Bölüm 20: İleri Sıralama Algoritmaları

C ile Veri Yapıları ve Algoritmalar

Quick sort | Ortalama durum | Özyinelemesiz | Medyan-3 | qsort | Merge sort | Heap sort | Radix sort

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Hızlı sıralama

Bölümleme (*partition*) değişmezini açıklamak; `bol()` ve özyinelemeyi adım adım izlemek.

Analiz

En iyi n lg n, en kötü n^2, ortalama ≈ 1,39 n lg n sonuçlarını özyinelemeli bağıntılardan elde etmek.

İyileştirmeler

Özyinelemesiz sürüm (O(lg n) yığın), medyan-3, küçük parçada eklemeli; C'de `qsort`.

Birleştirme sıralaması

merge işlemini, yığın çerçevelerini ve bağlı (indisli) sürümü izlemek; O(n) ek bellek.

Heap sort

Aşağıdan yukarıya O(n)'de heap kurup yerinde O(n lg n) sıralamak.

Basamağa göre

LSD radix sort'u kovalı bağlı listelerle uygulamak ve karmaşıklığını yorumlamak.

Hızlı (Çabuk) Sıralama: Quick Sort

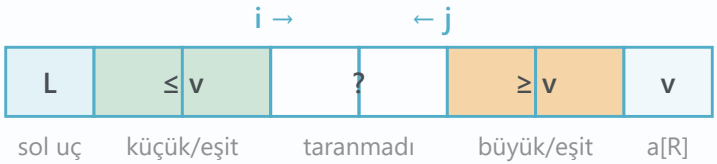
- 1962'de **C. A. R. (Tony) Hoare** tarafından geliştirildi.
- **Parçala ve yönet** (*divide and conquer*): problem doğrudan çözülecek kadar küçülene dek özyinelemeli olarak alt problemlere bölünür.
- Ortalama n lg n ile orantılı zaman; en kötü durumda n^2. Yalnızca küçük bir yardımcı **yığın** kullanır.
- **Kararlı değildir**; yine de bilinen en hızlı genel amaçlı algoritmalardan biridir. Standart C kitaplığında `qsort` vardır.

```
int bol(int a[], int L, int R);

void quicksort1(int a[], int L, int R) {
    if (R <= L) return;
    int i = bol(a, L, R);
    quicksort1(a, L, i - 1);
    quicksort1(a, i + 1, R);
}
```

HOCADAN "Bu kod, Sedgewick'in kitabından alınmıştır. Oldukça güzel yazılmış bir kod. (Sedgewick, Knuth'un doktora öğrencisidir; quick sort üzerinde fazlaca çalışmıştır.)"

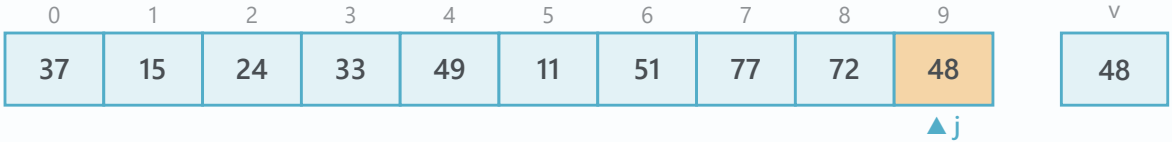
`bol()`: Bölümleme (Partition)




```
int bol(int a[], int L, int R) {
    int i = L - 1, j = R, v = a[R]; // pivot a[R]
    for (;;) {
        while (kucuk(a[++i], v))
            ;
        while (kucuk(v, a[--j]))
            if (j == L) break;
        if (i >= j) break;
        SWAP(a[i], a[j]);
    }
    SWAP(a[i], a[R]);
    return i;
}
```

- [2] Pivot (*partitionary element*) `v = a[R]`.
- [4] i soldan, v'den **büyük ya da eşit** ilk öğede durur. Sınır sınaması gerekmez: en kötü i = R'de `a[R] = v` durdurur.
- [6–7] j sağdan, v'den **küçük ya da eşit** öğede durur; bu döngüde taşmayı `j == L` önler.
- [9] İki öğe yanlış taraftadır → takas. [8] i ile j buluşunca döngü biter; [11] pivot a[i]'ye: solu ≤ v ≤ sağı.

Kod İzleme: `bol(a, 0, 9)`



Pivot **v = a[9] = 48**. i = L − 1 = −1, j = R = 9: taramalar ++i ve --j ile başlayacak.

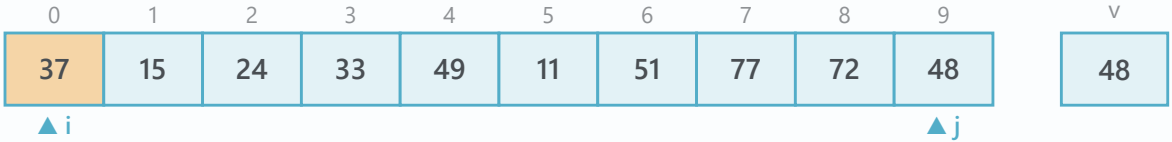
Adım 1 / 15

```
1  int bol(int a[], int L, int R) {
▶      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`L = 0` `R = 9` `i = -1` `j = 9` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[0] = 37 < 48 → i ilerlemeye devam.

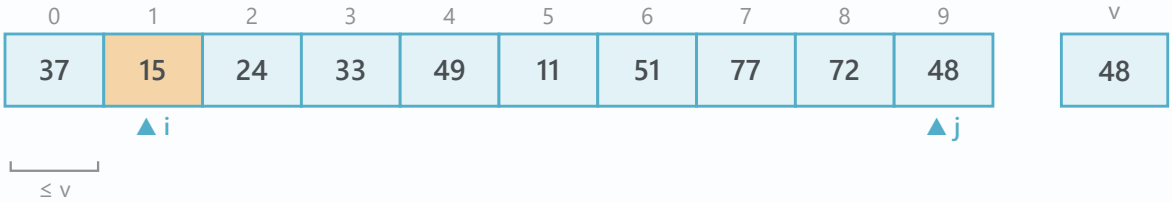
Adım 2 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 0` `j = 9` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[1] = 15 < 48 → i ilerlemeye devam.

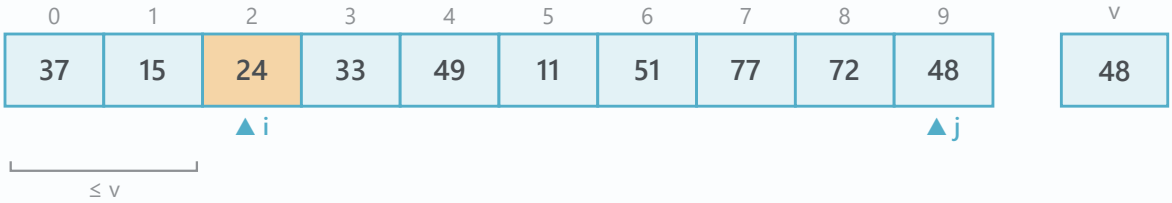
Adım 3 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 1` `j = 9` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[2] = 24 < 48 → i ilerlemeye devam.

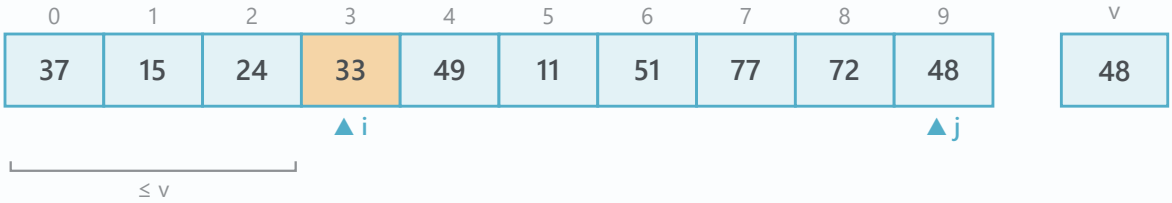
Adım 4 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 2` `j = 9` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[3] = 33 < 48 → i ilerlemeye devam.

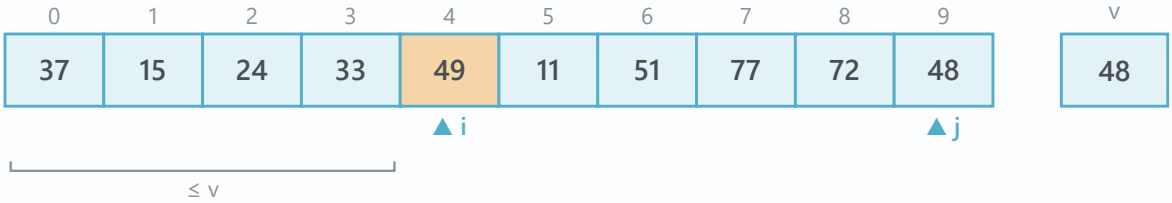
Adım 5 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 3` `j = 9` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[4] = 49 ≥ 48 → **i = 4**'te durdu (sağ tarafa ait bir öğe).

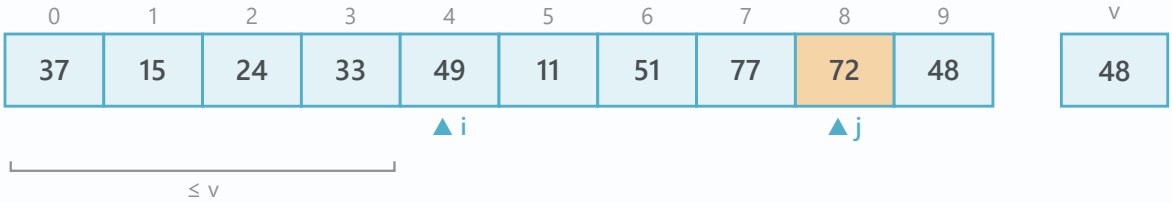
Adım 6 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 4` `j = 9` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[8] = 72 > 48 → j sola ilerliyor.

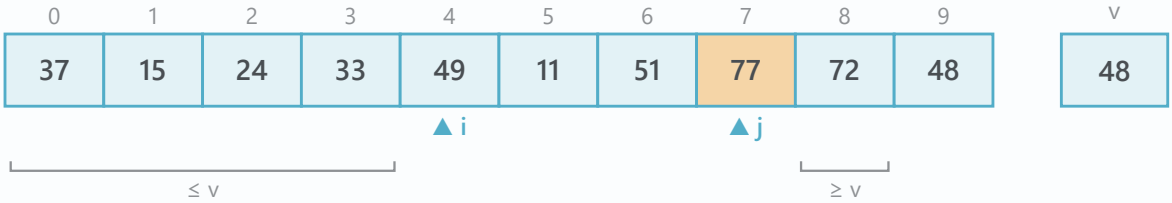
Adım 7 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
▶          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 4` `j = 8` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[7] = 77 > 48 → j sola ilerliyor.

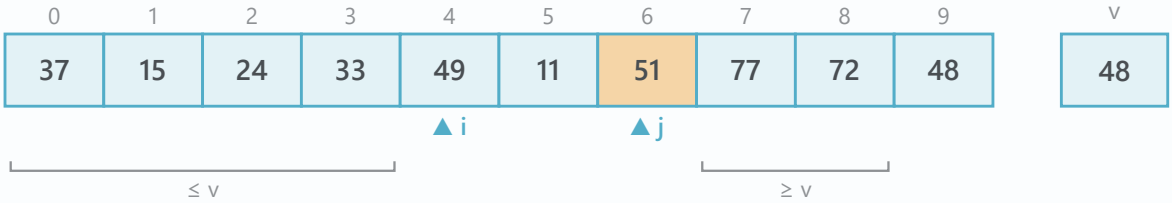
Adım 8 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
▶          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 4` `j = 7` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[6] = 51 > 48 → j sola ilerliyor.

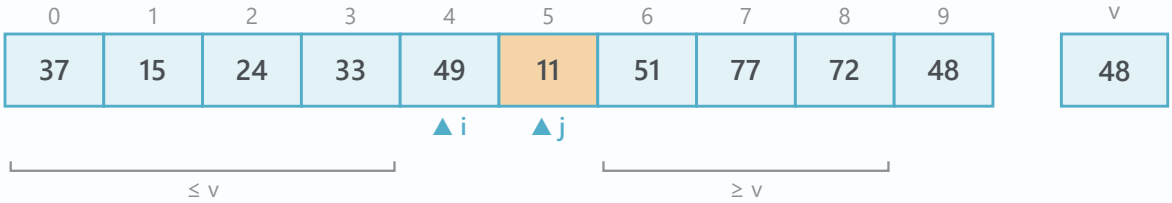
Adım 9 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
▶          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 4` `j = 6` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[5] = 11 ≤ 48 → **j = 5**'te durdu.

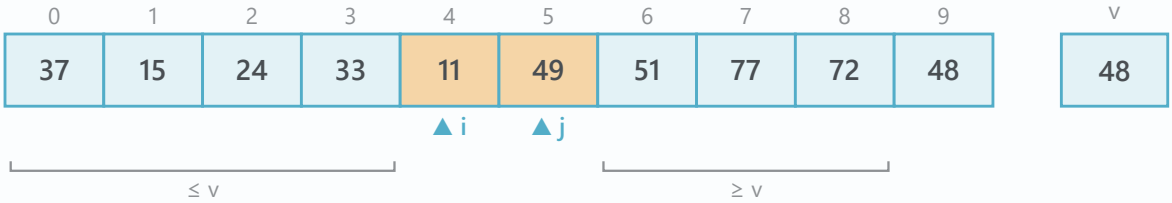
Adım 10 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
▶          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 4` `j = 5` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



i < j: a[4] ile a[5] yanlış tarafta → takas: [37, 15, 24, 33, 11, 49, 51, 77, 72, 48].

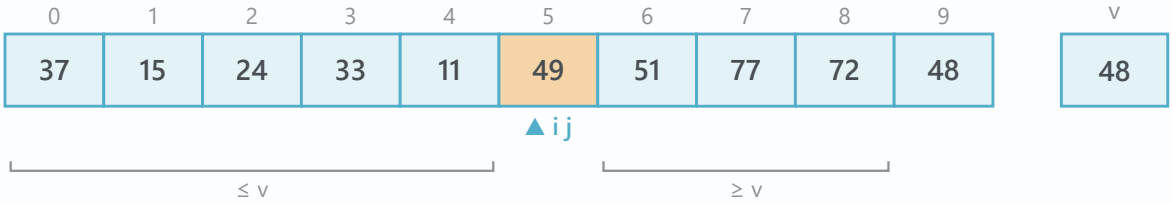
Adım 11 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
▶          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 4` `j = 5` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[5] = 49 ≥ 48 → **i = 5**'te durdu (sağ tarafa ait bir öğe).

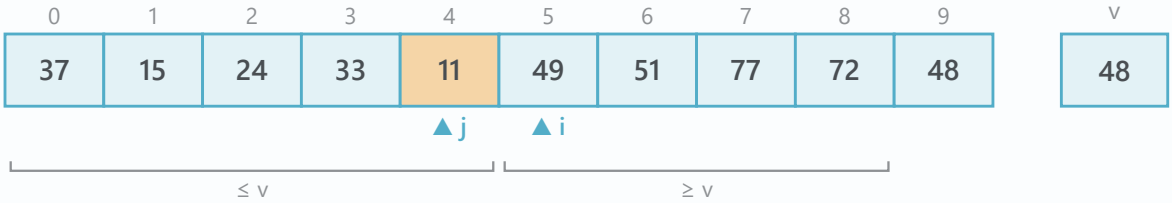
Adım 12 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 5` `j = 5` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



a[4] = 11 ≤ 48 → **j = 4**'te durdu.

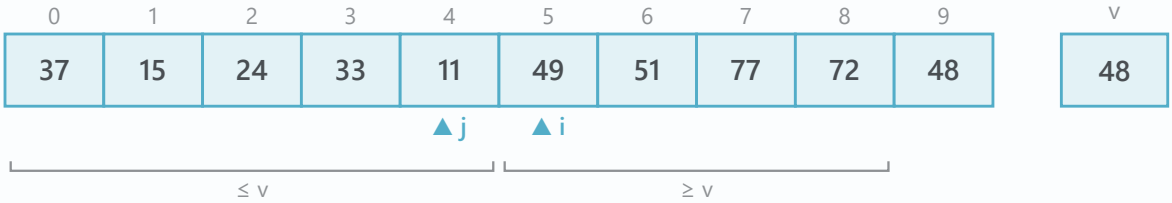
Adım 13 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
▶          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 5` `j = 4` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



i = 5 ≥ j = 4: indisler çaprazlandı → döngüden çık.

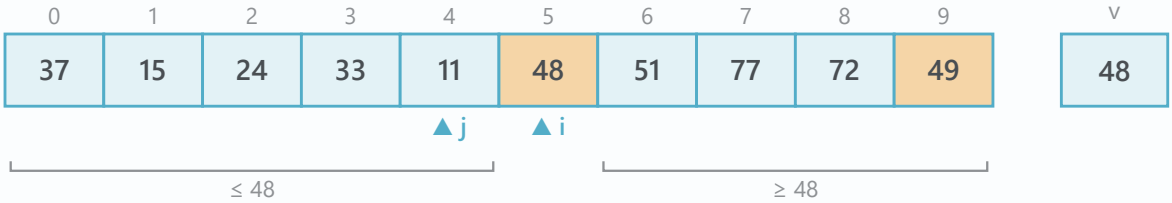
Adım 14 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
▶          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

DEĞİŞKENLER

`i = 5` `j = 4` `v = 48`

Kod İzleme: `bol(a, 0, 9)`



Pivot a[5]'e taşındı ve **5 döndürüldü**. Veriler sıralansaydı 48 tam burada olurdu.

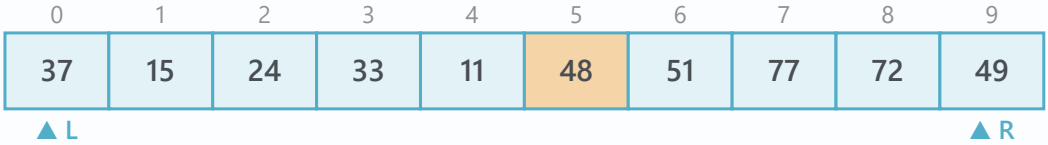
Adım 15 / 15

```
1  int bol(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (kucuk(a[++i], v))
5              ;
6          while (kucuk(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
▶      SWAP(a[i], a[R]);
▶      return i;
13 }
```

DEĞİŞKENLER

`i = 5` `j = 4` `v = 48`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(0, 9): pivot 48 → konum **5**. Sol a[0..4], sağ a[6..9] ayrı sıralanacak.

Adım 1 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

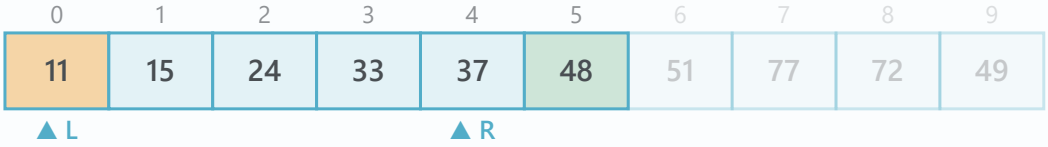
DEĞİŞKENLER

`L = 0` `R = 9` `i = 5`

ÇAĞRI YIĞINI

- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(0, 4): pivot 11 → konum **0**. Sol a[0..-1], sağ a[1..4] ayrı sıralanacak.

Adım 2 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

DEĞİŞKENLER

L = 0 | R = 4 | i = 0

ÇAĞRI YIĞINI

- **quicksort1(0, 4)**
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	51	77	72	49

quicksort1(0, -1): boş aralık → dön.

Adım 3 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

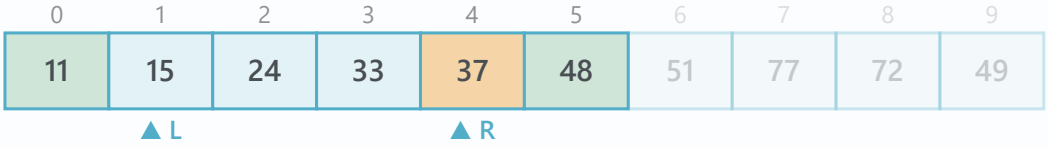
DEĞİŞKENLER

L = 0 R = -1

ÇAĞRI YIĞINI

- `quicksort1(0, -1)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(1, 4): pivot 37 → konum **4**. Sol a[1..3], sağ a[5..4] ayrı sıralanacak.

Adım 4 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

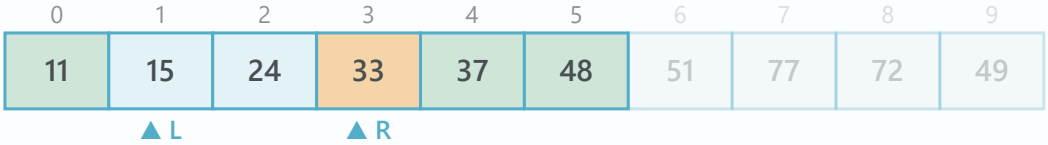
DEĞİŞKENLER

`L = 1` `R = 4` `i = 4`

ÇAĞRI YIĞINI

- **`quicksort1(1, 4)`**
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(1, 3): pivot 33 → konum **3**. Sol a[1..2], sağ a[4..3] ayrı sıralanacak.

Adım 5 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

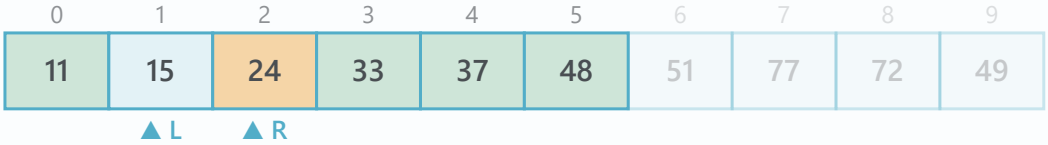
DEĞİŞKENLER

L = 1 | R = 3 | i = 3

ÇAĞRI YIĞINI

- `quicksort1(1, 3)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(1, 2): pivot 24 → konum **2**. Sol a[1..1], sağ a[3..2] ayrı sıralanacak.

Adım 6 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

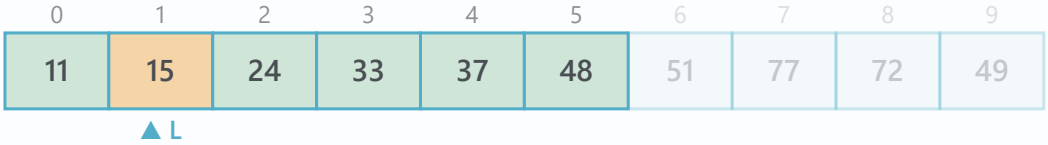
DEĞİŞKENLER

L = 1 | R = 2 | i = 2

ÇAĞRI YIĞINI

- **quicksort1(1, 2)**
- quicksort1(1, 3)
- quicksort1(1, 4)
- quicksort1(0, 4)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



quicksort1(1, 1): tek öğe → zaten yerinde, dön.

Adım 7 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

DEĞİŞKENLER

L = 1 R = 1

ÇAĞRI YIĞINI

- `quicksort1(1, 1)`
- `quicksort1(1, 2)`
- `quicksort1(1, 3)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	51	77	72	49

quicksort1(3, 2): boş aralık → dön.

Adım 8 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

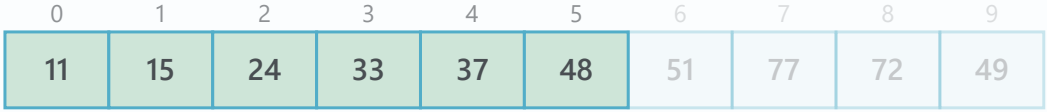
DEĞİŞKENLER

L = 3 R = 2

ÇAĞRI YIĞINI

- **quicksort1(3, 2)**
- quicksort1(1, 2)
- quicksort1(1, 3)
- quicksort1(1, 4)
- quicksort1(0, 4)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



quicksort1(4, 3): boş aralık → dön.

Adım 9 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

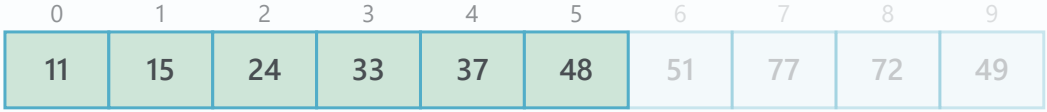
DEĞİŞKENLER

L = 4 R = 3

ÇAĞRI YIĞINI

- **quicksort1(4, 3)**
- quicksort1(1, 3)
- quicksort1(1, 4)
- quicksort1(0, 4)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



quicksort1(5, 4): boş aralık → dön.

Adım 10 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

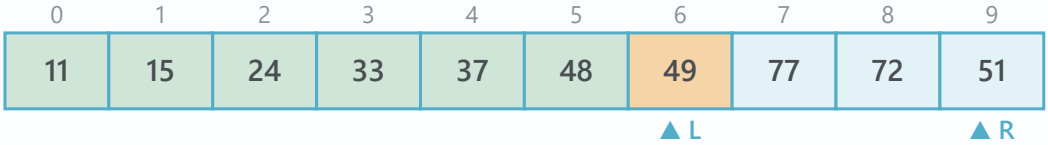
DEĞİŞKENLER

L = 5 R = 4

ÇAĞRI YIĞINI

- **quicksort1(5, 4)**
- quicksort1(1, 4)
- quicksort1(0, 4)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(6, 9): pivot 49 → konum **6**. Sol a[6..5], sağ a[7..9] ayrı sıralanacak.

Adım 11 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

DEĞİŞKENLER

L = 6 | R = 9 | i = 6

ÇAĞRI YIĞINI

- **quicksort1(6, 9)**
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	49	77	72	51

quicksort1(6, 5): boş aralık → dön.

Adım 12 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

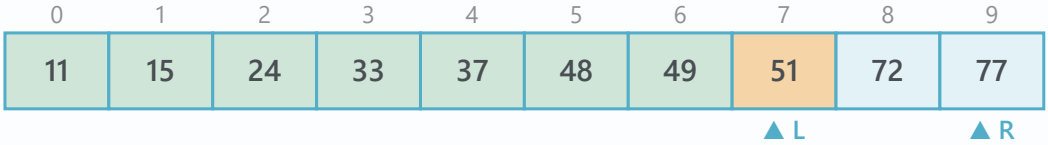
DEĞİŞKENLER

L = **6** R = **5**

ÇAĞRI YIĞINI

- **quicksort1(6, 5)**
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(7, 9): pivot 51 → konum **7**. Sol a[7..6], sağ a[8..9] ayrı sıralanacak.

Adım 13 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

DEĞİŞKENLER

L = 7 | R = 9 | i = 7

ÇAĞRI YIĞINI

- `quicksort1(7, 9)`
- `quicksort1(6, 9)`
- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	49	51	72	77

quicksort1(7, 6): boş aralık → dön.

Adım 14 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

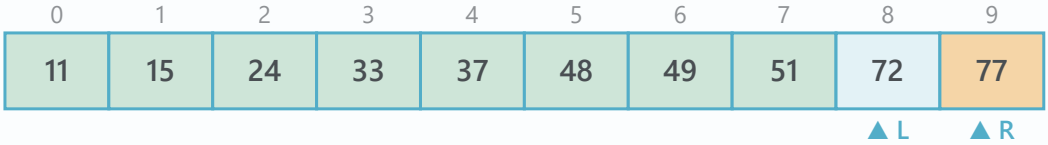
DEĞİŞKENLER

L = **7** R = **6**

ÇAĞRI YIĞINI

- **quicksort1(7, 6)**
- quicksort1(7, 9)
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



bol(8, 9): pivot 77 → konum **9**. Sol a[8..8], sağ a[10..9] ayrı sıralanacak.

Adım 15 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

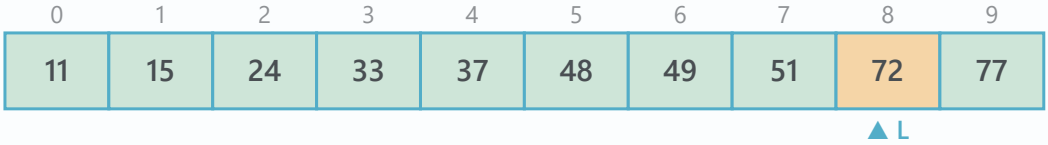
DEĞİŞKENLER

`L = 8` `R = 9` `i = 9`

ÇAĞRI YIĞINI

- **`quicksort1(8, 9)`**
- `quicksort1(7, 9)`
- `quicksort1(6, 9)`
- `quicksort1(0, 9)`
- `main()`

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını



quicksort1(8, 8): tek öğe → zaten yerinde, dön.

Adım 16 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

DEĞİŞKENLER

L = **8** R = **8**

ÇAĞRI YIĞINI

- **quicksort1(8, 8)**
- quicksort1(8, 9)
- quicksort1(7, 9)
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Kod İzleme: `quicksort1(a, 0, 9)` — Çağrı Yığını

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	49	51	72	77

quicksort1(10, 9): boş aralık → dön.

Adım 17 / 17

```
1 int bol(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = bol(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

DEĞİŞKENLER

L = 10 R = 9

ÇAĞRI YIĞINI

- **quicksort1(10, 9)**
- quicksort1(8, 9)
- quicksort1(7, 9)
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Değerlendirme: Neden Hızlı?

Doğru konum + küçülen problem

Kabarcık ve seçimli de her adımda bir öğeyi doğru konuma koyar; ama sonraki adımda alt problem n − 1 kalır. Hızlı sıralama hem pivotu yerleştirir hem de problemi **iki küçük parçaya** böler.

Özyineleme ağacı

`quicksort1(a, 0, 9)` izinde 10 öğe için 17 çağrı: her iç çağrı bir pivotu kesin yerine koydu (izde yeşil hücreler).

HOCADAN "Denilebilir ki, kabarcık ya da seçimli sıralamada da her adımda bir öğe için doğru konum bulunuyordu. Evet! Onlar da buluyor, ama problemi küçülterek sonraki adıma katkı yapamıyorlar."

HOCADAN "Quick sort'la ilgili üç-dört algoritma yazacağım. O yüzden bunun anlaşılması önemli. [...] Küçük kodlar defalarca, farklı şekillerde yazılmalı."

Okuma Parçası: Algoritma ve Bilgisayar

HOCADAN "— Hocam! Adam bu quick sort'u 1960'larda nerde denemiş, nasıl test etmiş?
Algoritma kelimesi nerden geliyor? 8. yüzyılda el-Harezmî'nin 4GB'lık laptopu mu vardı? ☺"

HOCADAN "Algoritma demek; hiç bilmeyen adam onu alacak, adım adım işletecek ve doğru sonucu bulacak, demek. Kaldı ki, 1960'larda bilgisayar zaten vardı. Fakat bizim dersimiz, bilgisayardan bağımsızdır. Zaten bir algoritmanın makalesinin çıkması yeterlidir."

El-Harezmî (780–850)

Bağdat'taki Beytü'l-Hikme'de çalışan matematikçi ve gökbilimci. Birinci ve ikinci dereceden denklemlerin sistematik çözümünü veren *el-Kitâbü'l-Muhtasar fî Hisâbi'l-Cebr ve'l-Mukâbele* kitabının yazarı.

Kelimelerin kökeni

"Cebir/algebra" bu kitaptaki *el-cebr* yönteminden; "algoritma" ise adının Latince biçimi *Algoritmi*'den gelir. Onluk konumsal sayı sistemini Batı'ya tanıtan eserleriyle bilinir.

Quick Sort: En Kötü ve En İyi Durum

En kötü: sıralı dizi

Pivot a[R] her seferinde en büyüktür, yerinde kalır; sonraki çağrı bir eksik boyutla yapılır:
$n + (n-1) + \ldots + 1 = n(n+1)/2 = O(n^2)$

En iyi: tam ortadan bölme

Her adımda i ve j toplam n adım ilerler, n/2'lik iki alt problem kalır:
$C_n = 2C_{n/2} + n = 2^k C_{n/2^k} + k \cdot n = n \lg n$

HOCADAN "Şimdi "Bunun neresi hızlı?" diye düşünülecek."

En iyi durum n lg n; optimal sıralama sınırı n lg n − 1,44n idi. Yani katsayı yine 1, ama 1,44n kadar fazlalık var.

Sayım: Sıralı ve Rastgele Girdide quicksort1

n	sıralı: karşılaştırma	derinlik	rastgele: karşılaştırma	derinlik	n lg n
10	63	9	36	5	33
100	5.148	99	721	9	664
1000	501.498	999	12.831	25	9.966

bol() içindeki karşılaştırmalar Python simülasyonuyla sayıldı (rastgele girdi: tohum 20).

Sıralı girdi

Karşılaştırma ≈ $n^2/2$ ve özyineleme derinliği n − 1: C'de büyük n için **yığın taşması** riski.

Rastgele girdi

Karşılaştırma n lg n mertebesinde, derinlik yalnızca birkaç lg n.

Ortalama Durum Analizi: Özyinelemeli Bağıntı

$$C_n = (n+1) + (1/n)\,\Sigma_{k=1}^{n}\,(C_{k-1} + C_{n-k}),\quad C_0 = C_1 = 0$$

- Bölme noktası k, 1 ≤ k ≤ n arasında **eşit olasılıklı**: n! dizilişin hepsinde bol() çağrılınca frekanslar eşit çıkar.
- n + 1: bol()'da pivotla yapılan karşılaştırmalar; k'da bölünürse (k−1) ve (n−k) öğeli iki alt dizi kalır.
- İki toplam aynı terimleri içerir: $\Sigma(C_{k-1} + C_{n-k}) = 2\,\Sigma_{k=0}^{n-1}\,C_k \Rightarrow n\cdot C_n = n(n+1) + 2\,\Sigma\,C_{k-1}$ … (1)

HOCADAN “Yazı tura atılıyor, kaç kez yazı geleceği araştırılıyor. 10 kez, 100 kez, 1000 kez, hatta 1 milyon kez atıldı. Biri dedi ki; ½'ye yakınsar! İşte, nerede yoğunlaşacağını ortalama durum analizi yaparak saptarız.”

HOCADAN ““Bana göre, k'nın hepsinde olma ihtimali eşittir.” Bana göresi, sana göresi yok bu işin! Matematiği var bu işin.”

Ortalama Durum: Çıkarma Hilesi ve Sonuç

- (1)'de n yerine n − 1: $(n-1)C_{n-1} = n(n-1) + 2\sum_{k=1}^{n-1} C_{k-1}$ … (2)
- (1) − (2): $n \cdot C_n - (n-1)C_{n-1} = 2n + 2C_{n-1} \Rightarrow n \cdot C_n = (n+1)C_{n-1} + 2n$
- (n(n+1)'e bölünce) $C_n/(n+1) = C_{n-1}/n + 2/(n+1)$: artık iterasyonla açılabilir.
- Açınca $C_n \approx 2(n+1)(1/3 + 1/4 + \ldots + 1/(n+1)) \leq 2(n+1) \ln n$.

$$C_n \approx 2n \ln n = 2 \ln 2 \cdot n \lg n \approx 1{,}39\ n \lg n$$

n	C_n (bağıntıdan)	1,39 n lg n	oran
10	37,1	46,2	0,80
100	780,5	923,5	0,85
1000	12.318,6	13.852,4	0,89

Tablo: bağıntı Python ile tam hesaplandı. Oran yavaşça 1'e yaklaşır (alt dereceli −cn terimi).

HOCADAN “Kitap, bunu yerine koymayla çözüyor. Biz daha akıllıca bir taktikle çözeceğiz.”

Özyinelemesiz Quick Sort (quicksort2)

```
void quicksort2(int a[], int L, int R) {
  yigit_baslat(); yigita_at(L, R);
  while (!yigit_bosmu()) {
    L = yigit_al(); R = yigit_al();
    if (R <= L) continue;
    int i = bol(a, L, R);
    if (i - L > R - i) {       // büyük parça önce
      yigita_at(L, i - 1); yigita_at(i + 1, R);
    } else {
      yigita_at(i + 1, R); yigita_at(L, i - 1);
    }
  }
}
```

```
#define YMAX 64     /* > 2·lg n */
static int yigit[YMAX], ust;
void yigit_baslat(void) { ust = 0; }
int yigit_bosmu(void) { return ust == 0; }
void yigit_ekle(int x) { yigit[ust++] = x; }
int yigit_al(void) { return yigit[--ust]; }
#define yigita_at(A, B) \
  do { yigit_ekle(B); yigit_ekle(A); } while (0)
```

- Amaç: sistem yığını yerine kendi yığınımızı kullanmak. `yigita_at(A,B)` önce B'yi sonra A'yı iter; böylece ilk alınan L olur.
- [7–11] **Büyük parça önce** itilir, küçük parça sonra: son eklenen önce alındığından hep küçük parça işlenir → yığın derinliği **O(lg n)**'de kalır (en çok ≈ lg n çift).
- Bellek bağıntısı $T_n = T_{n/2} + 2 \Rightarrow T_n = 2 \lg n$ (ikili aramadaki gibi).

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın





Yığına ilk problem (0, 6) itildi: önce R = 6, sonra L = 0.

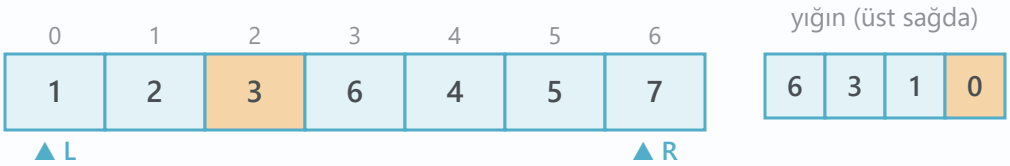
Adım 1 / 11

```
1  void quicksort2(int a[], int L, int R) {
▶    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
4      L = yigit_al(); R = yigit_al();
5      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`yigit = [6, 0]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



(0, 6): bol → pivot 3 konum 2. sağ (3,6) büyük/eşit → önce o, sonra sol (0,1) itildi; küçük parça üstte.

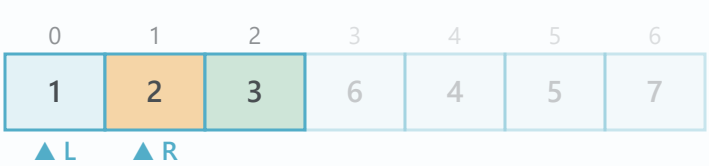

```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
4      L = yigit_al(); R = yigit_al();
5      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
▶      } else {
▶        yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 0` `R = 6` `i = 2`

`yigit = [6, 3, 1, 0]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın





(0, 1): bol → pivot 2 konum 1. sol (0,0) büyük → önce o, sonra sağ (2,1) itildi; küçük parça üstte.

Adım 3 / 11

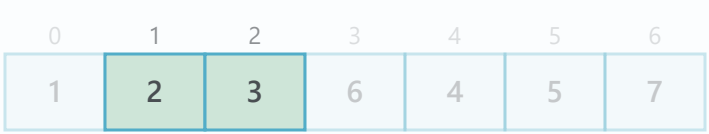
```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
4      L = yigit_al(); R = yigit_al();
5      if (R <= L) continue;
6      int i = bol(a, L, R);
▶      if (i - L > R - i) {        // büyük parça önce
▶        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 0` `R = 1` `i = 1`

`yigit = [6, 3, 0, 0, 1, 2]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



yığın (üst sağda)



(2, 1) alındı: boş aralık → continue.

Adım 4 / 11

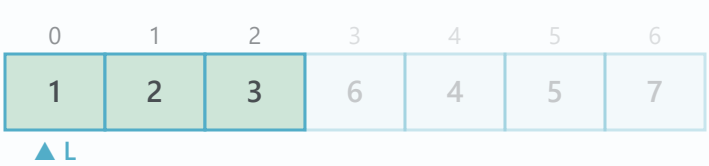
```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
▶      L = yigit_al(); R = yigit_al();
▶      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 2` `R = 1`

`yigit = [6, 3, 0, 0]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



(0, 0) alındı: tek öğe → continue.


```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
▶      L = yigit_al(); R = yigit_al();
▶      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 0` `R = 0` `yigit = [6, 3]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



(3, 6): bol → pivot 7 konum 6. sol (3,5) büyük → önce o, sonra sağ (7,6) itildi; küçük parça üstte.

Adım 6 / 11

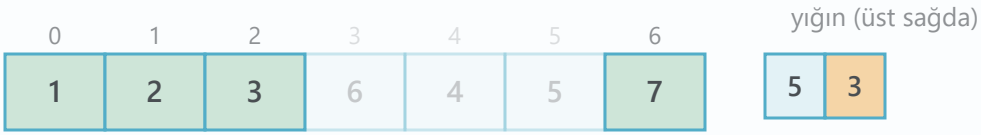
```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
4      L = yigit_al(); R = yigit_al();
5      if (R <= L) continue;
6      int i = bol(a, L, R);
▶      if (i - L > R - i) {        // büyük parça önce
▶        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 3` `R = 6` `i = 6`

`yigit = [5, 3, 6, 7]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



(7, 6) alındı: boş aralık → continue.

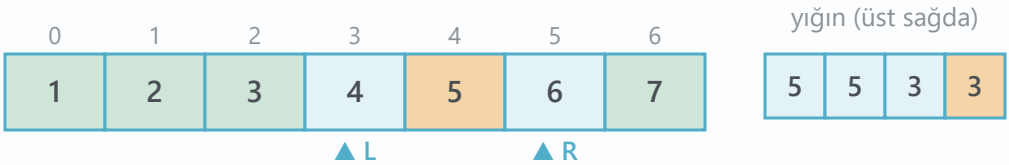
Adım 7 / 11

```
 1 void quicksort2(int a[], int L, int R) {
 2   yigit_baslat(); yigita_at(L, R);
 3   while (!yigit_bosmu()) {
▶      L = yigit_al(); R = yigit_al();
▶      if (R <= L) continue;
 6      int i = bol(a, L, R);
 7      if (i - L > R - i) {        // büyük parça önce
 8        yigita_at(L, i - 1); yigita_at(i + 1, R);
 9      } else {
10        yigita_at(i + 1, R); yigita_at(L, i - 1);
11      }
12    }
13  }
```

DEĞİŞKENLER

`L = 7` `R = 6` `yigit = [5, 3]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



(3, 5): bol → pivot 5 konum 4. sağ (5,5) büyük/eşit → önce o, sonra sol (3,3) itildi; küçük parça üstte.

Adım 8 / 11

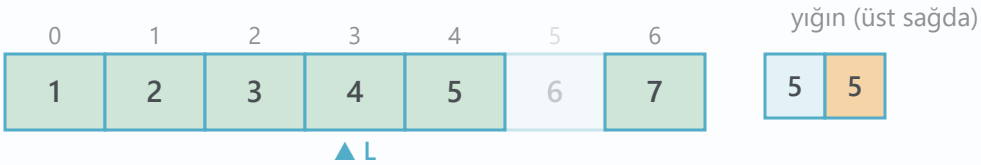
```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
4      L = yigit_al(); R = yigit_al();
5      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
▶      } else {
▶        yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 3` `R = 5` `i = 4`

`yigit = [5, 5, 3, 3]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



(3, 3) alındı: tek öğe → continue.

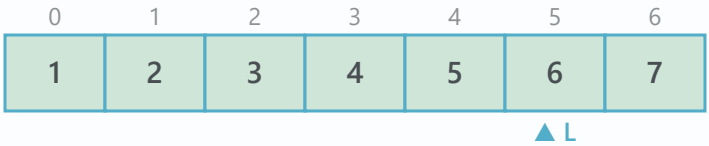
Adım 9 / 11

```
 1 void quicksort2(int a[], int L, int R) {
 2   yigit_baslat(); yigita_at(L, R);
 3   while (!yigit_bosmu()) {
 4     L = yigit_al(); R = yigit_al();
 5     if (R <= L) continue;
 6     int i = bol(a, L, R);
 7     if (i - L > R - i) {        // büyük parça önce
 8       yigita_at(L, i - 1); yigita_at(i + 1, R);
 9     } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 3` `R = 3` `yigit = [5, 5]`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



yığın (üst sağda)

boş

(5, 5) alındı: tek öğe → continue.

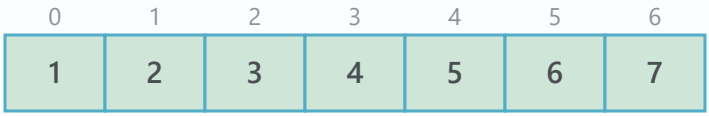
Adım 10 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
3    while (!yigit_bosmu()) {
▶      L = yigit_al(); R = yigit_al();
▶      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`L = 5` `R = 5` `yigit = []`

Kod İzleme: `quicksort2(a, 0, 6)` — Açık Yığın



yığın (üst sağda)

boş

Yığın boş → bitti: [1, 2, 3, 4, 5, 6, 7]. Yığında en çok 3 (L, R) çifti bekledi.

Adım 11 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    yigit_baslat(); yigita_at(L, R);
▶    while (!yigit_bosmu()) {
4      L = yigit_al(); R = yigit_al();
5      if (R <= L) continue;
6      int i = bol(a, L, R);
7      if (i - L > R - i) {        // büyük parça önce
8        yigita_at(L, i - 1); yigita_at(i + 1, R);
9      } else {
10       yigita_at(i + 1, R); yigita_at(L, i - 1);
11     }
12   }
13 }
```

DEĞİŞKENLER

`yigit = []`

Geliştirilmiş Quick Sort: Medyan-3 + Eklemeli

```c
#define M 10    /* bu boydan küçük parça kalsın */
void quicksort3(int a[], int L, int R) {
    if (R - L <= M) return;
    SWAP(a[(L + R) / 2], a[R - 1]);
    if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
    if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
    if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
    int i = bol(a, L + 1, R - 1);
    quicksort3(a, L, i - 1);
    quicksort3(a, i + 1, R);
}
void sirala(int a[], int L, int R) {
    quicksort3(a, L, R);
    insertion(a, L, R);   /* kalan küçük parçalar */
}
```

`if (kucuk(A, B)) SWAP(A, B)` "A < B ise takas" demektir; argümanlar `(a[L], a[R−1])` sırasıyla verilirse büyüğü a[L]'ye atar ve bölümlemeyi bozar (rastgele dizilerle denendi: pivotun solunda ondan büyük öğe kalıyor). Bu yüzden argümanlar ters sırada verilir: sonuçta **a[L] ≤ a[R−1] ≤ a[R]**.

HOCADAN "C'deki standart kütüphane fonksiyonu olan qsort() muhtemelen budur."

Neden Medyan-3 ve Küçük Parça Eşiği?

Medyan-3 (median-of-three)

a[L], ortadaki öğe ve a[R] sıraya konur; ortancası a[R−1]'e alınıp pivot olur. Quick sort en kötü performansını, insertion en iyi performansını sıralı veride gösteriyordu: sıralı girdideki n^2 durumu bozulur.

Doğal bekçiler

a[L] ≤ pivot ≤ a[R] olduğundan bölümleme L+1..R−1 üzerinde yapılır; uç öğeler taramaları durdurur.

Eşik M

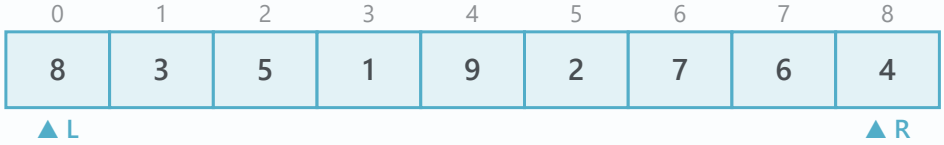
R − L ≤ M olan parçalar sıralanmadan bırakılır; `sirala()` sonunda tek bir `insertion` geçişi hepsini neredeyse doğrusal sürede düzeltir. 5 ≤ M ≤ 25 için ≈ %10 kazanç.

Sınırı

Medyan-3 sıralı/ters sıralı girdiyi kurtarır; ama özel olarak kurulmuş kötü girdiler hâlâ n^2 yapabilir → çare introsort.

HOCADAN “Asla n^2 ile karşı karşıya bırakmaz. Analizi biraz ağır…”

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



R − L = 8 > M = 3 (izde M = 3 alındı) → parça büyük, bölümlenecek.

Adım 1 / 7

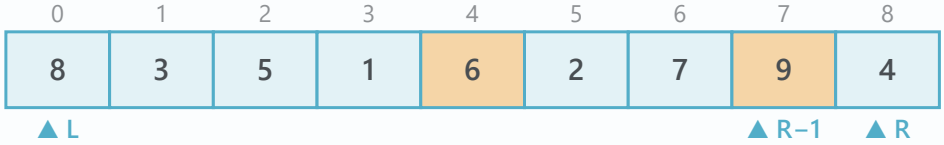
```
1  void quicksort3(int a[], int L, int R) {
▶      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <=
   a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <=
   a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
7      int i = bol(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [8, 3, 5, 1, 9, 2, 7, 6, 4]`

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



Ortadaki öğe a[4] ile a[R−1] = a[7] takaslandı: aday pivot 9 artık a[7]'de.

Adım 2 / 7

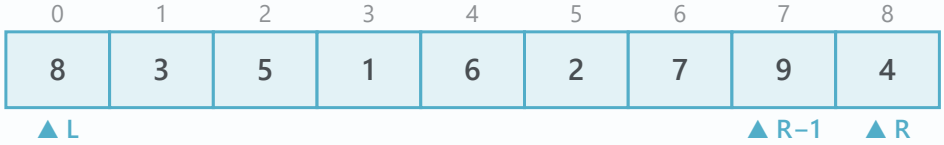
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
▶      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <=
   a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <=
   a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
7      int i = bol(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [8, 3, 5, 1, 6, 2, 7, 9, 4]`

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



a[7] = 9 ≥ a[0] = 8 → takas yok.

Adım 3 / 7

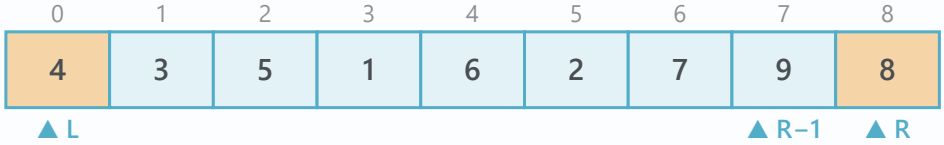
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
▶      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
7      int i = bol(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [8, 3, 5, 1, 6, 2, 7, 9, 4]`

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



a[8] = 4 < a[0] = 8 → takas.

Adım 4 / 7

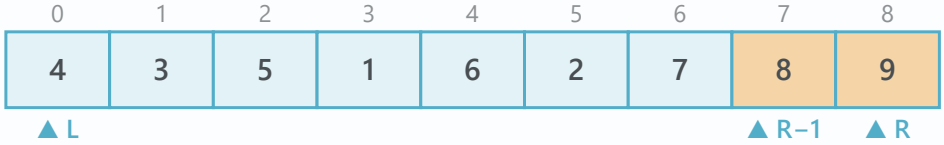
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
▶      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
7      int i = bol(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [4, 3, 5, 1, 6, 2, 7, 9, 8]`

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



a[8] = 8 < a[7] = 9 → takas.

Adım 5 / 7

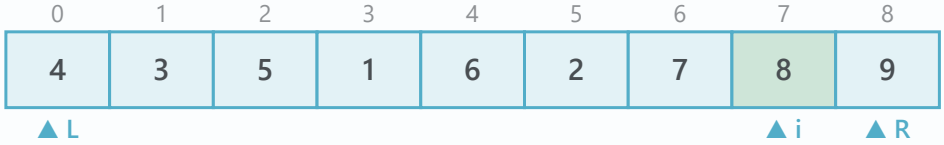
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <=
       a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <=
       a[R]
▶      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
       a[R]
7      int i = bol(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [4, 3, 5, 1, 6, 2, 7, 8, 9]`

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



a[L] = 4 ≤ pivot 8 ≤ a[R] = 9. bol(a, 1, 7) → pivot konum **7**; solu ≤ 8 ≤ sağı.

Adım 6 / 7

```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <=
   a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <=
   a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
▶      int i = bol(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

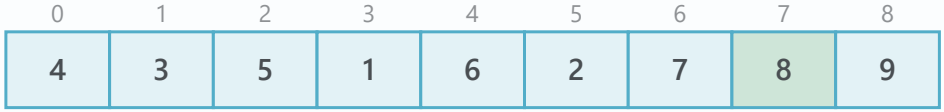
DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [4, 3, 5, 1, 6, 2, 7, 8, 9]`

`i = 7`

Kod İzleme: Medyan-3 Hazırlığı, `quicksort3(a, 0, 8)`



Sağ a[8..8] hemen döner. Sol a[0..6]'da R − L = 6 > M: medyan-3 yinelenir, bol(a, 1, 5) → 3 ve dizi [1, 3, 2, 4, 6, 5, 7, 8, 9] olur. Kalan parçalar (R − L = 2 ≤ M) sıralanmadan döner; sirala() içindeki insertion bitirir.

Adım 7 / 7

```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <=
   a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <=
   a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
7      int i = bol(a, L + 1, R - 1);
▶      quicksort3(a, L, i - 1);
▶      quicksort3(a, i + 1, R);
10 }
```

DEĞİŞKENLER

`L = 0` `R = 8` `M = 3`

`a = [4, 3, 5, 1, 6, 2, 7, 8, 9]`

`i = 7`

OKUMA PARÇASI

Okuma Parçası: Zorunlu Stajlar

HOCADAN “Stajlar, öyle zannediyorum ki, imkânsızlıktan dolayı konmuş olmalı. Eskiden program yazılır, laboratuvara getirilip kartlara deldirilirdi. Bir kişiye günde bir, en çok iki kez derleme sırası gelirdi. Şimdi günde kaç kez derleme imkânı var? 50 mi, 100 mü?”

HOCADAN “Şimdi ise durum farklı... Bizim öğrenciyi şunu yap, bunu yap diye ayak işlerinde kullanıyorlar. Klavyeyle toprak kazdırmak gibi... Bizim piyasayla işimiz yok ki! Bizim için önemli olan, bilim adamı yetiştirmek...”

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu

0	1	2	3	4
37	15	24	33	49

Dizi hazır; `sizeof a / sizeof a[0]` = 20 / 4 = 5 öğe (int 4 bayt varsayımıyla).

Adım 1 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
▶    int a[] = {37, 15, 24, 33, 49};
▶    size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], artan);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`a = [37, 15, 24, 33, 49]` `n = 5`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu

0	1	2	3	4
37	15	24	33	49

`qsort` 'a dizinin adresi, öğe sayısı, öğe boyu ve **karşılaştırma fonksiyonunun adresi** verilir. qsort öğe türünü bilmez; `void *` ile çalışır.

Adım 2 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 ▶     qsort(a, n, sizeof a[0], artan);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`a = [37, 15, 24, 33, 49]` `n = 5`

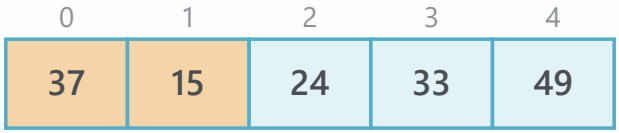
ÇAĞRI YIĞINI

`qsort(a, 5, 4, artan)`

`main()`

ÇIKTI

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



Örnek çağrı artan(&37, &15) → **1**: pozitif = ilk öğe sonra gelmeli. (qsort'un hangi çiftleri hangi sırayla karşılaştıracağı standartta belirtilmez.)

Adım 3 / 11

```
1  int artan(const void *p, const void *q) {
▶      int x = *(const int *)p;
▶      int y = *(const int *)q;
▶      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], artan);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

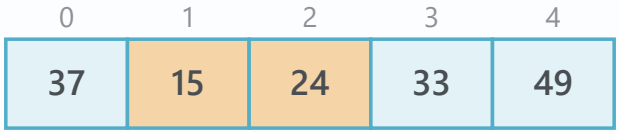
`x = 37` `y = 15` `dönüş = 1`

ÇAĞRI YIĞINI

- `artan(p, q)`
- `qsort(a, 5, 4, artan)`
- `main()`

ÇIKTI

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



Örnek çağrı artan(&15, &24) → **-1**: negatif = ilk öğe önce gelmeli. (qsort'un hangi çiftleri hangi sırayla karşılaştıracağı standartta belirtilmez.)

Adım 4 / 11

```
1  int artan(const void *p, const void *q) {
▶      int x = *(const int *)p;
▶      int y = *(const int *)q;
▶      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], artan);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

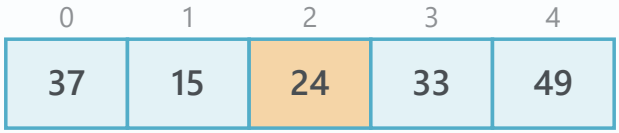
`x = 15` `y = 24` `dönüş = -1`

ÇAĞRI YIĞINI

- `artan(p, q)`
- `qsort(a, 5, 4, artan)`
- `main()`

ÇIKTI

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



Örnek çağrı artan(&24, &24) → **0**: sıfır = eşit (göreli sıra garanti değil). (qsort'un hangi çiftleri hangi sırayla karşılaştıracağı standartta belirtilmez.)

Adım 5 / 11

```
1  int artan(const void *p, const void *q) {
▶      int x = *(const int *)p;
▶      int y = *(const int *)q;
▶      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], artan);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`x = 24` `y = 24` `dönüş = 0`

ÇAĞRI YIĞINI

- `artan(p, q)`
- `qsort(a, 5, 4, artan)`
- `main()`

ÇIKTI

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu

0	1	2	3	4
15	24	33	37	49

qsort döndü: dizi artan sırada.

Adım 6 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 ▶     qsort(a, n, sizeof a[0], artan);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

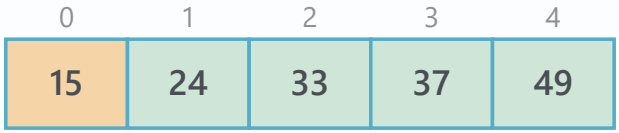
`a = [15, 24, 33, 37, 49]` `n = 5`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



printf: a[0] = 15.

Adım 7 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], artan);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`i = 0` `a = [15, 24, 33, 37, 49]`

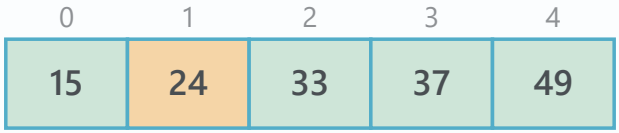
ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
15
```

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



printf: a[1] = 24.

Adım 8 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], artan);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`i = 1` `a = [15, 24, 33, 37, 49]`

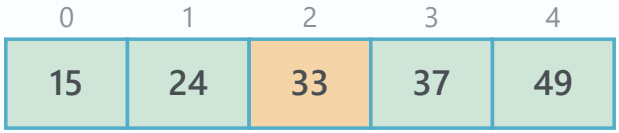
ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
15 24
```

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



printf: a[2] = 33.

Adım 9 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], artan);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`i = 2` `a = [15, 24, 33, 37, 49]`

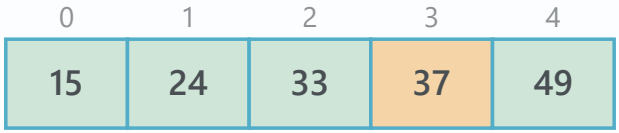
ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
15 24 33
```

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



printf: a[3] = 37.

Adım 10 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], artan);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`i = 3` `a = [15, 24, 33, 37, 49]`

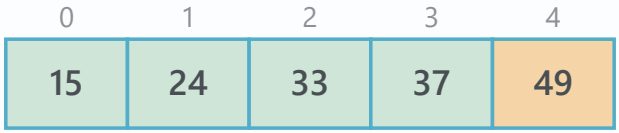
ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
15 24 33 37
```

Kod İzleme: `qsort` ve Karşılaştırma Fonksiyonu



printf: a[4] = 49.

Adım 11 / 11

```
 1 int artan(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], artan);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

DEĞİŞKENLER

`i = 4` `a = [15, 24, 33, 37, 49]`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
15 24 33 37 49
```

qsort ile Tutanak (struct) Sıralamak

```c
typedef struct { char ad[20]; int not_; } ogrenci;

int ada_gore(const void *p, const void *q) {
    const ogrenci *x = p, *y = q;
    return strcmp(x->ad, y->ad);        // alfabetik
}
int nota_gore_azalan(const void *p, const void *q) {
    const ogrenci *x = p, *y = q;
    return (y->not_ > x->not_) - (y->not_ < x->not_);
}
/* qsort(s, n, sizeof s[0], ada_gore); */
```

Sözleşme

Karşılaştırıcı negatif / sıfır / pozitif döndürür ve **tutarlı** olmalıdır (a < b ve b < c ise a < c).

Tuzak: taşma

`return x - y;` kısa ama `INT_MIN` ile `1` gibi değerlerde taşar; `(x > y) - (x < y)` güvenlidir.

Kararlılık yok

Standart, qsort'un kararlı olmasını ya da belirli bir algoritmayı kullanmasını istemez; eşitlerde sıra önemliyse ikincil anahtar eklenir.

Hibrit Sıralamalar: Algoritmaları Birleştirmek

Quick + eklemeli

Küçük parçalarda özyinelemenin maliyeti baskındır; M'den küçük parçalar eklemeliyle bitirilir (quicksort3 + sirala).

Introsort (Musser, 1997)

Hızlı sıralama ile başlar; özyineleme derinliği ≈ 2⌊lg n⌋'i aşarsa o parçaya **heap sort** uygular → en kötü durum O(n lg n). Küçük parçalar eklemeliyle. C++ `std::sort` gerçeklemelerinde yaygındır.

Timsort (Peters, 2002)

Birleştirme + eklemeli sıralama; veride hazır sıralı "koşuları" (*run*) bulup birleştirir. **Kararlı**; Python'un `sort` 'u ve Java'nın nesne dizileri için `Arrays.sort` 'u.

Ortak fikir

Büyük ölçekte iyi asimptotik davranış (n lg n) + küçük ölçekte düşük sabit (eklemeli) + en kötü durum sigortası.

Birleştirme Sıralaması (Merge Sort)

1. Liste 0 ya da 1 öğeliyse zaten sıralıdır.
2. Değilse liste olabildiğince eşit iki parçaya ayrılır.
3. Her parça merge sort özyinelemeli uygulanarak sıralanır.
4. Kendi içinde sıralı iki alt liste tek sıralı listede **birleştirilir**.

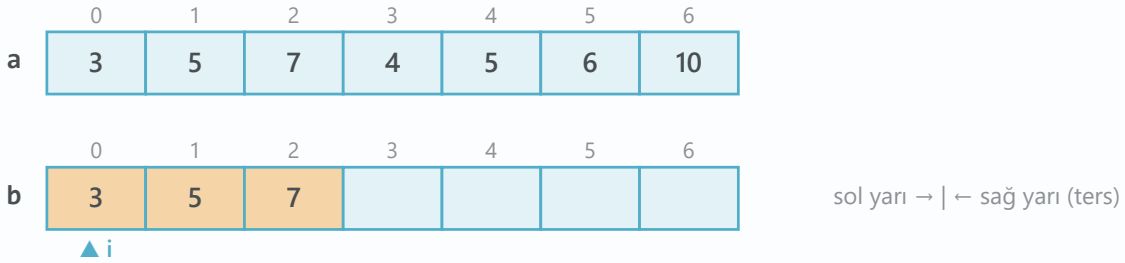
- 1945, **John von Neumann**; parçala ve yönet.
- Fikir 1: küçük listeyi sıralamak daha az adım ister.
- Fikir 2: sıralı iki listeden sıralı liste elde etmek, sırasız ikisinden elde etmekten ucuzdur.

```
void mergesort(int a[], int b[], int L, int R) {
    if (R <= L) return;
    int m = (R + L) / 2;
    mergesort(a, b, L, m);
    mergesort(a, b, m + 1, R);
    merge(a, b, L, m, R);
}
```

```
void merge(int a[], int b[], int L, int m, int R) {
    int i, j, k;
    for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
    for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
    for (k = L; k <= R; k++)
        if (kucuk(b[j], b[i]))
            a[k] = b[j--];
        else
            a[k] = b[i++];
}
```

Yardımcı `b[]` dizisi parametre olarak geçirilir (a ile aynı boyda); tanımlanmadan kullanılamaz.

Kod İzleme: `merge(a, b, 0, 2, 6)`



1. döngü: sol yarı a[0..2] olduğu gibi b[0..2]'ye kopyalandı; döngü **i = L = 0** ile çıkar.

Adım 1 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
▶      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
6          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

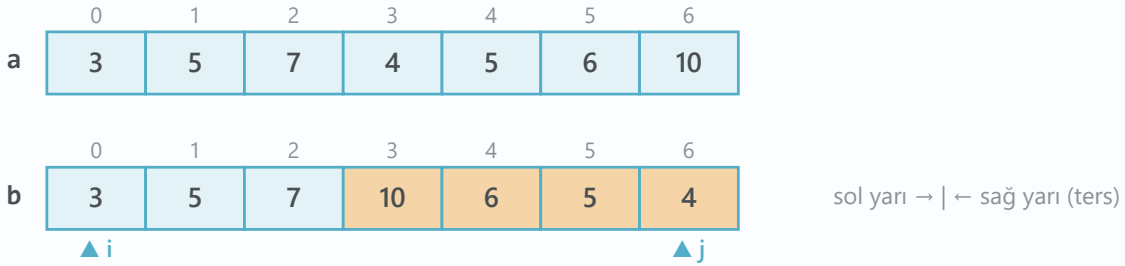
DEĞİŞKENLER

`i = 0` `j = ?` `k = ?`

`a = [3, 5, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, _, _, _, _]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



2. döngü: sağ yarı b[3..6]'ya **ters sırada** kopyalandı → b iki uçtan ortaya doğru artan (bitonik). Döngü **j = R = 6** ile çıkar.

Adım 2 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
▶      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
6          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

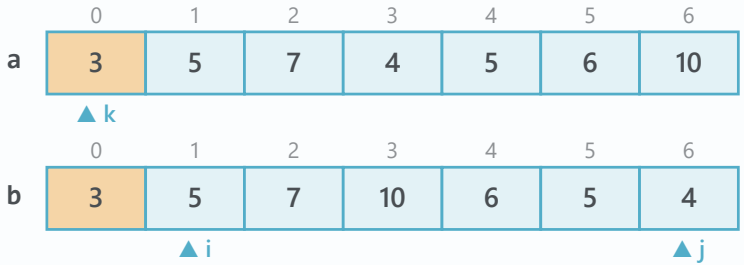
DEĞİŞKENLER

`i = 0` `j = 6` `k = ?`

`a = [3, 5, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 4 ≥ b[i] = 3 → a[0] = b[0]; i++.

Adım 3 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

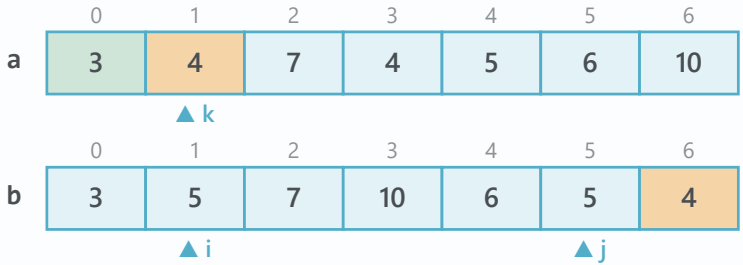
DEĞİŞKENLER

`i = 1` `j = 6` `k = 0`

`a = [3, 5, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 4 < b[i] = 5 → a[1] = b[6]; j--.

Adım 4 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
▶              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

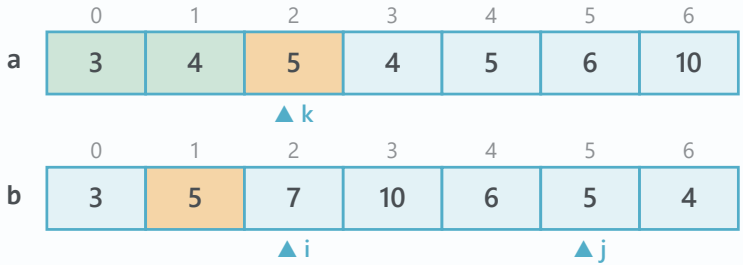
DEĞİŞKENLER

`i = 1` `j = 5` `k = 1`

`a = [3, 4, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 5 ≥ b[i] = 5 → a[2] = b[1]; i++.

Adım 5 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

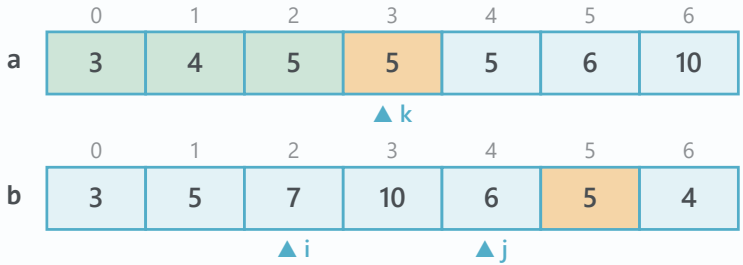
DEĞİŞKENLER

`i = 2` `j = 5` `k = 2`

`a = [3, 4, 5, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 5 < b[i] = 7 → a[3] = b[5]; j--.

Adım 6 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
▶              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

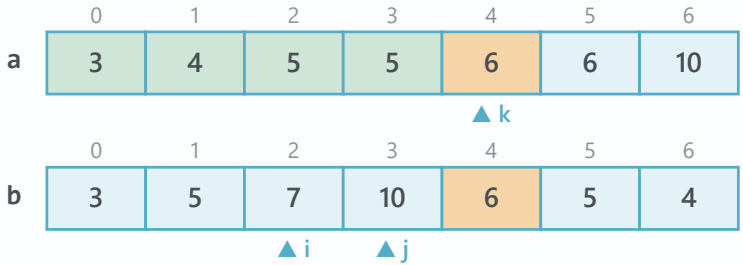
DEĞİŞKENLER

`i = 2` `j = 4` `k = 3`

`a = [3, 4, 5, 5, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 6 < b[i] = 7 → a[4] = b[4]; j--.

Adım 7 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
▶              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

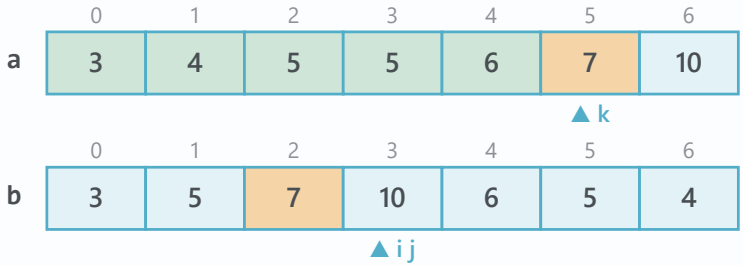
DEĞİŞKENLER

`i = 2` `j = 3` `k = 4`

`a = [3, 4, 5, 5, 6, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 10 ≥ b[i] = 7 → a[5] = b[2]; i++.

Adım 8 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

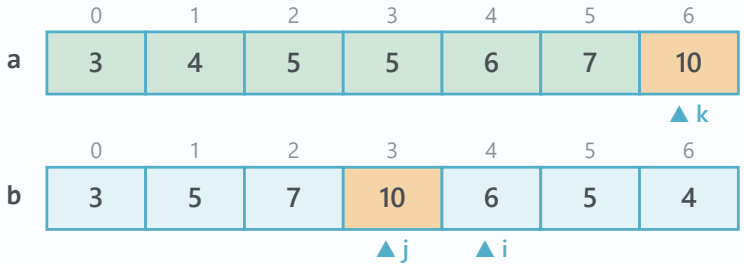
DEĞİŞKENLER

`i = 3` `j = 3` `k = 5`

`a = [3, 4, 5, 5, 6, 7, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



sol yarı → | ← sağ yarı (ters)

b[j] = 10 ≥ b[i] = 10 → a[6] = b[3]; i++.

Adım 9 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

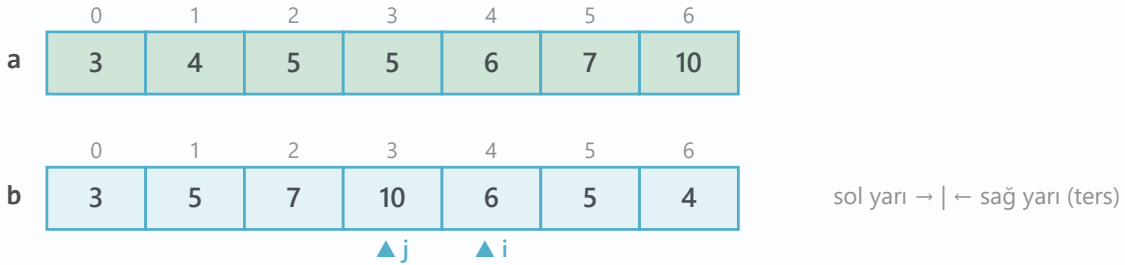
DEĞİŞKENLER

`i = 4` `j = 3` `k = 6`

`a = [3, 4, 5, 5, 6, 7, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `merge(a, b, 0, 2, 6)`



Bitti: a = [3, 4, 5, 5, 6, 7, 10]. Sonda i = 4, j = 3: her iki indis ortada buluştu; sınır sınaması gerekmedi. Eşit 5'lerde soldaki (b[i]) önce alındı.

Adım 10 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
▶      for (k = L; k <= R; k++)
6          if (kucuk(b[j], b[i]))
7              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

DEĞİŞKENLER

`i = 4` `j = 3` `k = 6`

`a = [3, 4, 5, 5, 6, 7, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Kod İzleme: `mergesort` — 9 merge Çağrısı

0	1	2	3	4	5	6	7	8	9
89	67	59	62	23	93	24	20	28	60

mergesort(a, b, 0, 9): önce sol yarı tamamen sıralanır; ilk merge ancak en derindeki (0, 1) çağrısında olur.

Adım 1 / 10

```
▶ void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
6     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

`L = 0` `R = 9`

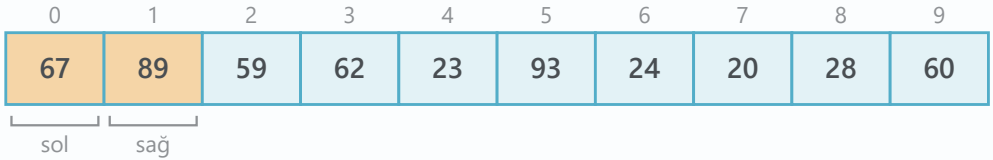
`a = [89, 67, 59, 62, 23, 93, 24, 20, 28, 60]`

ÇAĞRI YIĞINI

`mergesort(0, 9)`

`main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #1: a[0..0] ile a[1..1] birleşti → a[0..1] sıralı.

Adım 2 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

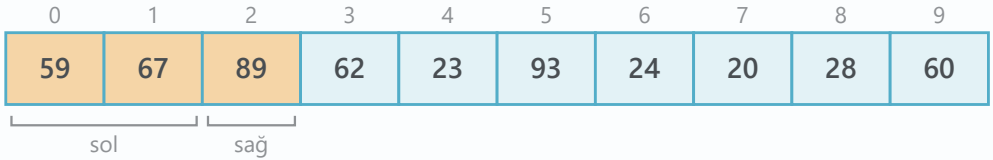
`L = 0` `m = 0` `R = 1`

`a = [67, 89, 59, 62, 23, 93, 24, 20, 28, 60]`

ÇAĞRI YIĞINI

- `merge(0, 0, 1)`
- `mergesort(0, 1)`
- `mergesort(0, 2)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #2: a[0..1] ile a[2..2] birleşti → a[0..2] sıralı.

Adım 3 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

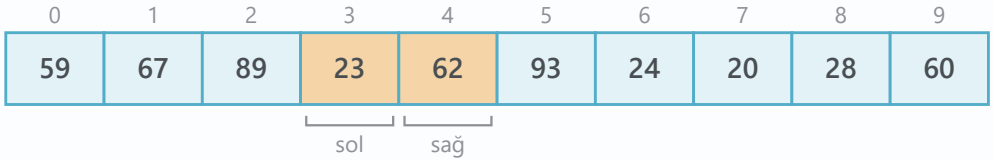
`L = 0` `m = 1` `R = 2`

`a = [59, 67, 89, 62, 23, 93, 24, 20, 28, 60]`

ÇAĞRI YIĞINI

- `merge(0, 1, 2)`
- `mergesort(0, 2)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #3: a[3..3] ile a[4..4] birleşti → a[3..4] sıralı.

Adım 4 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

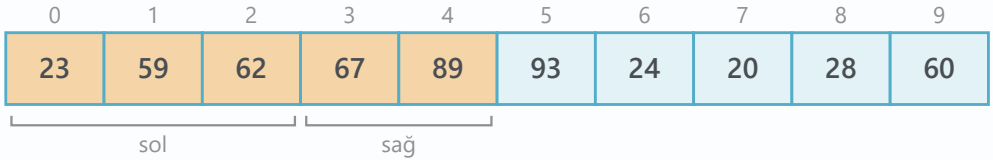
`L = 3` `m = 3` `R = 4`

`a = [59, 67, 89, 23, 62, 93, 24, 20, 28, 60]`

ÇAĞRI YIĞINI

- `merge(3, 3, 4)`
- `mergesort(3, 4)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #4: a[0..2] ile a[3..4] birleşti → a[0..4] sıralı.

Adım 5 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

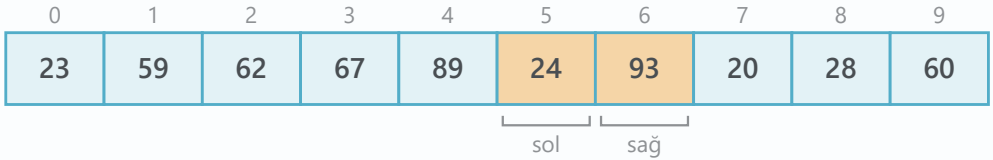
`L = 0` `m = 2` `R = 4`

`a = [23, 59, 62, 67, 89, 93, 24, 20, 28, 60]`

ÇAĞRI YIĞINI

- `merge(0, 2, 4)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #5: a[5..5] ile a[6..6] birleşti → a[5..6] sıralı.

Adım 6 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

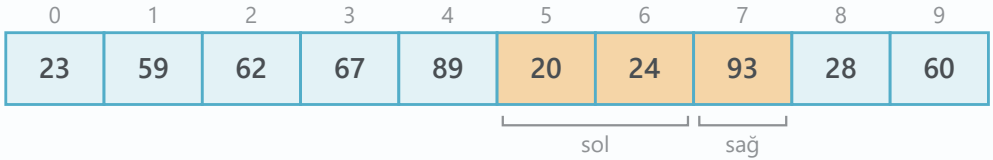
`L = 5` `m = 5` `R = 6`

`a = [23, 59, 62, 67, 89, 24, 93, 20, 28, 60]`

ÇAĞRI YIĞINI

- `merge(5, 5, 6)`
- `mergesort(5, 6)`
- `mergesort(5, 7)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #6: a[5..6] ile a[7..7] birleşti → a[5..7] sıralı.

Adım 7 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

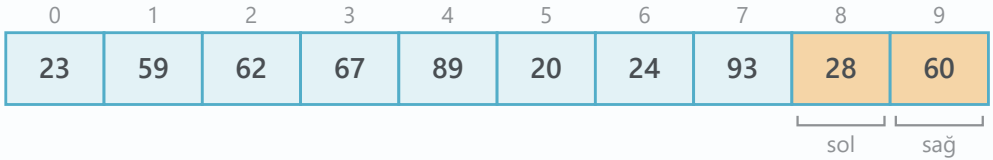
`L = 5` `m = 6` `R = 7`

`a = [23, 59, 62, 67, 89, 20, 24, 93, 28, 60]`

ÇAĞRI YIĞINI

- `merge(5, 6, 7)`
- `mergesort(5, 7)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #7: a[8..8] ile a[9..9] birleşti → a[8..9] sıralı.

Adım 8 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

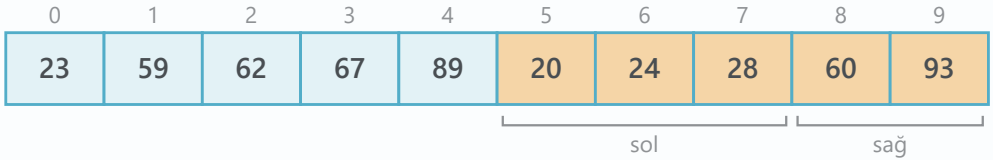
`L = 8` `m = 8` `R = 9`

`a = [23, 59, 62, 67, 89, 20, 24, 93, 28, 60]`

ÇAĞRI YIĞINI

- `merge(8, 8, 9)`
- `mergesort(8, 9)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #8: a[5..7] ile a[8..9] birleşti → a[5..9] sıralı.

Adım 9 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

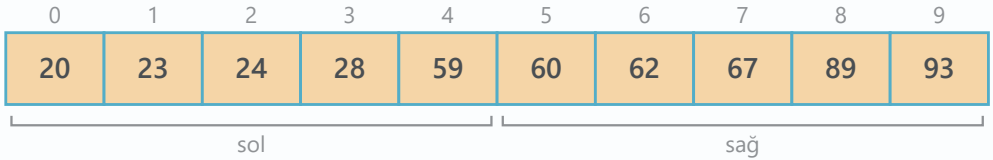
`L = 5` `m = 7` `R = 9`

`a = [23, 59, 62, 67, 89, 20, 24, 28, 60, 93]`

ÇAĞRI YIĞINI

- `merge(5, 7, 9)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Kod İzleme: `mergesort` — 9 merge Çağrısı



merge #9: a[0..4] ile a[5..9] birleşti → a[0..9] sıralı.

Adım 10 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

DEĞİŞKENLER

`L = 0` `m = 4` `R = 9`

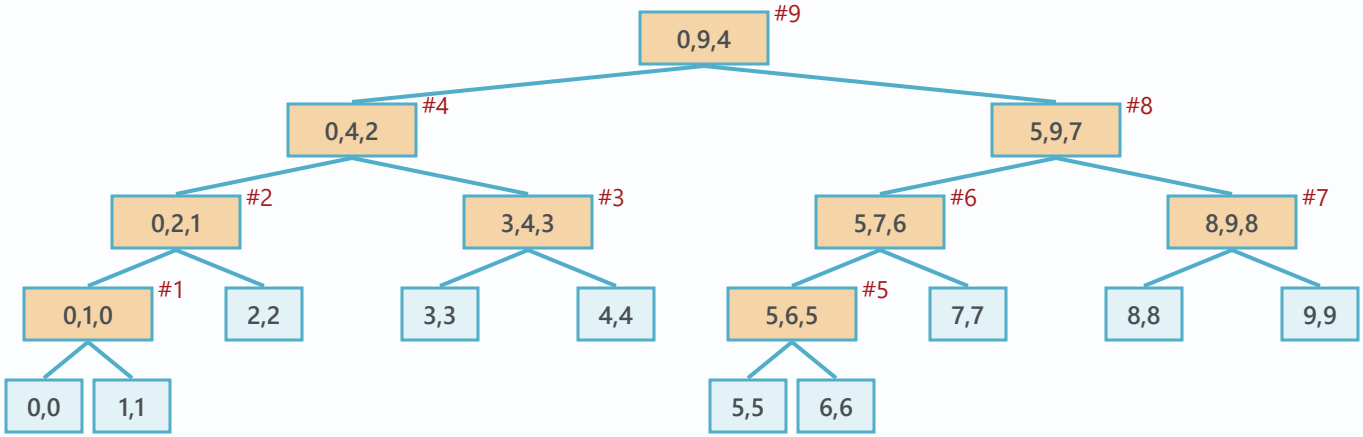
`a = [20, 23, 24, 28, 59, 60, 62, 67, 89, 93]`

ÇAĞRI YIĞINI

- `merge(0, 4, 9)`
- `mergesort(0, 9)`
- `main()`

BİRLEŞTİRME SIRALAMASI

Yığın Çerçevesi (Stack Activation Frame) Ağacı



Kutu: L, R, m · turuncu: merge çağrısı yapan çerçeve, #k: kaçıncı merge. Soldan derine iner, dönüşte birleştirir.

HOCADAN "Veri Yapıları 1 dersinde 'Bu mantığa merge diyoruz.' diye hep söylemişimdir."

Soru: İki mergesort Satırının Yeri Değişirse?

Soru: Yordamdaki iki `mergesort` satırının yerleri değiştirilirse nasıl bir tablo oluşur, gözlemleyiniz.

▶ **Cevap: önce sağ yarılar birleşir; sonuç aynı, ara tablolar farklı**

Merge Sort: Zaman ve Alan Karmaşıklığı

$$a_n = a_{\lceil n/2 \rceil} + a_{\lfloor n/2 \rfloor} + n, \quad a_1 = 0$$

$$n = 2^k: a_n = 2a_{n/2} + n \Rightarrow a_n = \Theta(n \lg n)$$

- 16 öğe 8 + 8, 17 öğe 8 + 9 bölünür; merge n adım sürer.
- Çağrı sayısı $T_n = 2T_{n/2} + 1 \Rightarrow 2n - 1$ (10 öğede 19 çağrı, 9'u merge yapar).
- Dizi durumundan **bağımsız**: en iyi = ortalama = en kötü = n lg n.
- Dezavantaj: **O(n) ek bellek**. Aşmak maliyeti artırır; ek bellek istenmiyorsa heap sort öne çıkar.
- Sıralı parçaları sırayla birleştirdiği için disk üzerindeki **dış sıralamanın** temelidir.

Kararlılık

Klasik merge (eşitte soldakini alan) **kararlıdır** (quick ve heap sort kararlı değildir). Buradaki ters kopyalı (bitonik) merge kararlı değildir: [2a, 2b, 2c, 2d] → [2a, 2b, 2d, 2c] (Python ile denendi).

HOCADAN "Mademki girdiyi ortadan ikiye bölmek marifet, işte bu da ortadan bölüyor. Ama öncekilerde pivot bir adımda doğru konumuna yerleşiyordu; bunda ise yerleşmiyor. Sıralı iki diziden 'sıralı olma' avantajını kullanarak yeni bir sıralı dizi elde ediyor."

Alfabetik Verileri Sıralamak: Bağlı Merge Sort

```c
typedef struct {
    int anahtar;    /* + büyük veri alanları */
    int bag;        /* sonraki öğenin indisi, 0 = son */
} eleman;

int mergesort(eleman a[], int L, int R) {
    if (R <= L) return L;           /* tek öğeli liste */
    int m = (R + L) / 2;
    int sag = mergesort(a, m + 1, R);
    int sol = mergesort(a, L, m);
    return merge(a, sol, sag);      /* baş indisi */
}
```

merge() iki sıralı listeyi bag alanları üzerinden birleştirip baş indisini döndürür.

- Büyük tutanaklar (ör. alfabetik bilgiler) kopyalanırsa zaman alıcıdır. Çözüm: **bag** alanıyla dizi içinde bağlı liste kurmak; yalnız indisler değişir.
- Veriler a[1..n]'dedir; **a[0]** merge sırasında geçici baş (takas alanı). Başta tüm bag alanları 0.
- mergesort, sıralanmış alt listenin **baş indisini** döndürür.
- **Sık hatalar:** `void` yordamdan değer döndürmek; `else` dalında i2 yerine i1; a[0] yerine `a[c].bag` döndürmek.
- C'de argümanların değerlendirilme sırası **belirsizdir**; burada sıra açıkça yazıldı.

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	4	2	5	0	1	3	9	6	0	8	7
	▲ sonuc				▲ i1						▲ i2

merge #9: birinci liste (baş 4) 38→71→77→119→122; ikinci liste (baş 10) 13→22→41→43→115. sonuc = 0 (a[0] geçici baş).

Adım 1 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
▶      int sonuc = 0;              // a[0]: geçici baş
▶      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
8          } else {
9              a[sonuc].bag = i2; sonuc = i2;
10             i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 4` `i2 = 10` `sonuc = 0`

`bag = [4, 2, 5, 0, 1, 3, 9, 6, 0, 8, 7]`

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	10	2	5	0	1	3	9	6	0	8	7
					▲ i1			▲ i2			▲ sonuc

38 > 13 → a[0].bag = 10; sonuc = 10; i2 = 7.

Adım 2 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
▶          } else {
▶              a[sonuc].bag = i2; sonuc = i2;
▶              i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 4` `i2 = 7` `sonuc = 10`

`bag = [10, 2, 5, 0, 1, 3, 9, 6, 0, 8, 7]`

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	10	2	5	0	1	3	9	6	0	8	7
					▲ i1		▲ i2	▲ sonuc			

38 > 22 → a[10].bag = 7; sonuc = 7; i2 = 6.

Adım 3 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
▶          } else {
▶              a[sonuc].bag = i2; sonuc = i2;
▶              i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 4` `i2 = 6` `sonuc = 7`

bag = `[10, 2, 5, 0, 1, 3, 9, 6, 0, 8, 7]`

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	10	2	5	0	1	3	9	4	0	8	7
		▲ i1			▲ sonuc		▲ i2				

38 ≤ 41 → a[7].bag = 4; sonuc = 4; i1 = 1.

Adım 4 / 10

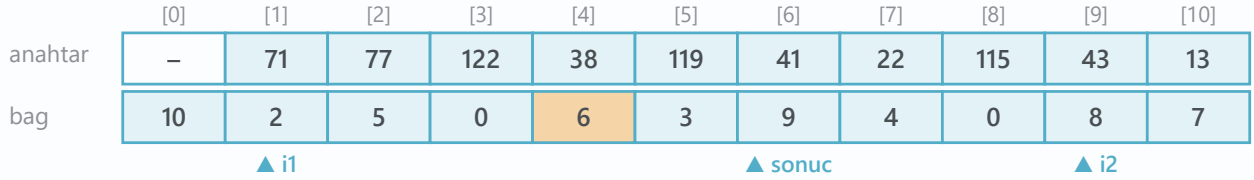
```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
▶          if (a[i1].anahtar <= a[i2].anahtar) {
▶              a[sonuc].bag = i1; sonuc = i1;
▶              i1 = a[i1].bag;
8          } else {
9              a[sonuc].bag = i2; sonuc = i2;
10             i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 1` `i2 = 6` `sonuc = 4`

`bag = [10, 2, 5, 0, 1, 3, 9, 4, 0, 8, 7]`

Kod İzleme: Bağlı `merge` (#9)



71 > 41 → a[4].bag = 6; sonuc = 6; i2 = 9.

Adım 5 / 10

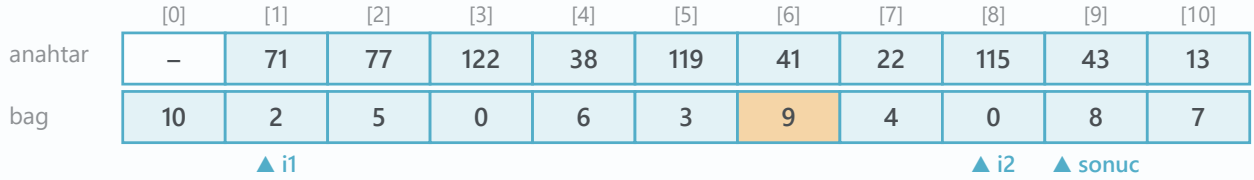
```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
▶          } else {
▶              a[sonuc].bag = i2; sonuc = i2;
▶              i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 1` `i2 = 9` `sonuc = 6`

`bag = [10, 2, 5, 0, 6, 3, 9, 4, 0, 8, 7]`

Kod İzleme: Bağlı `merge` (#9)



71 > 43 → a[6].bag = 9; sonuc = 9; i2 = 8.

Adım 6 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
▶          } else {
▶              a[sonuc].bag = i2; sonuc = i2;
▶              i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 1` `i2 = 8` `sonuc = 9`

`bag = [10, 2, 5, 0, 6, 3, 9, 4, 0, 8, 7]`

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	10	2	5	0	6	3	9	4	0	1	7
		▲ sonuc	▲ i1						▲ i2		

71 ≤ 115 → a[9].bag = 1; sonuc = 1; i1 = 2.

Adım 7 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
▶          if (a[i1].anahtar <= a[i2].anahtar) {
▶              a[sonuc].bag = i1; sonuc = i1;
▶              i1 = a[i1].bag;
8          } else {
9              a[sonuc].bag = i2; sonuc = i2;
10             i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 2` `i2 = 8` `sonuc = 1`

`bag = [10, 2, 5, 0, 6, 3, 9, 4, 0, 1, 7]`

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	10	2	5	0	6	3	9	4	0	1	7
			▲ sonuc			▲ i1			▲ i2		

77 ≤ 115 → a[1].bag = 2; sonuc = 2; i1 = 5.

Adım 8 / 10

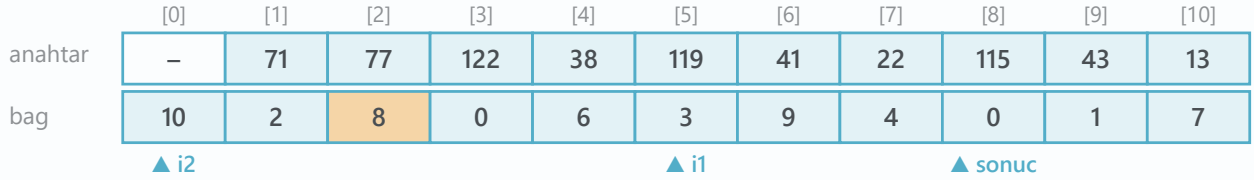
```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
▶          if (a[i1].anahtar <= a[i2].anahtar) {
▶              a[sonuc].bag = i1; sonuc = i1;
▶              i1 = a[i1].bag;
8          } else {
9              a[sonuc].bag = i2; sonuc = i2;
10             i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 5` `i2 = 8` `sonuc = 2`

bag = `[10, 2, 5, 0, 6, 3, 9, 4, 0, 1, 7]`

Kod İzleme: Bağlı `merge` (#9)



119 > 115 → a[2].bag = 8; sonuc = 8; i2 = 0.

Adım 9 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
▶          } else {
▶              a[sonuc].bag = i2; sonuc = i2;
▶              i2 = a[i2].bag;
11         }
12     a[sonuc].bag = i1 ? i1 : i2;
13     return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 5` `i2 = 0` `sonuc = 8`

`bag = [10, 2, 8, 0, 6, 3, 9, 4, 0, 1, 7]`

Kod İzleme: Bağlı `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
anahtar	–	71	77	122	38	119	41	22	115	43	13
bag	10	2	8	0	6	3	9	4	5	1	7

▲ sonuc

i2 = 0 (ikinci liste bitti) → a[8].bag = i1 = 5; baş **a[0].bag = 10** döner:
13→22→38→41→43→71→77→115→119→122.

Adım 10 / 10

```
1  int merge(eleman a[], int birinci, int ikinci) {
2      int sonuc = 0;              // a[0]: geçici baş
3      int i1 = birinci, i2 = ikinci;
4      while (i1 && i2)
5          if (a[i1].anahtar <= a[i2].anahtar) {
6              a[sonuc].bag = i1; sonuc = i1;
7              i1 = a[i1].bag;
8          } else {
9              a[sonuc].bag = i2; sonuc = i2;
10             i2 = a[i2].bag;
11         }
▶      a[sonuc].bag = i1 ? i1 : i2;
▶      return a[0].bag;
14 }
```

DEĞİŞKENLER

`i1 = 5` `i2 = 0` `sonuc = 8`

`bag = [10, 2, 8, 0, 6, 3, 9, 4, 5, 1, 7]`

Bağlı Merge Sort: bag Alanlarının Değişimi

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	dönüş	L, R
anahtar	71	77	122	38	119	41	22	115	43	13		
#1	0	0	0	0	0	0	0	0	0	**9**	10	9,10
#2	0	0	0	0	0	0	**6**	0	0	9	7	6,7
#3	0	0	0	0	0	**8**	6	0	0	9	7	6,8
#4	0	0	0	0	0	**9**	6	0	**8**	**7**	10	6,10
#5	0	0	0	**5**	0	9	6	0	8	7	4	4,5
#6	**2**	0	0	5	0	9	6	0	8	7	1	1,2
#7	2	**3**	0	5	0	9	6	0	8	7	1	1,3
#8	2	**5**	0	**1**	**3**	9	6	0	8	7	4	1,5
#9	2	**8**	0	**6**	3	9	**4**	**5**	**1**	7	10	1,10

Kalın: o merge'de değişen bag. Sıralı liste: 13 → 22 → 38 → 41 → 43 → 71 → 77 → 115 → 119 → 122.

SINAV SORULARI

Sınav Sorusu

Aşağıdaki uyarlamayla `eleman` dizisi a[1..9] anahtara göre sıralanıyor. (i) Özyinelemeli çağrıların **yığın çerçevelerini** çizin; merge çağrılmışsa dönüş değerini yazın. (ii) İşletim tablosunu merge numaraları, dönüş değerleri ve her adımdaki bag alanlarıyla doldurun.

```
int mergesort(eleman a[], int L, int R) {
    int m;
    if (R <= L) return L;
    m = (R + L) / 2;
    return merge(a, mergesort(a, m + 1, R),
                    mergesort(a, L, m));
}
```

Anahtarlar: [1] ?, [2] ?, [3] 18, [4] −5, [5] 27, [6] 11, [7] ?, [8] 83, [9] 30. Soru kâğıdındaki "?" yerine burada 42, 7, 55 alındı. merge() bağlı merge sort'taki gibidir; basılı sürümün `else` dalında i1 yazıyor (yazım hatası; doğrusu i2).

Sınav Sorusu (3. soru): Çözüm

▼ **Çözüm**

Argümanlar soldan sağa değerlendirilirse (önce sağ yarı) çağrı sırası: (1,9) → (6,9) → (8,9) → (9,9) → (8,8) → (6,7) → (7,7) → (6,6) → (1,5) …

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	dön.	L,R,m
#1	0	0	0	0	0	0	0	0	8	9	8,9,8
#2	0	0	0	0	0	7	0	0	8	6	6,7,6
#3	0	0	0	0	0	9	8	0	7	6	6,9,7
#4	0	0	0	5	0	9	8	0	7	4	4,5,4
#5	0	1	0	5	0	9	8	0	7	2	1,2,1
#6	0	3	1	5	0	9	8	0	7	2	1,3,2
#7	0	3	5	2	1	9	8	0	7	4	1,5,3
#8	7	6	5	2	9	3	8	0	1	4	1,9,5

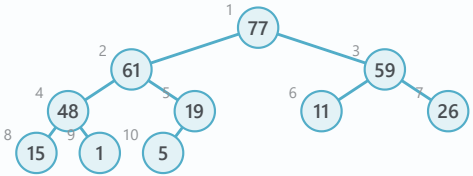
Sonuç listesi (baş 4): -5 → 7 → 11 → 18 → 27 → 30 → 42 → 55 → 83. Tablo simülasyonla üretildi.

Heap Sort

```c
void heapify(int a[], int kok, int n) {
    int t = a[kok], j;
    for (j = 2 * kok; j <= n; j *= 2) {
        if (j < n && a[j] < a[j + 1]) j++; // büyük
        if (t >= a[j]) break;   // t buraya sığar
        a[j / 2] = a[j];        // çocuğu yukarı al
    }
    a[j / 2] = t;
}
void heapsort(int a[], int n) {     /* a[1..n] */
    int i;
    for (i = n / 2; i >= 1; i--)     /* heap kur */
        heapify(a, i, n);
    for (i = n - 1; i >= 1; i--) {   /* sırala */
        SWAP(a[1], a[i + 1]);    /* en büyüğü sona */
        heapify(a, 1, i);            /* heap'i düzelt */
    }
}
```

- Karmaşıklık merge sort gibi **O(n lg n)**; ama **fazladan bellek kullanmaz** (yerinde).
- Dizi a[1..n] (0 kullanılmaz) **tam ikili ağaç** gibi düşünülür: a[i]'nin çocukları a[2i], a[2i+1].
- `heapify(a, kok, n)` : kok'ün sol ve sağ alt ağaçları zaten max-heap iken kök değerine uygun yeri arar.
- Yapraklar atlanır: ⌊n/2⌋, ⌊n/2⌋−1, ..., 1 düğümleri için sondan başa heapify → max-heap.
- Sonra kökteki en büyük sona taşınır, kalan i öğe için heap düzeltilir.
- `for` başlığında tanımlanan `j` ve `i` döngü dışında kullanılamaz (C99'da kapsam dışı).

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
77	61	59	48	19	11	26	15	1	5

heap bölgesi a[1..10]

Girdi [26, 5, 77, 1, 61, 11, 59, 15, 48, 19] (Horowitz örneği), i = 5...1 için heapify ile aşağıdan yukarıya bu max-heap'e çevrildi (toplam 5 kaydırma). "Tamam hocam! Onu anladık da, bu dizi ne zaman sıralanacak?" — Şimdi: kökteki en büyük (77) her turda sona taşınır, kalan heap düzeltilir.

Adım 1 / 19

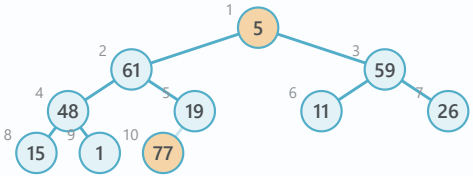
```
 1 void heapify(int a[], int kok, int n) {
 2     int t = a[kok], j;
 3     for (j = 2 * kok; j <= n; j *= 2) {
 4         if (j < n && a[j] < a[j + 1]) j++; // büyük
 5         if (t >= a[j]) break;   // t buraya sığar
 6         a[j / 2] = a[j];        // çocuğu yukarı al
 7     }
 8     a[j / 2] = t;
 9 }
▶    for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`n = 10`

`a = [77, 61, 59, 48, 19, 11, 26, 15, 1, 5]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	61	59	48	19	11	26	15	1	77

heap bölgesi a[1..9]

i = 9: SWAP(a[1], a[10]) → en büyük 77 sona; kökte 5 heap'i bozuyor.

Adım 2 / 19

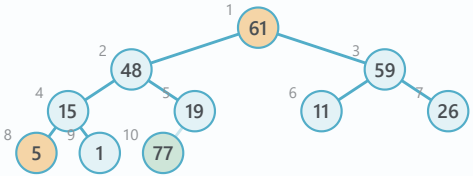
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 9`

`a = [5, 61, 59, 48, 19, 11, 26, 15, 1, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
61	48	59	15	19	11	26	5	1	77

heap bölgesi a[1..9] · sıralı kısım yeşil

heapify(a, 1, 9): 5 aşağı indi (a[8]); yeni kök **61** = kalanların en büyüğü.

Adım 3 / 19

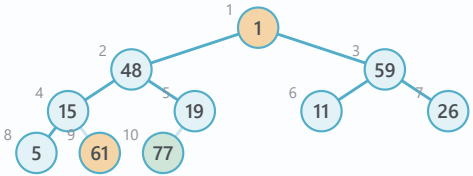
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 9`

`a = [61, 48, 59, 15, 19, 11, 26, 5, 1, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	48	59	15	19	11	26	5	61	77

heap bölgesi a[1..8] · sıralı kısım yeşil

i = 8: SWAP(a[1], a[9]) → en büyük 61 sona; kökte 1 heap'i bozuyor.

Adım 4 / 19

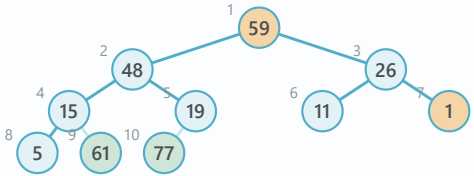
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 8`

`a = [1, 48, 59, 15, 19, 11, 26, 5, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
59	48	26	15	19	11	1	5	61	77

heap bölgesi a[1..8] · sıralı kısım yeşil

heapify(a, 1, 8): 1 aşağı indi (a[7]); yeni kök **59** = kalanların en büyüğü.

Adım 5 / 19

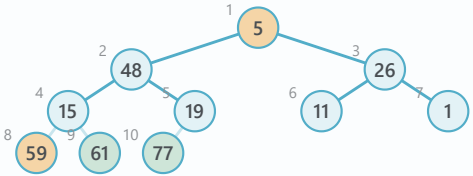
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 8`

`a = [59, 48, 26, 15, 19, 11, 1, 5, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	48	26	15	19	11	1	59	61	77

heap bölgesi a[1..7] · sıralı kısım yeşil

i = 7: SWAP(a[1], a[8]) → en büyük 59 sona; kökte 5 heap'i bozuyor.

Adım 6 / 19

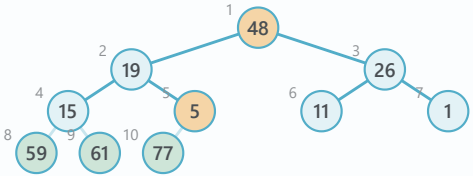
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 7`

`a = [5, 48, 26, 15, 19, 11, 1, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
48	19	26	15	5	11	1	59	61	77

heap bölgesi a[1..7] · sıralı kısım yeşil

heapify(a, 1, 7): 5 aşağı indi (a[5]); yeni kök **48** = kalanların en büyüğü.

Adım 7 / 19

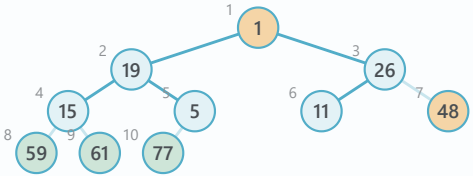
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 7`

`a = [48, 19, 26, 15, 5, 11, 1, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	19	26	15	5	11	48	59	61	77

heap bölgesi a[1..6] · sıralı kısım yeşil

i = 6: SWAP(a[1], a[7]) → en büyük 48 sona; kökte 1 heap'i bozuyor.

Adım 8 / 19

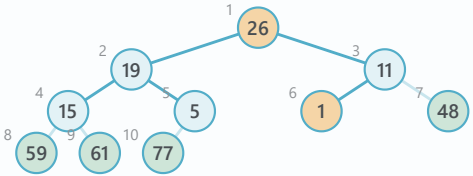
```
 1 void heapify(int a[], int kok, int n) {
 2     int t = a[kok], j;
 3     for (j = 2 * kok; j <= n; j *= 2) {
 4         if (j < n && a[j] < a[j + 1]) j++; // büyük
 5         if (t >= a[j]) break;   // t buraya sığar
 6         a[j / 2] = a[j];        // çocuğu yukarı al
 7     }
 8     a[j / 2] = t;
 9 }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶           SWAP(a[1], a[i + 1]);
12          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 6`

`a = [1, 19, 26, 15, 5, 11, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
26	19	11	15	5	1	48	59	61	77

heap bölgesi a[1..6] · sıralı kısım yeşil

heapify(a, 1, 6): 1 aşağı indi (a[6]); yeni kök **26** = kalanların en büyüğü.

Adım 9 / 19

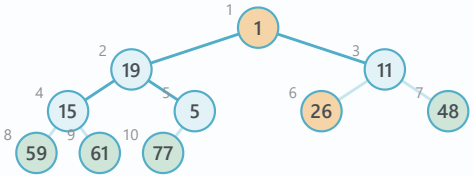
```
 1 void heapify(int a[], int kok, int n) {
 2     int t = a[kok], j;
 3     for (j = 2 * kok; j <= n; j *= 2) {
 4         if (j < n && a[j] < a[j + 1]) j++; // büyük
 5         if (t >= a[j]) break;   // t buraya sığar
 6         a[j / 2] = a[j];        // çocuğu yukarı al
 7     }
 8     a[j / 2] = t;
 9 }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶           heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 6`

`a = [26, 19, 11, 15, 5, 1, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	19	11	15	5	26	48	59	61	77

heap bölgesi a[1..5] · sıralı kısım yeşil

i = 5: SWAP(a[1], a[6]) → en büyük 26 sona; kökte 1 heap'i bozuyor.

Adım 10 / 19

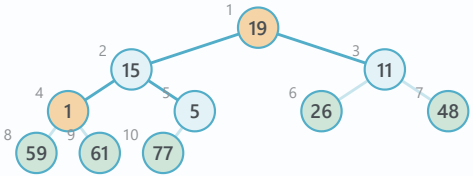
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 5`

`a = [1, 19, 11, 15, 5, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
19	15	11	1	5	26	48	59	61	77

heap bölgesi a[1..5] · sıralı kısım yeşil

heapify(a, 1, 5): 1 aşağı indi (a[4]); yeni kök **19** = kalanların en büyüğü.

Adım 11 / 19

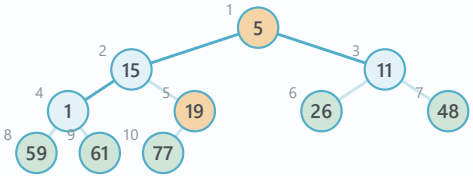
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 5`

`a = [19, 15, 11, 1, 5, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	15	11	1	19	26	48	59	61	77

heap bölgesi a[1..4] · sıralı kısım yeşil

i = 4: SWAP(a[1], a[5]) → en büyük 19 sona; kökte 5 heap'i bozuyor.

Adım 12 / 19

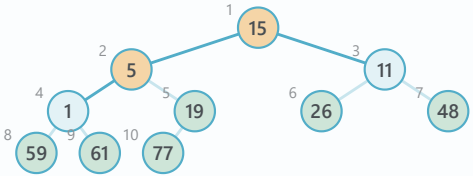
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 4`

`a = [5, 15, 11, 1, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
15	5	11	1	19	26	48	59	61	77

heap bölgesi a[1..4] · sıralı kısım yeşil

heapify(a, 1, 4): 5 aşağı indi (a[2]); yeni kök **15** = kalanların en büyüğü.

Adım 13 / 19

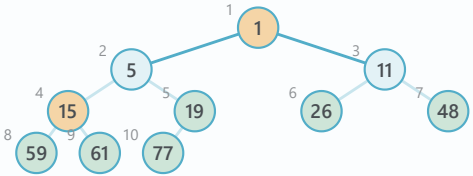
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 4`

`a = [15, 5, 11, 1, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap bölgesi a[1..3] · sıralı kısım yeşil

i = 3: SWAP(a[1], a[4]) → en büyük 15 sona; kökte 1 heap'i bozuyor.

Adım 14 / 19

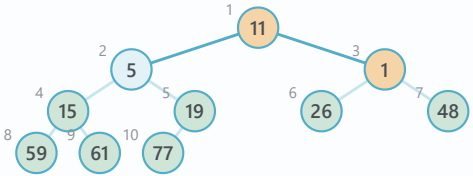
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 3`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
11	5	1	15	19	26	48	59	61	77

heap bölgesi a[1..3] · sıralı kısım yeşil

heapify(a, 1, 3): 1 aşağı indi (a[3]); yeni kök **11** = kalanların en büyüğü.

Adım 15 / 19

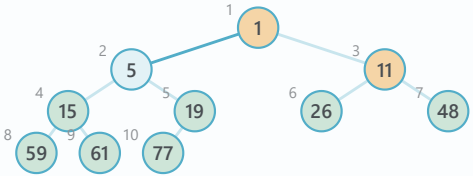
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 3`

`a = [11, 5, 1, 15, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap bölgesi a[1..2] · sıralı kısım yeşil

i = 2: SWAP(a[1], a[3]) → en büyük 11 sona; kökte 1 heap'i bozuyor.

Adım 16 / 19

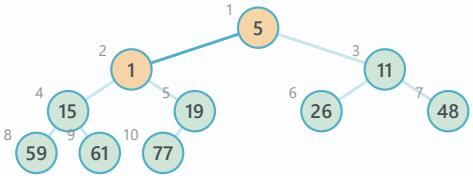
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 2`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	1	11	15	19	26	48	59	61	77

heap bölgesi a[1..2] · sıralı kısım yeşil

heapify(a, 1, 2): 1 aşağı indi (a[2]); yeni kök **5** = kalanların en büyüğü.

Adım 17 / 19

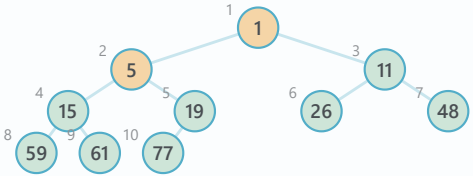
```
 1 void heapify(int a[], int kok, int n) {
 2     int t = a[kok], j;
 3     for (j = 2 * kok; j <= n; j *= 2) {
 4         if (j < n && a[j] < a[j + 1]) j++; // büyük
 5         if (t >= a[j]) break;   // t buraya sığar
 6         a[j / 2] = a[j];        // çocuğu yukarı al
 7     }
 8     a[j / 2] = t;
 9 }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶           heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 2`

`a = [5, 1, 11, 15, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap bölgesi a[1..1] · sıralı kısım yeşil

i = 1: SWAP(a[1], a[2]) → en büyük 5 sona; kökte kalan tek öğe (1) zaten bir heap'tir.

Adım 18 / 19

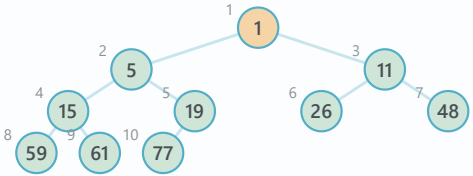
```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 1`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Kod İzleme: Sıralama Aşaması — SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap bölgesi a[1..1] · sıralı kısım yeşil

heapify(a, 1, 1): j = 2 > 1, döngüye girilmez; 1 yerinde kalır. Dizi sıralı: **[1, 5, 11, 15, 19, 26, 48, 59, 61, 77]**.

Adım 19 / 19

```
1  void heapify(int a[], int kok, int n) {
2      int t = a[kok], j;
3      for (j = 2 * kok; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // büyük
5          if (t >= a[j]) break;   // t buraya sığar
6          a[j / 2] = a[j];        // çocuğu yukarı al
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sırala */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

DEĞİŞKENLER

`i = 1`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Heap Sort: Zaman Karmaşıklığı

- **Kaba analiz:** heapify en çok lg n kaydırma yapar, n/2 kez çağrılır → heap kurma n lg n.
- **İnce analiz:** k = düzey sayısı ≈ lg(n+1); i. düzeyde en çok 2^{i-1} düğüm, her biri en çok (k − i) kayar:

$$\Sigma_{i=1}^{k} 2^{i-1}(k-i) = 2^k - k - 1 \leq 2n = O(n)$$

- Sağdaki Σ i·2^{i-1} terimi, geometrik seri S(y) = (y^{k+1} − 1)/(y − 1)'in türevinde y = 2 konarak bulunur.
- Sıralama aşaması: n − 1 kez heapify(a, 1, i) → O(n lg n). Toplam **O(n lg n)**, en kötü durumda da; ek bellek O(1); kararlı değil.
- "≤ 2n" sınırı **O(n)** demektir, o(n) değil (2n ile sınırlı olmak küçük-o değildir).

k	Σ $2^{i-1}(k-i)$	2^k-k-1	n = 2^k-1
2	1	1	3
3	4	4	7
4	11	11	15
6	57	57	63
10	1013	1013	1023

HOCADAN "Kruskal gerçeklenirken öğeler tek tek min-heap'e eklenerek heap oluşturulmuştu. O şekilde gerçekten heap oluşur. Ama öğelerin bulunduğu diziyi, onun yarısı kadar bir maliyetle min-heap hâline getirdik."

İleri Heapsort Algoritmaları ve Veriyi Tanımak

Ne zaman?

50.000'den az veri için heapsort makul bir algoritmadır (≈ 2n lg n karşılaştırma). Daha büyük veride karşılaştırma sayısını azaltan **bottom-up heapsort** ve kuramsal olarak daha da iyi sürümler vardır.

Bottom-up fikri

Kök boşluğu, t ile karşılaştırılmadan büyük çocuklar boyunca yaprağa kadar indirilir; sonra t yukarı doğru yerleştirilir. Düzey başına iki karşılaştırma yerine çoğunlukla bir.

HOCADAN "Counting sort örneğinde olduğu gibi, sıralanacak verilere ait bazı özellikleri biliyorsak, ona göre bir yaklaşım seçebiliriz. Bu bağlamda shell sort ve package sort'un birlikte kullanımları var. Temel birkaç tanesi bilinmeli."

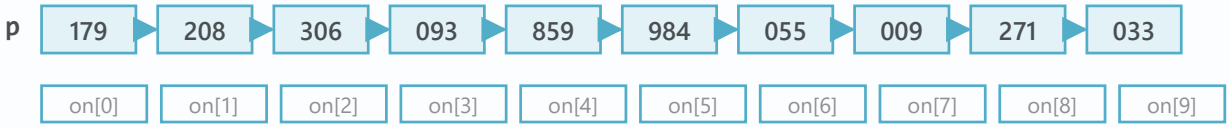
Basamağa Göre Sıralama (Radix Sort)

- Birden çok anahtara göre sıralama: ör. kişileri önce **soyadına**, soyadı aynı olanları **ada** göre.
- Anahtarlar K_0, K_1, …, K_{r-1}: K_0 **en anlamlı** (MSD), K_{r-1} **en az anlamlı** (LSD). Örnekte soyadı en anlamlı.
- **LSD radix sort:** önce en az anlamlı basamağa göre, sonra sırayla daha anlamlılara göre; en son MSD'ye göre sıralanır.
- Her geçiş **kararlı** olmalı: önceki geçişin düzeni eşit basamaklarda korunur.
- Sayıları birbiriyle karşılaştırmaz → karşılaştırmasız (*non-comparison*) sıralama.

```c
#define MAX_DIGIT 3
#define RADIX 10
typedef struct oge *oge_gost;
struct oge {
    int deger[MAX_DIGIT];  /* 179 → {1, 7, 9} */
    oge_gost bag;
};
```

K_0: yüzler (MSD) · K_1: onlar · K_2: birler (LSD). [0, 999] aralığı. `typedef struct oge {…};` gibi tür adı verilmeyen bir typedef anlamsızdır.

Kod İzleme: `radixsort` — i = 2 (birler)



Geçiş i = 2 (birler basamağı): 10 kova boşaltıldı.

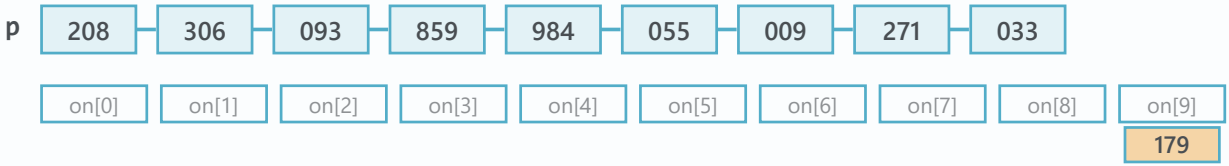
Adım 1 / 12

```
▶ for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2     for (int j = 0; j < RADIX; j++)
▶         on[j] = arka[j] = NULL;
4     for (; p; p = p->bag) {     // dağıt
5         int b = p->deger[i];
6         if (!on[b]) on[b] = p;
7         else arka[b]->bag = p;
8         arka[b] = p;
9     }
10    for (int j = RADIX - 1; j >= 0; j--) // topla
11        if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `p = 179→208→306→093…`

Kod İzleme: `radixsort` — i = 2 (birler)



179: deger[2] = 9 → kova 9'un **arkasına** eklendi (geliş sırası korunur → kararlı).

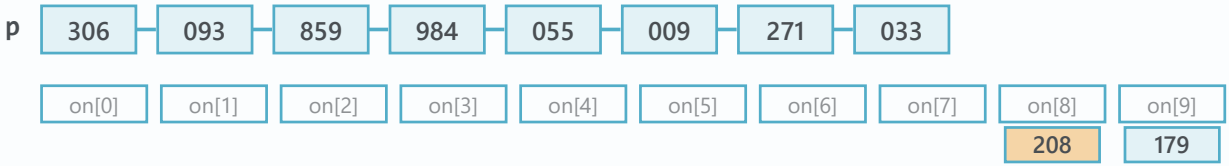
Adım 2 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 9` `p = 179`

Kod İzleme: `radixsort` — i = 2 (birler)



208: deger[2] = 8 → kova 8'in **arkasına** eklendi (geliş sırası korunur → kararlı).

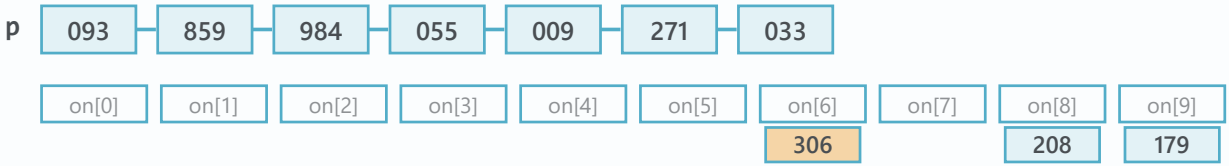
Adım 3 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 8` `p = 208`

Kod İzleme: `radixsort` — i = 2 (birler)



306: deger[2] = 6 → kova 6'nın **arkasına** eklendi (geliş sırası korunur → kararlı).

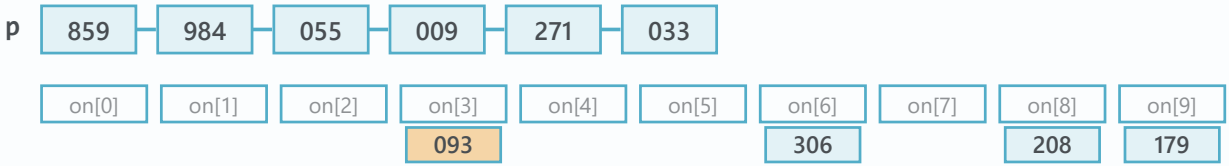
Adım 4 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 6` `p = 306`

Kod İzleme: `radixsort` — i = 2 (birler)



093: deger[2] = 3 → kova 3'ün **arkasına** eklendi (geliş sırası korunur → kararlı).

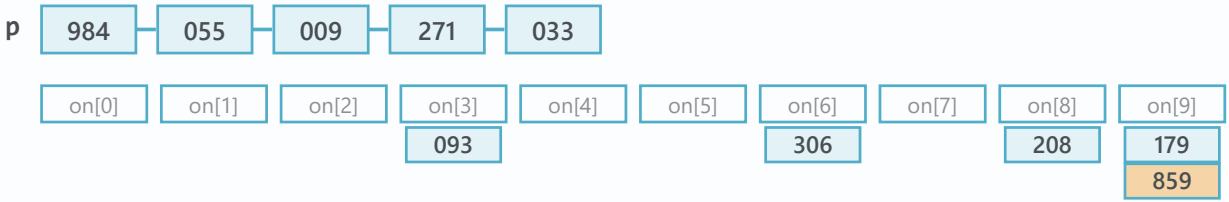
Adım 5 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 3` `p = 093`

Kod İzleme: `radixsort` — i = 2 (birler)



859: deger[2] = 9 → kova 9'un **arkasına** eklendi (geliş sırası korunur → kararlı).

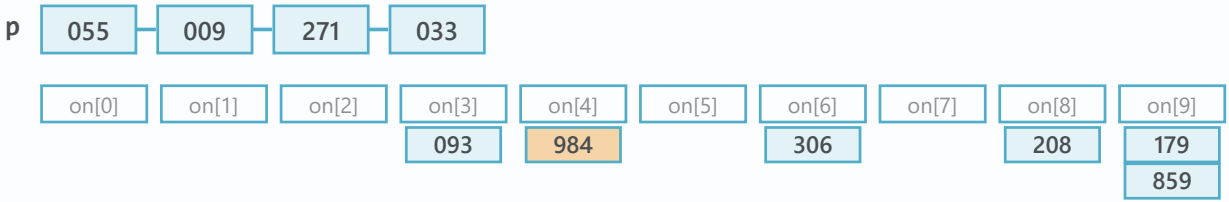
Adım 6 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 9` `p = 859`

Kod İzleme: `radixsort` — i = 2 (birler)



984: deger[2] = 4 → kova 4'ün **arkasına** eklendi (geliş sırası korunur → kararlı).

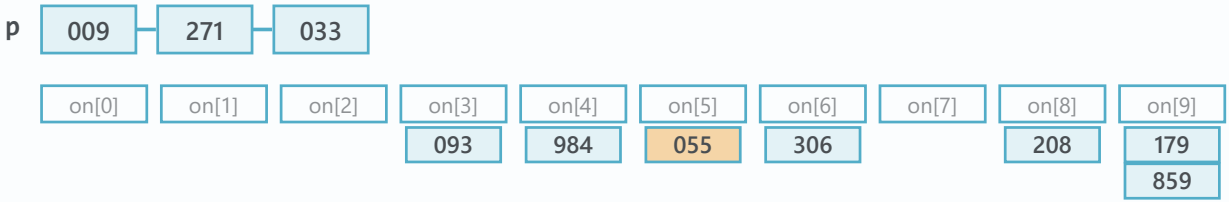
Adım 7 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 4` `p = 984`

Kod İzleme: `radixsort` — i = 2 (birler)



055: deger[2] = 5 → kova 5'in **arkasına** eklendi (geliş sırası korunur → kararlı).

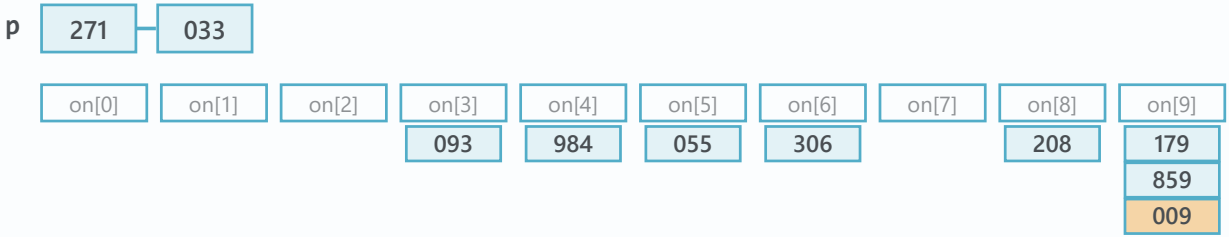
Adım 8 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 5` `p = 055`

Kod İzleme: `radixsort` — i = 2 (birler)



009: deger[2] = 9 → kova 9'un **arkasına** eklendi (geliş sırası korunur → kararlı).

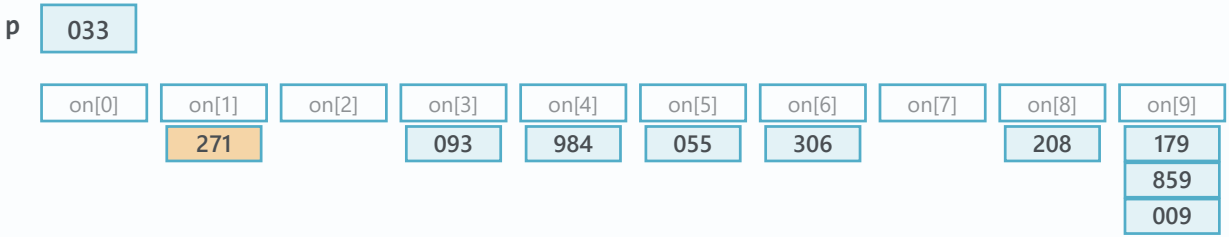
Adım 9 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 9` `p = 009`

Kod İzleme: `radixsort` — i = 2 (birler)



271: deger[2] = 1 → kova 1'in **arkasına** eklendi (geliş sırası korunur → kararlı).

Adım 10 / 12

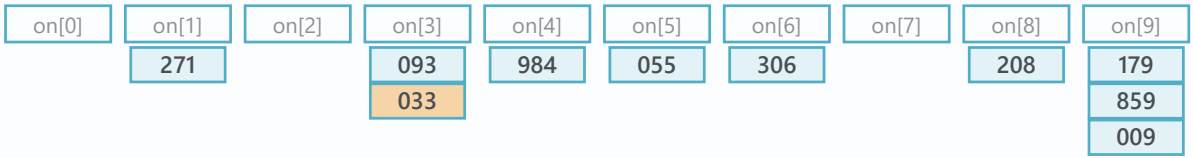
```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 1` `p = 271`

Kod İzleme: `radixsort` — i = 2 (birler)

p NULL



033: deger[2] = 3 → kova 3'ün **arkasına** eklendi (geliş sırası korunur → kararlı).

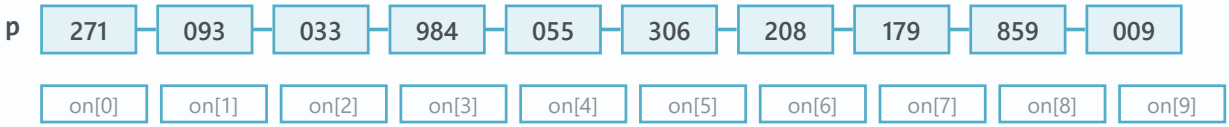
Adım 11 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
▶          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `b = 3` `p = 033`

Kod İzleme: `radixsort` — i = 2 (birler)



Toplama: kovalar 9'dan 0'a gezilip her kova listenin **önüne** bağlanır → küçük kova önde: 271 093 033 984 055 306 208 179 859 009.

Adım 12 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
5          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
8          arka[b] = p;
9      }
▶      for (int j = RADIX - 1; j >= 0; j--) // topla
▶          if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 2` `p = 271`

Kod İzleme (devam): i = 1 ve i = 0



Geçiş i = 1 (onlar basamağı): 10 kova boşaltıldı.

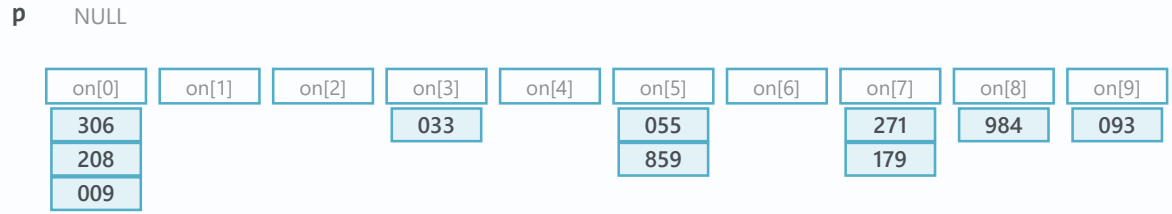
Adım 1 / 7

```
▶ for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2     for (int j = 0; j < RADIX; j++)
▶         on[j] = arka[j] = NULL;
4     for (; p; p = p->bag) {     // dağıt
5         int b = p->deger[i];
6         if (!on[b]) on[b] = p;
7         else arka[b]->bag = p;
8         arka[b] = p;
9     }
10    for (int j = RADIX - 1; j >= 0; j--) // topla
11        if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 1` `p = 271→093→033→984…`

Kod İzleme (devam): i = 1 ve i = 0



Dağıtım: her öğe onlar basamağına göre kovasının arkasına.

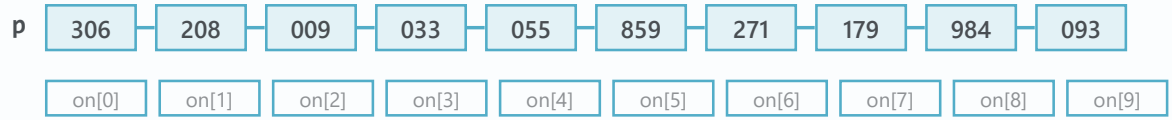
Adım 2 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
▶      for (; p; p = p->bag) {     // dağıt
5          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 1`

Kod İzleme (devam): i = 1 ve i = 0



Toplama: kovalar 9'dan 0'a gezilip her kova listenin **önüne** bağlanır → küçük kova önde: 306 208 009 033 055 859 271 179 984 093.

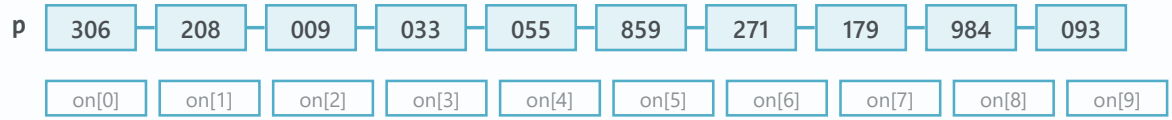
Adım 3 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {     // dağıt
5          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
8          arka[b] = p;
9      }
▶      for (int j = RADIX - 1; j >= 0; j--) // topla
▶          if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 1` `p = 306`

Kod İzleme (devam): i = 1 ve i = 0



Geçiş i = 0 (yüzler basamağı): 10 kova boşaltıldı.

Adım 4 / 7

```
▶ for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2     for (int j = 0; j < RADIX; j++)
▶         on[j] = arka[j] = NULL;
4     for (; p; p = p->bag) {     // dağıt
5         int b = p->deger[i];
6         if (!on[b]) on[b] = p;
7         else arka[b]->bag = p;
8         arka[b] = p;
9     }
10    for (int j = RADIX - 1; j >= 0; j--) // topla
11        if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

i = 0 | p = 306→208→009→033…

Kod İzleme (devam): i = 1 ve i = 0

p NULL



Dağıtım: her öğe yüzler basamağına göre kovasının arkasına.

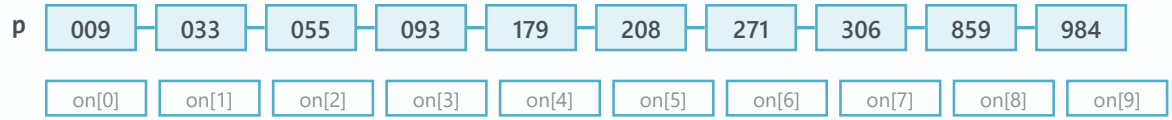
Adım 5 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
▶      for (; p; p = p->bag) {     // dağıt
5          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
▶          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

i = 0

Kod İzleme (devam): i = 1 ve i = 0



Toplama: kovalar 9'dan 0'a gezilip her kova listenin **önüne** bağlanır → küçük kova önde: 009 033 055 093 179 208 271 306 859 984.

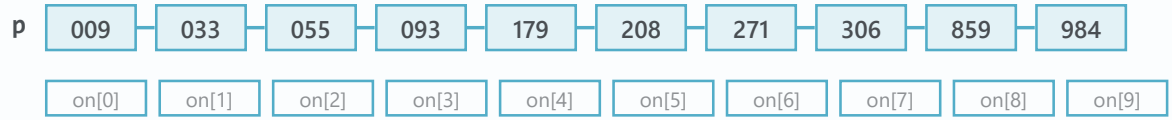
Adım 6 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
5          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
8          arka[b] = p;
9      }
▶      for (int j = RADIX - 1; j >= 0; j--) // topla
▶          if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
13 return p;
```

DEĞİŞKENLER

`i = 0` `p = 009`

Kod İzleme (devam): i = 1 ve i = 0



Üç geçiş sonunda liste sıralı; baş işaretçisi döndürülür.

Adım 7 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          on[j] = arka[j] = NULL;
4      for (; p; p = p->bag) {      // dağıt
5          int b = p->deger[i];
6          if (!on[b]) on[b] = p;
7          else arka[b]->bag = p;
8          arka[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // topla
11         if (on[j]) { arka[j]->bag = p; p = on[j]; }
12 }
▶ return p;
```

DEĞİŞKENLER

p = **009**

Basamağa Göre Sıralama: Analiz

$$O(MAX_DIGIT \cdot (RADIX + n + RADIX)) = O(MAX_DIGIT \cdot (RADIX + n))$$

- Örnekte O(3 · (10 + n)): doğrusal görünür ama sayarak sıralama gibi **özel bir durumdur**: basamak sayısı ve taban sabit kabul edilmiştir.
- Veriler (a) pozitif, (b) tamsayı, (c) küçük aralıkta olmalı. Aralık büyürse yine sıralar ama karmaşıklık hızla artar.
- Ek bellek: 2·RADIX işaretçi (liste yerinde yeniden bağlanır).

HOCADAN "Sadece tamsayıları sıralayabilmesi bir dezavantajdır. Kayan noktalı sayı türleri için uyarlanabilir, ama bu uyarlamayı yapana kadar diğer algoritmalar sıralamış olur."

geçiş	liste
başlangıç	179 208 306 093 859 984 055 009 271 033
i = 2	271 093 033 984 055 306 208 179 859 009
i = 1	306 208 009 033 055 859 271 179 984 093
i = 0	009 033 055 093 179 208 271 306 859 984

SINAV SORULARI

Sınav Sorusu: Değiştirilmiş Radix

Bir öğrenci, yordamın önce en anlamlı basamağı ele alması gerektiğini düşünüp dış döngüyü `for (i = 0; i ≤ MAX_DIGIT-1; i++)`, toplama döngüsünü `for (j = 0; j ≤ RADIX_SIZE-1; j++)` yapmıştır. Aşağıdaki liste için yordamı işletin, her adımda listeyi yazın ve sonucu yorumlayın.

`p → 346 180 008 930 023 451 670 098 203 007`

▶ **Çözüm**

ÖZET

Karmaşıklık Özeti ve Karşılaştırma

algoritma	en iyi	ortalama	en kötü	ek bellek	kararlı	yerinde
Seçimli	n^2	n^2	n^2	O(1)	hayır	evet
Eklemeli	n	n^2	n^2	O(1)	evet	evet
Kabarcık (erken dur.)	n	n^2	n^2	O(1)	evet	evet
Shell (Knuth)	n lg n	—	$n^{3/2}$	O(1)	hayır	evet
Hızlı (quick)	n lg n	1,39 n lg n	n^2	O(lg n) yığın*	hayır	evet
Birleştirme (merge)	n lg n	n lg n	n lg n	O(n)	evet**	hayır
Heap sort	n lg n	n lg n	n lg n	O(1)	hayır	evet
Sayarak (counting)	n + k	n + k	n + k	O(n + k)	evet	hayır
Basamak (radix, d basamak)	d(n + r)	d(n + r)	d(n + r)	O(r)***	evet	—

* Küçük parça önce işlenirse; aksi hâlde en kötü O(n). ** Klasik merge ile. *** Bağlı liste üzerinde. Karşılaştırmalı sıralamada alt sınır $\lceil \lg n! \rceil \approx n \lg n - 1{,}44\, n$.

ÖZET

Hangi Durumda Hangisi?

Genel amaçlı, bellekte

Quick sort (medyan-3 + küçük parçada eklemeli) ya da kitaplık `qsort`.

En kötü durum garantisi

Heap sort (yerinde) ya da **introsort**.

Kararlılık / bağlı liste / disk

Merge sort: kararlı, sıralı parçaları birleştirir, dış sıralamanın temeli.

Büyük tutanaklar

Tutanaklar değil **indisler/bağlar** sıralanır (bağlı merge sort).

Sabit uzunluklu tamsayı/anahtar

Radix: d geçiş × O(n + r).

Küçük aralık

Sayarak sıralama.

ÖZET

Sık Yapılan Hatalar

1. bol()'da j döngüsünde `j == L` sınamasını unutmak: pivot en küçükse dizi başından taşar.
2. Sıralı girdide pivot hep a[R]: derinlik n olur, yığın taşar. Çare: medyan-3 ya da rastgele pivot.
3. Özyinelemesiz sürümde parçaları rastgele sırada itmek: yığın O(n)'e büyür.
4. merge sort'ta `b[]`'yi her çağrıda `malloc`'lamak (yavaş) ya da hiç ayırmamak.
5. heapify'da 1 tabanlı indisle 0 tabanlı diziyi karıştırmak: çocuklar 2i, 2i+1 ancak a[1..n]'de doğru.
6. qsort karşılaştırıcısında `return x - y;`: taşma; ya da tutarsız (geçişsiz) karşılaştırıcı.

ÖZET

Alıştırmalar

1. [5, 9, 1, 7, 3] için `bol(a, 0, 4)` sonucu diziyi ve dönüş değerini yazın.
2. Quick sort n = 8 sıralı dizide kaç bol() çağrısı yapar? Özyineleme derinliği nedir?
3. Merge sort 16 öğe için kaç merge çağrısı yapar? En çok kaç karşılaştırma?
4. [4, 10, 3, 5, 1] (a[1..5]) için heap kurma sonrası diziyi yazın.
5. Radix sort 170, 45, 75, 90, 802, 24, 2, 66 listesinde birler geçişinden sonraki sırayı yazın.
6. Heap sort neden kararlı değildir? Küçük bir örnek verin.

▶ **Cevapları göster**

ÖZET

Bölüm Özeti

Quick sort

Pivotu yerine koyar ve problemi böler: en iyi n lg n, ortalama 1,39 n lg n, en kötü n^2.

İyileştirmeler

Küçük parça önce (yığın lg n), medyan-3, küçük parçada eklemeli; qsort ve introsort.

Merge sort

Her durumda n lg n, O(n) ek bellek; bağlı sürüm büyük tutanakları kopyalamaz.

Heap sort

Heap kurma O(n), sıralama O(n lg n); yerinde ama kararsız.

Radix sort

Kararlı kovalarla LSD'den MSD'ye: O(d(n + r)); pozitif tamsayılar ve küçük aralık.

Analiz araçları

Özyinelemeli bağıntılar, iterasyon, çıkarma hilesi, seri türevi — dersin bütün analizleri burada buluştu.

ÖZET

Kaynaklar

- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 7–10 (quicksort, mergesort, priority queues/heapsort, radix sorting).
- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Bölüm 7.
- T. H. Cormen vd., *Introduction to Algorithms*, Bölüm 6–8; D. Musser, "Introspective Sorting and Selection Algorithms", 1997.
- ISO/IEC 9899 (C standardı), `qsort` tanımı (§7.22.5.2).

← Bölüm 19: Temel Sıralama Algoritmaları

Ders sayfası

Son

Bölüm 20: İleri Sıralama Algoritmaları

"Ustada kalırsa bu öksüz yapı, / Onu sürdürmeyen çırak utansın!" — N. F. K.

Dersin sonu · Ders sayfası