# Bölüm 19: Temel Sıralama Algoritmaları

## C ile Veri Yapıları ve Algoritmalar

Kararlılık · Seçimli · Eklemeli · Ortalama durum · Kabarcık · Sayarak · Shell · Kova · Alt sınır

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

# Öğrenme Çıktıları

**Kavramlar**

Tutanak, anahtar, iç/dış sıralama, **kararlılık** ve **yerinde** sıralamayı ayırt etmek.

**Üç temel algoritma**

Seçimli, eklemeli ve kabarcık sıralamasını C'de yazıp adım adım izlemek.

**Analiz**

Karşılaştırma/değiştirme sayısını; en iyi, en kötü ve **ortalama durumu** hesaplamak.

**Karşılaştırmasız sıralama**

Sayarak (counting) ve kova (bucket) sıralamanın ne zaman O(n) olduğunu açıklamak.

**Shell sıralama**

Aralıklı eklemeli sıralamanın neden eklemeliden hızlı olduğunu göstermek.

**Alt sınır**

Karar ağacıyla karşılaştırmalı sıralamanın Ω(n lg n) sınırını kanıtlamak.

# Sıralama: Tutanak, Anahtar, İç/Dış

- Alanların (*fields*) oluşturduğu **tutanaklar** (*records*) bellekte tutulunca **liste**, bellek dışında tutulunca **kütük** (*file*) adını alır.
- Tutanağı **biricik** (*unique*) olarak tanıtan alan(lar) **anahtardır** (*key*); sıralama anahtara göre yapılır.
- **İç sıralama** (*internal*): verinin tamamı bellekte. **Dış sıralama** (*external*): disk/teyp gibi yardımcı bellek kullanılır; disk erişim sayısı belirleyicidir.

tutanak (record) = alanlar (fields)

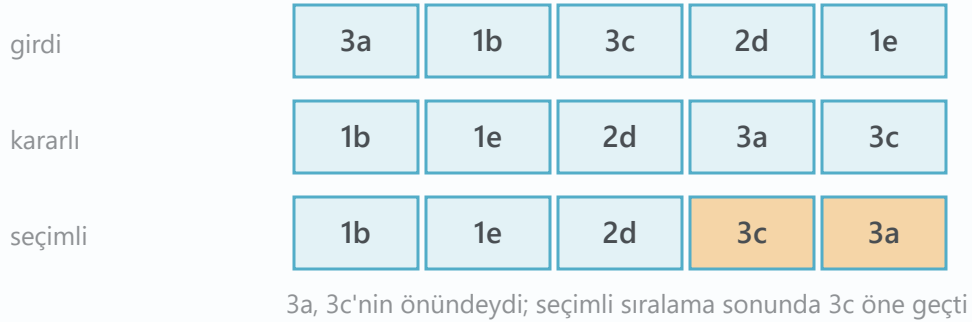
| no (anahtar) | ad | soyad | ort. |
|---|---|---|---|
| 1042 | Ayşe | Yılmaz | 3.1 |
| 0977 | Can | Demir | 2.8 |
| 1310 | Ece | Kaya | 3.6 |

bellekte: liste · bellek dışında: kütük (file)

Anahtar: öğrenci no

**HOCADAN** "Biz, iç sıralamayla ilgileneceğiz. Bu arada dış sıralama için de bir temel vermiş olacağız."

# Kararlılık (Stability) ve Yerinde (In-place) Sıralama




3a, 3c'nin önündeydi; seçimli sıralama sonunda 3c öne geçti

**Kararlı (stable)**

Eşit anahtarlı tutanakların **göreli sırası** korunur. Önce ada, sonra soyada göre kararlı sıralanırsa soyadı aynı olanlar ad sırasında kalır (basamak sıralamanın temeli).

**Yerinde (in-place)**

Girdinin dışında yalnızca **O(1)** (bazı tanımlarda O(lg n)) ek bellek kullanır. Seçimli, eklemeli, kabarcık yerindedir; sayarak sıralama değildir (c ve b dizileri).

# Ortak Altyapı: `kucuk` ve `SWAP`

```
#define N 15
#define kucuk(A, B) ((A) < (B))
#define SWAP(A, B) do { int t_ = (A); (A) = (B); (B) = t_; } while (0)
```

```
int main(void) {
    int *a = malloc(N * sizeof *a);
    if (a == NULL) return 1;
    for (int i = 0; i < N; i++)
        a[i] = rand() % 50;
    selection(a, 0, N - 1);   /* sıralama yordamına çağrı */
    for (int i = 0; i < N; i++) printf("%d ", a[i]);
    free(a);
    return 0;
}
```

### Makro tuzağı

Makroyu `{int t = A; A=B; B=t;}` biçiminde yazmak `if (…) SWAP(…); else` içinde derleme hatası verir; `do { … } while (0)` bu tuzağı kapatır. `if (kucuk(A, B)) SWAP(A, B);` kalıbı **A < B ise** takas eder (karşılaştır-takas).

# Seçimli Sıralama (Selection Sort)

```
void selection(int a[], int L, int R) {
    for (int i = L; i < R; i++) {
        int min = i;
        for (int j = i + 1; j <= R; j++)
            if (kucuk(a[j], a[min]))
                min = j;
        SWAP(a[i], a[min]);
    }
}
```

1. Listedeki en küçük değerli öğe bulunur.
2. Bu öğe ilk konumdaki öğeyle yer değiştirir.
3. Aynı adımlar 2. öğeden başlayarak kalan alt liste için yinelenir.

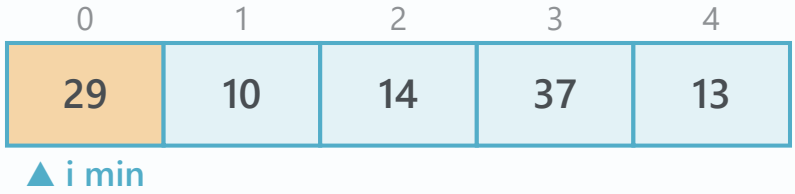
### Değişmez

Her dış tur sonunda `a[L..i]` sıralı ve dizinin en küçük öğelerini içerir.

### Neden sona kadar?

En küçüğü seçmek için kalan alt liste **kesinlikle** sonuna kadar taranmalıdır; veri sıralı olsa bile.

# Kod İzleme: `selection(a, 0, 4)`




Tur i = 0: kalan alt liste a[0..4]. Başlangıçta en küçüğün indisi **min = 0** (a[0] = 29).

Adım 1 / 18

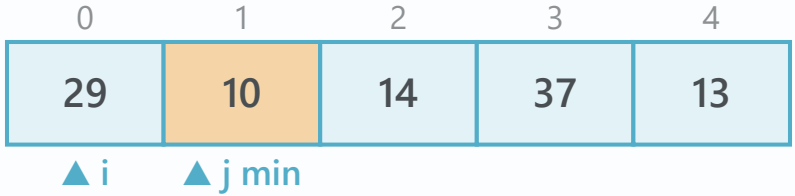
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 0` `min = 0`

`a = [29, 10, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[1] = 10 < a[0] = 29 → yeni en küçük: **min = 1**.

Adım 2 / 18

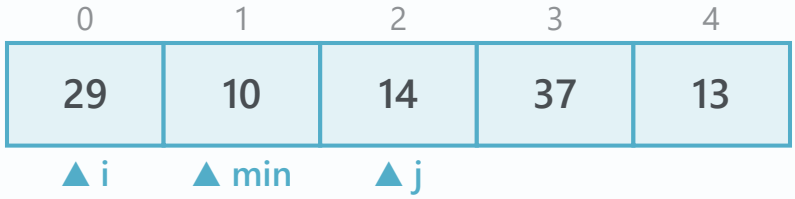
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
▶                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 0` `j = 1` `min = 1`

`a = [29, 10, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[2] = 14 ≥ a[1] = 10 → min değişmez.

Adım 3 / 18

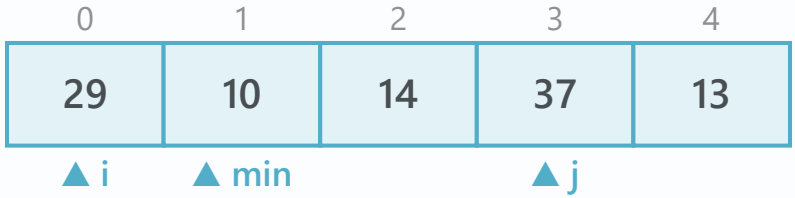
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 0` `j = 2` `min = 1`

`a = [29, 10, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[3] = 37 ≥ a[1] = 10 → min değişmez.

Adım 4 / 18

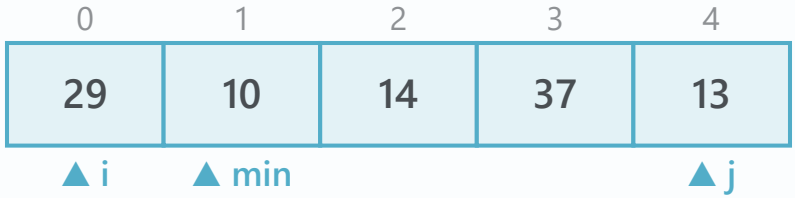
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 0` `j = 3` `min = 1`

`a = [29, 10, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[4] = 13 ≥ a[1] = 10 → min değişmez.

Adım 5 / 18

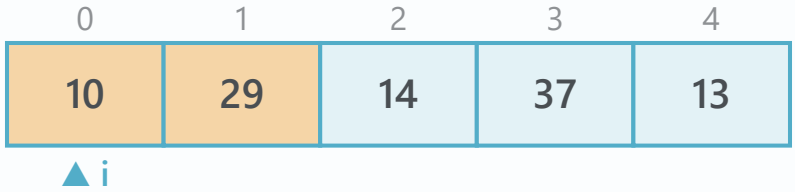
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 0` `j = 4` `min = 1`

`a = [29, 10, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




SWAP(a[0], a[1]): 10 yerine oturdu; a[0..0] artık sıralı.

Adım 6 / 18

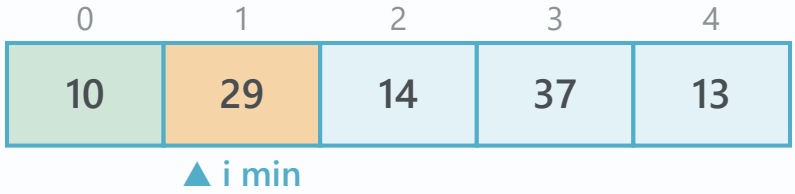
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 0` `min = 1`

`a = [10, 29, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




Tur i = 1: kalan alt liste a[1..4]. Başlangıçta en küçüğün indisi **min = 1** (a[1] = 29).

Adım 7 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 1` `min = 1`

`a = [10, 29, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[2] = 14 < a[1] = 29 → yeni en küçük: **min = 2**.

Adım 8 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
▶                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 1` `j = 2` `min = 2`

`a = [10, 29, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[3] = 37 ≥ a[2] = 14 → min değişmez.

Adım 9 / 18

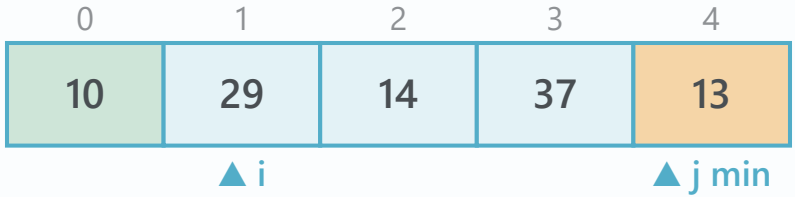
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 1` `j = 3` `min = 2`

`a = [10, 29, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




a[4] = 13 < a[2] = 14 → yeni en küçük: **min = 4**.

Adım 10 / 18

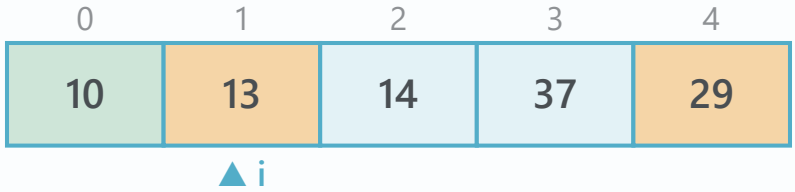
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
▶                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 1` `j = 4` `min = 4`

`a = [10, 29, 14, 37, 13]`

# Kod İzleme: `selection(a, 0, 4)`




SWAP(a[1], a[4]): 13 yerine oturdu; a[0..1] artık sıralı.

Adım 11 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 1` `min = 4`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




Tur i = 2: kalan alt liste a[2..4]. Başlangıçta en küçüğün indisi **min = 2** (a[2] = 14).

Adım 12 / 18

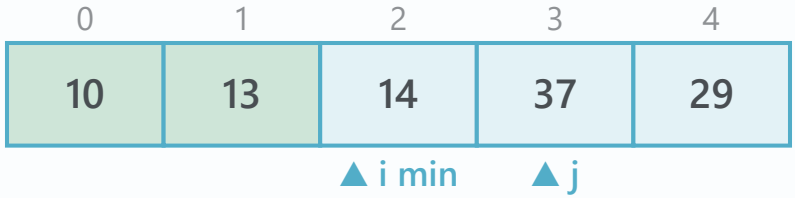
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 2` `min = 2`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




a[3] = 37 ≥ a[2] = 14 → min değişmez.

Adım 13 / 18

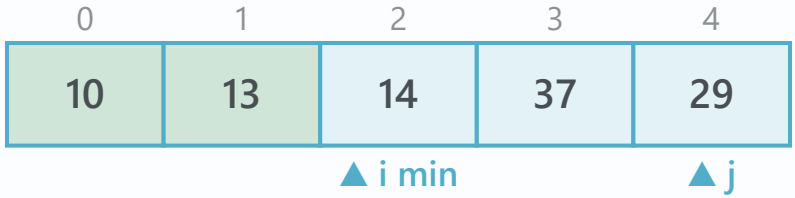
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 2` `j = 3` `min = 2`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




a[4] = 29 ≥ a[2] = 14 → min değişmez.

Adım 14 / 18

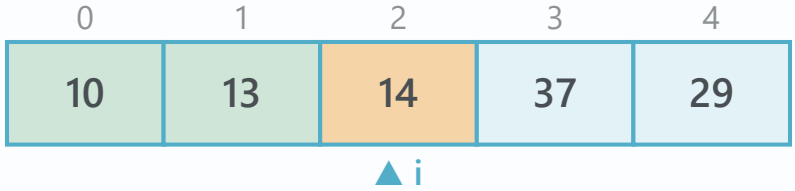
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 2` `j = 4` `min = 2`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




min = i olduğundan takas kendisiyle yapılır (değişiklik yok); a[0..2] sıralı.

Adım 15 / 18

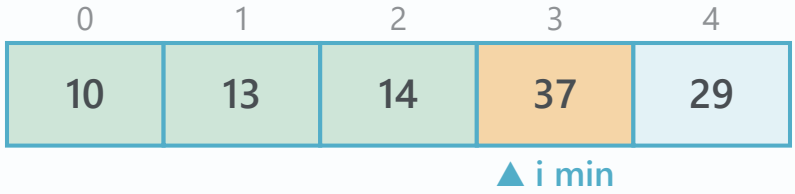
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 2` `min = 2`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




Tur i = 3: kalan alt liste a[3..4]. Başlangıçta en küçüğün indisi **min = 3** (a[3] = 37).

Adım 16 / 18

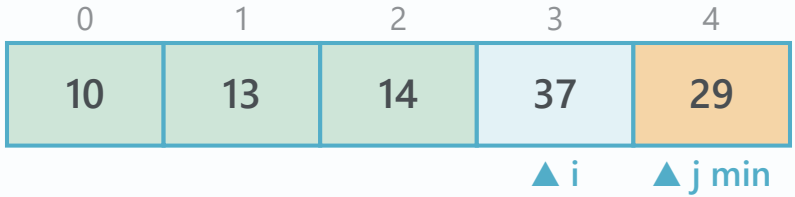
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 3` `min = 3`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




a[4] = 29 < a[3] = 37 → yeni en küçük: **min = 4**.

Adım 17 / 18

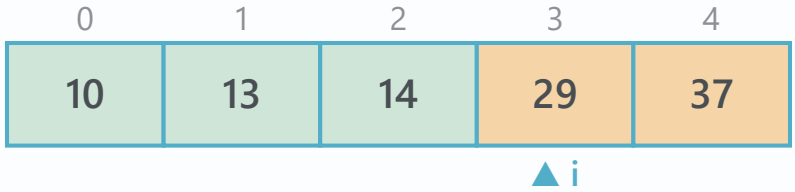
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (kucuk(a[j], a[min]))
▶                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 3` `j = 4` `min = 4`

`a = [10, 13, 14, 37, 29]`

# Kod İzleme: `selection(a, 0, 4)`




SWAP(a[3], a[4]): 29 yerine oturdu; a[0..3] artık sıralı.

Adım 18 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (kucuk(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

**DEĞİŞKENLER**

`i = 3` `min = 4`

`a = [10, 13, 14, 29, 37]`

# Seçimli Sıralamanın Analizi

$$(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 \approx n^2/2$$

- Karşılaştırma sayısı girdiden **bağımsızdır**: en iyi = ortalama = en kötü ≈ $n^2/2$.
- Değiştirme (takas) sayısı **n − 1**: son öğeye gelindiğinde o zaten doğru yerdedir.
- Karmaşıklık **$O(n^2)$**; ek bellek O(1) (yerinde).
- Uzak takaslar yüzünden **kararlı değildir** (önceki 3a/3c örneği).

| n | karşılaştırma | takas |
|---|---|---|
| 5 | 10 | 4 |
| 10 | 45 | 9 |
| 100 | 4950 | 99 |
| 1000 | 499500 | 999 |

Takas pahalıysa (büyük tutanaklar, flash bellek yazımı) az takas yapması seçimliyi değerli kılar.

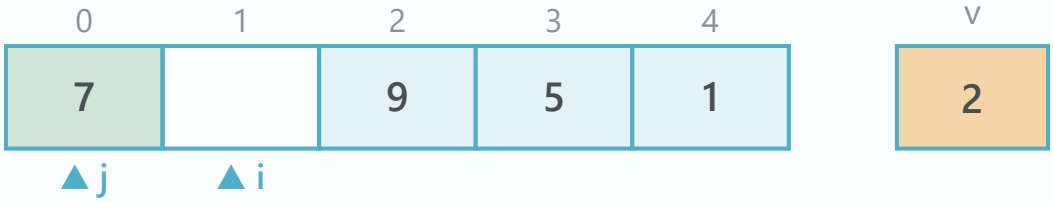
# Eklemeli Sıralama (Insertion Sort): İki Sürüm

```
void insertion(int a[], int L, int R) {
    for (int i = L + 1; i <= R; i++) {
        int v = a[i];      // yerleştirilecek
        int j = i - 1;     // a[L..i-1] sıralı
        while (j >= L && a[j] > v) {
            a[j + 1] = a[j]; // sağa kaydır
            j--;
        }
        a[j + 1] = v;
    }
}
```

```
void insertion1(int a[], int L, int R) {
    for (int i = R; i > L; i--) // bekçi
        if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
    for (int i = L + 2; i <= R; i++) {
        int v = a[i], j = i;
        while (kucuk(v, a[j - 1])) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = v;
    }
}
```

- Her turda girdiden bir öğe alınır ve **sıralı ön ek** içinde uygun konuma eklenir (iskambil kâğıdı dizer gibi).
- Sağdaki `insertion1`, soldakinin **ilerlemiş hâlidir**: önce en küçüğü `a[L]`'ye taşır; bu **bekçi** (*sentinel*) sayesinde iç döngüde `j > L` sınaması gerekmez.
- Koşulu `j ≥ 0` yazmak yanlıştır; alt dizi `L > 0`'dan başlarsa sınırı aşar, doğrusu `j ≥ L`.

# Kod İzleme: `insertion(a, 0, 4)`




i = 1: **v = a[1] = 2** bir kenara alındı; a[0..0] sıralı, v için yer aranacak.

Adım 1 / 15

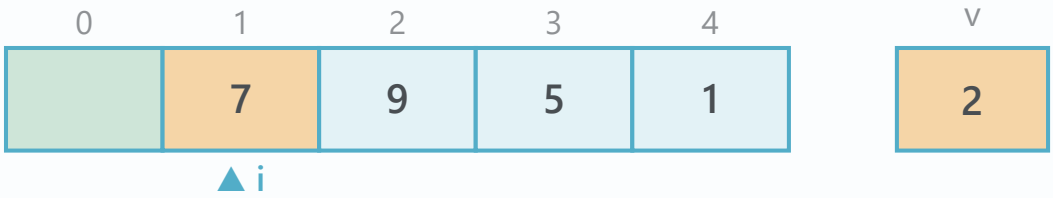
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];      // yerleştirilecek
▶          int j = i - 1;     // a[L..i-1] sıralı
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `v = 2` `j = 0`

`a = [7, 2, 9, 5, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




a[0] = 7 > v = 2 → 7 bir sağa kaydı (a[1]); j = -1.

Adım 2 / 15

```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `v = 2` `j = -1`

`a = [7, 7, 9, 5, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




j = −1, dizinin başına gelindi → döngü biter; **a[0] = 2**. Sıralı ön ek artık a[0..1].

Adım 3 / 15

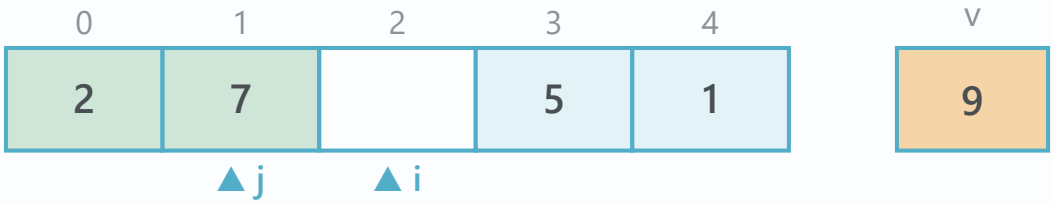
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `v = 2` `j = -1`

`a = [2, 7, 9, 5, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




i = 2: **v = a[2] = 9** bir kenara alındı; a[0..1] sıralı, v için yer aranacak.

Adım 4 / 15

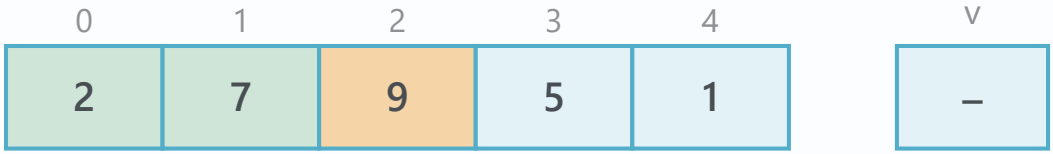
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];      // yerleştirilecek
▶          int j = i - 1;     // a[L..i-1] sıralı
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 2` `v = 9` `j = 1`

`a = [2, 7, 9, 5, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




a[1] = 7 ≤ v → döngü biter; **a[2] = 9**. Sıralı ön ek artık a[0..2].

Adım 5 / 15

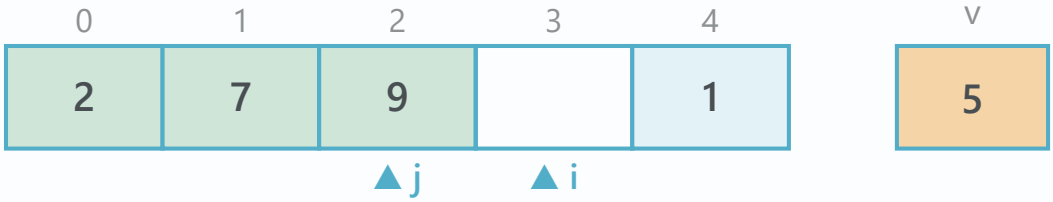
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 2` `v = 9` `j = 1`

`a = [2, 7, 9, 5, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




i = 3: **v = a[3] = 5** bir kenara alındı; a[0..2] sıralı, v için yer aranacak.

Adım 6 / 15

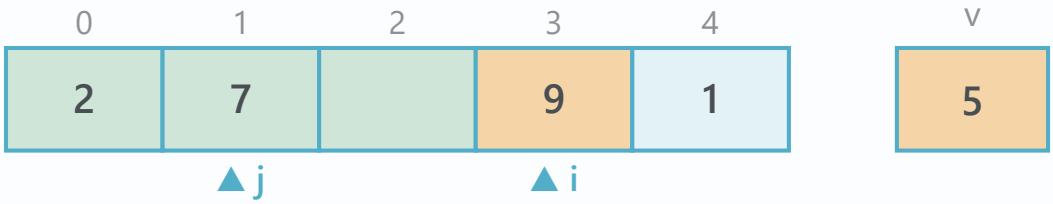
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];      // yerleştirilecek
▶          int j = i - 1;     // a[L..i-1] sıralı
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 3` `v = 5` `j = 2`

`a = [2, 7, 9, 5, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




a[2] = 9 > v = 5 → 9 bir sağa kaydı (a[3]); j = 1.

Adım 7 / 15

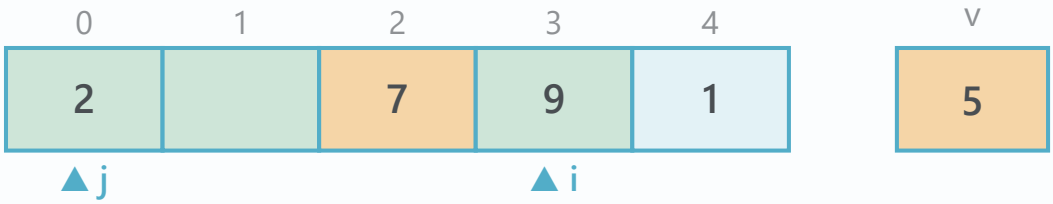
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 3` `v = 5` `j = 1`

`a = [2, 7, 9, 9, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




a[1] = 7 > v = 5 → 7 bir sağa kaydı (a[2]); j = 0.

Adım 8 / 15

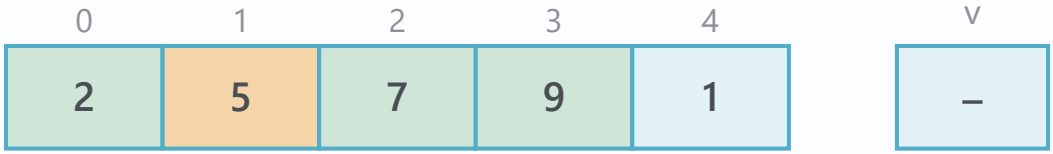
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 3` `v = 5` `j = 0`

`a = [2, 7, 7, 9, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




a[0] = 2 ≤ v → döngü biter; **a[1] = 5**. Sıralı ön ek artık a[0..3].

Adım 9 / 15

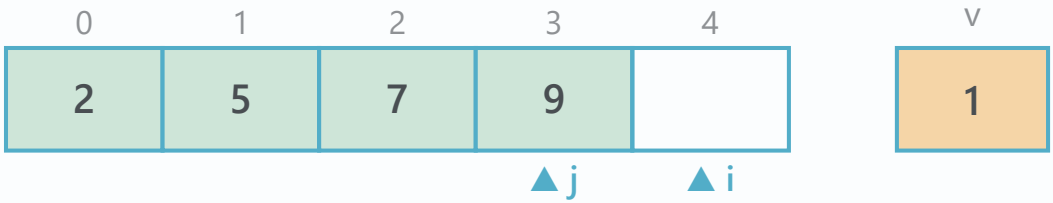
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 3` `v = 5` `j = 0`

`a = [2, 5, 7, 9, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




i = 4: **v = a[4] = 1** bir kenara alındı; a[0..3] sıralı, v için yer aranacak.

Adım 10 / 15

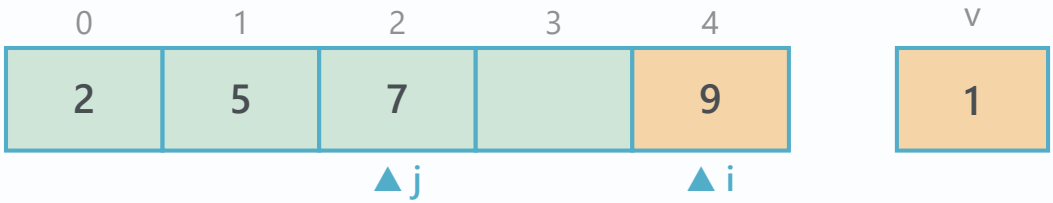
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];      // yerleştirilecek
▶          int j = i - 1;     // a[L..i-1] sıralı
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 4` `v = 1` `j = 3`

`a = [2, 5, 7, 9, 1]`

# Kod İzleme: `insertion(a, 0, 4)`




a[3] = 9 > v = 1 → 9 bir sağa kaydı (a[4]); j = 2.

Adım 11 / 15

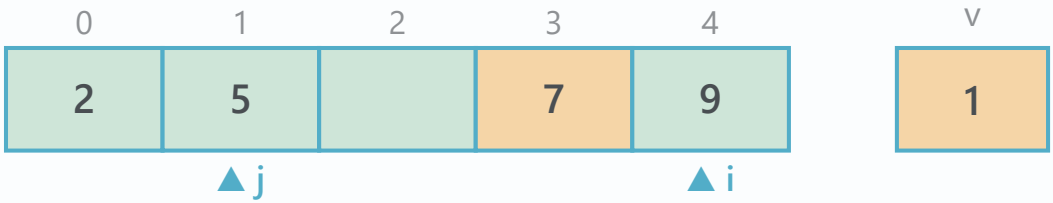
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // yerleştirilecek
4          int j = i - 1;      // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 4` `v = 1` `j = 2`

`a = [2, 5, 7, 9, 9]`

# Kod İzleme: `insertion(a, 0, 4)`




a[2] = 7 > v = 1 → 7 bir sağa kaydı (a[3]); j = 1.

Adım 12 / 15

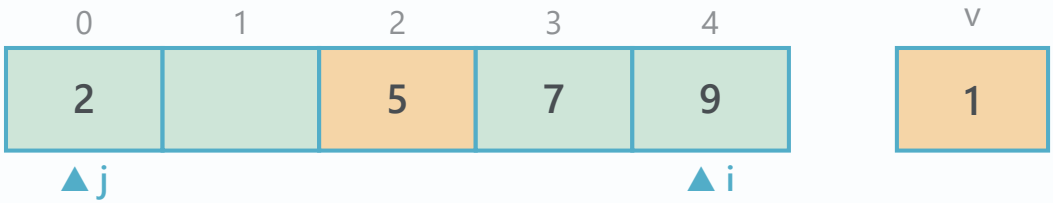
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 4` `v = 1` `j = 1`

`a = [2, 5, 7, 7, 9]`

# Kod İzleme: `insertion(a, 0, 4)`




a[1] = 5 > v = 1 → 5 bir sağa kaydı (a[2]); j = 0.

Adım 13 / 15

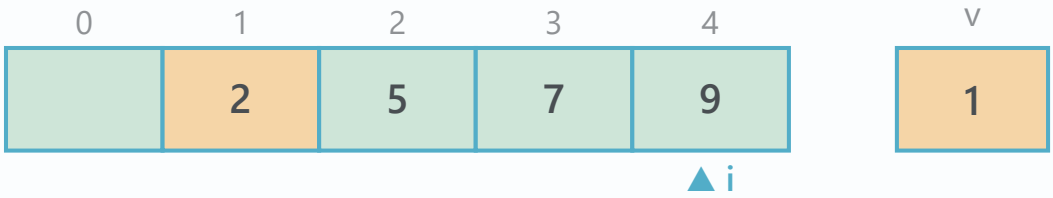
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 4` `v = 1` `j = 0`

`a = [2, 5, 5, 7, 9]`

# Kod İzleme: `insertion(a, 0, 4)`




a[0] = 2 > v = 1 → 2 bir sağa kaydı (a[1]); j = -1.

Adım 14 / 15

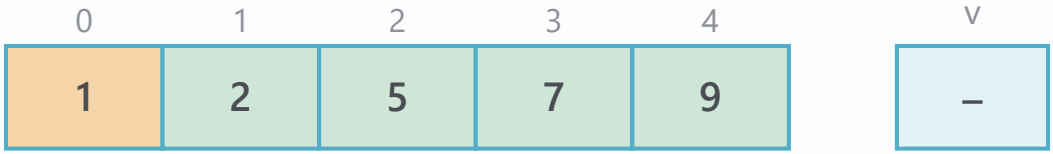
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // sağa kaydır
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 4` `v = 1` `j = -1`

`a = [2, 2, 5, 7, 9]`

# Kod İzleme: `insertion(a, 0, 4)`




j = −1, dizinin başına gelindi → döngü biter; **a[0] = 1**. Sıralı ön ek artık a[0..4].

Adım 15 / 15

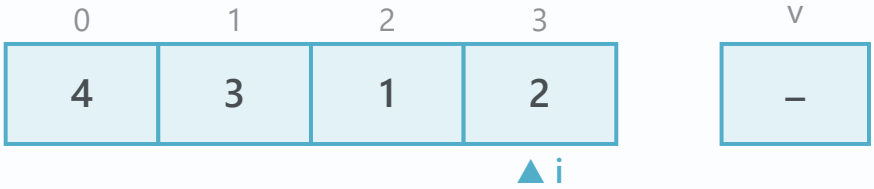
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];      // yerleştirilecek
4          int j = i - 1;     // a[L..i-1] sıralı
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // sağa kaydır
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

**DEĞİŞKENLER**

`i = 4` `v = 1` `j = -1`

`a = [1, 2, 5, 7, 9]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




a[3] = 2 ≥ a[2] = 1 → takas yok. En küçük öğe sola doğru ilerliyor.

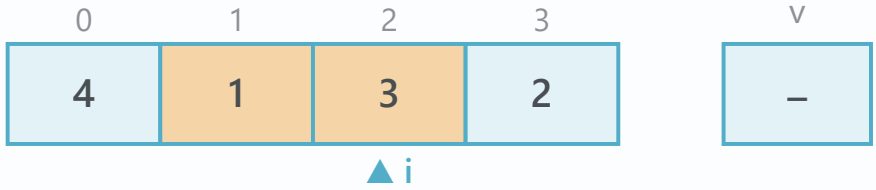
Adım 1 / 10

```
1  void insertion1(int a[], int L, int R) {
▶      for (int i = R; i > L; i--) // bekçi
▶          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
6          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

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

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




a[2] = 1 < a[1] = 3 → takas. En küçük öğe sola doğru ilerliyor.

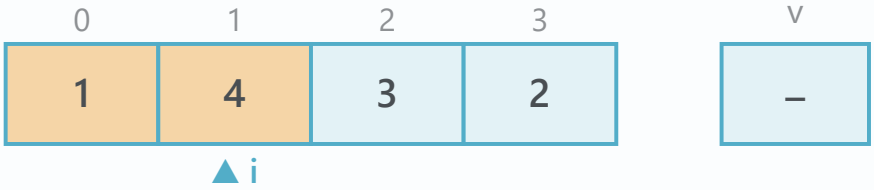
Adım 2 / 10

```
1  void insertion1(int a[], int L, int R) {
▶      for (int i = R; i > L; i--) // bekçi
▶          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
6          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

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

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




a[1] = 1 < a[0] = 4 → takas. **a[0] = 1** artık en küçük: bekçi hazır.

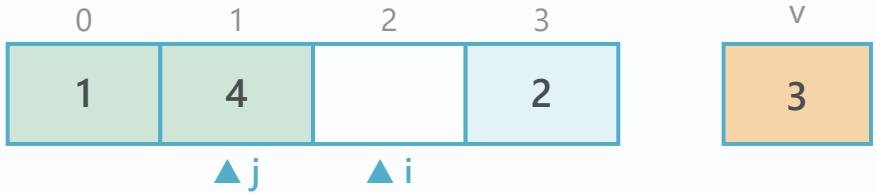
Adım 3 / 10

```
1  void insertion1(int a[], int L, int R) {
▶      for (int i = R; i > L; i--) // bekçi
▶          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
6          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

DEĞİŞKENLER

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

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




i = 2: v = 3. a[0..1] sıralı.

Adım 4 / 10

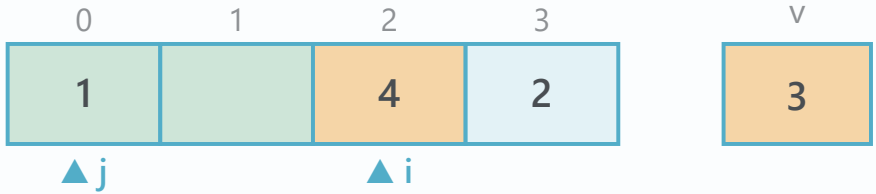
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
▶          int v = a[i], j = i;
6          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 2` `v = 3` `j = 2`

`a = [1, 4, 3, 2]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




v = 3 < a[1] = 4 → kaydır; j = 1. Sınır sınaması yok: bekçi a[0] durduracak.

Adım 5 / 10

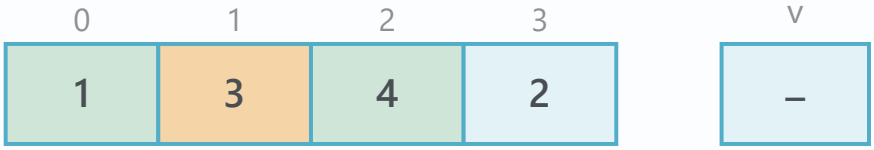
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (kucuk(v, a[j - 1])) {
▶              a[j] = a[j - 1];
▶              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 2` `v = 3` `j = 1`

`a = [1, 4, 4, 2]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




a[0] = 1 ≤ v → dur; **a[1] = 3**.

Adım 6 / 10

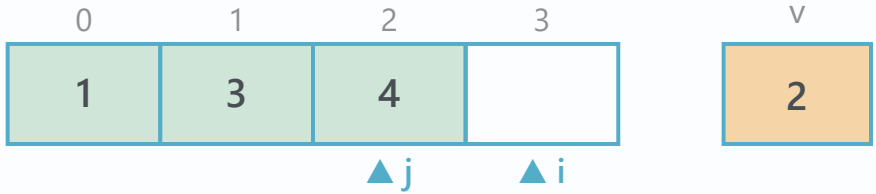
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
▶          a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 2` `v = 3` `j = 1`

`a = [1, 3, 4, 2]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




i = 3: v = 2. a[0..2] sıralı.

Adım 7 / 10

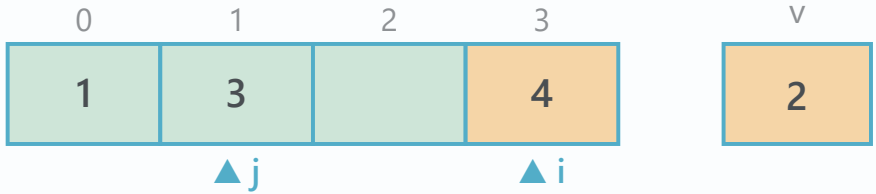
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
▶          int v = a[i], j = i;
6          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 3` `v = 2` `j = 3`

`a = [1, 3, 4, 2]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




v = 2 < a[2] = 4 → kaydır; j = 2. Sınır sınaması yok: bekçi a[0] durduracak.

Adım 8 / 10

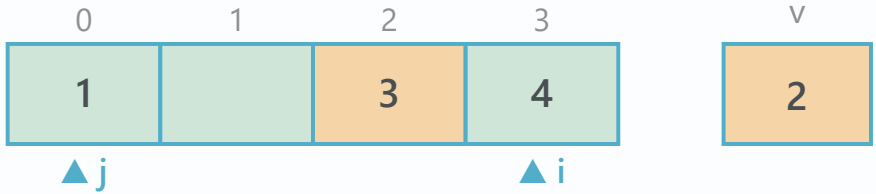
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (kucuk(v, a[j - 1])) {
▶              a[j] = a[j - 1];
▶              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 3` `v = 2` `j = 2`

`a = [1, 3, 4, 4]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




v = 2 < a[1] = 3 → kaydır; j = 1. Sınır sınaması yok: bekçi a[0] durduracak.

Adım 9 / 10

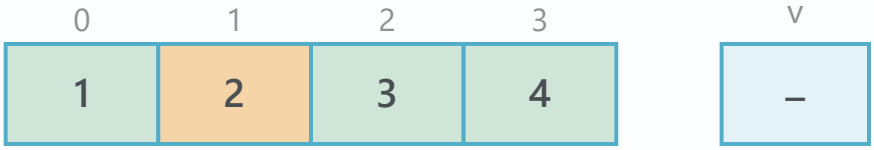
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (kucuk(v, a[j - 1])) {
▶              a[j] = a[j - 1];
▶              j--;
9          }
10         a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 3` `v = 2` `j = 1`

`a = [1, 3, 3, 4]`

# Kod İzleme: `insertion1(a, 0, 3)` (bekçili)




a[0] = 1 ≤ v → dur; **a[1] = 2**.

Adım 10 / 10

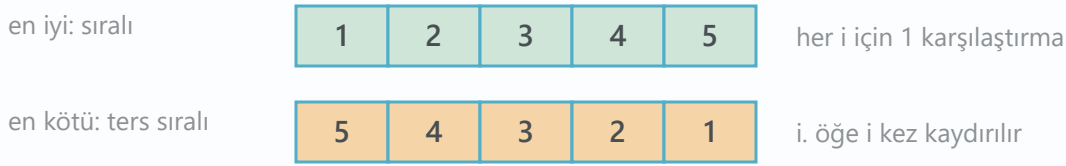
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // bekçi
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (kucuk(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
▶          a[j] = v;
11     }
12 }
```

**DEĞİŞKENLER**

`i = 3` `v = 2` `j = 1`

`a = [1, 2, 3, 4]`

# Eklemeli Sıralama: En İyi ve En Kötü Durum




**En iyi: Ω(n)**

Girdi sıralıysa seçilen öğe yalnızca sıralı listenin en sağındaki değerle kıyaslanır, daha büyük görülünce hemen yerine konur: toplam $n - 1$ karşılaştırma.

**En kötü: $O(n^2)$**

Girdi ters sıralıysa her öğe sıralı listenin tümüyle kıyaslanıp en sola eklenir; sıralı dizi her turda 1 sağa kayar: $1 + 2 + \dots + (n-1) = n(n-1)/2$.

Eklemeli sıralama **kararlıdır** ( `a[j] > v` katı eşitsizlik: eşit öğe atlanmaz) ve yerindedir. Sıralıya yakın veride neredeyse doğrusaldır.

# Ortalama Durum Analizi: j = 3 İçin Olası Durumlar

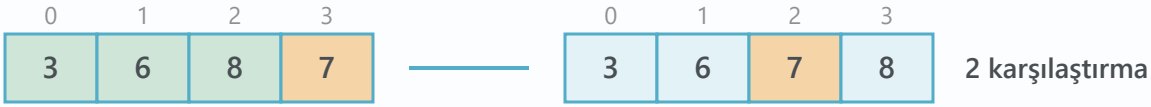



Durum 1: 8 > 10 yanlış → **1 karşılaştırma**.

Adım 1 / 4

j öğesi j + 1 konumdan birine yerleşir: durumların 1'inde 1, 1'inde 2, **2'sinde 3** karşılaştırma. Olasılıklar eşit; yalnız sonuncu “3 karşılaştırma” sonucunu iki kez verir. Ortalama: (1 + 2 + 3 + 3)/4 = **2,25**.

# Ortalama Durum Analizi: j = 3 İçin Olası Durumlar

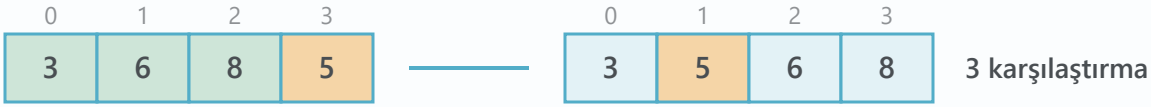



Durum 2: 8 > 7 doğru, 6 > 7 yanlış → **2 karşılaştırma**.

Adım 2 / 4

j öğesi j + 1 konumdan birine yerleşir: durumların 1'inde 1, 1'inde 2, **2'sinde 3** karşılaştırma. Olasılıklar eşit; yalnız sonuncu “3 karşılaştırma” sonucunu iki kez verir. Ortalama: (1 + 2 + 3 + 3)/4 = **2,25**.

# Ortalama Durum Analizi: j = 3 İçin Olası Durumlar




Durum 3: 8 > 5 doğru, 6 > 5 doğru, 3 > 5 yanlış → **3 karşılaştırma**.



j öğesi j + 1 konumdan birine yerleşir: durumların 1'inde 1, 1'inde 2, **2'sinde 3** karşılaştırma. Olasılıklar eşit; yalnız sonuncu “3 karşılaştırma” sonucunu iki kez verir. Ortalama: (1 + 2 + 3 + 3)/4 = **2,25**.

# Ortalama Durum Analizi: j = 3 İçin Olası Durumlar




j = −1 olunca j ≥ 0 yanlış: bu sınama sayılmaz

Durum 4: 8 > 2 doğru, 6 > 2 doğru, 3 > 2 doğru; j = −1 olunca `j ≥ 0` yanlış, `a[j] > v` hiç sınanmaz — sondan bir önceki durumla **aynı sayıda → 3 karşılaştırma**.

Adım 4 / 4

j öğesi j + 1 konumdan birine yerleşir: durumların 1'inde 1, 1'inde 2, **2'sinde 3** karşılaştırma. Olasılıklar eşit; yalnız sonuncu “3 karşılaştırma” sonucunu iki kez verir. Ortalama: (1 + 2 + 3 + 3)/4 = **2,25**.

# Ortalama Durum: Beklenen Karşılaştırma Sayısı

$$E(C_j) = (1/(j+1)) \cdot [2j + \Sigma_{k=1}^{j-1} k] = (j+2)/2 - 1/(j+1)$$

$$E(Y) = \Sigma_{j=1}^{N-1} E(C_j) = (N^2 + 3N)/4 - H_N \approx N^2/4$$

- $C_j$: j. öğeyi yerleştirirken yapılan karşılaştırma sayısı, bir **rastlantı değişkeni**; $E(C_j)$ beklenen değeri.
- $H_N = 1 + 1/2 + \ldots + 1/N$ harmonik sayı: $H_N = \ln N + O(1)$.
- Ortalamada $\approx N^2/4$ karşılaştırma ve $\approx N^2/4$ **yarı değişim** (takas değil kaydırma) → **$O(n^2)$**.

| N | N! | tüm permütasyonlar | formül | $N^2/4$ |
|---|---|---|---|---|
| 3 | 6 | 2,667 | 2,667 | 2,25 |
| 4 | 24 | 4,917 | 4,917 | 4,00 |
| 5 | 120 | 7,717 | 7,717 | 6,25 |
| 6 | 720 | 11,050 | 11,050 | 9,00 |
| 7 | 5040 | 14,907 | 14,907 | 12,25 |

Tablo: N! dizilişin hepsi Python ile sayıldı; formülle birebir aynı.

**HOCADAN** "En iyi ve en kötü durumu biliyoruz; ikisinin ortalamasını alınca ortalama durumu analiz etmiş olur muyuz? Hayır! Ortalama durum analizinde, örnekler uzayındaki durumların tamamını gözlememiz gerekir."

# Kabarcık Sıralaması (Bubble Sort)

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

**HOCADAN** “Dizi sıralıysa, hemen çıkacak şekilde daha dikkatli kodlanabilir. Bir döngü boyunca hiç takas yapılmamışsa sonraki döngüye girmeye gerek yoktur.”




Başlangıç: [7, 2, 9, 5, 1]. Her tur sağdan sola komşuları karşılaştırır.

Adım 1 / 5

# Kabarcık Sıralaması (Bubble Sort)

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

**HOCADAN** “Dizi sıralıysa, hemen çıkacak şekilde daha dikkatli kodlanabilir. Bir döngü boyunca hiç takas yapılmamışsa sonraki döngüye girmeye gerek yoktur.”




Tur i = 0: en küçük kalan değer **1** kabarcık gibi a[0]'a yükseldi → [1, 7, 2, 9, 5].

# Kabarcık Sıralaması (Bubble Sort)

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

**HOCADAN** “Dizi sıralıysa, hemen çıkacak şekilde daha dikkatli kodlanabilir. Bir döngü boyunca hiç takas yapılmamışsa sonraki döngüye girmeye gerek yoktur.”

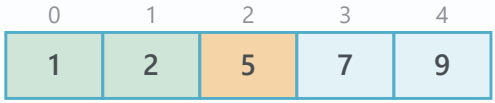



Tur i = 1: en küçük kalan değer **2** kabarcık gibi a[1]'e yükseldi → [1, 2, 7, 5, 9].

# Kabarcık Sıralaması (Bubble Sort)

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

**HOCADAN** “Dizi sıralıysa, hemen çıkacak şekilde daha dikkatli kodlanabilir. Bir döngü boyunca hiç takas yapılmamışsa sonraki döngüye girmeye gerek yoktur.”

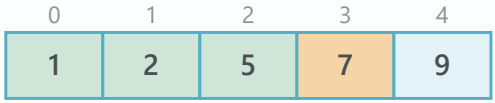



Tur i = 2: en küçük kalan değer **5** kabarcık gibi a[2]'ye yükseldi → [1, 2, 5, 7, 9].

# Kabarcık Sıralaması (Bubble Sort)

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

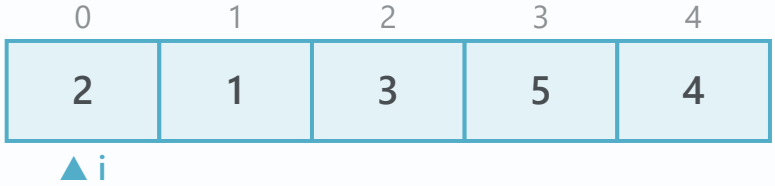
**HOCADAN** “Dizi sıralıysa, hemen çıkacak şekilde daha dikkatli kodlanabilir. Bir döngü boyunca hiç takas yapılmamışsa sonraki döngüye girmeye gerek yoktur.”




Tur i = 3: takas yok; kalanların en küçüğü **7** zaten a[3]'te → [1, 2, 5, 7, 9].

Adım 5 / 5

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




Tur i = 0 başlıyor; takas bayrağı sıfırlandı.

Adım 1 / 11

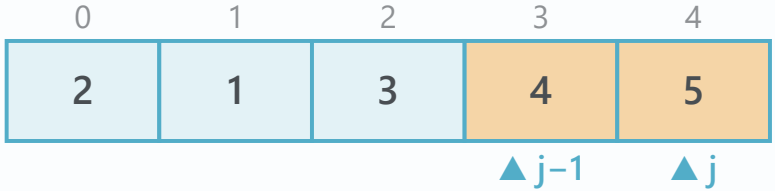
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
▶          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
5              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 0` `takas = 0`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[4] < a[3] → takas: [2, 1, 3, 4, 5]; **takas = 1**.

Adım 2 / 11

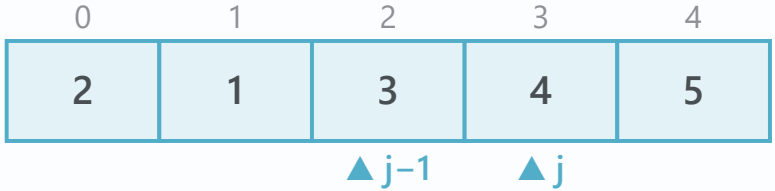
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;       // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
▶                  SWAP(a[j], a[j - 1]);
▶                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 0` `j = 4` `takas = 1`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[3] = 4 ≥ a[2] = 3 → takas yok.

Adım 3 / 11

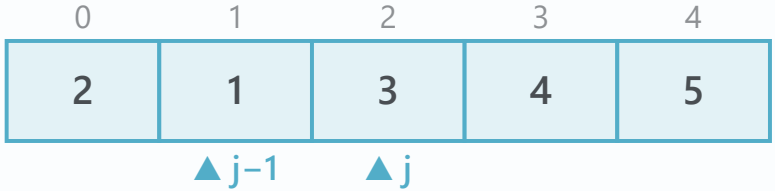
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 0` `j = 3` `takas = 1`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[2] = 3 ≥ a[1] = 1 → takas yok.

Adım 4 / 11

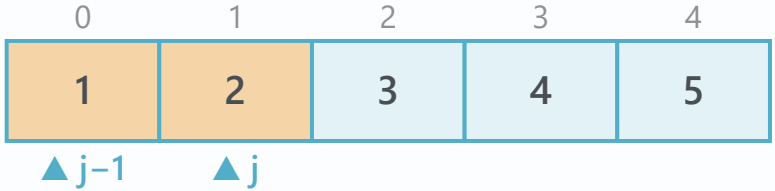
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 0` `j = 2` `takas = 1`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[1] < a[0] → takas: [1, 2, 3, 4, 5]; **takas = 1**.

Adım 5 / 11

```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;       // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
▶                  SWAP(a[j], a[j - 1]);
▶                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 0` `j = 1` `takas = 1`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




takas = 1 → dizi henüz sıralı sayılamaz, sonraki tura geç.

Adım 6 / 11

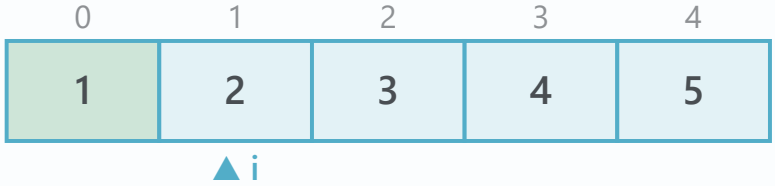
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
5              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
▶          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 0` `takas = 1`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




Tur i = 1 başlıyor; takas bayrağı sıfırlandı.

Adım 7 / 11

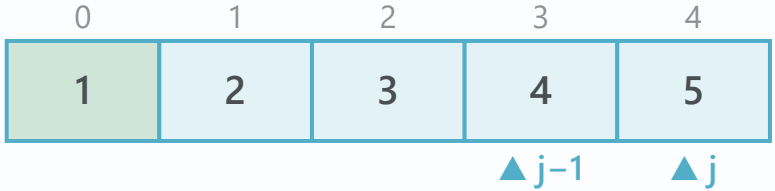
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
▶          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
5              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `takas = 0`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[4] = 5 ≥ a[3] = 4 → takas yok.

Adım 8 / 11

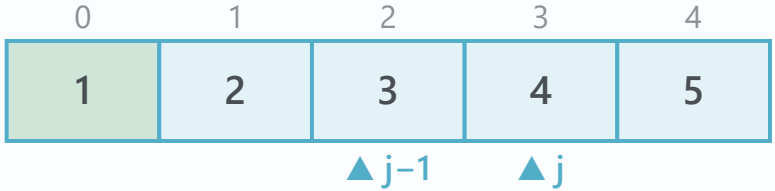
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `j = 4` `takas = 0`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[3] = 4 ≥ a[2] = 3 → takas yok.

Adım 9 / 11

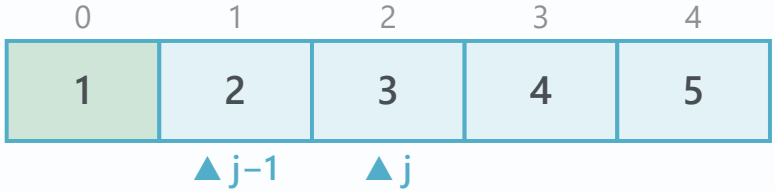
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `j = 3` `takas = 0`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




a[2] = 3 ≥ a[1] = 2 → takas yok.

Adım 10 / 11

```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
▶              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
9          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `j = 2` `takas = 0`

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

# Kod İzleme: Erken Durmalı `bubble2(a, 0, 4)`




Bu turda hiç takas olmadı → dizi sıralı; **erken çıkış**. 2 tur boşuna yapılmadı.

Adım 11 / 11

```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int takas = 0;      // bu turda takas oldu mu?
4          for (int j = R; j > i; j--)
5              if (kucuk(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  takas = 1;
8              }
▶          if (!takas) return; // sıralı: erken çık
10     }
11 }
```

**DEĞİŞKENLER**

`i = 1` `takas = 0`

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

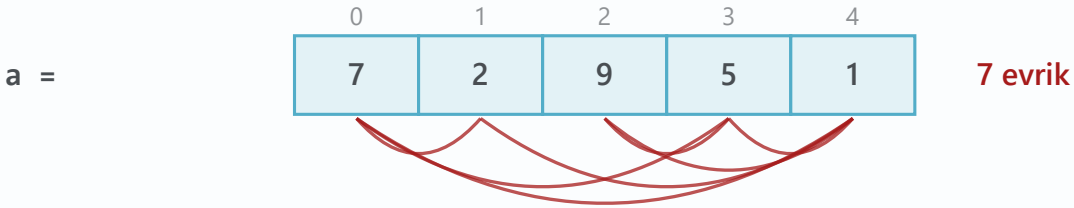
# Kabarcık Sıralamasının Analizi

- Karşılaştırma: her durumda $n(n-1)/2 \approx \mathbf{n^2/2}$ (erken durma yoksa).
- Değiştirme: en kötü durumda (ters sıralı) $\mathbf{n^2/2}$; ortalamada **evrik** (*inversion*) sayısı kadar, yani $\approx \mathbf{n^2/4}$.
- Karmaşıklık $\mathbf{O(n^2)}$; erken durmalı sürüm sıralı girdide $\mathbf{O(n)}$.
- Yalnız komşular takaslanır ve eşitler takaslanmaz → **kararlı**, yerinde.

| n | karş. | takas (en kötü) | takas (ort.) |
|---|---|---|---|
| 4 | 6 | 6 | 3 |
| 5 | 10 | 10 | 5 |
| 6 | 15 | 15 | 7,5 |
| 7 | 21 | 21 | 10,5 |

Ortalama sütunu: n! dizilişin tamamı sayıldı. Bazı kaynaklar (ör. Sedgewick) ortalama için de $n^2/2$ değiştirme verir; sayım ortalamanın $n^2/4$ olduğunu gösteriyor.

# Evrik (Inversion): Ortak Ölçü




Evrik: i < j iken a[i] > a[j] olan çift. [7, 2, 9, 5, 1] → (7,2), (7,5), (7,1), (2,1), (9,5), (9,1), (5,1)

- Komşu takası (kabarcık) ya da bir konum kaydırma (eklemeli) tam olarak **bir** evriği giderir.
- Bu yüzden ikisinde de takas/kaydırma sayısı = evrik sayısı: bu dizide 7 (izlerde de 7 kaydırma vardı).
- Ters sıralı dizide n(n−1)/2 evrik; rastgele dizide ortalama **n(n−1)/4** (her çift ½ olasılıkla ters).
- Komşu öğelerle çalışan her algoritma ortalamada $\Omega(n^2)$: uzak takas yapan Shell ve hızlı sıralama bu duvarı aşar.

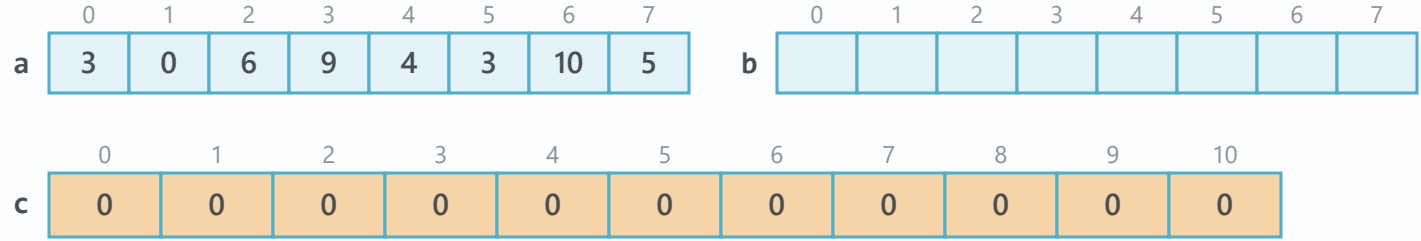
# Sayarak Sıralama (Counting Sort)

```c
int counting_sort(const int a[], int b[],
                  int r, int k) {
    int *c = malloc((k + 1) * sizeof *c);
    if (c == NULL) return -1;
    for (int i = 0; i <= k; i++) c[i] = 0;  // L1
    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
    for (int i = 1; i <= k; i++)            // L3
        c[i] += c[i - 1];
    for (int j = r; j >= 0; j--) {          // L4
        b[c[a[j]] - 1] = a[j];
        c[a[j]]--;
    }
    free(c);
    return 0;
}
```

| ad | anlamı |
|---|---|
| a[] | girdi dizisi |
| b[] | sıralı çıktı dizisi |
| c[] | yardımcı sayaç dizisi (k+1 öğe) |
| r | a[0..r] sıralanır (r+1 öğe) |
| k | değerler 0…k aralığında |

- 1954'te **Harold H. Seward**. Aralığı bilinen tamsayıları sıralar.
- Değerler birbiriyle **karşılaştırılmaz** → *non-comparison sort* (kova sıralama da bu sınıftadır).
- `int c[K]` derleme zamanında sabit boyut ister; burada `malloc` + kontrol + `free`.

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L1: c[0..10] sıfırlandı. Her değerden kaç tane olduğu burada sayılacak.

Adım 1 / 19

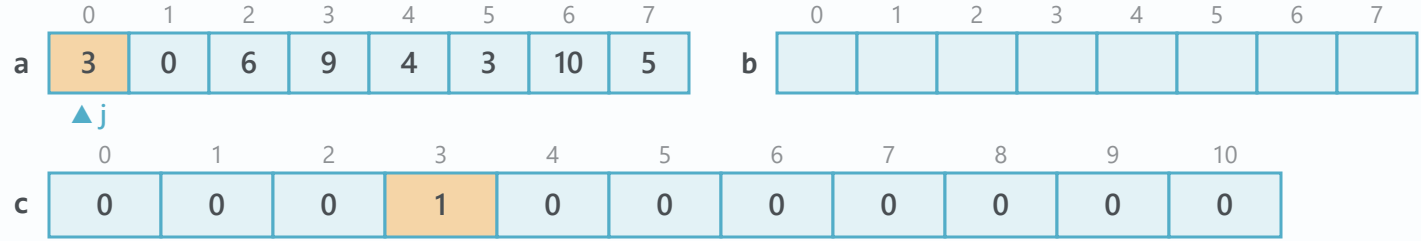
```
▶   for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2   for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3   for (int i = 1; i <= k; i++)            // L3
4       c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`r = 7` `k = 10`

`c = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[0] = 3 → **c[3]++** = 1.

Adım 2 / 19

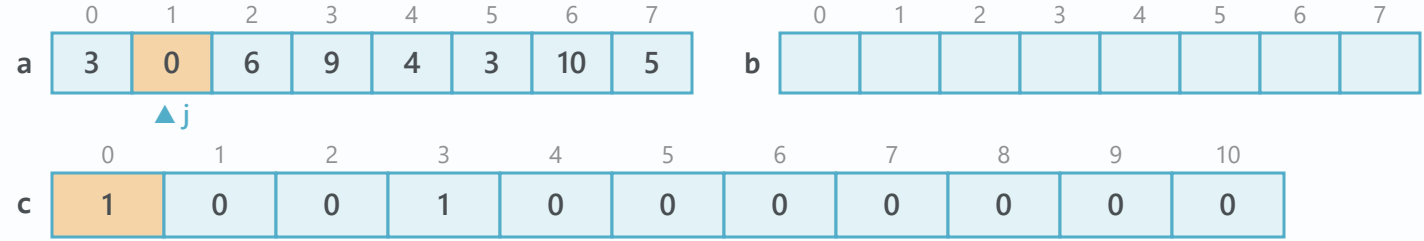
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 0` `a[j] = 3`

`c = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[1] = 0 → **c[0]++** = 1.

Adım 3 / 19

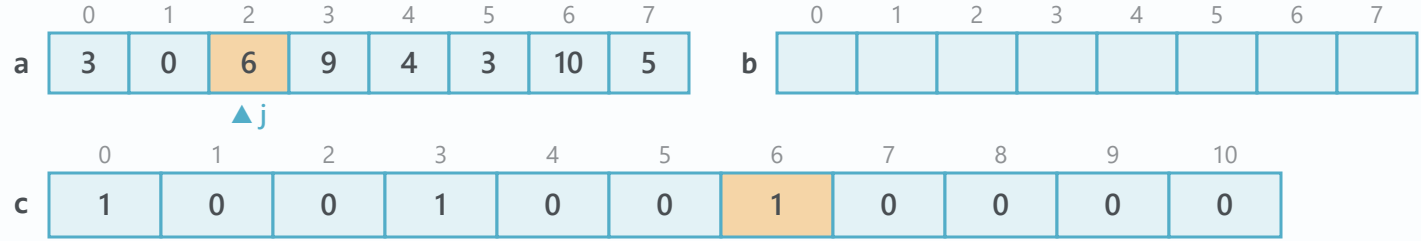
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 1` `a[j] = 0`

`c = [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[2] = 6 → **c[6]++** = 1.

Adım 4 / 19

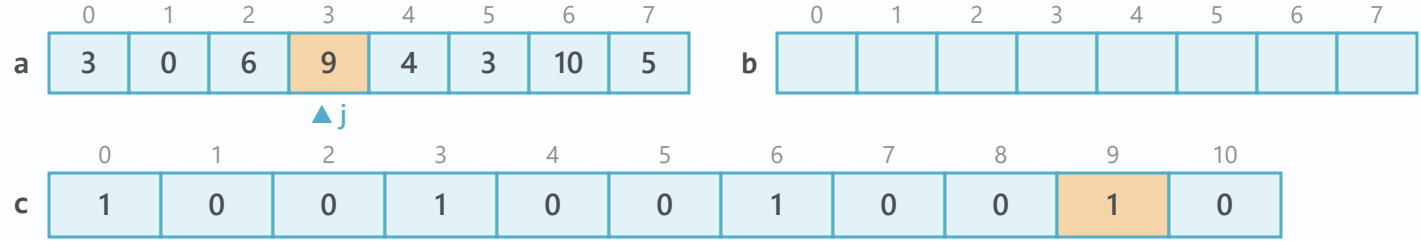
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 2` `a[j] = 6`

`c = [1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[3] = 9 → **c[9]++** = 1.

Adım 5 / 19

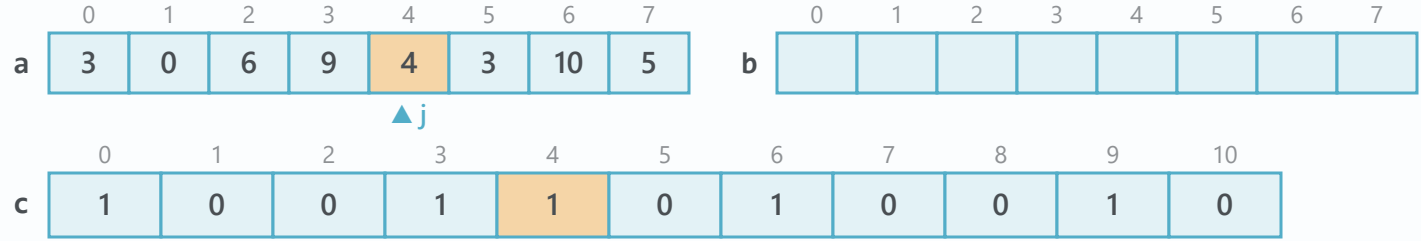
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 3` `a[j] = 9`

`c = [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[4] = 4 → **c[4]++** = 1.

Adım 6 / 19

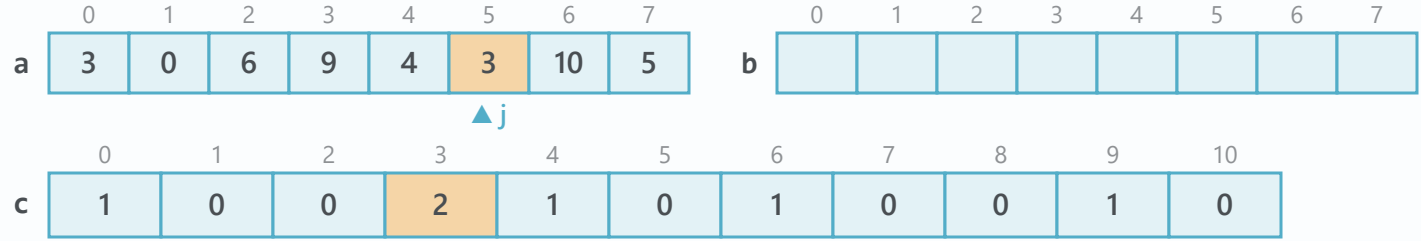
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 4` `a[j] = 4`

`c = [1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[5] = 3 → **c[3]++** = 2. (İkinci 3!)

Adım 7 / 19

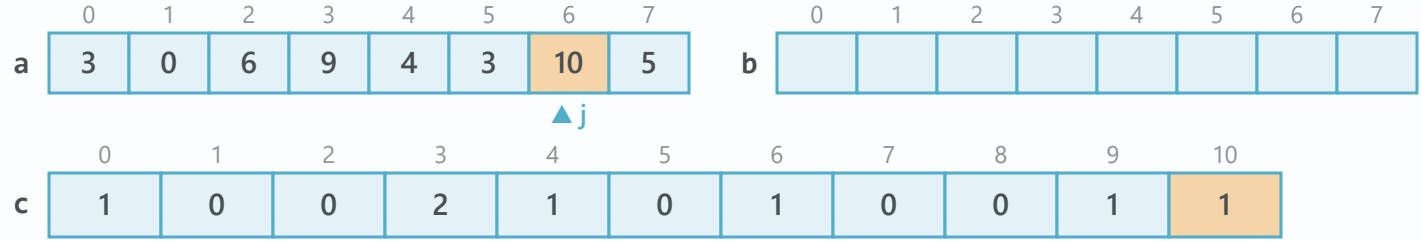
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 5` `a[j] = 3`

`c = [1, 0, 0, 2, 1, 0, 1, 0, 0, 1, 0]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[6] = 10 → **c[10]++** = 1.

Adım 8 / 19

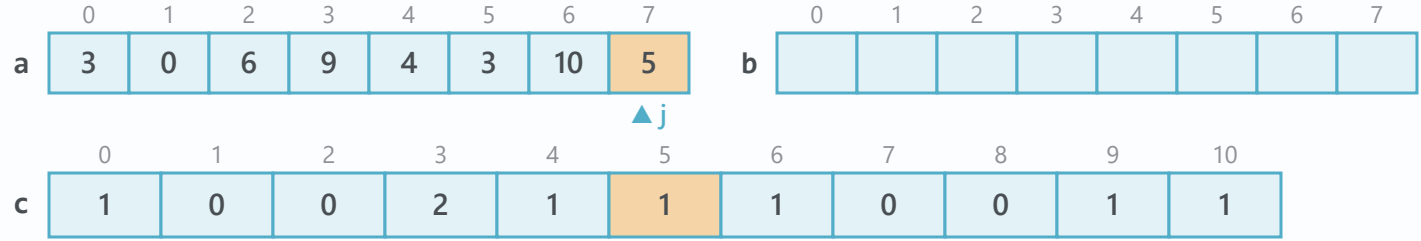
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 6` `a[j] = 10`

`c = [1, 0, 0, 2, 1, 0, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L2: a[7] = 5 → **c[5]++** = 1.

Adım 9 / 19

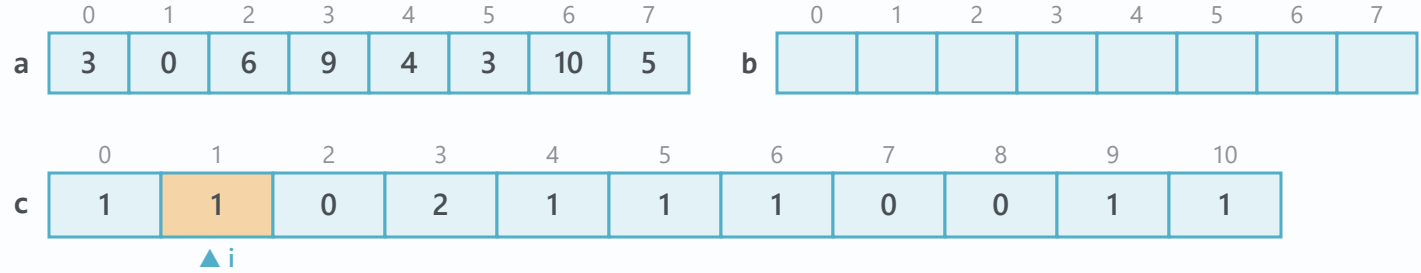
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`j = 7` `a[j] = 5`

`c = [1, 0, 0, 2, 1, 1, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[1] += c[0] → 1: “≤ 1 olan 1 öğe var”.

Adım 10 / 19

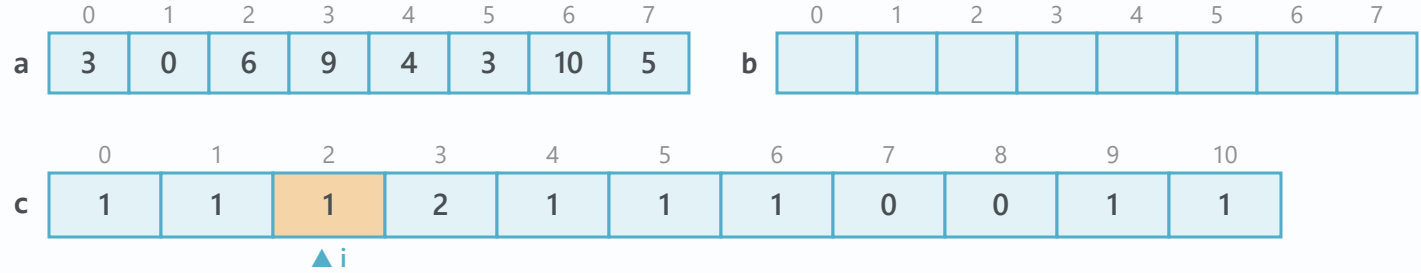
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 1`

`c = [1, 1, 0, 2, 1, 1, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[2] += c[1] → 1: “≤ 2 olan 1 öğe var”.

Adım 11 / 19

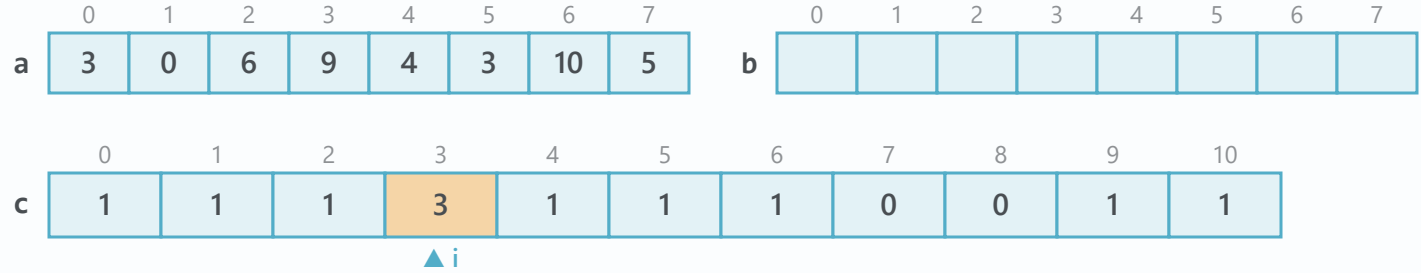
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 2`

`c = [1, 1, 1, 2, 1, 1, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[3] += c[2] → 3: "≤ 3 olan 3 öğe var". 3 gelirse 2. konuma.

Adım 12 / 19

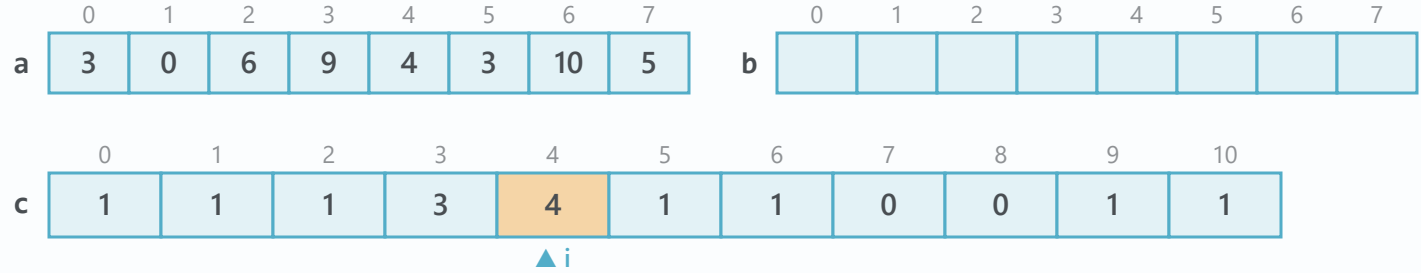
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 3`

`c = [1, 1, 1, 3, 1, 1, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[4] += c[3] → 4: "≤ 4 olan 4 öğe var". 4 gelirse 3. konuma.

Adım 13 / 19

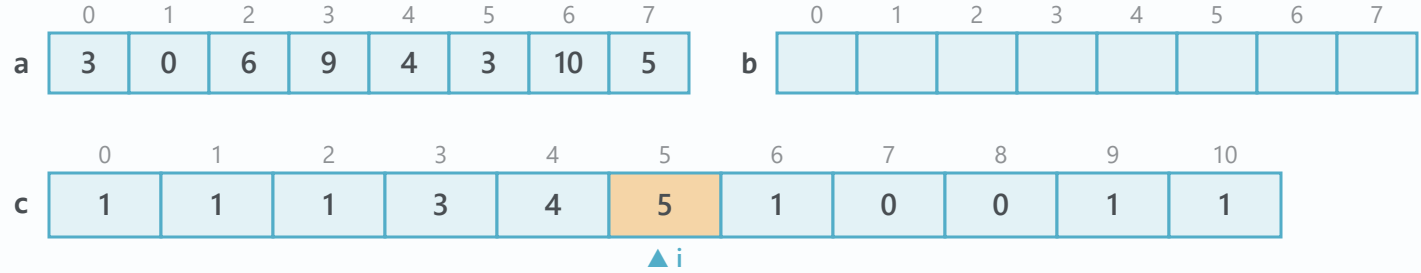
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 4`

`c = [1, 1, 1, 3, 4, 1, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[5] += c[4] → 5: "≤ 5 olan 5 öğe var". 5 gelirse 4. konuma.

Adım 14 / 19

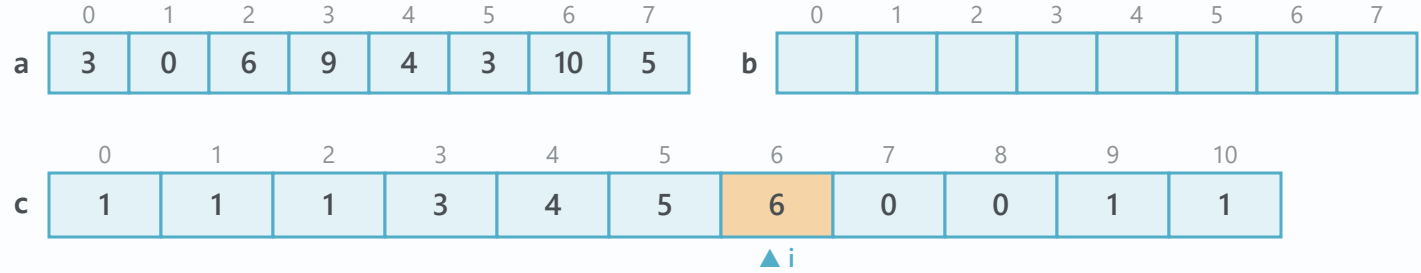
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 5`

`c = [1, 1, 1, 3, 4, 5, 1, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[6] += c[5] → 6: "≤ 6 olan 6 öğe var". 6 gelirse 5. konuma.

Adım 15 / 19

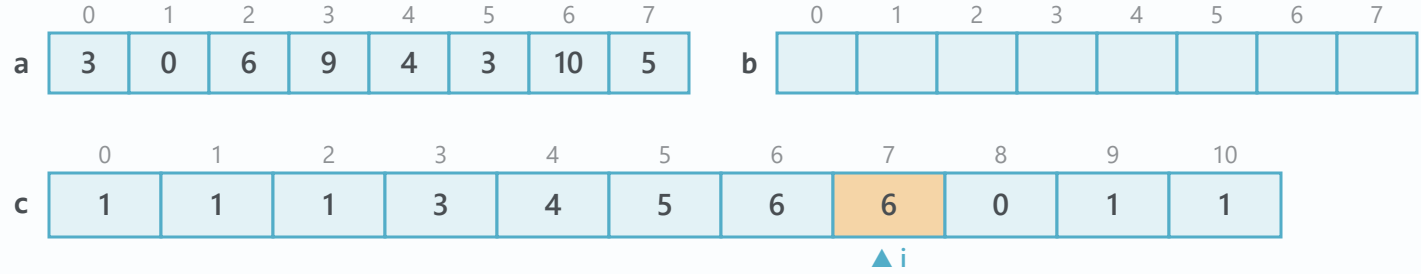
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 6`

`c = [1, 1, 1, 3, 4, 5, 6, 0, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[7] += c[6] → 6: “≤ 7 olan 6 öğe var”.

Adım 16 / 19

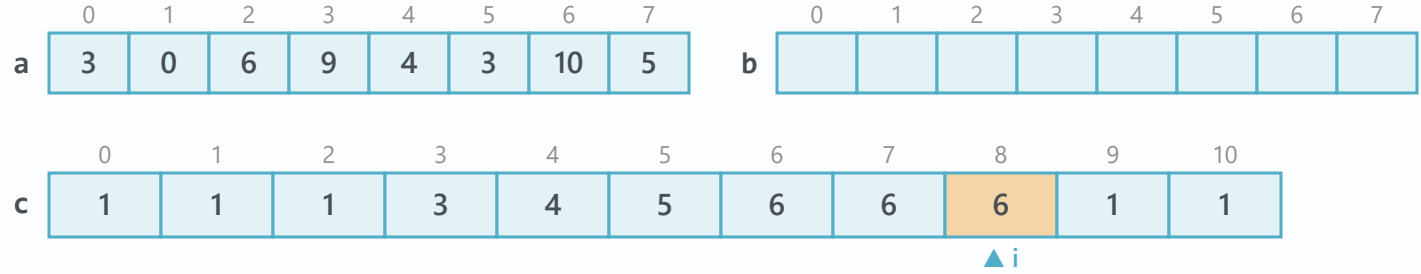
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 7`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 0, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[8] += c[7] → 6: “≤ 8 olan 6 öğe var”.

Adım 17 / 19

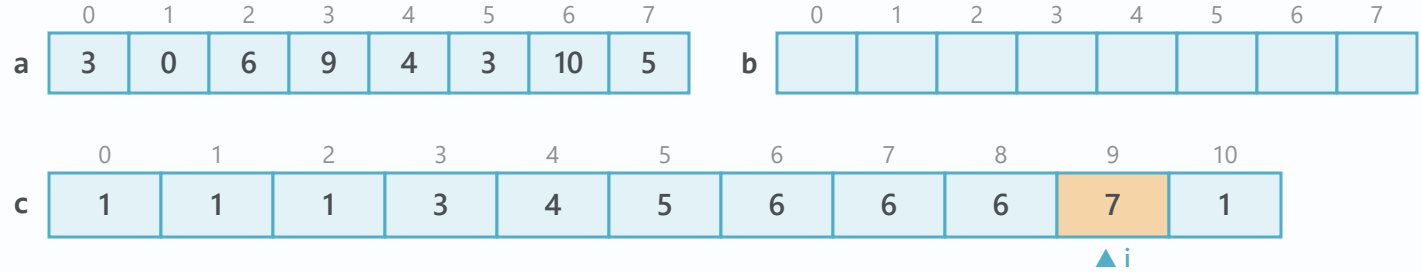
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 8`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 1, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[9] += c[8] → 7: "≤ 9 olan 7 öğe var". 9 gelirse 6. konuma.

Adım 18 / 19

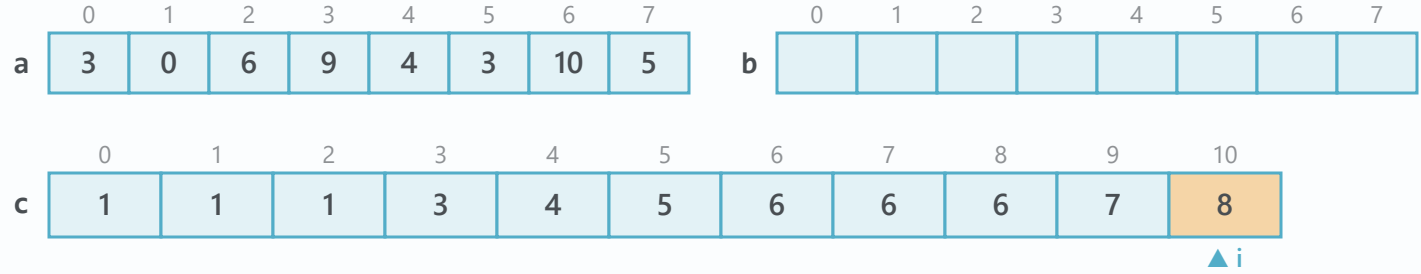
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 9`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 7, 1]`

# Kod İzleme: `counting_sort` — Sayma (L1–L3)




L3: c[10] += c[9] → 8: “≤ 10 olan 8 öğe var”. 10 gelirse 7. konuma.

Adım 19 / 19

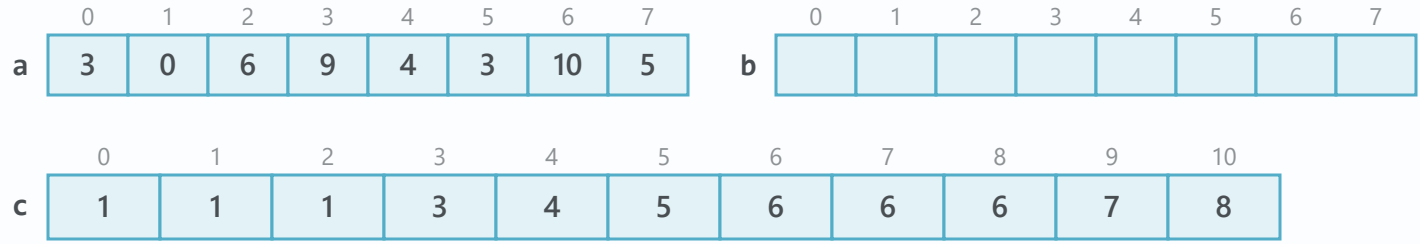
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

**DEĞİŞKENLER**

`i = 10`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 7, 8]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L3 sonu: c[x] = “x'ten küçük ya da eşit öğe sayısı”. Ör. c[9] = 7 → 9 gelirse 7 − 1 = **6**. konuma.

Adım 1 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
2        b[c[a[j]] - 1] = a[j];
3        c[a[j]]--;
4    }
```

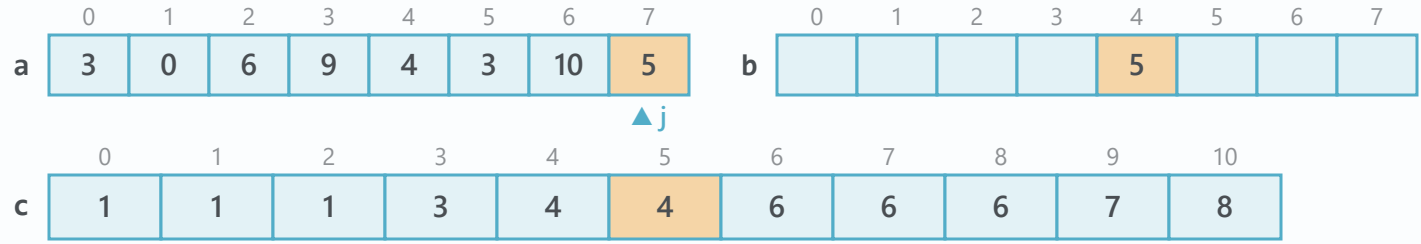
**DEĞİŞKENLER**

`r = 7`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 7, 8]`

`b = [_, _, _, _, _, _, _, _]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[7] = 5, c[5] = 5 → **b[4] = 5**; c[5]-- = 4.

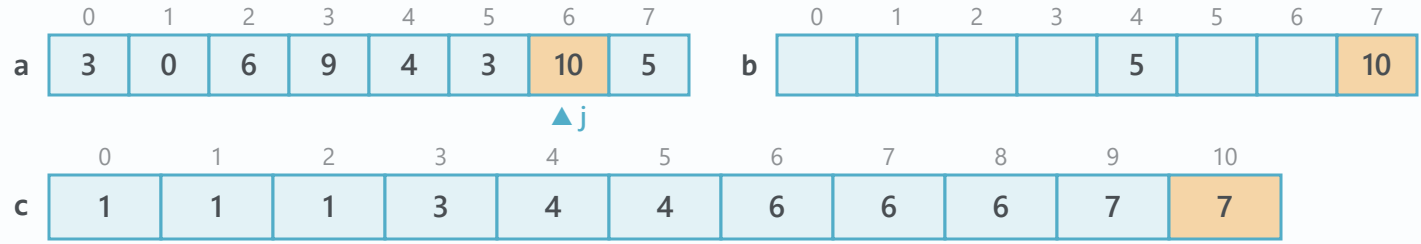
Adım 2 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

**DEĞİŞKENLER**

- `j = 7`
- `a[j] = 5`
- `c = [1, 1, 1, 3, 4, 4, 6, 6, 6, 7, 8]`
- `b = [_, _, _, _, 5, _, _, _]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[6] = 10, c[10] = 8 → **b[7] = 10**; c[10]-- = 7.

Adım 3 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

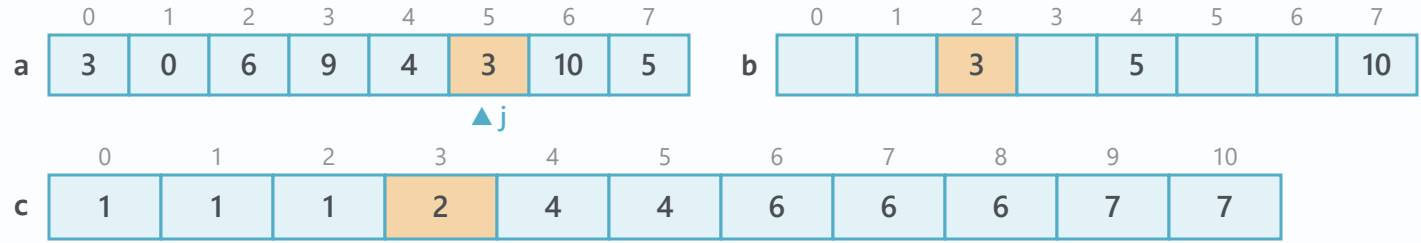
**DEĞİŞKENLER**

`j = 6` `a[j] = 10`

`c = [1, 1, 1, 3, 4, 4, 6, 6, 6, 7, 7]`

`b = [_, _, _, _, 5, _, _, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[5] = 3, c[3] = 3 → **b[2] = 3**; c[3]-- = 2 (aynı değerli bir sonraki öğe bir sola yerleşecek).

Adım 4 / 10

```
1     for (int j = r; j >= 0; j--) {          // L4
▶         b[c[a[j]] - 1] = a[j];
▶         c[a[j]]--;
4     }
```

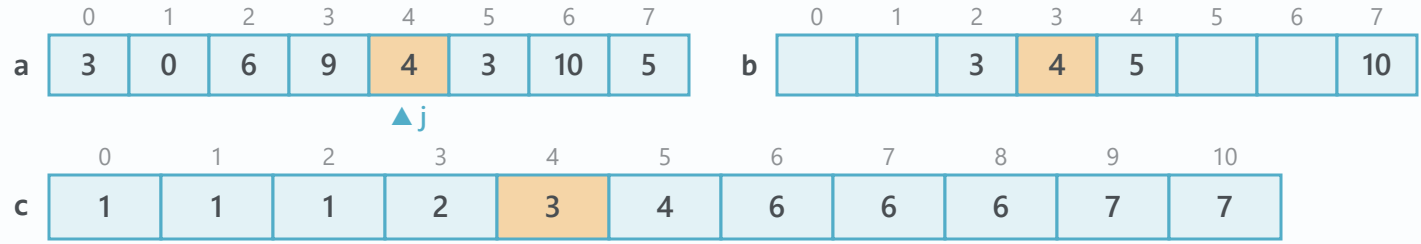
**DEĞİŞKENLER**

`j = 5` `a[j] = 3`

`c = [1, 1, 1, 2, 4, 4, 6, 6, 6, 7, 7]`

`b = [_, _, 3, _, 5, _, _, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[4] = 4, c[4] = 4 → **b[3] = 4**; c[4]-- = 3.

Adım 5 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

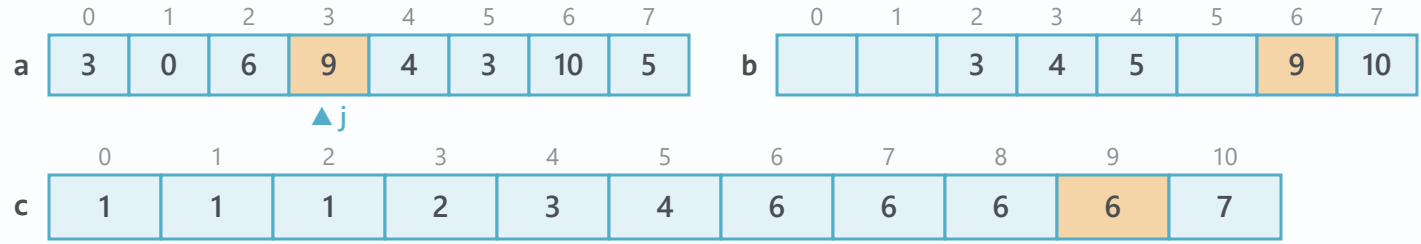
**DEĞİŞKENLER**

`j = 4` `a[j] = 4`

`c = [1, 1, 1, 2, 3, 4, 6, 6, 6, 7, 7]`

`b = [_, _, 3, 4, 5, _, _, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[3] = 9, c[9] = 7 → **b[6] = 9**; c[9]-- = 6.

Adım 6 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

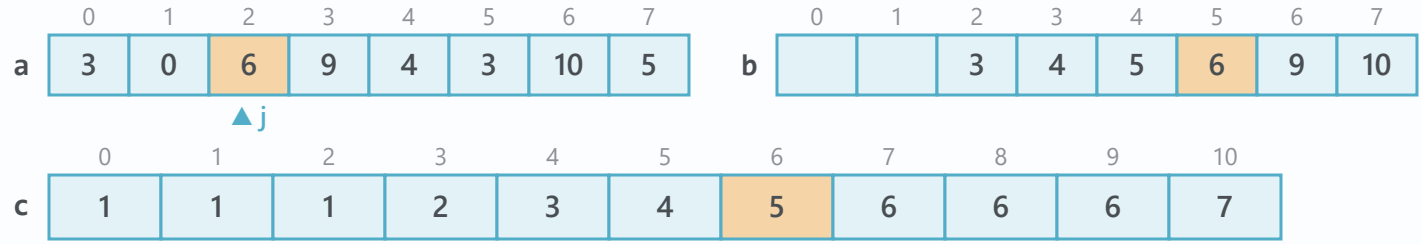
**DEĞİŞKENLER**

`j = 3` `a[j] = 9`

`c = [1, 1, 1, 2, 3, 4, 6, 6, 6, 6, 7]`

`b = [_, _, 3, 4, 5, _, 9, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[2] = 6, c[6] = 6 → **b[5] = 6**; c[6]-- = 5.

Adım 7 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

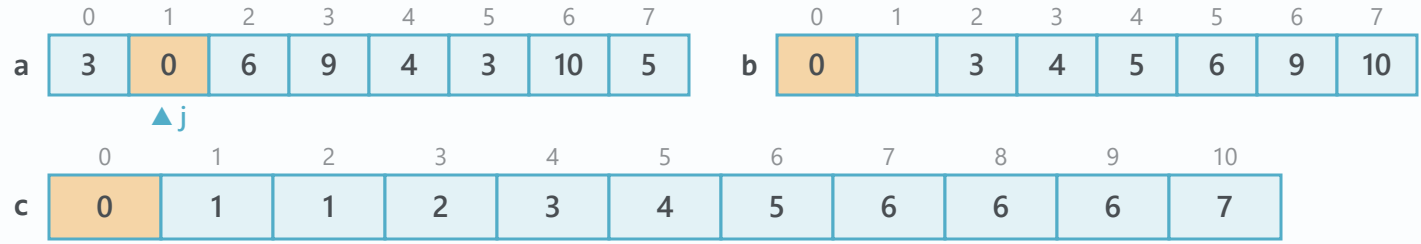
**DEĞİŞKENLER**

`j = 2` `a[j] = 6`

`c = [1, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7]`

`b = [_, _, 3, 4, 5, 6, 9, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[1] = 0, c[0] = 1 → **b[0] = 0**; c[0]-- = 0.

Adım 8 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

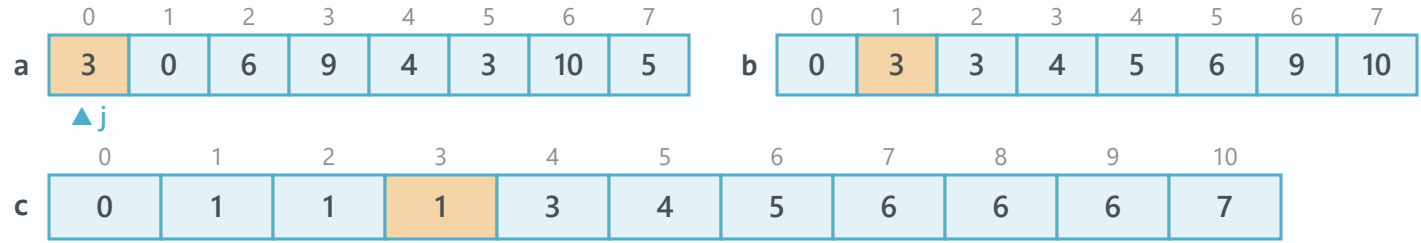
**DEĞİŞKENLER**

`j = 1` `a[j] = 0`

`c = [0, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7]`

`b = [0, _, 3, 4, 5, 6, 9, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




L4: a[0] = 3, c[3] = 2 → **b[1] = 3**; c[3]-- = 1 (aynı değerli bir sonraki öğe bir sola yerleşecek).

Adım 9 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

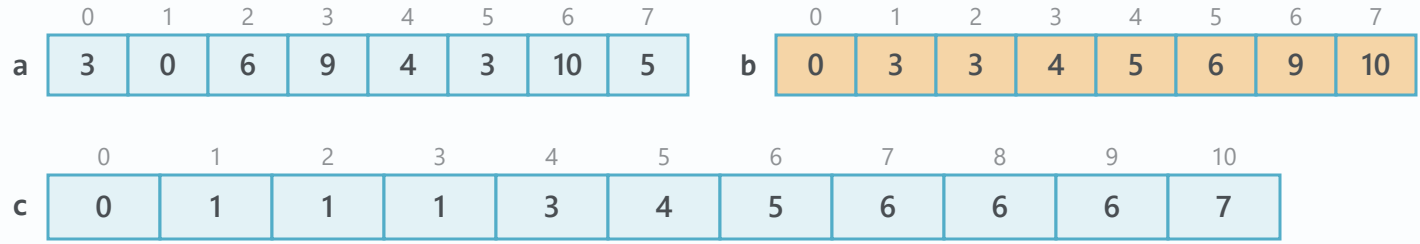
**DEĞİŞKENLER**

`j = 0` `a[j] = 3`

`c = [0, 1, 1, 1, 3, 4, 5, 6, 6, 6, 7]`

`b = [0, 3, 3, 4, 5, 6, 9, 10]`

# Kod İzleme (devam): L4 ile b Dizisine Yerleştirme




Sonuç b = [0, 3, 3, 4, 5, 6, 9, 10]. Sağdan sola gezildiği için iki 3'ün göreli sırası korundu: **kararlı**.

Adım 10 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
2        b[c[a[j]] - 1] = a[j];
3        c[a[j]]--;
▶    }
```

**DEĞİŞKENLER**

c = `[0, 1, 1, 1, 3, 4, 5, 6, 6, 6, 7]`

b = `[0, 3, 3, 4, 5, 6, 9, 10]`

# Sayarak Sıralama: Analiz ve Sınırlar

| döngü | iş | maliyet |
|---|---|---|
| L1 | c'yi sıfırla | O(k) |
| L2 | say | O(n) |
| L3 | kümülatif toplam | O(k) |
| L4 | yerleştir | O(n) |

- Toplam **O(n + k)**; k = O(n) ise **O(n)**. Ek bellek O(n + k): **yerinde değil**.
- L4 sağdan sola ilerlediği için **kararlı** — basamağa göre sıralamada (radix sort) bu özellik şart.

**HOCADAN** "Doğrusal zamanda sıralama yapmak için veri aralığını bilmeliyiz. 10 değer sıralanacaksa aralık 0 ile 1000 arasında değişiyorsa $n^3$'lük algoritma olur... Dolayısıyla aralığın çok küçük olması lazımdır."

**HOCADAN** "İlk geçişte frekansları sayıyoruz, ikinci geçişte sıralıyoruz. Hızlı transpoz alma tek bellekle olur demiştik ya, işte bu o..."

# Kod İzleme: `shell_sort(a, 8)`

h = 4

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 35 | 33 | 42 | 10 | 14 | 19 | 27 | 44 |

Aralık **h = 4**: birbirinden 4 uzak öğeler ayrı ayrı eklemeli sıralanır.

Adım 1 / 20

```
1  void shell_sort(int a[], int n) {
▶      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
5              while (j >= h && a[j - h] > v) {
6                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
9              a[j] = v;
10         }
11 }
```

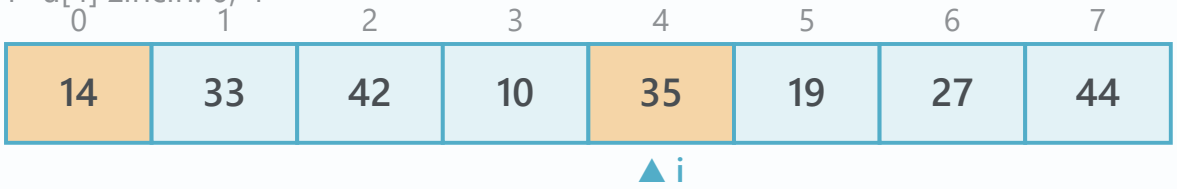
DEĞİŞKENLER

h = 4

a = [35, 33, 42, 10, 14, 19, 27, 44]

# Kod İzleme: `shell_sort(a, 8)`

h = 4 · a[4] zinciri: 0, 4




i = 4: v = 14; 1 kez 4 geri kaydırıldı, a[0] = 14.

Adım 2 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

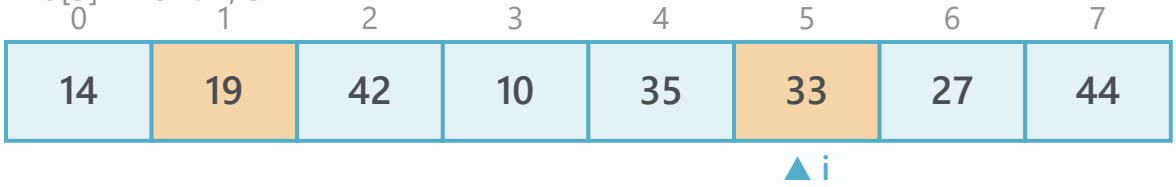
**DEĞİŞKENLER**

`h = 4` `i = 4` `v = 14` `j = 0`

`a = [14, 33, 42, 10, 35, 19, 27, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 4 · a[5] zinciri: 1, 5




i = 5: v = 19; 1 kez 4 geri kaydırıldı, a[1] = 19.

Adım 3 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

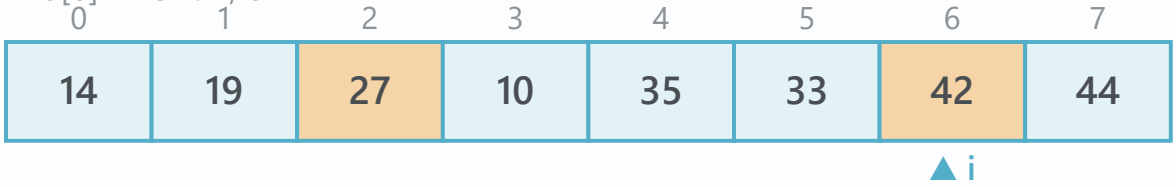
**DEĞİŞKENLER**

`h = 4` `i = 5` `v = 19` `j = 1`

`a = [14, 19, 42, 10, 35, 33, 27, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 4 · a[6] zinciri: 2, 6




i = 6: v = 27; 1 kez 4 geri kaydırıldı, a[2] = 27.

Adım 4 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

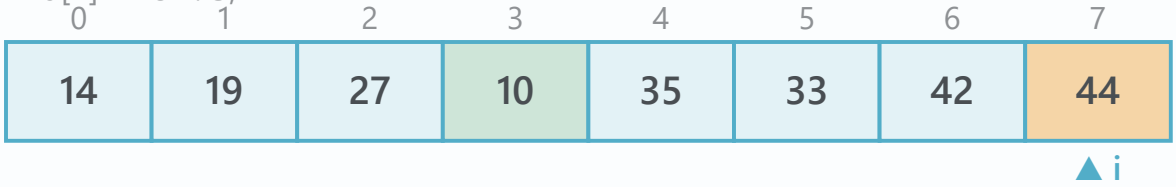
**DEĞİŞKENLER**

`h = 4` `i = 6` `v = 27` `j = 2`

`a = [14, 19, 27, 10, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 4 · a[7] zinciri: 3, 7




i = 7: v = 44; a[3] ≤ 44, yerinde kaldı.

Adım 5 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

**DEĞİŞKENLER**

`h = 4` `i = 7` `v = 44` `j = 7`

`a = [14, 19, 27, 10, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 14 | 19 | 27 | 10 | 35 | 33 | 42 | 44 |

Aralık **h = 2**: birbirinden 2 uzak öğeler ayrı ayrı eklemeli sıralanır.

Adım 6 / 20

```
1  void shell_sort(int a[], int n) {
▶      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
5              while (j >= h && a[j - h] > v) {
6                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
9              a[j] = v;
10         }
11 }
```

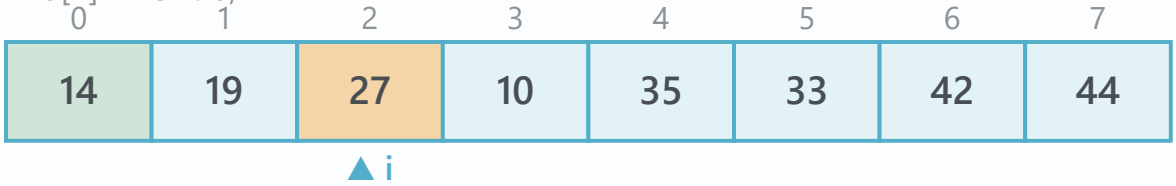
DEĞİŞKENLER

h = 2

a = [14, 19, 27, 10, 35, 33, 42, 44]

# Kod İzleme: `shell_sort(a, 8)`

h = 2 · a[2] zinciri: 0, 2




i = 2: v = 27; a[0] ≤ 27, yerinde kaldı.

Adım 7 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

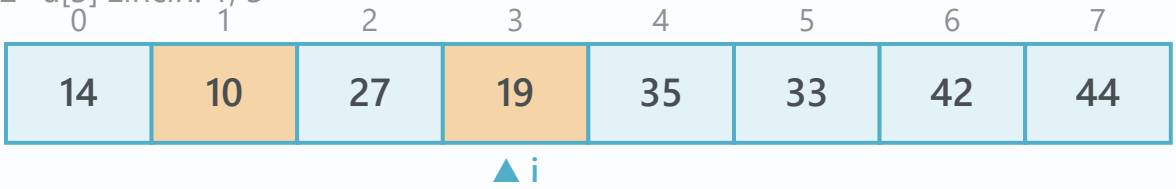
**DEĞİŞKENLER**

`h = 2` `i = 2` `v = 27` `j = 2`

`a = [14, 19, 27, 10, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 2 · a[3] zinciri: 1, 3




i = 3: v = 10; 1 kez 2 geri kaydırıldı, a[1] = 10.

Adım 8 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

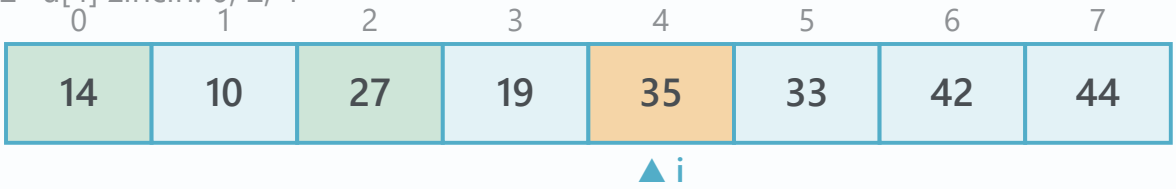
**DEĞİŞKENLER**

`h = 2` `i = 3` `v = 10` `j = 1`

`a = [14, 10, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 2 · a[4] zinciri: 0, 2, 4




i = 4: v = 35; a[2] ≤ 35, yerinde kaldı.

Adım 9 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

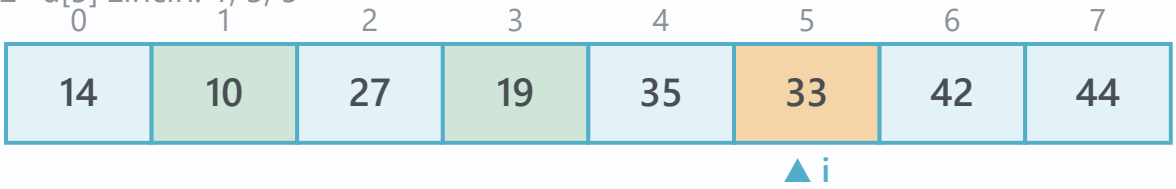
**DEĞİŞKENLER**

`h = 2` `i = 4` `v = 35` `j = 4`

`a = [14, 10, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 2 · a[5] zinciri: 1, 3, 5




i = 5: v = 33; a[3] ≤ 33, yerinde kaldı.

Adım 10 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

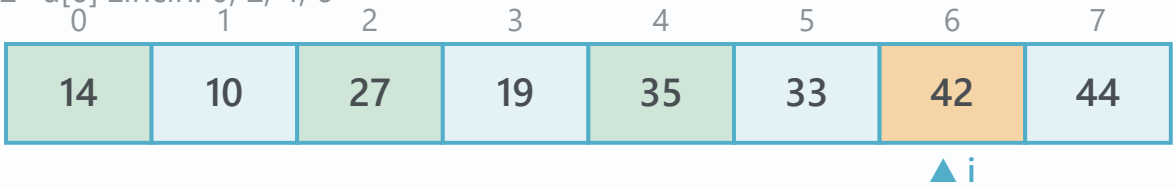
**DEĞİŞKENLER**

`h = 2` `i = 5` `v = 33` `j = 5`

`a = [14, 10, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 2 · a[6] zinciri: 0, 2, 4, 6




i = 6: v = 42; a[4] ≤ 42, yerinde kaldı.

Adım 11 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

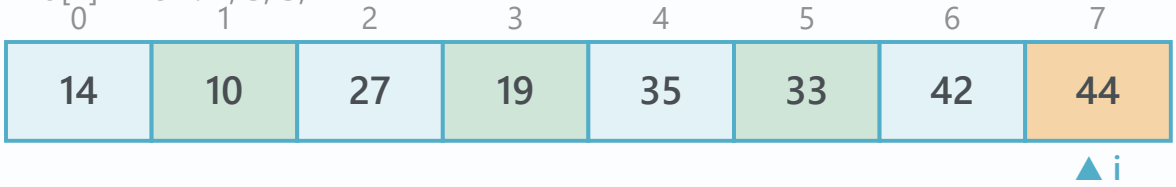
**DEĞİŞKENLER**

`h = 2` `i = 6` `v = 42` `j = 6`

`a = [14, 10, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 2 · a[7] zinciri: 1, 3, 5, 7




i = 7: v = 44; a[5] ≤ 44, yerinde kaldı.

Adım 12 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

**DEĞİŞKENLER**

`h = 2` `i = 7` `v = 44` `j = 7`

`a = [14, 10, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 14 | 10 | 27 | 19 | 35 | 33 | 42 | 44 |

Son tur **h = 1**: sıradan eklemeli sıralama — ama dizi artık neredeyse sıralı.

Adım 13 / 20

```
1  void shell_sort(int a[], int n) {
▶      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
5              while (j >= h && a[j - h] > v) {
6                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
9              a[j] = v;
10         }
11 }
```

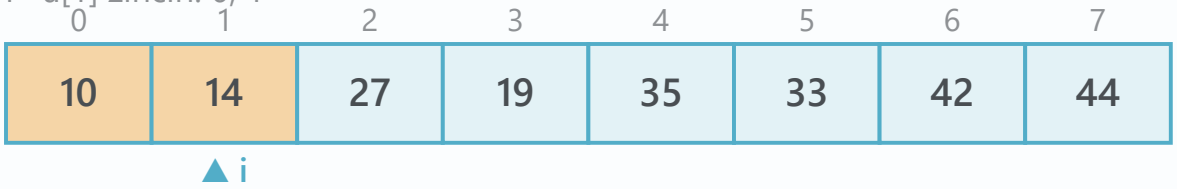
DEĞİŞKENLER

h = 1

a = [14, 10, 27, 19, 35, 33, 42, 44]

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[1] zinciri: 0, 1




i = 1: v = 10; 1 kez 1 geri kaydırıldı, a[0] = 10.

Adım 14 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

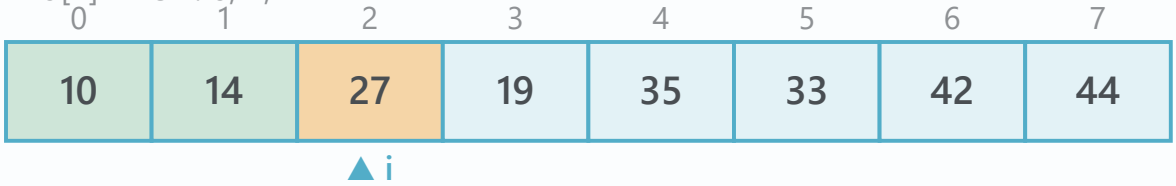
**DEĞİŞKENLER**

`h = 1` `i = 1` `v = 10` `j = 0`

`a = [10, 14, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[2] zinciri: 0, 1, 2




i = 2: v = 27; a[1] ≤ 27, yerinde kaldı.

Adım 15 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

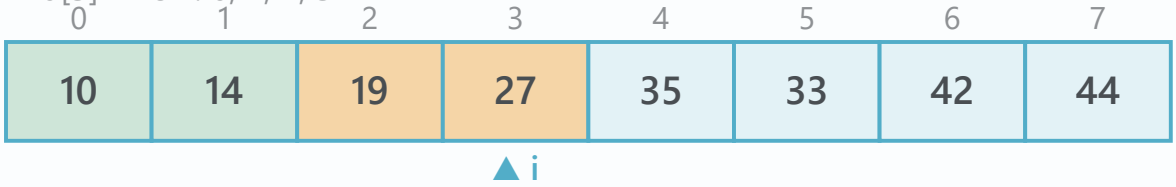
**DEĞİŞKENLER**

`h = 1` `i = 2` `v = 27` `j = 2`

`a = [10, 14, 27, 19, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[3] zinciri: 0, 1, 2, 3




i = 3: v = 19; 1 kez 1 geri kaydırıldı, a[2] = 19.



```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

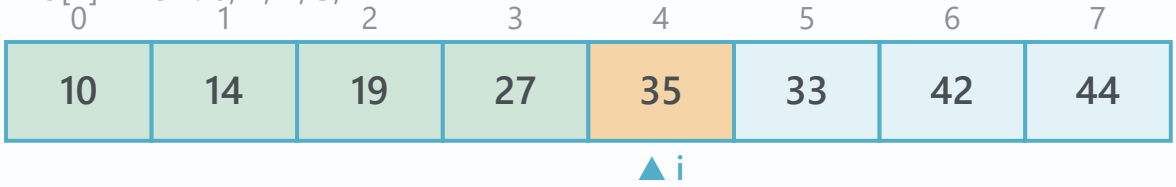
**DEĞİŞKENLER**

`h = 1` `i = 3` `v = 19` `j = 2`

`a = [10, 14, 19, 27, 35, 33, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[4] zinciri: 0, 1, 2, 3, 4




i = 4: v = 35; a[3] ≤ 35, yerinde kaldı.

Adım 17 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

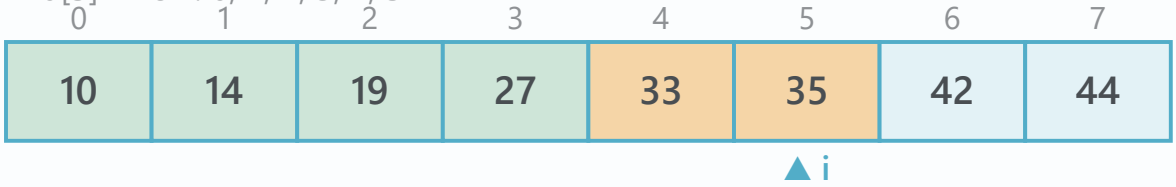
DEĞİŞKENLER

h = 1 | i = 4 | v = 35 | j = 4

a = [10, 14, 19, 27, 35, 33, 42, 44]

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[5] zinciri: 0, 1, 2, 3, 4, 5




i = 5: v = 33; 1 kez 1 geri kaydırıldı, a[4] = 33.

Adım 18 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // h geri kaydır
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

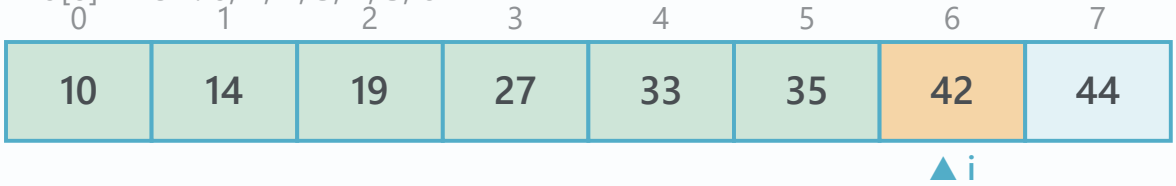
**DEĞİŞKENLER**

`h = 1` `i = 5` `v = 33` `j = 4`

`a = [10, 14, 19, 27, 33, 35, 42, 44]`

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[6] zinciri: 0, 1, 2, 3, 4, 5, 6




i = 6: v = 42; a[5] ≤ 42, yerinde kaldı.

Adım 19 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

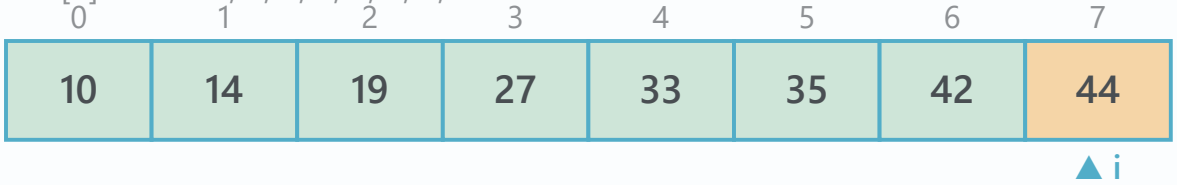
DEĞİŞKENLER

h = 1 | i = 6 | v = 42 | j = 6

a = [10, 14, 19, 27, 33, 35, 42, 44]

# Kod İzleme: `shell_sort(a, 8)`

h = 1 · a[7] zinciri: 0, 1, 2, 3, 4, 5, 6, 7




i = 7: v = 44; a[6] ≤ 44, yerinde kaldı.

Adım 20 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // aralık
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // h geri kaydır
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

**DEĞİŞKENLER**

`h = 1` `i = 7` `v = 44` `j = 7`

`a = [10, 14, 19, 27, 33, 35, 42, 44]`

# Shell Sıralama (Shell Sort, 1959)

- Eklemeli sıralamanın zayıflığı: öğe bir seferde yalnız **bir** konum ilerler; ters sıralıda $O(n^2)$.
- Shell: önce **uzak** öğeler (aralık h) sıralanır, h küçültülür; son tur h = 1 ile biter, bu yüzden sonuç **her zaman doğru**.
- Karmaşıklık aralık dizisine bağlıdır: h = n/2, n/4, … (Shell) için en kötü $O(n^2)$; Knuth'un 1, 4, 13, 40, … dizisi için $O(n^{3/2})$.
- Yerinde; ama uzak takaslar yüzünden **kararlı değildir**.

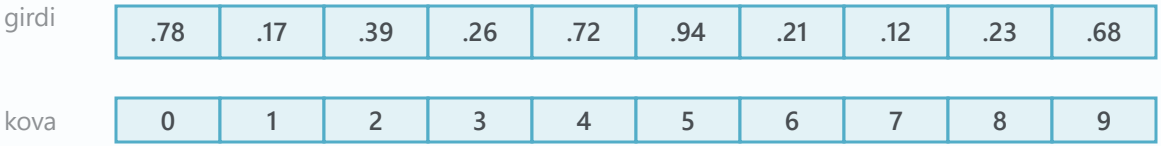
**Birlikte kullanım**

Veriye ait özellikler biliniyorsa shell sort ve paket (kova) sıralama birlikte kullanılabilir.

**Uygulamada**

Kısa kod + iyi pratik hız: gömülü sistemlerde ve küçük/orta dizilerde tercih edilir.

# Kova Sıralama (Bucket Sort)

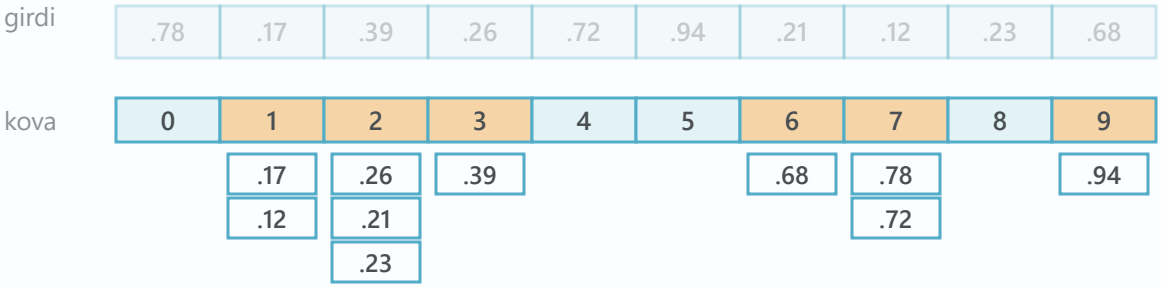



Girdi [0, 1) aralığında 10 sayı; 10 kova, her kova 0,1 genişliğinde.

Adım 1 / 4

Girdi aralıkta **düzgün dağılmışsa** kova başına beklenen öğe sayısı O(1) → ortalama O(n) (CLRS). Tüm öğeler aynı kovaya düşerse en kötü durum, kova içi sıralamanın maliyetidir (eklemeliyle $O(n^2)$). Ek bellek O(n + kova sayısı).

# Kova Sıralama (Bucket Sort)

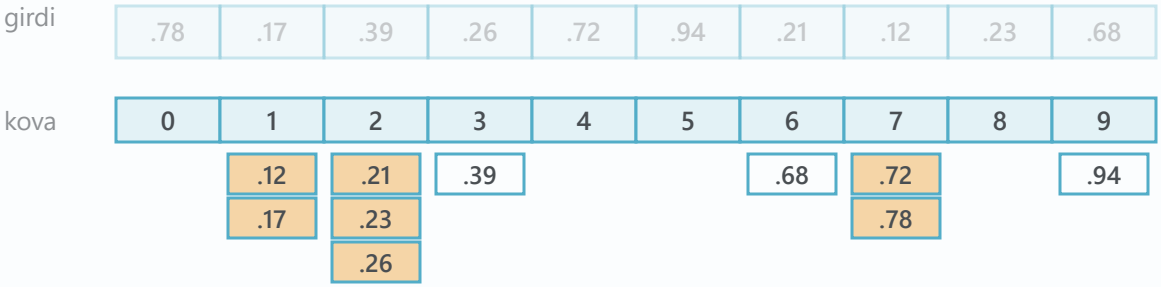



Dağıt: x değeri ⌊10·x⌋ numaralı kovaya eklenir (ör. .78 → 7, .17 → 1). O(n).



Girdi aralıkta **düzgün dağılmışsa** kova başına beklenen öğe sayısı O(1) → ortalama O(n) (CLRS). Tüm öğeler aynı kovaya düşerse en kötü durum, kova içi sıralamanın maliyetidir (eklemeliyle $O(n^2)$). Ek bellek O(n + kova sayısı).

# Kova Sıralama (Bucket Sort)

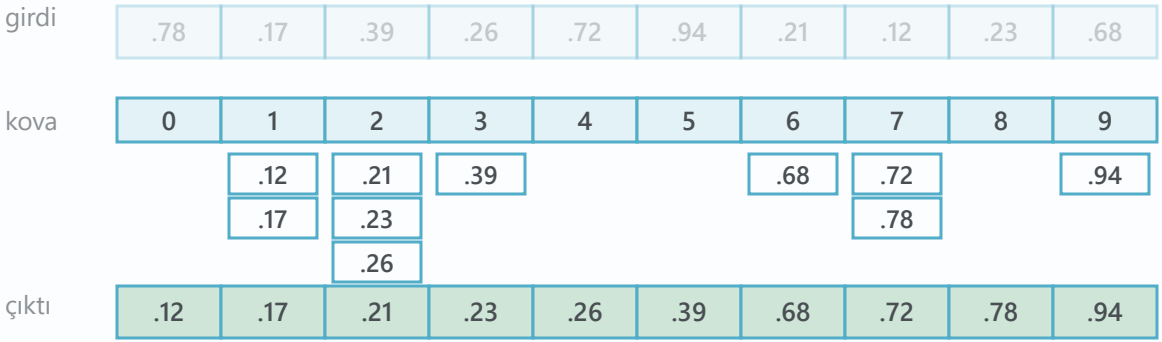



Her kova kendi içinde (eklemeli sıralamayla) sıralanır: kovalar küçük olduğundan ucuz.



Girdi aralıkta **düzgün dağılmışsa** kova başına beklenen öğe sayısı O(1) → ortalama O(n) (CLRS). Tüm öğeler aynı kovaya düşerse en kötü durum, kova içi sıralamanın maliyetidir (eklemeliyle $O(n^2)$). Ek bellek O(n + kova sayısı).

# Kova Sıralama (Bucket Sort)




Kovalar sırayla birleştirilir → sıralı çıktı. Düzgün dağılımda ortalama **O(n)**.



Girdi aralıkta **düzgün dağılmışsa** kova başına beklenen öğe sayısı O(1) → ortalama O(n) (CLRS). Tüm öğeler aynı kovaya düşerse en kötü durum, kova içi sıralamanın maliyetidir (eklemeliyle $O(n^2)$). Ek bellek O(n + kova sayısı).

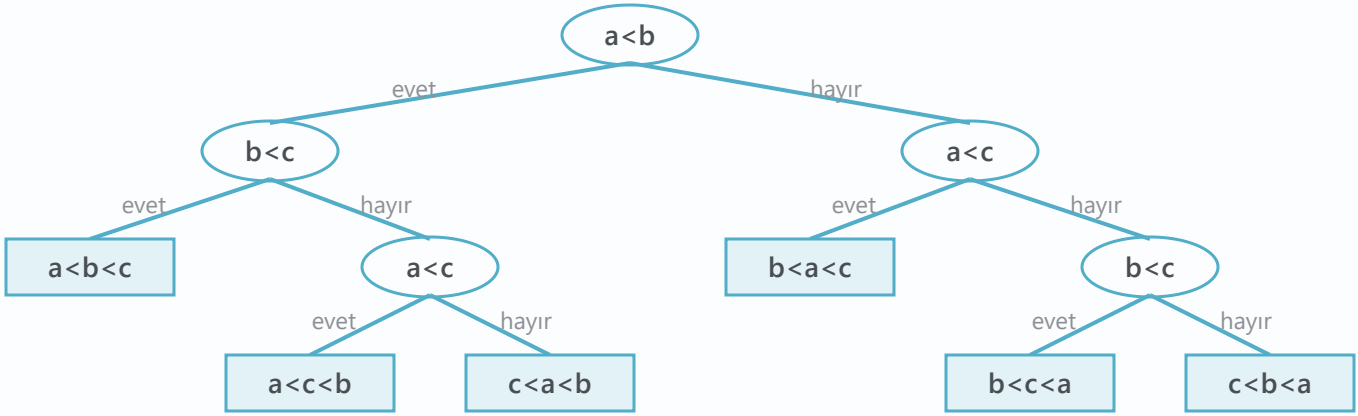
# Kova Sıralama: C ile Gerçekleme

```c
typedef struct kdugum { double x; struct kdugum *bag; } kdugum;

int bucket_sort(double a[], int n) {        /* 0 <= a[i] < 1 */
    kdugum **kova = calloc(n, sizeof *kova);
    kdugum *havuz = malloc(n * sizeof *havuz);
    if (!kova || !havuz) { free(kova); free(havuz); return -1; }
    for (int i = 0; i < n; i++) {           /* dağıt + sıralı ekle */
        kdugum **p = &kova[(int)(a[i] * n)];
        while (*p && (*p)->x <= a[i]) p = &(*p)->bag;
        havuz[i].x = a[i]; havuz[i].bag = *p; *p = &havuz[i];
    }
    int k = 0;                              /* kovaları birleştir */
    for (int b = 0; b < n; b++)
        for (kdugum *q = kova[b]; q; q = q->bag) a[k++] = q->x;
    free(kova); free(havuz);
    return 0;
}
```

- n kova; `a[i]` değeri `⌊a[i]·n⌋` numaralı bağlı listeye **sıralı** eklenir (kova içi eklemeli sıralama).
- `≤` sayesinde eşitler sona eklenir → kararlı. Düğümler tek bir `havuz` dizisinden alınır: n ayrı `malloc` yerine iki ayırma, tek `free` çifti.

# Optimal Sıralama Zamanı: Karar Ağacı

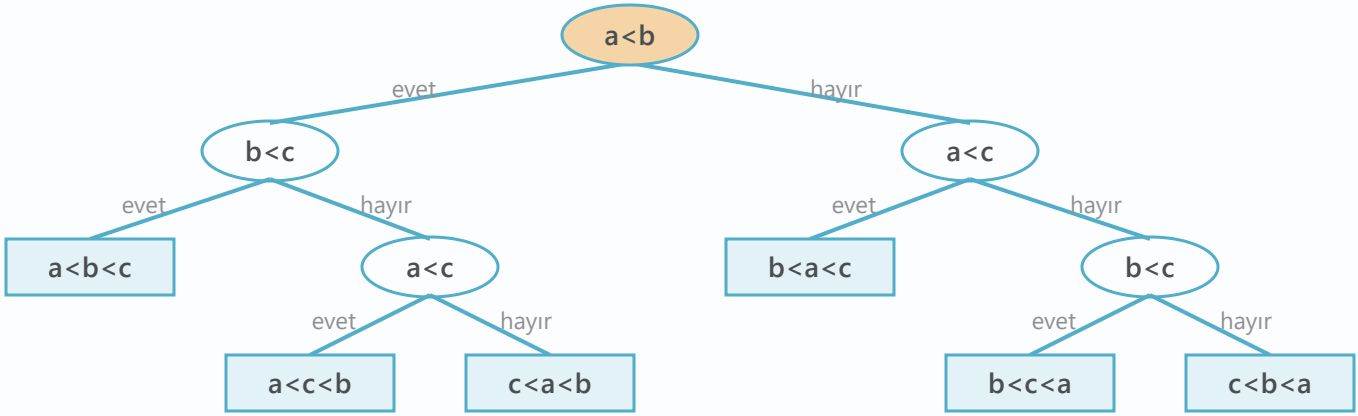



Sınamalar elips, kararlar (sıralamalar) dikdörtgen. n = 3 için yaprak sayısı 3! = 6.



Karşılaştırmalı her sıralama algoritması bir karar ağacıdır: her iç düğüm bir `kucuk()` çağrısı, her yaprak girdinin bir dizilişi. En kötü durumdaki karşılaştırma sayısı = ağacın **yüksekliği**.

# Optimal Sıralama Zamanı: Karar Ağacı

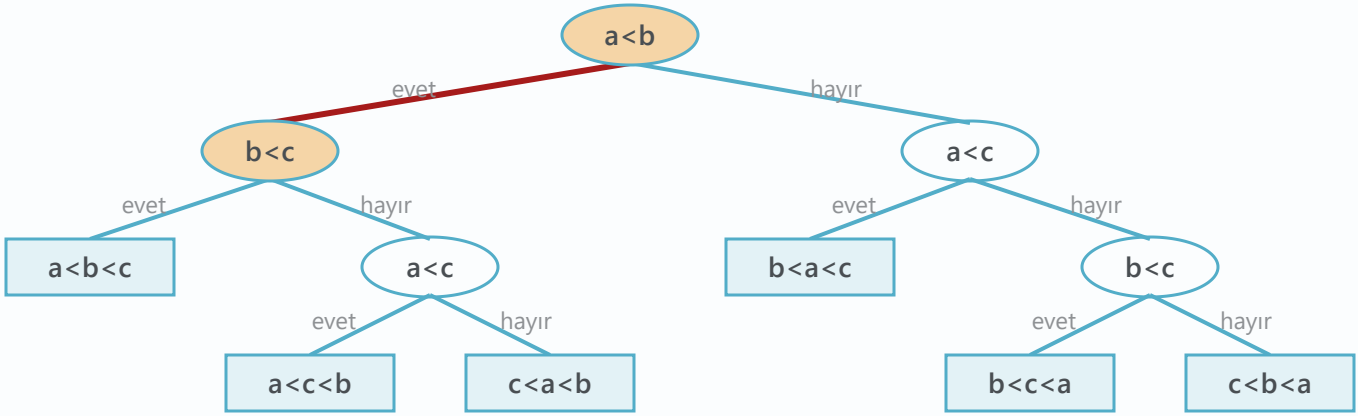



Girdi a = 5, b = 8, c = 2. Kök: a < b? 5 < 8 → **evet**.

Adım 2 / 5

Karşılaştırmalı her sıralama algoritması bir karar ağacıdır: her iç düğüm bir `kucuk()` çağrısı, her yaprak girdinin bir dizilişi. En kötü durumdaki karşılaştırma sayısı = ağacın **yüksekliği**.

# Optimal Sıralama Zamanı: Karar Ağacı

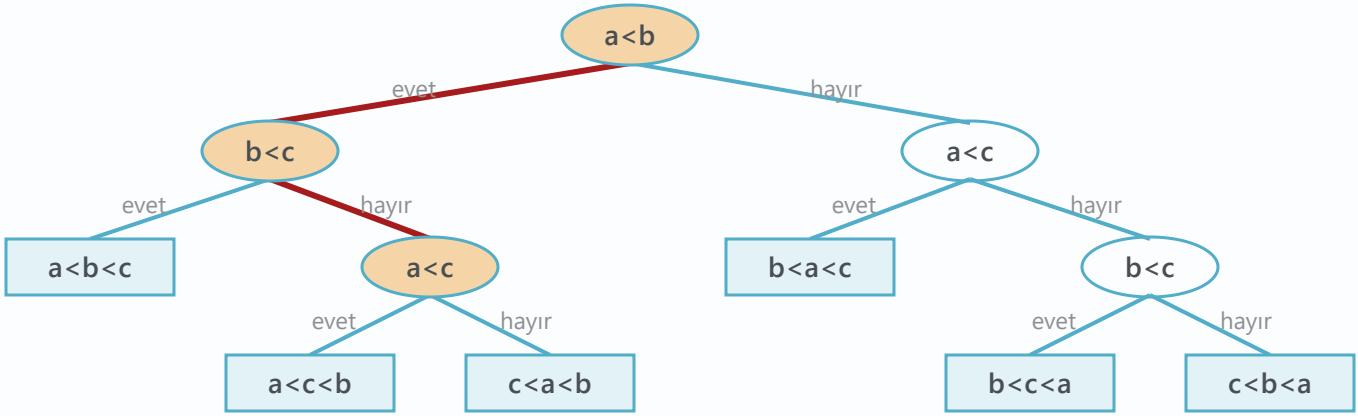



b < c? 8 < 2 → **hayır**: c, b'den önce gelmeli.



Karşılaştırmalı her sıralama algoritması bir karar ağacıdır: her iç düğüm bir `kucuk()` çağrısı, her yaprak girdinin bir dizilişi. En kötü durumdaki karşılaştırma sayısı = ağacın **yüksekliği**.

# Optimal Sıralama Zamanı: Karar Ağacı

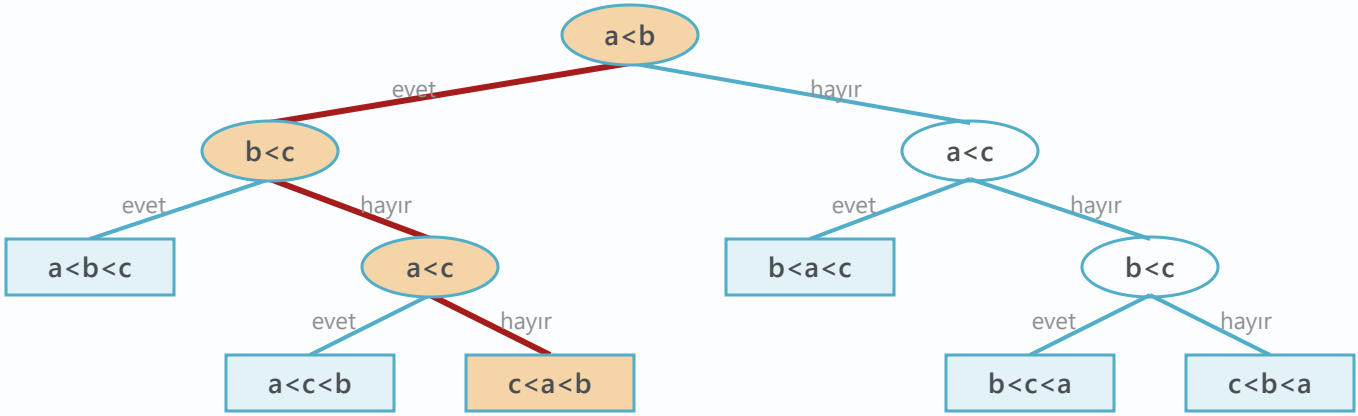



a < c? 5 < 2 → **hayır**.



Karşılaştırmalı her sıralama algoritması bir karar ağacıdır: her iç düğüm bir `kucuk()` çağrısı, her yaprak girdinin bir dizilişi. En kötü durumdaki karşılaştırma sayısı = ağacın **yüksekliği**.

# Optimal Sıralama Zamanı: Karar Ağacı




Yaprak **c < a < b** → 2, 5, 8. Bu girdi için 3 karşılaştırma = yolun uzunluğu.



Karşılaştırmalı her sıralama algoritması bir karar ağacıdır: her iç düğüm bir `kucuk()` çağrısı, her yaprak girdinin bir dizilişi. En kötü durumdaki karşılaştırma sayısı = ağacın **yüksekliği**.

# Teorem: Karşılaştırmalı Sıralamanın Alt Sınırı

**Teorem.** n farklı öğeyi sıralayan her karar ağacının derinliği (düzey sayısı) en az $\lceil \log_2(n!) \rceil + 1$'dir; yani en kötü durumda en az $\lceil \mathbf{lg(n!)} \rceil$ **karşılaştırma** gerekir.

- Yaprak sayısı en az n! (her diziliş bir yaprak).
- Yüksekliği h olan ikili ağacın en çok $2^h$ yaprağı vardır: $2^h \geq n! \Rightarrow h \geq \lg(n!)$.
- Stirling: $\lceil \lg(n!) \rceil = n \lg n - n \lg e + \Theta(\lg n) \approx$ **$n \lg n - 1{,}4427\, n$**.
- Sonuç: karşılaştırmalı sıralama **$\Omega(n \lg n)$**; sayarak/kova/basamak sıralama karşılaştırma yapmadığı için bu sınıra takılmaz.

| n | $\lceil \lg n! \rceil$ | $n \lg n - 1{,}44n$ | $n(n-1)/2$ |
|---|---|---|---|
| 3 | 3 | 0,4 | 3 |
| 5 | 7 | 4,4 | 10 |
| 10 | 22 | 18,8 | 45 |
| 20 | 62 | 57,6 | 190 |
| 100 | 525 | 520,1 | 4950 |

**HOCADAN** “Bundan sonra yazacağımız algoritmaları karşılaştırmak için bir mihenk taşı verdik.”

ÖZET

# Aynı Girdide Sayım (n = 1000)

| algoritma | sıralı: karş. / taşıma | ters: karş. / taşıma | rastgele: karş. / taşıma |
|---|---|---|---|
| Seçimli | 499.500 / 999 | 499.500 / 999 | 499.500 / 999 |
| Eklemeli | 999 / 0 | 499.500 / 499.500 | 247.876 / 246.884 |
| Kabarcık (erken dur.) | 999 / 0 | 499.500 / 499.500 | 499.035 / 246.884 |
| Shell (n/2, n/4, …) | 8.006 / 0 | 11.716 / 4.700 | 14.760 / 7.254 |

Değerler Python simülasyonuyla sayıldı (rastgele girdi: tohum 19). Taşıma: seçimlide takas, eklemeli/Shell'de kaydırma, kabarcıkta takas.

**Seçimli**

Girdiye kör: her durumda 499.500 karşılaştırma, 999 takas.

**Eklemeli**

Sıralıda 999 karşılaştırma; rastgelede ≈ $n^2/4$; tersinde $n^2/2$.

**Shell**

Rastgele girdide eklemeliden kat kat az iş: uzak kaydırmalar evrikleri hızla yok eder.

ÖZET

# Karmaşıklık Özeti

| algoritma | en iyi | ortalama | en kötü | ek bellek | kararlı | yerinde |
|---|---|---|---|---|---|---|
| Seçimli | $n^2/2$ | $n^2/2$ | $n^2/2$ | O(1) | hayır | evet |
| Eklemeli | O(n) | $\approx n^2/4$ | $n^2/2$ | O(1) | evet | evet |
| Kabarcık (erken dur.) | O(n) | $\approx n^2/2$ karş. | $n^2/2$ | O(1) | evet | evet |
| Shell (Knuth h) | O(n lg n) | — | $O(n^{3/2})$ | O(1) | hayır | evet |
| Sayarak | O(n+k) | O(n+k) | O(n+k) | O(n+k) | evet | hayır |
| Kova (düzgün dağ.) | O(n) | O(n) | $O(n^2)$ | O(n) | evet* | hayır |

Seçimli/eklemeli/kabarcık satırlarında sayılar karşılaştırma sayısıdır. * Kova içinde kararlı bir sıralama kullanılırsa. Shell ortalaması aralık dizisine bağlı ve kapalı biçimde bilinmiyor. Karşılaştırmalı sıralamada alt sınır $\lceil \lg n! \rceil \approx n \lg n$.

ÖZET

# Hangi Durumda Hangisi?

**Takas/yazma pahalı**

**Seçimli**: tam n − 1 takas.

**Küçük ya da neredeyse sıralı**

**Eklemeli**: kararlı, yerinde, sıralıya yakında O(n); hızlı sıralamanın küçük parçalarında da kullanılır.

**Öğretim / sıralılık testi**

**Kabarcık** + erken durma: bir turda sıralı olduğunu anlar; üretimde nadiren.

**Küçük tamsayı aralığı**

**Sayarak**: O(n + k), kararlı; k ≫ n ise bellek ve zaman boşa gider.

**Orta boy, ek bellek yok**

**Shell**: birkaç satırlık kodla $O(n^2)$'den belirgin hızlı.

**[0,1) düzgün dağılım**

**Kova**: ortalama O(n).

ÖZET

# Sık Yapılan Hatalar

1. Seçimlide en küçük **değeri** tutmak: takas için **indisi** (`min`) gerekir.
2. Eklemelide `v = a[i]`'yi kaydırmadan önce saklamamak: ilk kaydırma a[i]'yi ezer.
3. `j ≥ 0` yazmak: alt dizi a[L..R] ise sınır `j ≥ L`.
4. `if (kucuk(A, B)) SWAP(A, B)` kalıbında yönü karıştırmak: A < B ise takas edilir.
5. Sayarakta k'yı unutmak: negatif ya da k'dan büyük değer `c[]` dışına yazar.
6. Sayarakta L4'ü soldan sağa yazmak: sıralar ama **kararlılık** bozulur.

ÖZET

# Alıştırmalar

1. [5, 1, 4, 2, 8] için seçimli sıralamada her tur sonundaki diziyi yazın. Kaç takas yapıldı?
2. Eklemeli sıralama [1, 2, 3, …, n] ve [n, …, 2, 1] için kaç karşılaştırma yapar?
3. `bubble2` [1, 2, 3, 5, 4] girdisinde kaç tur çalışır?
4. a = [2, 5, 3, 0, 2, 3, 0, 3], k = 5 için counting sort'un L3 sonrası c dizisini yazın.
5. 3 öğe için karar ağacının en az yüksekliği nedir? 4 öğe için?
6. Seçimli sıralamanın kararsız olduğunu gösteren en küçük örneği bulun.

▶ **Cevapları göster**

ÖZET

# Bölüm Özeti

**Üç $O(n^2)$ algoritma**

Aynı sınıfta ama farklı karakter: seçimli az takas, eklemeli sıralıya yakında hızlı, kabarcık erken durabilir.

**Ortalama durum**

Tüm n! dizilişin beklenen değeri: eklemeli $\approx n^2/4$ karşılaştırma. En iyi ile en kötünün ortalaması değildir!

**Karşılaştırmasız**

Sayarak O(n+k) ve kova ortalama O(n): veri hakkında bilgi (aralık, dağılım) hız kazandırır.

**Kavramlar**

Kararlılık (eşitlerin sırası) ve yerindelik (ek bellek) algoritma seçiminde belirleyicidir.

**Shell**

Aralıklı eklemeli sıralama; son tur h = 1 doğruluğu garanti eder.

**Alt sınır**

Karşılaştırmalı sıralama $\lceil \lg n! \rceil \approx n \lg n$ karşılaştırmanın altına inemez → ileri sıralama algoritmalarının mihenk taşı.

ÖZET

# Kaynaklar ve Devam

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Bölüm 7 (Sorting).
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 6 (Elementary Sorting Methods): makro yaklaşımı, shell sort.
- T. H. Cormen vd., *Introduction to Algorithms* (CLRS), Bölüm 2 (insertion), 8 (alt sınır, counting, bucket).

← Bölüm 18: En Kısa Yol ve Aktivite Ağları | Ders sayfası | Bölüm 20: İleri Sıralama Algoritmaları →

# Son

## Bölüm 19: Temel Sıralama Algoritmaları

Sonraki bölüm: Bölüm 20: İleri Sıralama Algoritmaları · Ders sayfası