Bölüm 2: Açgözlü Algoritmalar

İleri Algoritmalar

Açgözlü yaklaşım | Para üstü | Etkinlik seçimi | İş sıralama | Kesirli sırt çantası | MST ve kesim özelliği | Huffman kodlama

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Bolum_02_Acgozlu_Algoritmalar.html

GİRİŞ

Öğrenme Çıktıları

Yaklaşım

Açgözlü (greedy) stratejiyi tanımlamak; açgözlü seçim özelliği ve optimal alt yapının ne zaman doğru sonuç garantilediğini açıklamak.

Klasik problemler

Para üstü, etkinlik seçimi, iş sıralama ve kesirli sırt çantası problemlerini açgözlü olarak çözmek; başarısız olduğu durumlara karşı örnek vermek.

Kapsayan ağaçlar

Kesim özelliğinin Kruskal, Prim ve Borůvka'nın açgözlü seçimini neden doğru kıldığını açıklamak; üç algoritmayı seçim kuralı, veri yapısı ve süre bakımından karşılaştırmak.

Veri sıkıştırma

Huffman ağacını kurmak, önek-özgür kodları türetmek, bir metni kodlayıp çözmek ve kazancı bit olarak hesaplamak.

AÇGÖZLÜ YAKLAŞIM

Adım Adım Kazananlar

- **Açgözlü algoritmalar** (greedy algorithms), her adımda **o an en iyi görünen** seçeneği seçerek ilerler.
- Tıpkı bir **define arayan** gibi: her aşamada en fazla kazancı sağlayacak seçeneği tercih eder.
- Basamaklı bir yaklaşımdır: problemi bütün olarak ele almaz, **mevcut en iyi seçeneğe** odaklanır.
- Verilen bir karar **geri alınmaz**; geri izleme (backtracking) yoktur.
- Yolculuk planlarken her kavşakta en kısa görünen yolu seçmek gibi.

Kazanç ve risk

Hızlı ve basit bir yöntem sunar; kod yapısı genellikle kısa ve anlaşılırdır. Ancak **en iyi (optimal) çözümü bulmayı garanti etmez**; çoğu durumda iyi bir yaklaşık çözüm üretir.

Açgözlü Paradigma: Genel Şablon

```
Cozum acgozlu(Aday[] adaylar) {
  Cozum s = new Cozum();    // boş çözüm
  sirala(adaylar);          // ölçüte göre
  for (Aday a : adaylar) {
    if (s.uygunMu(a))       // kısıt bozulmaz
      s.ekle(a);            // geri alınmaz
  }
  return s;
}
```

1. **Uygun çözümler** kümesi belirlenir (hangi seçimler kısıtları bozmaz?).
2. Bir **açgözlü ölçüt** seçilir: en ucuz, en erken biten, en kârlı, en sık…
3. Ölçüte göre **yerel olarak en iyi** aday seçilir; uygunsa çözüme eklenir.
4. Çözüm tamamlanana kadar bu adımlar tekrarlanır.



Çoğu açgözlü algoritmanın maliyeti **sıralama** ya da **öncelikli kuyruk** işlemlerinden gelir: tipik olarak **O(n log n)**.

AÇGÖZLÜ YAKLAŞIM

Ne Zaman Doğru Sonuç Verir?

Açgözlü seçim özelliği

(Greedy-choice property) Yerel olarak en iyi seçimi içeren **en az bir** en iyi çözüm vardır. İlk açgözlü seçim hiçbir zaman "yanlış yola" sokmaz.

Optimal alt yapı

(Optimal substructure) Seçim yapıldıktan sonra kalan alt problemin en iyi çözümü, ilk seçimle birlikte tüm problemin en iyi çözümünü verir.

Kanıt tekniği: değiştirme argümanı

(Exchange argument) Herhangi bir en iyi çözüm O alınır. O, açgözlü seçimi içermiyorsa, O'daki bir öğe açgözlü seçimle **değiştirilir**; çözüm uygun kalır ve daha kötü olmaz. Böylece açgözlü seçimi içeren bir en iyi çözüm bulunur; tümevarımla tüm adımlar için geçerlidir.

İki özellik de sağlanmıyorsa açgözlü algoritma **yalnızca bir sezgisel** (heuristic) yöntemdir: hızlıdır ama sonucu en iyiden kötü olabilir.

Avantajlar ve Dezavantajlar

Avantajlar

+ Anlaması ve uygulaması kolay; kod kısa.
+ Hızlı: çoğunlukla sıralama + tek geçiş, O(n log n).
+ Az bellek: yalnızca mevcut çözüm tutulur.
+ Doğru problemde (MST, Huffman, etkinlik seçimi) en iyi sonucu verir.
+ Zor problemlerde iyi bir yaklaşık çözüm ya da başlangıç noktası sağlar.

Dezavantajlar

– En iyi çözümü **garanti etmez** (para üstü {1, 3, 4}, 0/1 sırt çantası).
– Doğruluğu her problem için ayrıca **kanıtlanmalıdır**.
– Ölçüt seçimi kritik: yanlış ölçüt yanlış sonuç verir.
– Verilen karar geri alınmaz; hatayı düzeltme şansı yoktur.

AÇGÖZLÜ YAKLAŞIM

Para Üstü Problemi (Coin Change)

Belirli bir tutarı **en az sayıda** madeni parayla ödemek.

Açgözlü ölçüt: kalan tutardan büyük olmayan **en büyük** para seçilir; tutar sıfır olana kadar bu adım tekrarlanır.

```
// para[] büyükten küçüğe sıralı
static int paraUstu(int[] para, int tutar) {
  int adet = 0;
  for (int p : para) {
    while (tutar >= p) {
      tutar -= p;
      adet++;
    }
  }
  return adet;
}
```

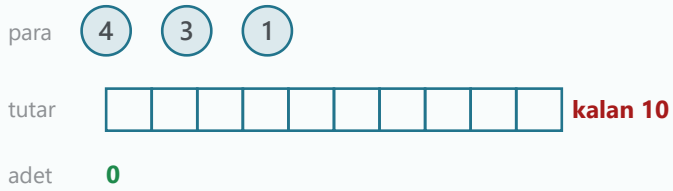
Her zaman doğru mu?

{50, 25, 10, 5, 1} gibi **kanonik** para sistemlerinde evet: 68 = 50 + 10 + 5 + 1 + 1 + 1.

{4, 3, 1} sisteminde 6 için açgözlü 4 + 1 + 1 (3 para) verir; en iyisi 3 + 3 (2 para). Genel durumda **dinamik programlama** gerekir. Zaman: O(k + adet), k = para türü sayısı.

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
▶    int adet = 0;
4    for (int p : para) {
5      while (tutar >= p) {
6        tutar -= p;
7        adet++;
8      }
9    }
10   return adet;
11 }
```



Paralar {4, 3, 1} büyükten küçüğe sıralı, tutar **10**. Bu sistem kanonik değil: açgözlü seçimin sonucu en iyi olmayabilir.

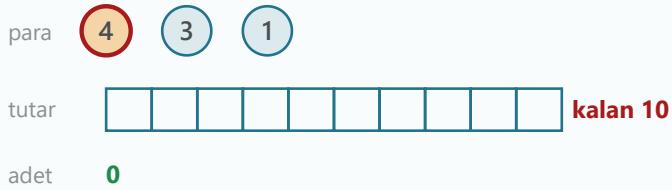
Adım 1 / 9

DEĞİŞKENLER

`p = –` `tutar = 10` `adet = 0`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
▶    for (int p : para) {
▶      while (tutar >= p) {
6        tutar -= p;
7        adet++;
8      }
9    }
10   return adet;
11 }
```



p = 4: 10 ≥ 4 → bu paradan alınabilir.

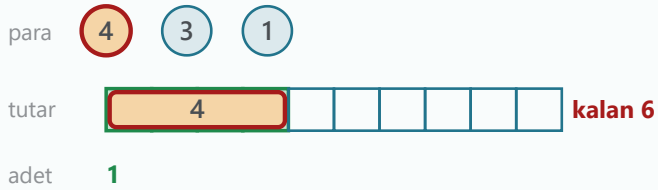
Adım 2 / 9

DEĞİŞKENLER

`p = 4` `tutar = 10` `adet = 0`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
4    for (int p : para) {
5      while (tutar >= p) {
▶        tutar -= p;
▶        adet++;
8      }
9    }
10   return adet;
11 }
```



tutar −= 4 → **6**, adet = 1. Döngü sürer.

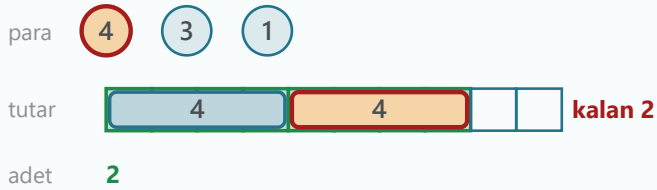
Adım 3 / 9

DEĞİŞKENLER

`p = 4` `tutar = 6` `adet = 1`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
4    for (int p : para) {
5      while (tutar >= p) {
▶        tutar -= p;
▶        adet++;
8      }
9    }
10   return adet;
11 }
```



tutar −= 4 → **2**, adet = 2. 2 < 4: while biter.

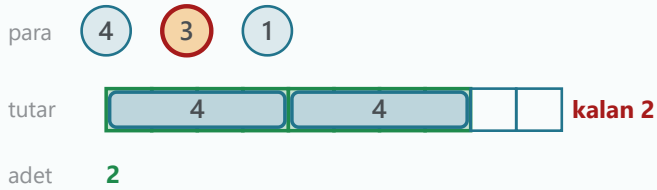
Adım 4 / 9

DEĞİŞKENLER

`p = 4` `tutar = 2` `adet = 2`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
▶    for (int p : para) {
▶      while (tutar >= p) {
6        tutar -= p;
7        adet++;
8      }
9    }
10   return adet;
11 }
```



p = 3: 2 < 3 → 3'lük para **hiç kullanılmaz**, sıradaki paraya geçilir.

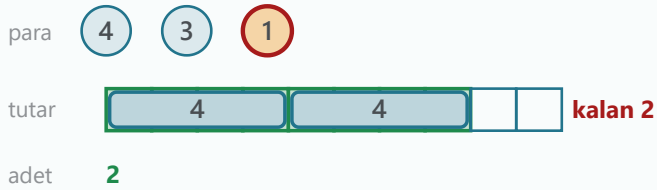
Adım 5 / 9

DEĞİŞKENLER

`p = 3` `tutar = 2` `adet = 2`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
▶    for (int p : para) {
▶      while (tutar >= p) {
6        tutar -= p;
7        adet++;
8      }
9    }
10   return adet;
11 }
```



p = 1: 2 ≥ 1 → bu paradan alınabilir.

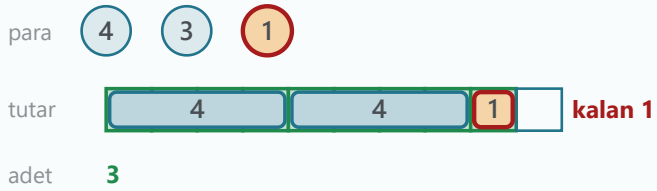
Adım 6 / 9

DEĞİŞKENLER

`p = 1` `tutar = 2` `adet = 2`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
4    for (int p : para) {
5      while (tutar >= p) {
▶        tutar -= p;
▶        adet++;
8      }
9    }
10   return adet;
11 }
```



tutar −= 1 → **1**, adet = 3. Döngü sürer.

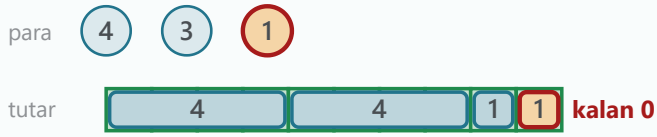
Adım 7 / 9

DEĞİŞKENLER

`p = 1` `tutar = 1` `adet = 3`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
4    for (int p : para) {
5      while (tutar >= p) {
▶        tutar -= p;
▶        adet++;
8      }
9    }
10   return adet;
11 }
```



tutar −= 1 → **0**, adet = 4. 0 < 1: while biter.

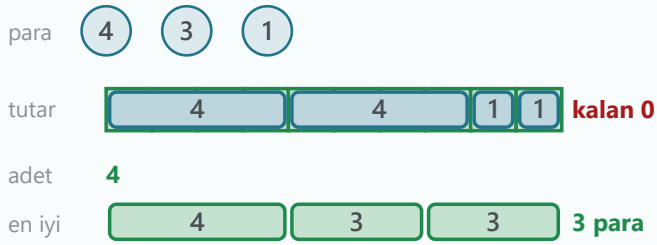
Adım 8 / 9

DEĞİŞKENLER

`p = 1` `tutar = 0` `adet = 4`

paraUstu({4, 3, 1}, 10): Kod İzleme

```
1  // para[] büyükten küçüğe sıralı
2  static int paraUstu(int[] para, int tutar) {
3    int adet = 0;
4    for (int p : para) {
5      while (tutar >= p) {
6        tutar -= p;
7        adet++;
8      }
9    }
▶    return adet;
11 }
```



Açgözlü 4 + 4 + 1 + 1 = **4 para** döndürür; oysa 4 + 3 + 3 = **3 para** yeterli (DP ile bulunur). İlk iki 4'lük seçim, 3'lüklerin kullanılmasını engelledi.

Adım 9 / 9

DEĞİŞKENLER

`p = –` `tutar = 0` `adet = 4`

Para Üstü: Adım Adım

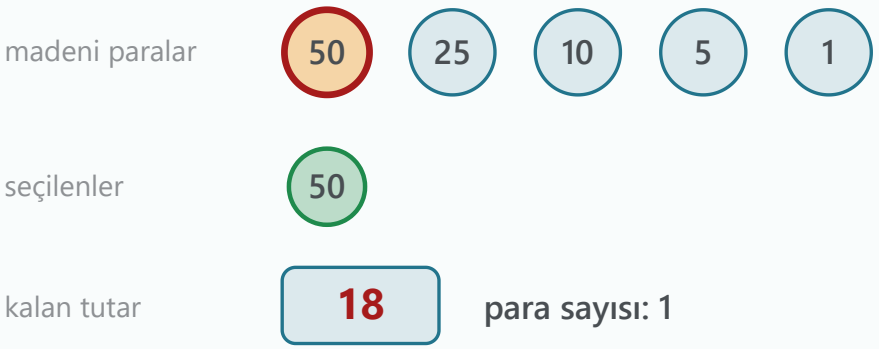
madeni paralar 50 25 10 5 1

seçilenler

kalan tutar **68** para sayısı: 0

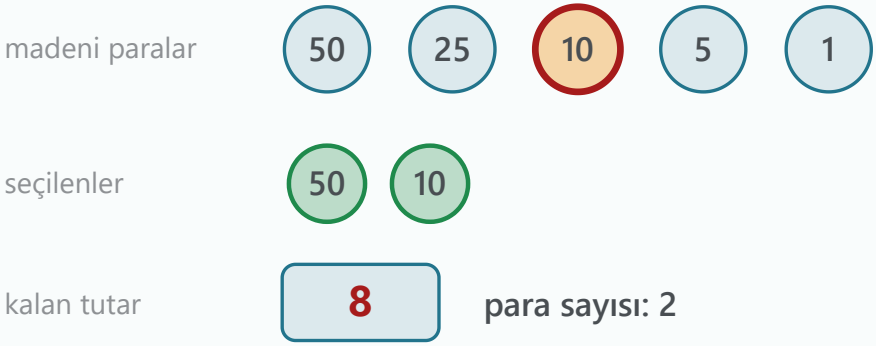
Örnek 1: paralar {50, 25, 10, 5, 1}, tutar **68**. Seçime en büyük paradan başlanır.

Para Üstü: Adım Adım



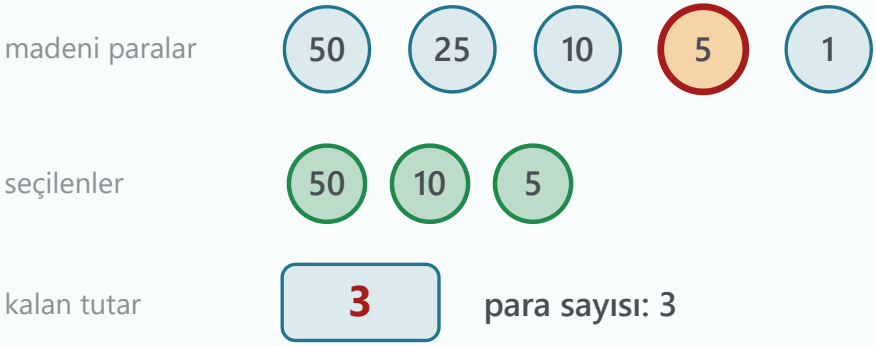
Kalan tutara sığan en büyük para **50** → kalan 68 − 50 = **18**.

Para Üstü: Adım Adım



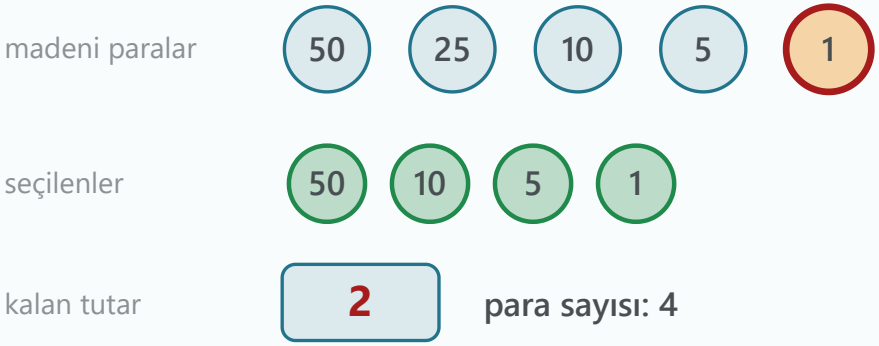
Kalan tutara sığan en büyük para **10** → kalan 18 − 10 = **8**.

Para Üstü: Adım Adım



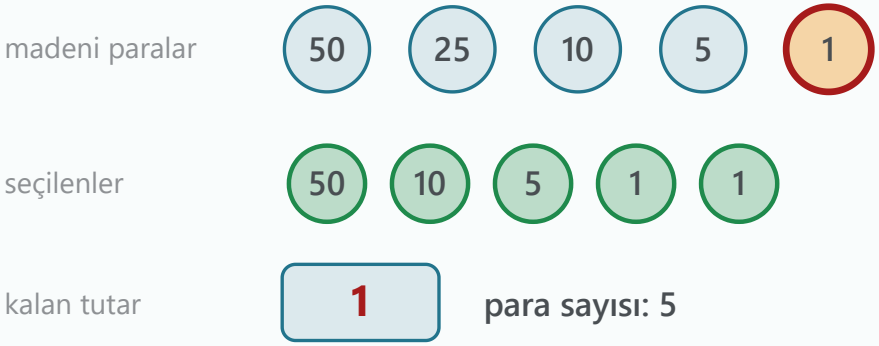
Kalan tutara sığan en büyük para **5** → kalan 8 − 5 = **3**.

Para Üstü: Adım Adım



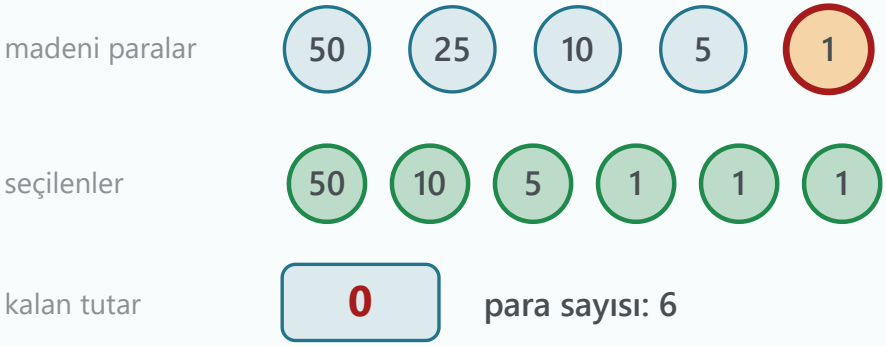
Kalan tutara sığan en büyük para **1** → kalan 3 − 1 = **2**.

Para Üstü: Adım Adım



Kalan tutara sığan en büyük para **1** → kalan 2 − 1 = **1**.

Para Üstü: Adım Adım



Kalan tutara sığan en büyük para **1** → kalan 1 − 1 = **0**. Toplam **6 para** — bu para sistemi için en iyi sonuç.

Para Üstü: Adım Adım

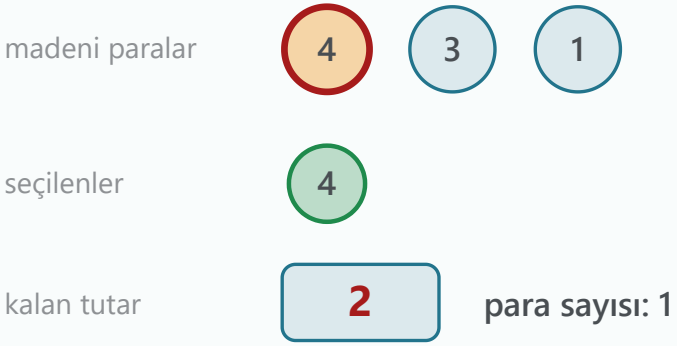
madeni paralar 4 3 1

seçilenler

kalan tutar **6** para sayısı: 0

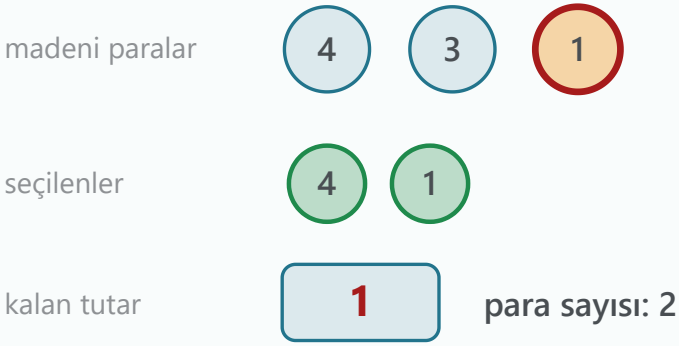
Örnek 2: paralar {4, 3, 1}, tutar **6**. Seçime en büyük paradan başlanır.

Para Üstü: Adım Adım



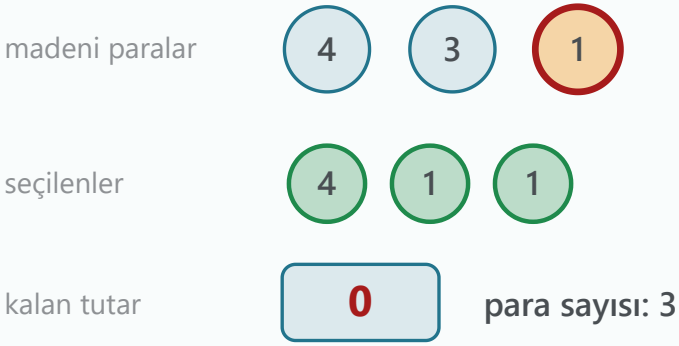
Kalan tutara sığan en büyük para **4** → kalan 6 − 4 = **2**.

Para Üstü: Adım Adım



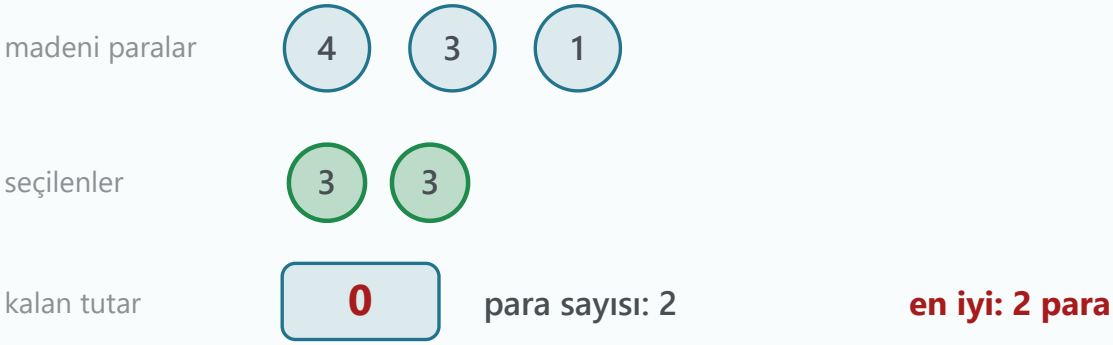
Kalan tutara sığan en büyük para **1** → kalan 2 − 1 = **1**.

Para Üstü: Adım Adım



Kalan tutara sığan en büyük para **1** → kalan 1 − 1 = **0**.

Para Üstü: Adım Adım



Açgözlü 4 + 1 + 1 = **3 para** verdi; oysa **3 + 3 = 2 para** yeterli. Yerel en iyi seçim (4) küresel en iyiyi kaçırdı.

AÇGÖZLÜ YAKLAŞIM

Deneyin: Açgözlü ve En İyi Para Üstü

Paralar `1, 3, 4` Tutar `6` **hesapla** **TL kuruş** **1, 7, 10**

Açgözlü

4 1 1

3 para

En iyi (DP)

3 3

2 para

Açgözlü **en iyi değil**: 3 para, en iyi 2 para (3 + 3).

En iyi sonuç dinamik programlama ile bulunur: enAz[t] = 1 + min enAz[t − p]. Paralar 1–999 arası tam sayı, en fazla 8 tür; tutar en fazla 999. Enter = hesapla.

Açgözlü Yaklaşım ve Böl ve Fethet

	Açgözlü	Böl ve fethet
Karar	her adımda tek, yerel en iyi seçim; geri alınmaz	sabit bir kuralla bölünür; seçim yapılmaz
Alt problemler	seçimden sonra tek alt problem kalır	birbirinden bağımsız birkaç alt problem
Doğruluk	açgözlü seçim özelliği + optimal alt yapı kanıtlanmalı	birleştirme adımının doğruluğuna bağlı
Maliyet	genelde O(n log n): sıralama + tek geçiş	yineleme bağıntısı ve ana teoremle
Örnek	etkinlik seçimi, Kruskal, Huffman	birleştirme sıralaması, Karatsuba

İkisi de problemi küçültür: böl ve fethet **bütün** parçaları çözüp birleştirir; açgözlü yaklaşım **tek** bir seçim yapar ve geriye kalan tek alt problemle devam eder.

Etkinlik Seçimi (Activity Selection)

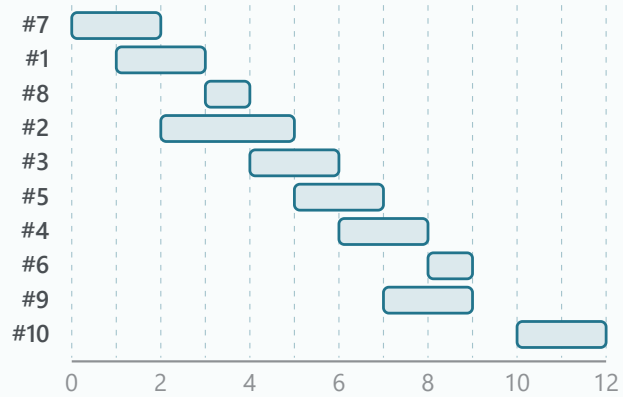
- Her etkinliğin bir **başlangıç** s_i ve **bitiş** f_i zamanı var; aynı anda tek etkinlik yapılabilir (tek salon).
- Amaç: birbiriyle çakışmayan **en fazla sayıda** etkinliği seçmek. $[s_i, f_i)$ ile $[s_j, f_j)$ çakışmaz $\Leftrightarrow$ $s_j \geq f_i$ ya da $s_i \geq f_j$.
- **Açgözlü ölçüt: en erken biten** etkinlik seçilir, onunla çakışanlar atılır, kalanlarla aynı adım tekrarlanır.

Etkinlik	1	2	3	4	5	6	7	8	9	10
başlangıç	1	2	4	6	5	8	0	3	7	10
bitiş	3	5	6	8	7	9	2	4	9	12

Zaman: sıralama O(n log n) + tek geçiş O(n). Etkinlikler zaten bitişe göre sıralıysa **O(n)**.

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
▶    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
6      if (e[j].bas >= e[i].bitis) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



Etkinlikler **bitiş zamanına göre** sıralandı (yukarıdan aşağı). Kesikli çizgi: son seçilenin bitişi.

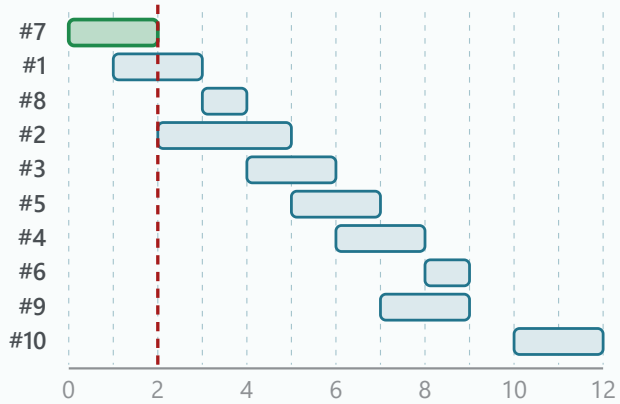
Adım 1 / 11

DEĞİŞKENLER

`i = –` `j = –` `e[i].bitis = –` `çıktı = –`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
▶    int i = 0;                    // son seçilen
▶    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
6      if (e[j].bas >= e[i].bitis) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



En erken biten etkinlik (#7, [0, 2)) **her zaman** seçilir.

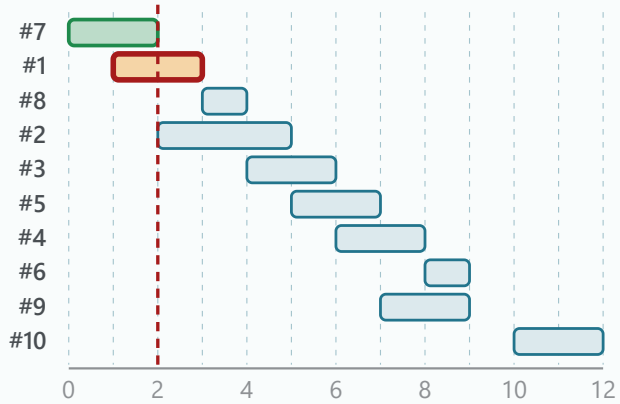
Adım 2 / 11

DEĞİŞKENLER

`i = 0` `j = –` `e[i].bitis = 2` `çıktı = 7`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                  // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#1 [1, 3): başlangıç 1 < 2 → çakışıyor, **atlandı**.

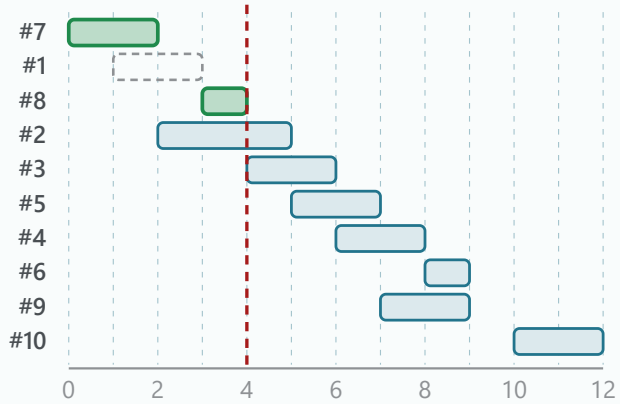
Adım 3 / 11

DEĞİŞKENLER

`i = 0` `j = 1` `e[i].bitis = 2` `çıktı = 7`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#8 [3, 4): başlangıç 3 ≥ 2 → **seçildi**, i = 2.

Adım 4 / 11

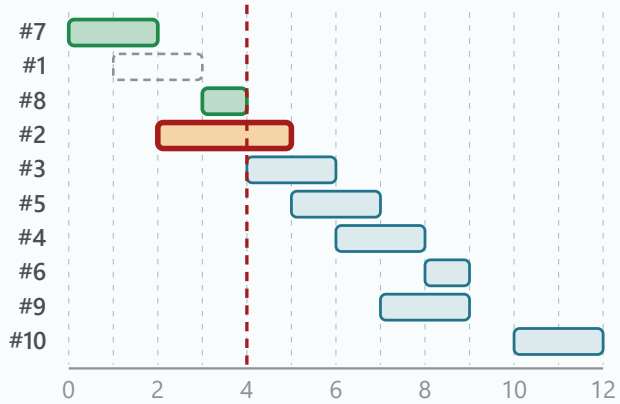
DEĞİŞKENLER

`i = 2` `j = 2` `e[i].bitis = 4`

`çıktı = 7 8`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#2 [2, 5): başlangıç 2 < 4 → çakışıyor, **atlandı**.

Adım 5 / 11

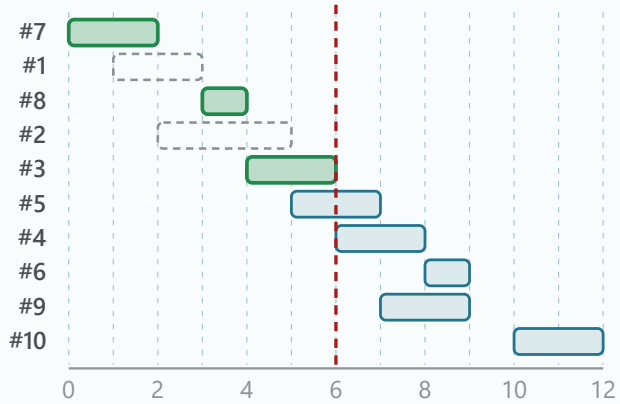
DEĞİŞKENLER

`i = 2` `j = 3` `e[i].bitis = 4`

`çıktı = 7 8`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#3 [4, 6): başlangıç 4 ≥ 4 → **seçildi**, i = 4.

Adım 6 / 11

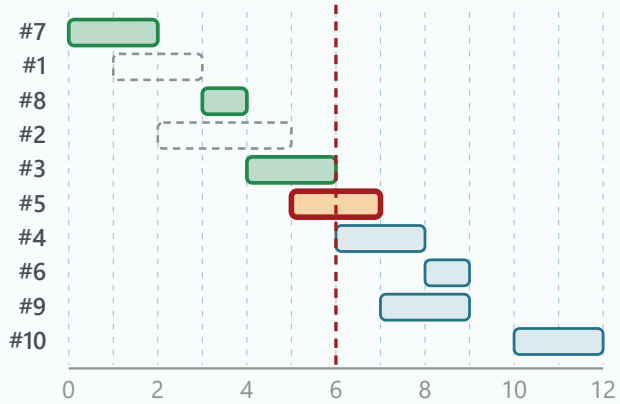
DEĞİŞKENLER

`i = 4` `j = 4` `e[i].bitis = 6`

`çıktı = 7 8 3`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#5 [5, 7): başlangıç 5 < 6 → çakışıyor, **atlandı**.

Adım 7 / 11

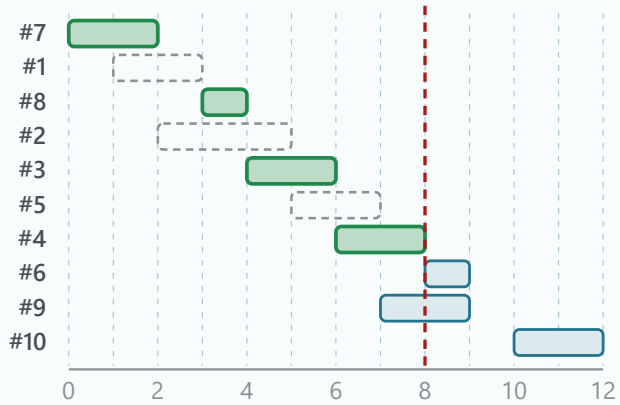
DEĞİŞKENLER

`i = 4` `j = 5` `e[i].bitis = 6`

`çıktı = 7 8 3`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#4 [6, 8): başlangıç 6 ≥ 6 → **seçildi**, i = 6.

Adım 8 / 11

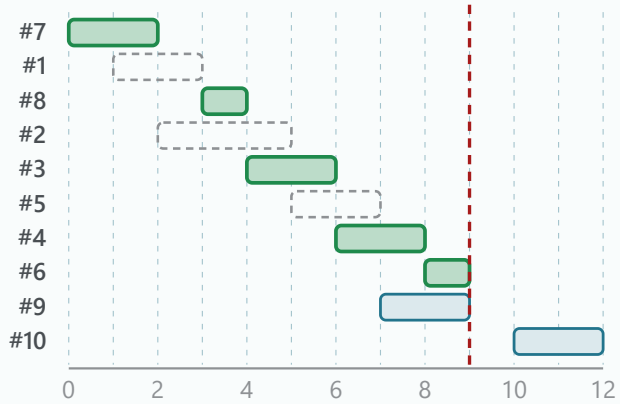
DEĞİŞKENLER

`i = 6` `j = 6` `e[i].bitis = 8`

`çıktı = 7 8 3 4`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#6 [8, 9): başlangıç 8 ≥ 8 → **seçildi**, i = 7.

Adım 9 / 11

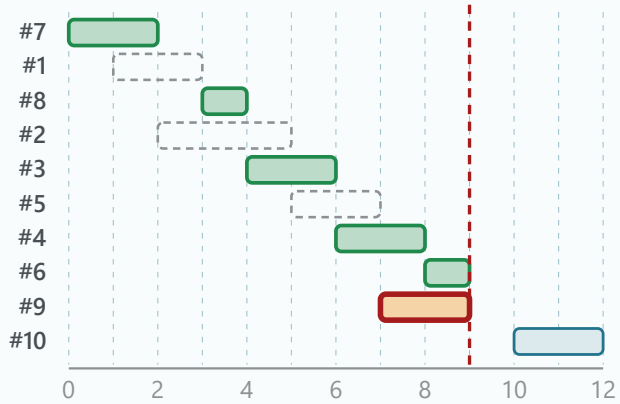
DEĞİŞKENLER

`i = 7` `j = 7` `e[i].bitis = 9`

`çıktı = 7 8 3 4 6`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#9 [7, 9): başlangıç 7 < 9 → çakışıyor, **atlandı**.

Adım 10 / 11

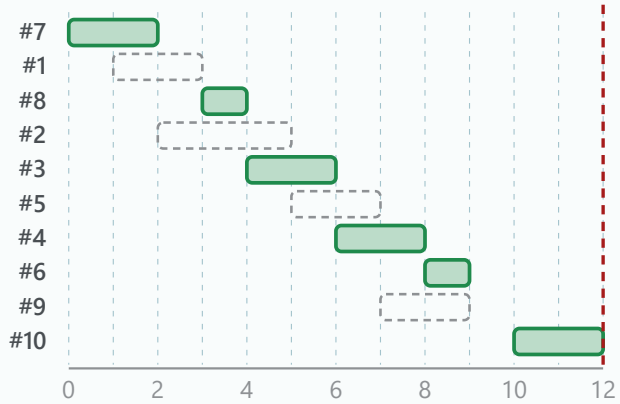
DEĞİŞKENLER

`i = 7` `j = 8` `e[i].bitis = 9`

`çıktı = 7 8 3 4 6`

sec(e): Kodu Adım Adım İzle

```
1  static void sec(Etkinlik[] e) {
2    Arrays.sort(e, (x, y) -> x.bitis -
   y.bitis);
3    int i = 0;                    // son seçilen
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].bas >= e[i].bitis) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#10 [10, 12): başlangıç 10 ≥ 9 → **seçildi**, i = 9.
Sonuç: **6 etkinlik** (7, 8, 3, 4, 6, 10).

Adım 11 / 11

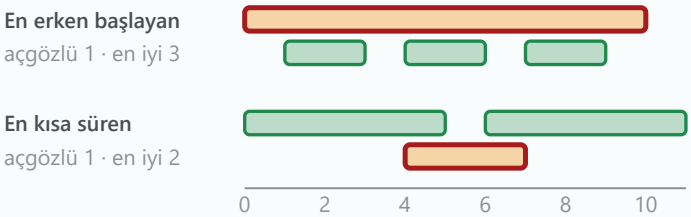
DEĞİŞKENLER

`i = 9` `j = 9` `e[i].bitis = 12`

`çıktı = 7 8 3 4 6 10`

Neden En Erken Biten?

Akla gelen diğer ölçütler en iyi sonucu **garanti etmez** (turuncu: açgözlünün seçtiği, yeşil: en iyi çözüm):



"En az çakışan" ölçütünün de karşı örneği vardır.

Değiştirme argümanı

a_1 en erken biten etkinlik, O ise bir en iyi çözüm olsun. O'nun ilk etkinliği $b \neq a_1$ ise b yerine a_1 konur: $f(a_1) \leq f(b)$ olduğundan O'nun diğer etkinlikleriyle çakışma oluşmaz ve etkinlik sayısı aynı kalır.
Demek ki a_1'i içeren bir en iyi çözüm vardır. a_1 ile çakışanlar atılınca kalan problem aynı yapıdadır (optimal alt yapı).

Son Tarihli İş Sıralama (Job Sequencing)

- Her iş **1 birim** sürer, bir **son tarihi** (deadline) ve bir **kârı** vardır.
- Kâr, iş yalnızca son tarihine kadar bitirilirse kazanılır; aynı anda tek iş yapılır.
- Amaç: **toplam kârı** en büyük yapmak.
- **Açgözlü ölçüt:** işler kâra göre **azalan** sırayla ele alınır; her iş son tarihinden önceki **en geç boş** zaman dilimine konur; boş dilim yoksa iş atlanır.
- En geç dilimi seçmek, erken dilimleri son tarihi daha yakın işlere bırakır.

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

Zaman: sıralama $O(n \log n)$ + slot arama $O(n \cdot d_{max})$ → en kötü **$O(n^2)$**. Union-Find ile slot arama neredeyse $O(n)$'e iner.

İşleri **son tarihe** göre sıralamak yanlıştır: (kâr 10, son tarih 1) ve (kâr 40, son tarih 1) için son tarih sırası 10, kâr sırası 40 kazanır. Kâra göre azalan sıra optimaldir: olurlu iş kümeleri bir **matroid** oluşturur.

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
▶    Arrays.sort(is, (a, b) -> b.kar - a.kar);
▶    int[] slot = new int[n];
▶    Arrays.fill(slot, -1);
▶    int toplam = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, is[i].sonTarih) -
   1;
8      for (; j >= 0; j--) {
9        if (slot[j] == -1) {
10         slot[j] = is[i].id;
11         toplam += is[i].kar;
12         break;
13       }
14     }
15   }
16   return toplam;
17 }
```

kâra göre sıralı işler

İş 5	İş 3	İş 4	İş 6	İş 1	İş 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
kâr 50	kâr 40	kâr 30	kâr 30	kâr 20	kâr 10

slot (zaman dilimi)

1	2	3	4	5	6
—	—	—	—	—	—

İşler **kâra göre azalan** sıralandı (eşit kârlı İş 4 ve İş 6 özgün sırasını korur). Tüm slotlar boş (−1).

Adım 1 / 8

DEĞİŞKENLER

`i = –` `j = –` `toplam = 0`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, is[i].sonTarih) -
1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = is[i].id;
▶          toplam += is[i].kar;
12           break;
13         }
14       }
15     }
16     return toplam;
17   }
```

kâra göre sıralı işler

İş 5	İş 3	İş 4	İş 6	İş 1	İş 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
kâr 50	kâr 40	kâr 30	kâr 30	kâr 20	kâr 10

slot (zaman dilimi)

1	2	3	4	5	6
—	—	İş 5	—	—	—

İş 5 (son tarih 3, kâr 50): en geç uygun slot **3** boş → yerleştirildi. toplam = 50.

Adım 2 / 8

DEĞİŞKENLER

`i = 0` `j = 2` `toplam = 50`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, is[i].sonTarih) -
1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = is[i].id;
▶          toplam += is[i].kar;
12           break;
13         }
14       }
15   }
16   return toplam;
17 }
```

kâra göre sıralı işler

İş 5	İş 3	İş 4	İş 6	İş 1	İş 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
kâr 50	kâr 40	kâr 30	kâr 30	kâr 20	kâr 10

slot (zaman dilimi)

1	2	3	4	5	6
İş 3	—	İş 5	—	—	—

İş 3 (son tarih 1, kâr 40): en geç uygun slot **1** boş → yerleştirildi. toplam = 90.

Adım 3 / 8

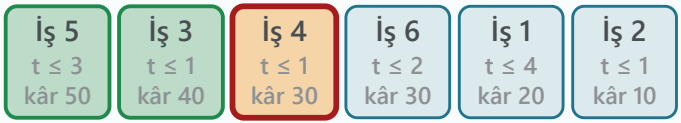
DEĞİŞKENLER

`i = 1` `j = 0` `toplam = 90`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, is[i].sonTarih) -
   1;
8      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
10         slot[j] = is[i].id;
11         toplam += is[i].kar;
12         break;
13       }
14     }
15   }
16   return toplam;
17 }
```

kâra göre sıralı işler



slot (zaman dilimi)



İş 4 (son tarih 1, kâr 30): slot 1 **dolu** (İş 3) ve daha erken slot yok → iş **atlanır**.

Adım 4 / 8

DEĞİŞKENLER

`i = 2` `j = 0` `toplam = 90`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, is[i].sonTarih) -
1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = is[i].id;
▶          toplam += is[i].kar;
12           break;
13         }
14       }
15   }
16   return toplam;
17 }
```

kâra göre sıralı işler

İş 5	İş 3	İş 4	İş 6	İş 1	İş 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
kâr 50	kâr 40	kâr 30	kâr 30	kâr 20	kâr 10

slot (zaman dilimi)

1	2	3	4	5	6
İş 3	İş 6	İş 5	—	—	—

İş 6 (son tarih 2, kâr 30): en geç uygun slot **2** boş → yerleştirildi. toplam = 120.

Adım 5 / 8

DEĞİŞKENLER

`i = 3` `j = 1` `toplam = 120`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, is[i].sonTarih) -
1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = is[i].id;
▶          toplam += is[i].kar;
12           break;
13         }
14       }
15   }
16   return toplam;
17 }
```

kâra göre sıralı işler

İş 5	İş 3	İş 4	İş 6	İş 1	İş 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
kâr 50	kâr 40	kâr 30	kâr 30	kâr 20	kâr 10

slot (zaman dilimi)

1	2	3	4	5	6
İş 3	İş 6	İş 5	İş 1	—	—

İş 1 (son tarih 4, kâr 20): en geç uygun slot **4** boş → yerleştirildi. toplam = 140.

Adım 6 / 8

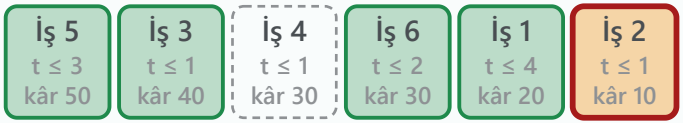
DEĞİŞKENLER

`i = 4` `j = 3` `toplam = 140`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, is[i].sonTarih) -
   1;
8      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
10         slot[j] = is[i].id;
11         toplam += is[i].kar;
12         break;
13       }
14     }
15   }
16   return toplam;
17 }
```

kâra göre sıralı işler



slot (zaman dilimi)



İş 2 (son tarih 1, kâr 10): slot 1 **dolu** (İş 3) ve daha erken slot yok → iş **atlanır**.

Adım 7 / 8

DEĞİŞKENLER

`i = 5` `j = 0` `toplam = 140`

isSirala(is, 6): Kodu Adım Adım İzle

```
1  static int isSirala(Is[] is, int n) {
2    Arrays.sort(is, (a, b) -> b.kar - a.kar);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int toplam = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, is[i].sonTarih) -
   1;
8      for (; j >= 0; j--) {
9        if (slot[j] == -1) {
10         slot[j] = is[i].id;
11         toplam += is[i].kar;
12         break;
13       }
14     }
15   }
▶    return toplam;
17 }
```

kâra göre sıralı işler

İş 5	İş 3	İş 4	İş 6	İş 1	İş 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
kâr 50	kâr 40	kâr 30	kâr 30	kâr 20	kâr 10

slot (zaman dilimi)

1	2	3	4	5	6
İş 3	İş 6	İş 5	İş 1	—	—

Sonuç: slotlar 1–4 = İş 3, İş 6, İş 5, İş 1 → **toplam kâr 140**. İş 4 ve İş 2 yapılamaz.

Adım 8 / 8

DEĞİŞKENLER

`i = 6` `j = –` `toplam = 140`

SIRT ÇANTASI

Kesirli Sırt Çantası (Fractional Knapsack)

Kapasite **W = 50**. Nesneler bölünebiliyorsa açgözlü ölçüt: **kâr / ağırlık oranı** en büyük olanı al.

Nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4

Kesirli: 1 + 2 tümüyle, 3'ün 20/30'u → 60 + 100 + 80 = **240** (en iyi). Zaman O(n log n).

Neden en iyi?

En yüksek oranlı nesnenin her birimi, aynı ağırlıktaki başka her seçenekten en az onun kadar kâr getirir. En iyi çözüm bu nesneden alabileceği kadar almıyorsa, başka bir nesnenin aynı ağırlıktaki kısmı onunla değiştirilir ve kâr azalmaz (değiştirme argümanı).

Nesneler bölünemiyorsa (0/1 sırt çantası) oran ölçütü en iyiyi garanti etmez; o problem **dinamik programlama** ile çözülür.



SIRT ÇANTASI · KODU ADIM ADIM ÇALIŞTIR

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4

çanta W = 50



kalan = 50 **kar = 0**

Nesneler kâr/ağırlık oranına göre azalan sıralandı: 1 (6), 2 (5), 3 (4). Çanta boş, kalan kapasite **50**.

Adım 1 / 8

```
1  static double kesirli(Nesne[] s, int W) {
▶    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
▶    double kar = 0;
▶    int kalan = W;
5    for (Nesne x : s) {
6      if (x.agirlik <= kalan) {        // tamamı sığar
7        kalan -= x.agirlik;
8        kar += x.kar;
9      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = –` `kalan = 50` `kar = 0`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4



Nesne 1: ağırlık 10 ≤ kalan 50 → **tamamı** sığar.

Adım 2 / 8

```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
▶    for (Nesne x : s) {
▶      if (x.agirlik <= kalan) {        // tamamı sığar
7        kalan -= x.agirlik;
8        kar += x.kar;
9      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = 1` `kalan = 50` `kar = 0`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4



kalan −= 10 → 40, kar += 60 → **60**.

Adım 3 / 8

```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
5    for (Nesne x : s) {
6      if (x.agirlik <= kalan) {        // tamamı sığar
▶        kalan -= x.agirlik;
▶        kar += x.kar;
9      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = 1` `kalan = 40` `kar = 60`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4



Nesne 2: ağırlık 20 ≤ kalan 40 → **tamamı** sığar.

Adım 4 / 8

```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
▶    for (Nesne x : s) {
▶      if (x.agirlik <= kalan) {        // tamamı sığar
7        kalan -= x.agirlik;
8        kar += x.kar;
9      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = 2` `kalan = 40` `kar = 60`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4



kalan −= 20 → 20, kar += 100 → **160**.

Adım 5 / 8

```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
5    for (Nesne x : s) {
6      if (x.agirlik <= kalan) {        // tamamı sığar
▶        kalan -= x.agirlik;
▶        kar += x.kar;
9      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = 2` `kalan = 20` `kar = 160`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4



Nesne 3: ağırlık 30 > kalan 20 → tamamı sığmaz, **else** dalı.

Adım 6 / 8

```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
▶    for (Nesne x : s) {
▶      if (x.agirlik <= kalan) {        // tamamı sığar
7        kalan -= x.agirlik;
8        kar += x.kar;
▶      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = 3` `kalan = 20` `kar = 160`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4

çanta W = 50



çanta dolu **kar = 240**

Kalan 20 birim için nesne 3'ün **20/30**'u: 120 · 20/30 = 80 → kar = **240**. Çanta doldu, `break`.

Adım 7 / 8

```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
5    for (Nesne x : s) {
6      if (x.agirlik <= kalan) {        // tamamı sığar
7        kalan -= x.agirlik;
8        kar += x.kar;
9      } else {                         // kesirli parça
▶        kar += x.kar * (double) kalan / x.agirlik;
▶        break;
12     }
13   }
14   return kar;
15 }
```

DEĞİŞKENLER

`x = 3` `kalan = 20` `kar = 240`

kesirli(s, 50): Kod İzleme

nesne	kâr	ağırlık	oran
1	60	10	6
2	100	20	5
3	120	30	4



Sonuç **240**: kesirli problemde en iyi. Maliyet sıralamadan gelir: **O(n log n)**; tarama O(n).

Adım 8 / 8

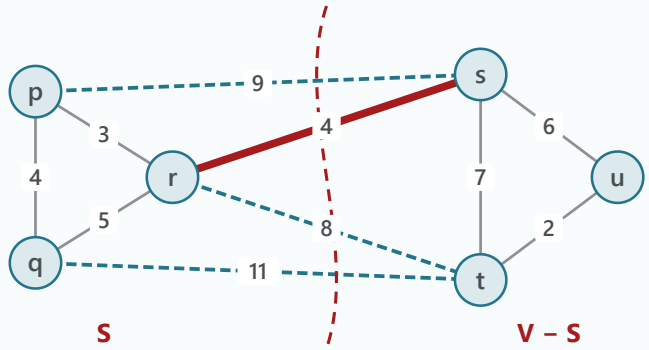
```
1  static double kesirli(Nesne[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.oran(), a.oran()));
3    double kar = 0;
4    int kalan = W;
5    for (Nesne x : s) {
6      if (x.agirlik <= kalan) {        // tamamı sığar
7        kalan -= x.agirlik;
8        kar += x.kar;
9      } else {                         // kesirli parça
10       kar += x.kar * (double) kalan / x.agirlik;
11       break;
12     }
13   }
▶  14   return kar;
15 }
```

DEĞİŞKENLER

`x = –` `kalan = 20` `kar = 240`

Minimum Kapsayan Ağaç ve Kesim Özelliği

- **Minimum kapsayan ağaç** (minimum spanning tree, MST): bağlı, ağırlıklı, yönsüz bir çizgenin tüm düğümlerini **en az toplam ağırlıkla** bağlayan ağaç; V − 1 kenar.
- **Kesim özelliği** (cut property): düğümler S ve V − S diye ikiye ayrılırsa, bu kesimi geçen **en hafif kenar** bir MST'de yer alır.
- Kruskal, Prim ve Borůvka bu özelliği farklı kesimlerle kullanan **açgözlü** algoritmalardır: her eklenen kenar, bir kesimi geçen en hafif kenardır; bu yüzden hiçbir seçim geri alınmaz.
- Ağırlıklar farklıysa MST **tektir**.



Kesimi geçen kenarlar: 9, 4, 8, 11. En hafifi r–s (4) MST'dedir.

MİNİMUM KAPSAYAN AĞAÇ

Kruskal, Prim ve Borůvka

	Kruskal	Prim	Borůvka
Açgözlü seçim	tüm çizgedeki, döngü kurmayan en hafif kenar	ağaçtan çıkan en hafif kenar	her bileşenden çıkan en hafif kenar
Büyüme	orman → ağaç	tek ağaç, düğüm düğüm	orman, turlarla birleşme
Kullandığı kesim	kenarın uçlarını ayıran iki bileşen	ağaç ile geri kalan düğümler	her bileşen ile geri kalanı
Veri yapısı	sıralama + Union-Find	öncelikli kuyruk	Union-Find
Zaman	O(E log V)	O(E log V) (ikili heap)	O(E log V)
Uygun olduğu	seyrek çizge, kenar listesi	yoğun çizge, komşuluk	paralel / dağıtık

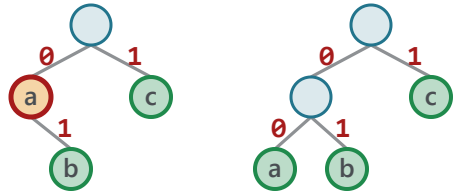
Üçü de aynı çizgede aynı MST'yi bulur (ağırlıklar farklıysa MST tektir); farkları kenarları ele alma sırasında ve gereken veri yapısındadır.

Huffman Kodlama

- **Kayıpsız veri sıkıştırma** için kullanılır (David Huffman, 1952).
- **Değişken uzunluklu** kodlar kullanır: **sık** kullanılan karakterlere **kısa**, seyrek olanlara uzun kodlar atanır.
- **Önek-özgür** (prefix-free) kodlar üretir: hiçbir kod, başka bir kodun öneki olamaz → ayırıcı gerekmeden tek anlamlı çözülür.
- Açgözlü adım: **en düşük frekanslı iki** ağaç birleştirilir.

Önek-özgür neden önemli?

Kodlar a = 0, b = 01, c = 1 olsun. "01" → "ab" mi, "b" mi? Belirsiz: 0, 01'in önekidir. Huffman kodları ikili ağacın **yapraklarında** durduğu için hiçbir kod diğerinin öneki olamaz.



✗ a, b'nin yolunda ✓ hepsi yaprak: 00, 01, 1

Kullanım: ZIP/DEFLATE, gzip, PNG, JPEG ve MP3'ün entropi kodlama aşaması.

Huffman Kodlama: Adımlar

1. **Adım 1:** Her karakterin **frekansı** hesaplanır.
2. **Adım 2:** Frekanslara göre bir **Huffman ağacı** oluşturulur: her karakter bir yaprak; en küçük iki frekanslı düğüm birleştirilerek yeni bir iç düğüm yapılır.
3. **Adım 3:** Ağaçta her karaktere karşılık gelen kodlar atanır: sol kenar 0, sağ kenar 1.
4. **Adım 4:** Veri, oluşturulan Huffman kodlarıyla kodlanır.

Örnek frekanslar (toplam 25 karakter):

karakter	F	O	R	G	E	T
frekans	2	3	4	4	5	7

n karakter için n − 1 birleştirme; öncelikli kuyruk ile **O(n log n)**.

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
▶    for (int k = 0; k < h.length; k++)
▶      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



Her karakter için frekansıyla bir yaprak oluşturulup öncelikli kuyruğa (min-heap) eklenir.

Adım 1 / 12

DEĞİŞKENLER

`kuyruk = [F2, O3, G4, R4, E5, T7]`

`sol = –` `sag = –`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
▶      Dugum sol = kuyruk.poll();
▶      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



`poll()` iki kez: en küçük frekanslar **F2** (sol) ve **O3** (sağ).

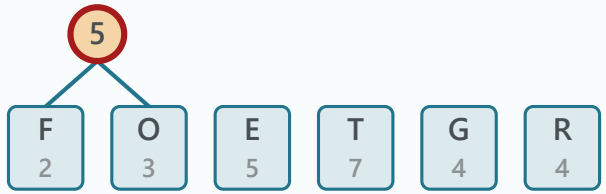
Adım 2 / 12

DEĞİŞKENLER

`kuyruk = [G4, R4, E5, T7]` `sol = F2` `sag = O3`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
▶      Dugum ata = new Dugum('\0', sol.f + sag.f);
▶      ata.sol = sol;
▶      ata.sag = sag;
▶      kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



Yeni iç düğüm **5 = 2 + 3**; çocukları F2 ve O3, kuyruğa geri eklenir.

Adım 3 / 12

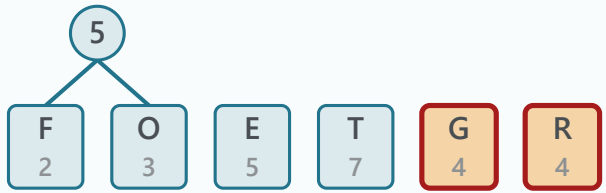
DEĞİŞKENLER

`kuyruk = [G4, R4, (5), E5, T7]`

`sol = F2` `sag = O3`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
▶      Dugum sol = kuyruk.poll();
▶      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



`poll()` iki kez: en küçük frekanslar **G4** (sol) ve **R4** (sağ). G4 ile R4 eşit; hangisinin önce çıkacağı heap'e bağlıdır.

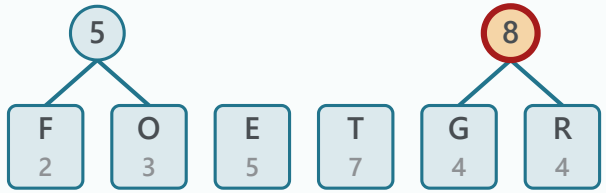
Adım 4 / 12

DEĞİŞKENLER

`kuyruk = [(5), E5, T7]` `sol = G4` `sag = R4`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
▶      Dugum ata = new Dugum('\0', sol.f + sag.f);
▶      ata.sol = sol;
▶      ata.sag = sag;
▶      kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



Yeni iç düğüm **8 = 4 + 4**; çocukları G4 ve R4, kuyruğa geri eklenir.

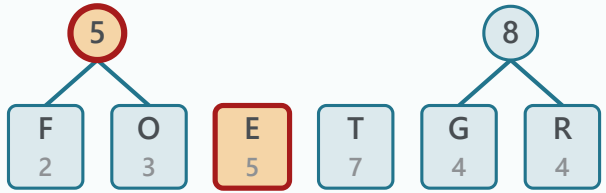
Adım 5 / 12

DEĞİŞKENLER

`kuyruk = [(5), E5, T7, (8)]` `sol = G4` `sag = R4`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
▶      Dugum sol = kuyruk.poll();
▶      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



`poll()` iki kez: en küçük frekanslar **(5)** (sol) ve **E5** (sağ). (5) ile E5 eşit; hangisinin önce çıkacağı heap'e bağlıdır, kod uzunlukları değişmez.

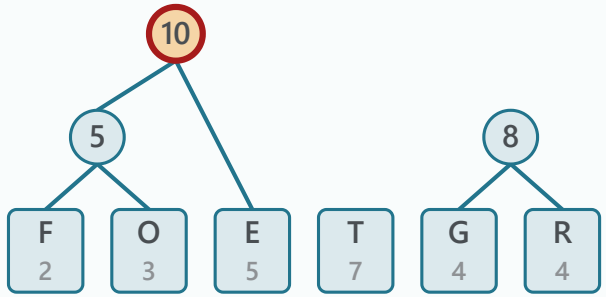
Adım 6 / 12

DEĞİŞKENLER

`kuyruk = [T7, (8)]` `sol = (5)` `sag = E5`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
▶      Dugum ata = new Dugum('\0', sol.f + sag.f);
▶      ata.sol = sol;
▶      ata.sag = sag;
▶      kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



Yeni iç düğüm **10 = 5 + 5**; çocukları (5) ve E5, kuyruğa geri eklenir.

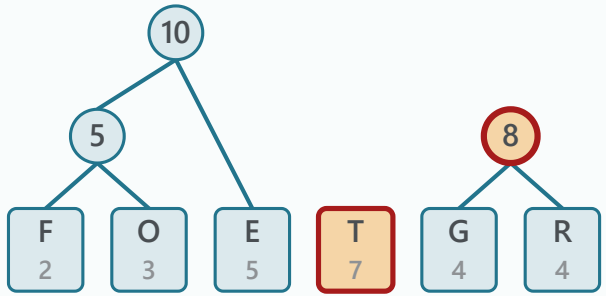
Adım 7 / 12

DEĞİŞKENLER

`kuyruk = [T7, (8), (10)]` `sol = (5)` `sag = E5`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
▶      Dugum sol = kuyruk.poll();
▶      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



`poll()` iki kez: en küçük frekanslar **T7** (sol) ve **(8)** (sağ).

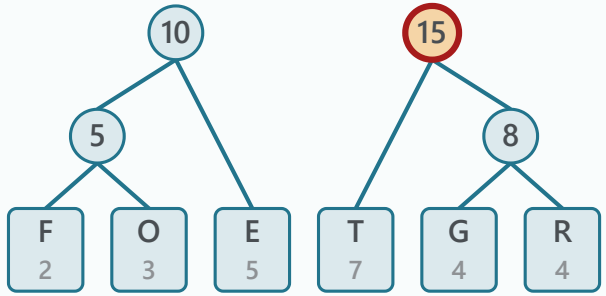
Adım 8 / 12

DEĞİŞKENLER

`kuyruk = [(10)]` `sol = T7` `sag = (8)`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
▶      Dugum ata = new Dugum('\0', sol.f + sag.f);
▶      ata.sol = sol;
▶      ata.sag = sag;
▶      kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



Yeni iç düğüm **15 = 7 + 8**; çocukları T7 ve (8), kuyruğa geri eklenir.

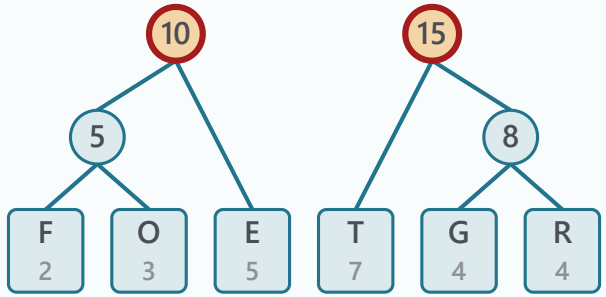
Adım 9 / 12

DEĞİŞKENLER

`kuyruk = [(10), (15)]` `sol = T7` `sag = (8)`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
▶      Dugum sol = kuyruk.poll();
▶      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



`poll()` iki kez: en küçük frekanslar **(10)** (sol) ve **(15)** (sağ).

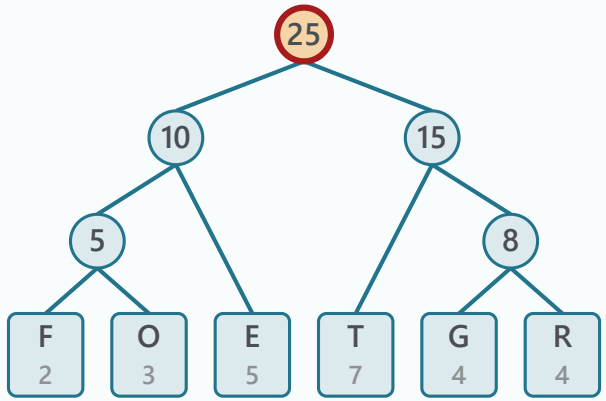
Adım 10 / 12

DEĞİŞKENLER

`kuyruk = []` `sol = (10)` `sag = (15)`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
▶      Dugum ata = new Dugum('\0', sol.f + sag.f);
▶      ata.sol = sol;
▶      ata.sag = sag;
▶      kuyruk.add(ata);
13   }
14   return kuyruk.poll();
15 }
```



Yeni iç düğüm **25 = 10 + 15**; çocukları (10) ve (15), kuyruğa geri eklenir. Kuyrukta tek düğüm kaldı: kök.

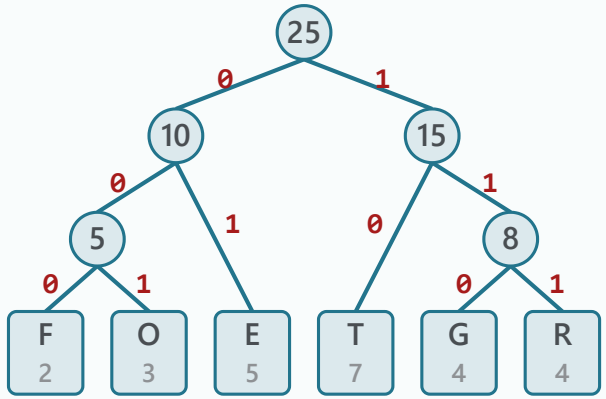
Adım 11 / 12

DEĞİŞKENLER

`kuyruk = [(25)]` `sol = (10)` `sag = (15)`

Huffman Ağacını Kur: Kodu Adım Adım İzle

```
1  static Dugum huffman(char[] h, int[] f) {
2    PriorityQueue<Dugum> kuyruk =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < h.length; k++)
5      kuyruk.add(new Dugum(h[k], f[k]));
6    while (kuyruk.size() > 1) {
7      Dugum sol = kuyruk.poll();
8      Dugum sag = kuyruk.poll();
9      Dugum ata = new Dugum('\0', sol.f + sag.f);
10     ata.sol = sol;
11     ata.sag = sag;
12     kuyruk.add(ata);
13   }
▶  return kuyruk.poll();
15 }
```



Döngü biter, kök (25) döndürülür. Sol kenar 0, sağ kenar 1: F=000, O=001, E=01, T=10, G=110, R=111.

Adım 12 / 12

DEĞİŞKENLER

`kuyruk = [(25)]` `sol = (10)` `sag = (15)`

Kodlar, Kodlama ve Çözme

kar.	frekans	kod	uzunluk	bit
F	2	`000`	3	6
O	3	`001`	3	9
E	5	`01`	2	10
T	7	`10`	2	14
G	4	`110`	3	12
R	4	`111`	3	12
toplam				**63**

- Sabit uzunluk: 6 karakter → 3 bit, 25 × 3 = **75 bit**. Huffman: **63 bit** (%16 kazanç, ortalama 2,52 bit).
- **Kodlama:** "FORTE" → 000 001 111 10 01 = 13 bit (sabit kodla 15).
- **Çözme:** kökten başlanır, 0 → sol, 1 → sağ; yaprağa gelince karakter yazılır ve köke dönülür: `1100110` → 110 | 01 | 10 → "GET".
- Sol ve sağ çocuğun yeri keyfîdir (ör. T sağ çocuk yapılabilir): kodlar değişir ama kod **uzunlukları** ve 63 bit değişmez.

Ağaçtan Kodları Üret: Kod İzleme

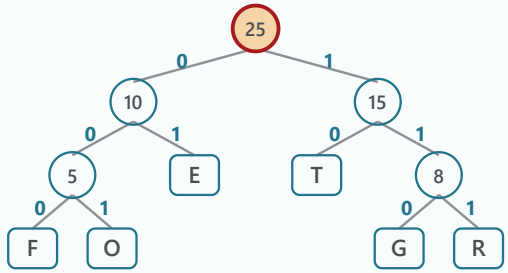
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
3     System.out.println(d.h + " = " + kod);
4     return;
5   }
▶   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

`kodUret((25), "")`

`main()`

ÇIKTI



Huffman ağacının kökü (25) ile `kodUret(kok, "")` çağrıldı: kök yaprak değil, önce sol çocuğa **"0"** eklenerek inilir.

Adım 1 / 11

DEĞİŞKENLER

`d = (25)` `kod = ""`

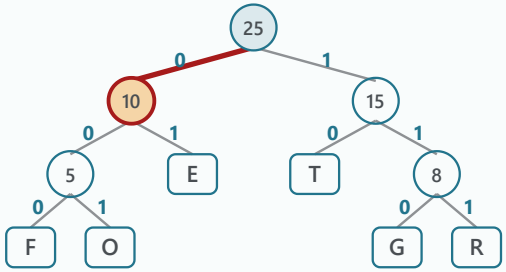
Ağaçtan Kodları Üret: Kod İzleme

```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
3     System.out.println(d.h + " = " + kod);
4     return;
5   }
▶   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- **kodUret((10), "0")**
- kodUret((25), "")
- main()

ÇIKTI



İç düğüm (10), kod = "0" (sol kenardan gelindi) → yaprak değil, sol çocuğa "00" ile inilir.

Adım 2 / 11

DEĞİŞKENLER

`d = (10)` `kod = "0"`

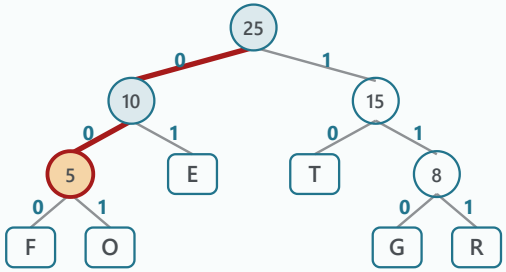
Ağaçtan Kodları Üret: Kod İzleme

```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
3     System.out.println(d.h + " = " + kod);
4     return;
5   }
▶   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- **kodUret((5), "00")**
- kodUret((10), "0")
- kodUret((25), "")
- main()

ÇIKTI



İç düğüm (5), kod = "00" (sol kenardan gelindi) → yaprak değil, sol çocuğa "000" ile inilir.

Adım 3 / 11

DEĞİŞKENLER

`d = (5)` `kod = "00"`

Ağaçtan Kodları Üret: Kod İzleme

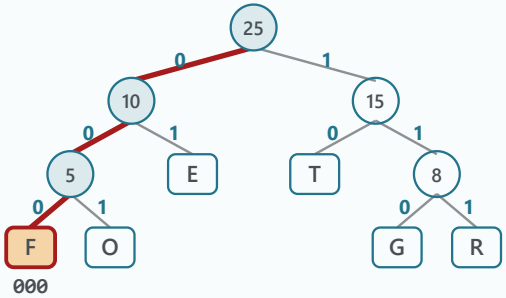
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
▶     System.out.println(d.h + " = " + kod);
4     return;
5   }
6   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- `kodUret(F, "000")`
- `kodUret((5), "00")`
- `kodUret((10), "0")`
- `kodUret((25), "")`
- `main()`

ÇIKTI

```
F = 000
```



Yaprak **F** (frekans 2): yoldaki bitler yazılır → **F = 000**.

Adım 4 / 11

DEĞİŞKENLER

`d = F` `kod = "000"`

Ağaçtan Kodları Üret: Kod İzleme

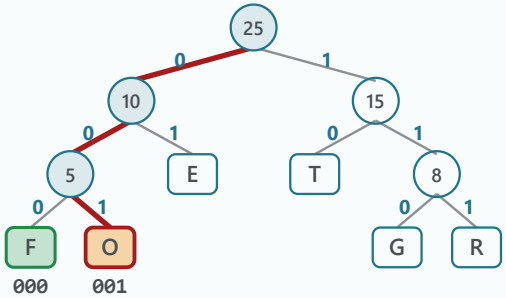
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
▶     System.out.println(d.h + " = " + kod);
4     return;
5   }
6   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- `kodUret(O, "001")`
- `kodUret((5), "00")`
- `kodUret((10), "0")`
- `kodUret((25), "")`
- `main()`

ÇIKTI

```
F = 000
O = 001
```



Yaprak **O** (frekans 3): yoldaki bitler yazılır → **O = 001**.

Adım 5 / 11

DEĞİŞKENLER

`d = O` `kod = "001"`

Ağaçtan Kodları Üret: Kod İzleme

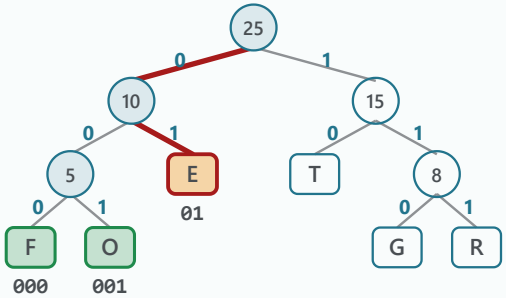
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
▶     System.out.println(d.h + " = " + kod);
4     return;
5   }
6   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- **kodUret(E, "01")**
- kodUret((10), "0")
- kodUret((25), "")
- main()

ÇIKTI

```
F = 000
O = 001
E = 01
```



Yaprak **E** (frekans 5): yoldaki bitler yazılır → **E = 01**.

DEĞİŞKENLER

`d = E` `kod = "01"`

Ağaçtan Kodları Üret: Kod İzleme

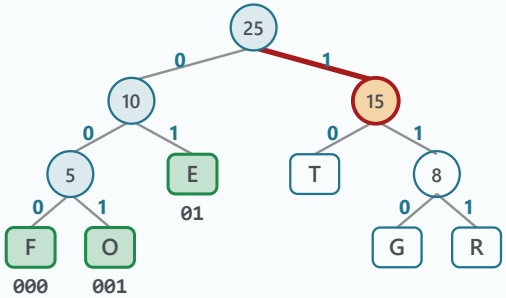
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
3     System.out.println(d.h + " = " + kod);
4     return;
5   }
▶   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- `kodUret((15), "1")`
- `kodUret((25), "")`
- `main()`

ÇIKTI

```
F = 000
O = 001
E = 01
```



İç düğüm (15), kod = "1" (sağ kenardan gelindi) → yaprak değil, sol çocuğa "10" ile inilir.

DEĞİŞKENLER

`d = (15)` `kod = "1"`

Ağaçtan Kodları Üret: Kod İzleme

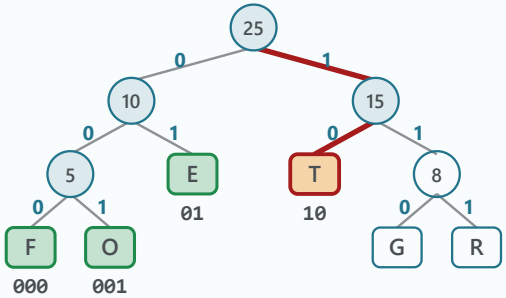
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
▶     System.out.println(d.h + " = " + kod);
4     return;
5   }
6   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- **kodUret(T, "10")**
- kodUret((15), "1")
- kodUret((25), "")
- main()

ÇIKTI

```
F = 000
O = 001
E = 01
T = 10
```



Yaprak **T** (frekans 7): yoldaki bitler yazılır → **T = 10**.

DEĞİŞKENLER

`d = T` `kod = "10"`

Ağaçtan Kodları Üret: Kod İzleme

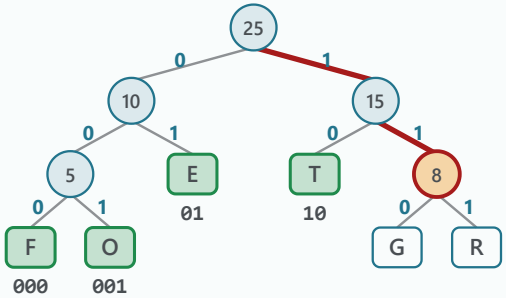
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
3     System.out.println(d.h + " = " + kod);
4     return;
5   }
▶   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- **kodUret((8), "11")**
- kodUret((15), "1")
- kodUret((25), "")
- main()

ÇIKTI

```
F = 000
O = 001
E = 01
T = 10
```



İç düğüm (8), kod = "11" (sağ kenardan gelindi) → yaprak değil, sol çocuğa "110" ile inilir.

Adım 9 / 11

DEĞİŞKENLER

`d = (8)` `kod = "11"`

Ağaçtan Kodları Üret: Kod İzleme

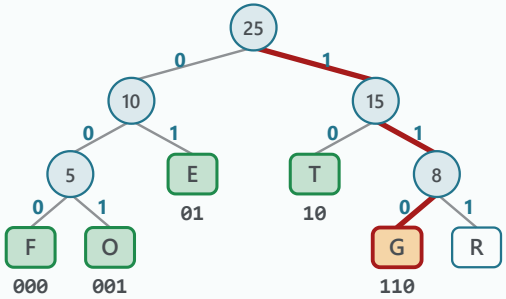
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
▶     System.out.println(d.h + " = " + kod);
4     return;
5   }
6   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- **kodUret(G, "110")**
- kodUret((8), "11")
- kodUret((15), "1")
- kodUret((25), "")
- main()

ÇIKTI

```
F = 000
O = 001
E = 01
T = 10
G = 110
```



Yaprak **G** (frekans 4): yoldaki bitler yazılır → **G = 110**.

Adım 10 / 11

DEĞİŞKENLER

`d = G` `kod = "110"`

Ağaçtan Kodları Üret: Kod İzleme

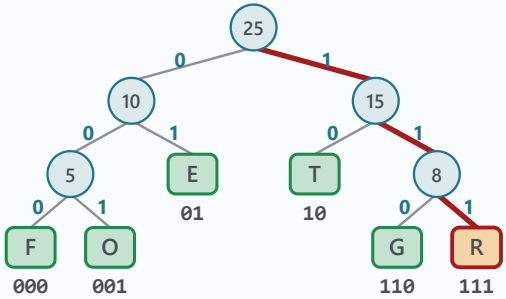
```
1 static void kodUret(Dugum d, String kod) {
▶   if (d.sol == null) {      // yaprak
▶     System.out.println(d.h + " = " + kod);
4     return;
5   }
6   kodUret(d.sol, kod + "0"); // sol: 0
7   kodUret(d.sag, kod + "1"); // sağ: 1
8 }
```

ÇAĞRI YIĞINI

- `kodUret(R, "111")`
- `kodUret((8), "11")`
- `kodUret((15), "1")`
- `kodUret((25), "")`
- `main()`

ÇIKTI

```
F = 000
O = 001
E = 01
T = 10
G = 110
R = 111
```



Yaprak **R** (frekans 4): yoldaki bitler yazılır → **R = 111**. Tüm yapraklar yazıldı: sık karakterler (E, T) 2 bit, seyrekler 3 bit. Ağaç bir kez gezilir: **O(n)**.

Adım 11 / 11

DEĞİŞKENLER

`d = R` `kod = "111"`

Huffman Neden En İyi?

Açgözlü seçim

En düşük frekanslı iki karakter x ve y'nin, en derin düzeyde **kardeş** olduğu bir en iyi kod ağacı vardır: daha derindeki bir karakterle yer değiştirmek toplam biti artırmaz (değiştirme argümanı).

Optimal alt yapı

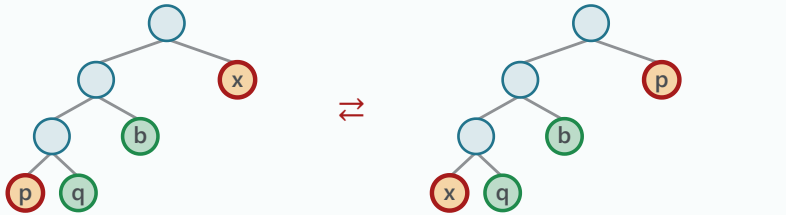
x ve y, frekansı f(x) + f(y) olan tek bir "z" karakteri yapılırsa, küçülen problemin en iyi ağacından z açılarak asıl problemin en iyi ağacı elde edilir.

Maliyet

Toplam bit = Σ frekans(c) · derinlik(c). Huffman, önek-özgür kodlar arasında bunu **en küçük** yapar.

Karmaşıklık

Frekans sayımı O(m) (m = metin uzunluğu); ağaç O(n log n); kodlama O(m). Ağaç (ya da kod tablosu) da çözücüye gönderilmelidir.



x en az frekanslı ama sığ

x en derine: toplam bit artmaz

UYGULAMALAR

Diğer Açgözlü Algoritmalar

Dijkstra ve Prim

Her adımda uzaklığı / bağlantı maliyeti en küçük düğümü kalıcı olarak seçer. Dijkstra negatif kenarsız çizgede en kısa yolları bulur; Prim her ağırlıkta MST'yi bulur.

Çizge boyama

Düğümler sırayla, komşularının kullanmadığı en küçük renkle boyanır. Hızlı ama renk sayısı en az olmayabilir (NP-zor).

Kutulara yerleştirme

(Bin packing) İlk sığan: her nesne sığdığı ilk kutuya konur (en iyi sığan: en az boşluk bırakan kutuya). Azalan sırayla ilk sığan en fazla ≈ 11/9 · OPT + 1 kutu kullanır.

En büyük kapsama

(Maximum coverage) Her adımda en çok yeni öğe kapsayan küme seçilir: en iyinin en az (1 − 1/e) ≈ %63'ünü garanti eder.

NP-zor problemlerde açgözlü algoritmalar, kanıtlanmış oranlı **yaklaşım algoritmaları** olarak değerlidir.

ÖZET

Özet

Problem	Açgözlü ölçüt	Zaman	En iyi mi?
Para üstü	sığan en büyük para	O(k + adet)	kanonikse
Etkinlik seçimi	en erken biten	O(n log n)	evet
İş sıralama	en kârlı iş, en geç boş slot	$O(n^2)$	evet
Kesirli sırt çantası	en büyük kâr/ağırlık	O(n log n)	evet (0/1: hayır)
MST	kesimi geçen en hafif kenar (Kruskal, Prim, Borůvka)	O(E log V)	evet
Huffman	en düşük frekanslı iki ağaç	O(n log n)	evet

ÖZET

Etkileşimli Simülatörler

Minimum Kapsayan Ağaç

Rastgele bağlı ağırlıklı çizgelerde Prim ve Kruskal adım adım, yan yana: seçilen kenarlar ve toplam ağırlık.

Dizgi Sıkıştırma

RLE (Run Length Encoding) ve LZW sıkıştırması adım adım, sıkıştırma oranıyla. Huffman bu sayfada yok; sözlük tabanlı LZW ile frekans tabanlı Huffman'ı karşılaştırmaya elverişlidir.

ÖZET

Sık Yapılan Hatalar

Açgözlüyü kanıtsız doğru saymak

{1, 3, 4} paralarıyla 6: açgözlü 3 para, en iyi 2. Her problemde açgözlü seçim özelliği gösterilmelidir.

Yanlış ölçüt

Etkinlik seçiminde en erken **başlayan** ya da en **kısa** etkinlik ölçütü yanlıştır; doğru ölçüt en erken **biten**.

0/1 sırt çantasında oran

Kesirli problemde en iyi olan oran ölçütü, bölünemeyen nesnelerde en iyi sonucu garanti etmez.

En erken boş slot

İş sıralamada işi son tarihinden önceki en **erken** boş slota koymak, son tarihi yakın işlere yer bırakmaz; en **geç** boş slot seçilir.

Kesim özelliğini yanlış okumak

Güvenli olan, bir kesimi **geçen** en hafif kenardır. Tüm çizgedeki en hafif kenarlar sırayla alınırsa döngü oluşabilir; Kruskal bu yüzden döngü kuranı atar.

Huffman'da ağaçsız çözme

Kod tablosu ya da ağaç gönderilmezse veri çözülemez. Tek karakterli metinde kök yapraktır: koda en az 1 bit verilmelidir.

ÖZET

Örnek Problemler

1. {1, 5, 10, 25} paralarıyla 63 açgözlü olarak kaç parayla ödenir? {1, 7, 10} ile 14 için açgözlü ve en iyi sonuç nedir?
2. Etkinlikler (s, f): (1,4), (3,5), (0,6), (5,7), (3,9), (5,9), (6,10), (8,11), (8,12), (2,14), (12,16). En fazla kaç etkinlik seçilir?
3. Frekanslar a:5, b:9, c:12, d:13, e:16, f:45. Huffman kod uzunluklarını ve toplam biti bulun.
4. Düğümler {A, B} ve {C, D} olarak ayrılıyor; kesimi geçen kenarlar A–C 5, B–C 3, B–D 4. Hangi kenar kesinlikle her MST'dedir?
5. Kruskal, Prim ve Borůvka aynı çizgede farklı kapsayan ağaçlar bulabilir mi?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. İşler (id, son tarih, kâr): (a,2,100), (b,1,19), (c,2,27), (d,1,25), (e,3,15). Açgözlü iş sıralamasını uygulayın.
2. W = 10, nesneler (kâr, ağırlık): (10,5), (40,4), (30,6), (50,3). Kesirli sırt çantasının değeri nedir?
3. "ABRACADABRA" için Huffman kodlaması kaç bit tutar? Sabit uzunluklu kodla karşılaştırın.
4. Kruskal'ın tüm kenarlar aynı ağırlıkta olduğunda ne ürettiğini ve O(E log V) maliyetinin hangi adımdan geldiğini açıklayın.
5. Açgözlü para üstünün en iyi sonuç vermediği, en küçük tutarlı bir para sistemi bulun ve gerekçelendirin.

▶ **Cevaplar / ipuçları**

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022 (Bölüm 15: Greedy Algorithms; Bölüm 21: MST).
- Kleinberg ve Tardos, *Algorithm Design*, 2006 (Bölüm 4: Greedy Algorithms).
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, 2011 (4.3 MST, 5.5 Data Compression).
- D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE* 40(9), 1952.
- J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. AMS* 7(1), 1956.

Son

Bölüm 2: Açgözlü Algoritmalar

Sonraki bölüm: Bölüm 3: Dinamik Programlama · Terim sözlüğü