_{k+1} = (a \cdot X_k + c) \bmod m$. `java.util.Random` : a = 25214903917, c = 11, m = 2^{48}.
- Aynı **tohum** (seed) aynı diziyi verir: `new Random(42)` ile çalıştırmalar tekrarlanabilir (hata ayıklama, deney).
- Kriptografi için `SecureRandom` ; çok iş parçacıklı kodda `ThreadLocalRandom` .

Rastgele aralıklar

- `zar.nextInt(n)` → 0 … n − 1
- `alt + zar.nextInt(ust − alt + 1)` → alt … ust
- `zar.nextDouble()` → [0, 1)
- `zar.nextInt(6) + 1` → zar atışı: 1 … 6

ÖZET

Özet

Konu	Hatırlanacaklar
Rastgele algoritma	girdi + rastgele bitler; süre/çıktı rastgele değişken; beklenti algoritmanın seçimleri üzerinden
Las Vegas	her zaman doğru, süre rastgele: QuickSort, QuickSelect, BogoSort
Monte Carlo	süre sınırlı, küçük hata olasılığı; tekrarla q^k'ya iner
Fisher–Yates	`j = nextInt(i + 1)`, $\Theta(n)$, $n!$ sonuç eşit olasılıklı; saf karıştırma yanlı
BogoSort	$p = 1/n!$, beklenen $\Theta(n \cdot n!)$, en kötü sınırsız
Rastgele QuickSort	beklenen $\approx 1{,}39\, n \log_2 n$ karşılaştırma, her girdi için $O(n \log n)$; en kötü $O(n^2)$
Miller–Rabin	$n - 1 = 2^s d$; hata $\le 4^{-k}$; "bileşik" kesin
Karger	rastgele kenar büzme; başarı $\ge 2/(n(n-1))$; çok kez tekrarlanır
π tahmini	$4 \cdot$ iç / n; hata $\propto 1/\sqrt{n}$

ÖZET

Etkileşimli Simülatörler

Monte Carlo ile π Tahmini

Birim kareye atılan noktalarla yapılan π tahmini n büyüdükçe yakınsar; hata yaklaşık 1/√n ile küçülür.

ÖZET

Sık Yapılan Hatalar

Saf karıştırma

Her i için `nextInt(n)` ile takas yanlıdır (n^n yol, n! permütasyon). Doğrusu `nextInt(i + 1)`.

"Rastgele pivot en kötüyü yok eder"

Hayır: en kötü durum hâlâ $O(n^2)$. Değişen, hiçbir girdinin sürekli kötü olmaması ve kötü durumun olasılığının çok küçük olmasıdır.

Ortalama ile beklenen süreyi karıştırmak

Ortalama durum girdi dağılımı varsayar; beklenen süre her girdi için algoritmanın yazı-turaları üzerinden hesaplanır.

Monte Carlo yanıtını kesin saymak

Miller–Rabin'de "asal" yanıtı yalnızca **muhtemelen** asaldır; tur sayısı k hata sınırını belirler.

Taşma

`int` ile `x * x % n`, n > 46 341 için taşabilir. `long`, `Math.multiplyHigh` ya da `BigInteger.modPow` kullanılır.

Her çağrıda new Random()

Gereksiz nesne üretir ve tohumlar ilişkili olabilir. Tek bir `Random` paylaşılır; Karger'de kenar **düzgün** seçilir (önce düğüm, sonra komşu seçmek düzgün değildir).

ÖZET

Örnek Problemler

1. Fisher–Yates `[1, 2, 3, 4]` dizisine uygulanıyor; i = 3, 2, 1 için çekilen j değerleri 1, 2, 0. Sonuç nedir?
2. 5 farklı öğeli bir dizide BogoSort'un ilk karıştırmada bitme olasılığı ve beklenen karıştırma sayısı nedir?
3. Rastgele QuickSort'ta en küçük ve en büyük öğe hangi olasılıkla karşılaştırılır? Sıralı hâlde komşu iki öğe?
4. Miller–Rabin: n = 25 için a = 7 ve a = 2 tabanlarıyla testi uygulayın.
5. Monte Carlo ile 1000 noktanın 787'si çeyrek dairenin içine düştü. π tahmini nedir?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. `[3, 8, 2, 5, 1, 4, 7, 6]` dizisinde `bol` çağrılıyor ve `r = 5` (pivot 4) çekiliyor. Bölümleme sonrası dizi ve dönüş değeri nedir?
2. Sıralı 8 öğeli bir dizide ilk öğeyi pivot seçen QuickSort kaç karşılaştırma yapar?
3. Miller–Rabin'de hata olasılığını 10^{-6}'nın altına indirmek için en az kaç tur gerekir?
4. Karger'i 6 düğümlü bir çizgede (başarı ≥ 1/15) kaç kez tekrarlarsanız başarısızlık olasılığı 1/6'nın altına iner?
5. Başarı olasılığı 1/3 olan ve sonucu doğrulanabilen bir Monte Carlo algoritmasını Las Vegas'a çevirin. Beklenen tekrar sayısı nedir?

▶ **Cevapları göster**

Kaynaklar

- R. Motwani ve P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
- M. Mitzenmacher ve E. Upfal, *Probability and Computing*, 2. baskı, Cambridge University Press, 2017.
- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022 (Bölüm 5, 7, 9, 31).
- C. A. R. Hoare, "Quicksort", *The Computer Journal* 5(1), 1962.
- M. O. Rabin, "Probabilistic Algorithm for Testing Primality", *Journal of Number Theory* 12(1), 1980.
- D. R. Karger, "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm", *SODA*, 1993.
- D. E. Knuth, *The Art of Computer Programming*, Cilt 2, 3. baskı, 1997 (Bölüm 3.4.2: karıştırma).

Son

Bölüm 12: Rastgele Algoritmalar

Sonraki bölüm: Bölüm 13: Çevrim İçi Algoritmalar · Terim sözlüğü