# VERİ YAPILARI

## DATA STRUCTURE

Bilgisayar Mühendisliği Dr.Günay TEMÜR

# Algoritmalarda Analiz

- Bir problemin çözümü için öne sürülen basamaklı kurallar bütününe **algoritma** denir.
- Bir algoritmanın iyi olabilmesi için şu özelliklere sahip olması gerekir:
  - **Çalışma Hızı Yüksek Olması**
  - **Karmaşıklığının Az Olması**
  - **Anlaşılır Olması**
  - **Bellek Kullanımı Minimum Olması**

# Algoritmalarda Analiz

- Algoritmaların değerlendirilmesi işlemlerinin sonucunda iyi veya kötü, yeterli veya yetersiz, hızlı veya yavaş, bellek kullanımı iyi veya kötü gibi kıyaslamaların bütününe **Algoritma Analizi** denir.

# Algoritmalarda Analiz

- Bir algoritmanın analizinde bakılacak önemli iki unsur vardır.

Bunlar ;

- **zaman** *(time complexity)* ve
- **hafıza** *(space complexity)* kullanımlarıdır.

Bu ikisinin karmaşıklıkları hesaplanırsa bir algoritma hakkında yorum yapabiliriz

# Neden algoritmayı analiz ederiz?

- Algoritmanın performansını ölçmek için
- Farklı algoritmalarla karşılaştırmak için
- Daha iyisi mümkün mü? Değerlendirmek için
- Olabileceklerin en iyisi mi?
  - Sorularına cevap verebilmek için.

# Neden algoritmayı analiz ederiz?

- Yazdığınız algoritmanın veri büyüklüğüne göre ne kadar zaman alacağı, karmaşıklığının nasıl arttığı çok önemlidir.
- Yazdığınız algoritma kaynak kullandığı için, bu kaynaklarda para anlamına geldiği için yazdığınız algoritmanın complexity (karmaşıklığı) çok önemlidir..

# Temel Kavramlar

- Yürütme Zamanı:
  - Bir programın veya algoritmanın işlevini yerine getirebilmesi için, temel kabul edilen işlemlerden kaç adet yürütülmesi gerektiğini veren bir bağıntıdır.
    - n elemanlı bir küme için yürütme zamanı T(n) dir.

# Algoritmanın Yürütme Zamanı

- Yürütme Zamanı T(n) = 1
  - `cout << "Günay Temür";`

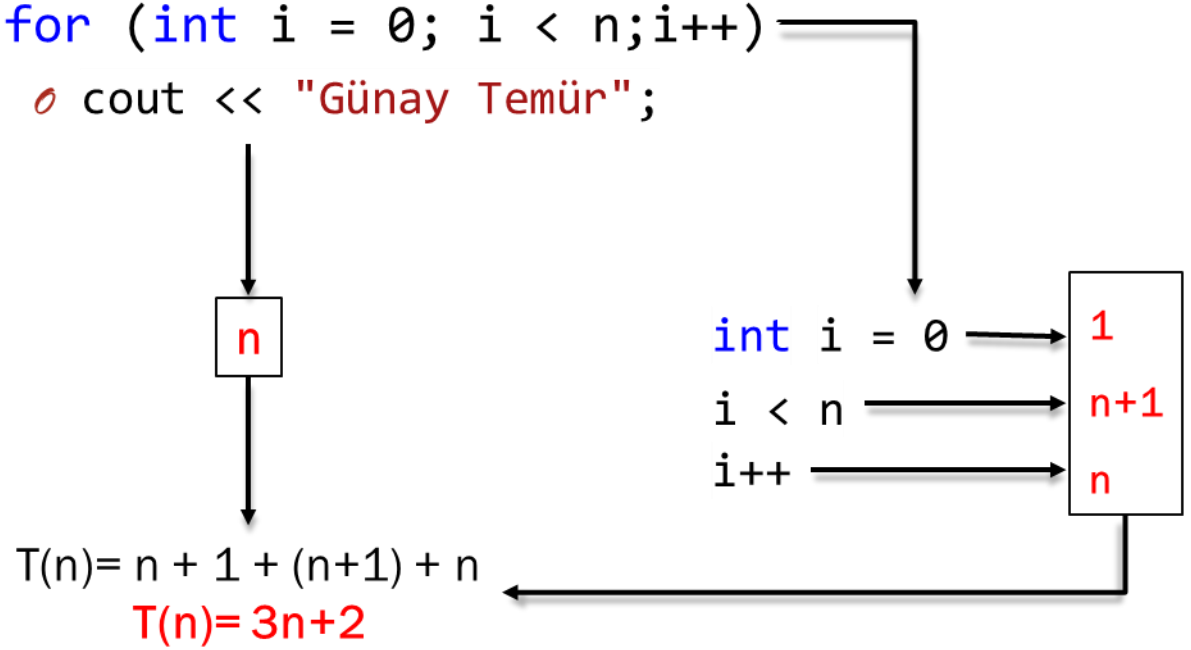
  Her durumda yaptığı işlem 1 tanedir. Bu sebeple **1**

# Algoritmanın Yürütme Zamanı

- n olarak ifade edilen birim işlemlerin toplamı T(n) yürütme zamanı fonksiyonunu verecektir

# Algoritmanın Yürütme Zamanı

- Örnek 2
  - Eğer bu işlem bir döngü ise;
    - `for (int i = 0; i < n;i++)`
      - `cout << "Günay Temür";`



# Algoritmanın Yürütme Zamanı

```
1  for (int i = 0; i < n; i++)
2
3      cout << "Günay Temür";          ⟶ 3n+2
4
5  for (int i = 0; i < n; i++)
6
7      cout << "Günay Temür";          ⟶ 3n+2
8
9  for (int i = 0; i < n; i++)
10
11     cout << "Günay Temür";          ⟶ 3n+2
```

İç içe olamayan döngülerde yürütme zamanları toplanır. T(n)= 9n+6

# Algoritmanın Yürütme Zamanı

```
1  for (int i = 0; i < n; i++)
2
3  for (int j = 0; j < n; j++)
4
5    o cout << "Günay Temür";
```

2n+2

3n+2

İç içe olan döngülerde yürütme zamanları çarpılır. $T(n)= (3n+2)*(2n+2)$

$$T(n)= 6n^2+10n+4$$

# Algoritmanın Yürütme Zamanı

- Reel Time= T(n)*C(n)
  - T(n): Yürütme adım sayısı
  - C(n): İşlemci iş zamanı

# Temel Kavramlar

- Zaman Karmaşıklığı: Büyüme Oranı
  - Karmaşıklık ifadesi, özellikle çok sayıda ya da büyüyen veri sayısı karşısında farklı algoritmaların nasıl davrandığını gösteren bir kavramsal ifadedir.
  - Bir algoritmanın asimtotik notasyona göre karmaşıklık mertebesini gösteren bir ifadedir. } Asimtotik ifade
  - O(n) -> big Oh

# Karmaşıklık Durumları

- Wast Case (Kötü Durum)
- Best Case (İyi Durum)
- Avarage Case (Ortalama Durum)

# Zaman Karmaşıklığı Dereceleri

- **O(1) Sabit Zaman :**bir ya da sabit bir sayıda komutun çalıştığı algoritmaların karmaşıklığıdır
- **O(n) Doğrusal zaman :** Giriş sayısı ile işlem miktarı oranı bir katsayı ile belirlenebilen algoritmaların karmaşıklığıdır
- **O($\log_2 n$) Logaritmik zaman :** Büyük problemlerin bölünüp küçültüldüğü algoritmaların karmaşıklığıdır.

# Zaman Karmaşıklığı Dereceleri

- **$O(n.\log_2 n)$ :** Genellikle problemin küçük parçalara bölünüp ayrı ayrı çözümlerinin birleştirildiği türde algoritmaların karmaşıklığıdır.
- **$O(n^2)$ Karesel zaman :** Problemin çözümünde iç içe döngüler kullanılıyor ise bu algoritmaların karmaşıklığı karesel zaman notasyonu ile gösterilir.

# zaman karmaşıklığını hesaplarken

1. Dizinin boyutuyla ilişkisi olmayan komutlar **O(1)** zamanı alınır.
2. Döngüler **O(n)** zamanı alınır. *(n dizi boyutu)*
3. İç içe döngüler zaman karmaşıklıklarının çarpılması şeklinde hesaplanır.
4. Birinci döngü n ikinci döngü n kere dönüyor ise zaman karmaşıklığı **O($n^2$)** zamanı alınır.
5. Art arda döngüler zaman karmaşıklıklarının toplanması şeklinde hesaplanır.

# zaman karmaşıklığını hesaplarken

6. If-then-else gibi şartlı yapılarda iki koşuldan zaman karmaşıklığı yüksek olan alınır.
7. Her bir icra, problemi, dizi boyutunun yarısı şeklinde ikiye bölüyorsa, zaman karmaşıklığı **$O(\log_2 n)$** zamanı alınır.
8. Katsayısı daha küçük olan zaman karmaşıklıkları ihmal edilir.
9. Sabit çarpanlar karmaşıklığı etkilemez. Yani;

# zaman karmaşıklığı

- Daha önce hesapladığımız fonksiyonun
- T(n) yürütme zamanı;
  - T(n) = 3n + 2
- Karmaşıklık notasyonu;
  - O(n) olur.

# zaman karmaşıklığı

- T(n) yürütme zamanı;
  - **$T(n)= 6n^2+10n+4$**
- Karmaşıklık notasyonu;
  - $O(n^2)$ olur.

# Algoritma Analizi

## YÜRÜTME ZAMANI ANALİZİ

```
int ortalama(int dizi[], int len)
{
	int i, ort, top = 0;
	for (i = 0; i<len; i++)
		top += dizi[i];
	ort = top / len;
	return ort;
}
```

## BÜYÜME ORANI ANALİZİ

```
int ortalama(int dizi[], int len)
{
	int i, ort, top = 0;
	for (i = 0; i<len; i++)
		top += dizi[i];
	ort = top / len;
	return ort;
}
```

# Algoritma Analizi

## YÜRÜTME ZAMANI ANALİZİ

```
int ortalama(int dizi[], int len)       // len= n
{
        int i, ort, top = 0;            -> 1
        for (i = 0; i<len; i++)         -> 1+(n+1)+n
                top += dizi[i];         -> n
        ort = top / len;                -> 1
        return ort;                     -> 1
}                           T(n)        => 5+3n
```

## BÜYÜME ORANI ANALİZİ

```
int ortalama(int dizi[], int len)
{
        int i, ort, top = 0;
        for (i = 0; i<len; i++)
                top += dizi[i];
        ort = top / len;
        return ort;
}
```

3n => O(n)

Sadece for döngüsünün yaptığı işe, diğer sabitler atılmış olarak baktığımızda

# Rekürsive Fonk. İçin Yürütme Zamanı Analizi

```
int faktoryel(int n)
{
    if (n <= 1)
    return 1;
    else
    return n*faktoryel(n - 1);
}
```

Programın çalışma süresi, programın bir parçasının çalışma süresine bağlıdır.

# Rekürsive Fonk. İçin Yürütme Zamanı Analizi

- n elemanlı bir hesaplama için, n-1 olan çalışma süresine bağlıdır.
- n=1 için T(1)=1'dir. Yani
- T(n) = 1+ T(n-1)
- T(n-1) = 1+ T(n-2)
- T(n-2) = 1+ T(n-3)
- T(n-3) = 1+ T(n-4)
- T(1) = 1+ 0

# Rekürsive Fonk. İçin Yürütme Zamanı Analizi

- n elemanlı bir hesaplama için, n-1 olan çalışma süresine bağlıdır.
- n=1 için T(1)=1'dir. Yani
- T(n) = 1+ T(n-1)
- T(n-1) = 1+ T(n-2)
- T(n-2) = 1+ T(n-3)
- T(n-3) = 1+ T(n-4)
- . . .
- T(1) = 1+ 0

n

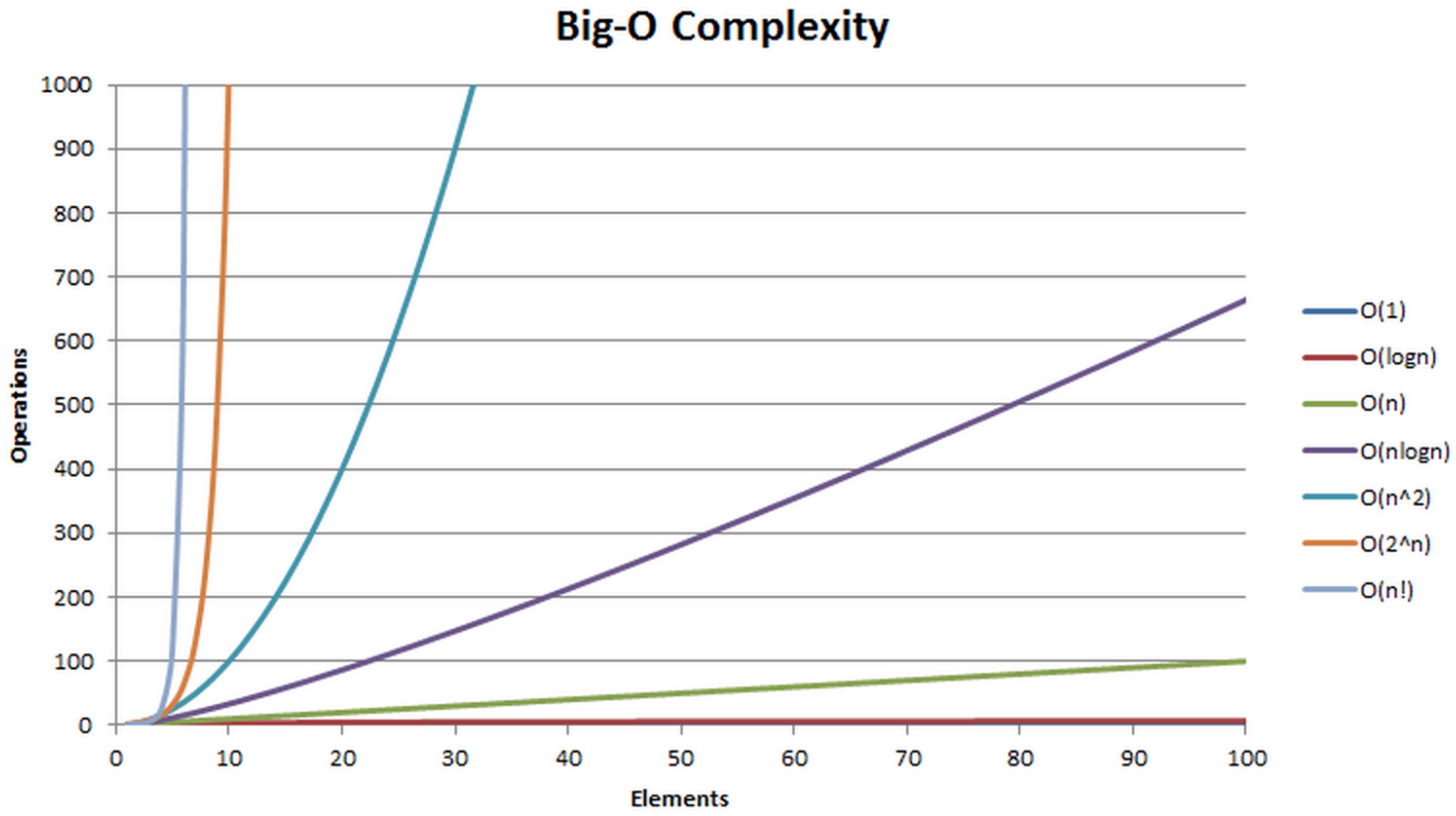
---

- T(n) = 1.n
- n ∈ O(n)

# Karmaşıklık Tablosu ve Anlamları

| Büyük O | Değişim Şekli |
|---|---|
| O(1) | Sabit |
| O(logn) | Logaritmik artış |
| O(n) | Doğrusal artış |
| O(nlogn) | Doğrusal çarpanlı logaritmik |
| $O(n^2)$ | Karesel artış |
| $O(n^3)$ | Kübik artış |
| $O(2^n)$ | İki tabanında üssel artış |
| $O(10^n)$ | On tabanında üssel artış |
| O(n!) | Faktöryel artış |

# Karmaşıklık Grafiği



# Data Structure Operations

| Data Structure | Time Complexity | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |
| Array | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) |
| Stack | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| Singly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| Doubly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| | | | | | | | | |
| Hash Table | - | O(1) | O(1) | O(1) | - | O(n) | O(n) | O(n) |
| Binary Search Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) |

# Program Bellek Gereksinimi

**Kod için;** Programın tasarımına bağlıdır.

**Veri için;** değişken, sabit sayısı ve türüne bağlıdır.

**Yığın için;** bilgilerin geçici olarak saklandığı bellek alanına bağlıdır.

# Bellek Gereksinimi Hesabı

```
float aritmatikort(int A[], int N)
{
    int A[20] = { 7,65,3,23,64,58,69,35,34,79,
                  62,12,38,62,52,46,47,95,24,27 };
    int k, toplam = 0;
    float ort;
    for (k = 0; k<20; k++)
    toplam += A[k];
    ort = (float)toplam / 20.0;
    printf("Ortalama değer = %f\n", ort);
}
```

# Bellek Gereksinimi Hesabı

- Öncelikle bellekte yer işgal edecek değişken bilgileri edinilir.
- A, k, toplam, ort
- İnt ve Float değişkenlerinin bellekte 4 byte yer tuttuğunu bildiğimizden
- 4*20+4*1+4*1+4*1= **92 byte** bellek ihtiyacımız vardır.

# ALGORİTMALAR

## SIRALAMA ve ARAMA

# Sıralama Algoritmaları

- Bilgisayar bilimlerinde verilmiş olan bir grup sayının küçükten büyüğe (veya tersi) sıralanması işlemini yapan algoritmalara verilen isimdir. Bunlardan bazıları
- Seçerek Sıralama (Selection Sort)
- Ekleme Sıralaması ( Insertion Sort)
- Kabarcık Sıralaması (Baloncuk sıralaması, Bubble Sort)
- Hızlı Sıralama Algoritması (Quick Sort Algorithm)
- Birleştirme Sıralaması (Merge Sort)
- Sayarak Sıralama (Counting Sort)
- Yığınlama Sıralaması (Heap Sort)

# Arama Algoritmaları

- Bilgisayar bilimlerinde, çeşitli veri yapılarının (data structures) üzerinde bir bilginin aranması sırasına kullanılan algoritmaların genel ismidir. Örneğin bir dosyada bir kelimenin aranması, bir ağaç yapısında (tree) bir düğümün (node) aranması veya bir dizi (array) üzerinde bir verinin aranması gibi durumlar bu algoritmaların çalışma alanlarına girer.
- Yapısal olarak arama algoritmalarını iki grupta toplamak mümkündür.
  - Uninformed Search (Bilmeden arama)
  - Informed Search (Bilerek arama)

# Arama Algoritmaları

- Her durumda aynı şekilde çalışan algoritmalara uninformed search (bilmeden arama) ismi verilir.
- Genelde kullanılan arama algoritmalarının temelini bu grup oluşturmaktadır.
- Bu aramaların bazıları şunlardır:
  - Listeler (diziler (array)) üzerinde çalışan arama algoritmaları:
    - Doğrusal Arama (Linear Search)
    - İkili arama (binary search)

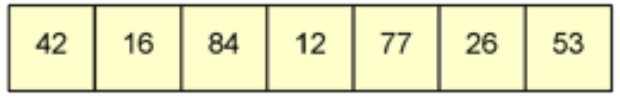
# Arama Algoritmaları

- Şekiller (graflar (Graphs) ) üzerinde çalışan algoritmalar
  - Kruskal Algoritması
  - Dijkstra Algoritması
  - Bellman Ford Algoritması
  - İkili arama ağacı (Binary Search Tree)
  - Sığ öncelikli arama (Breadth First Search, BFS)
  - Derin öncelikli arama (depth first search)

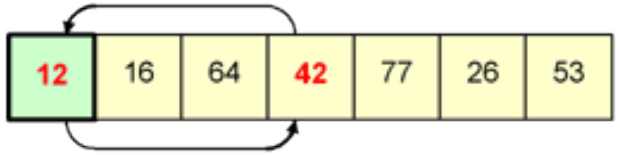
# Arama Algoritmaları

- Metin arama algoritmaları (bir yazı içerisinde belirli bir dizgiyi (string) arayan algoritmalar)
  - Kaba Kuvvet Metin Arama Algoritması (Brute Force Text Search, Linear Text Search)
  - Horspool Arama Algoritması
  - Knuth-Morris Prat arama algoritması
  - Boyer-Moore Arama algoritması

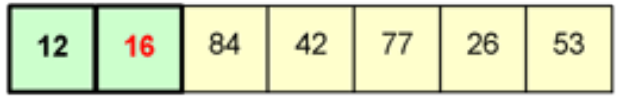
# Selection Sort



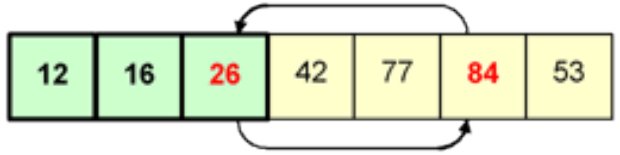

The array, before the selection sort operation begins.



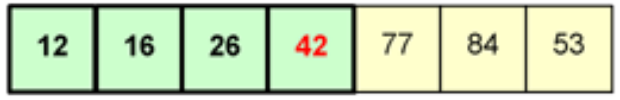

The smallest number (**12**) is swapped into the first element in the structure.



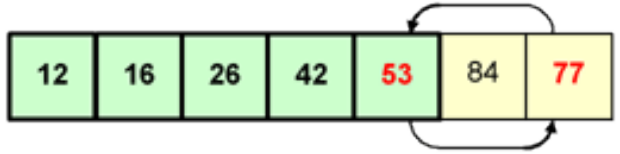

In the data that remains, **16** is the smallest; and it does not need to be moved.



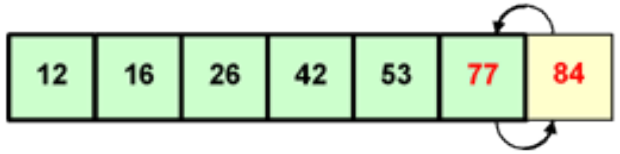

**26** is the next smallest number, and it is swapped into the third position.




**42** is the next smallest number; it is already in the correct position.

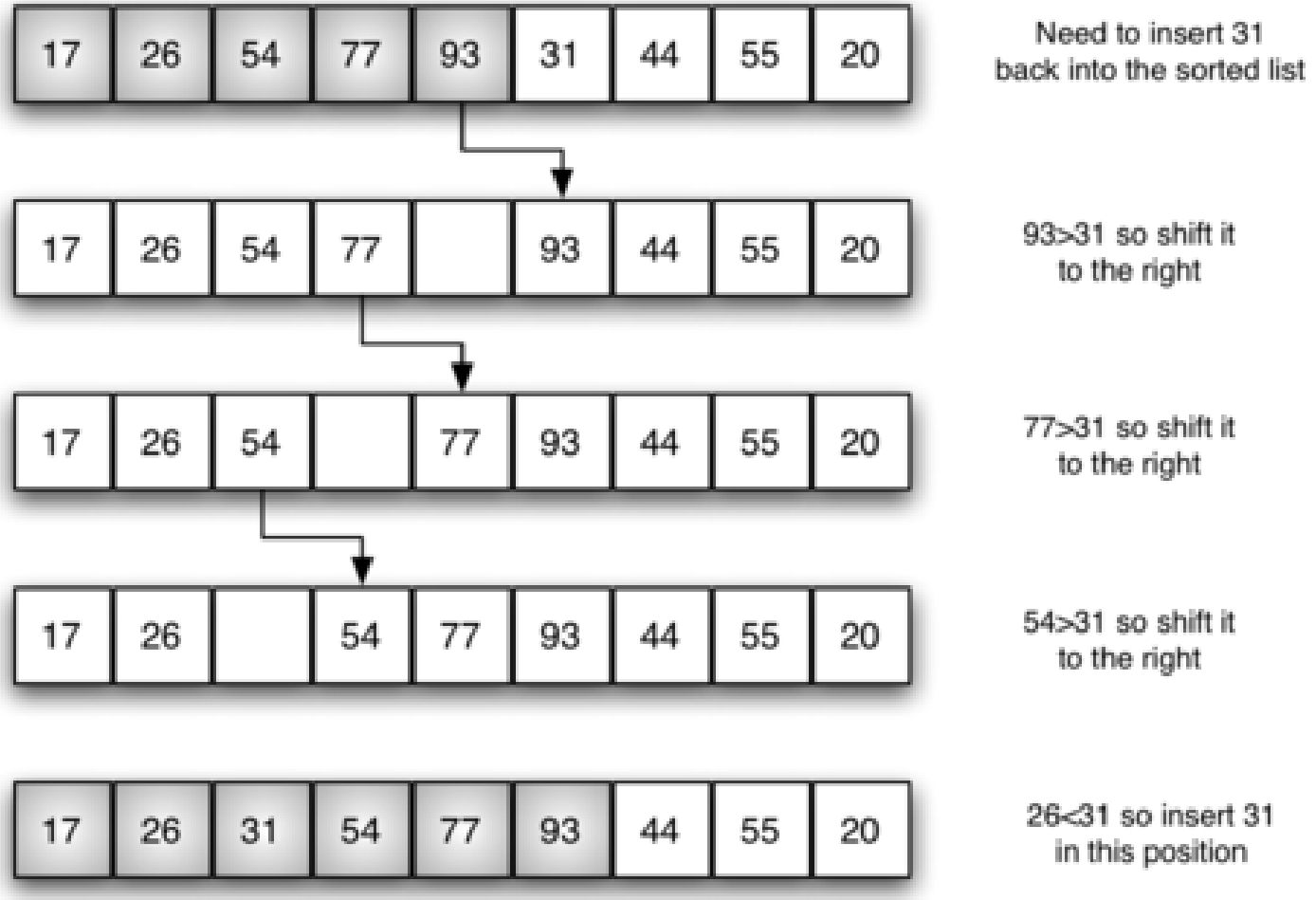



**53** is the smallest number in the data that remains; and it is swapped to the appropriate position.

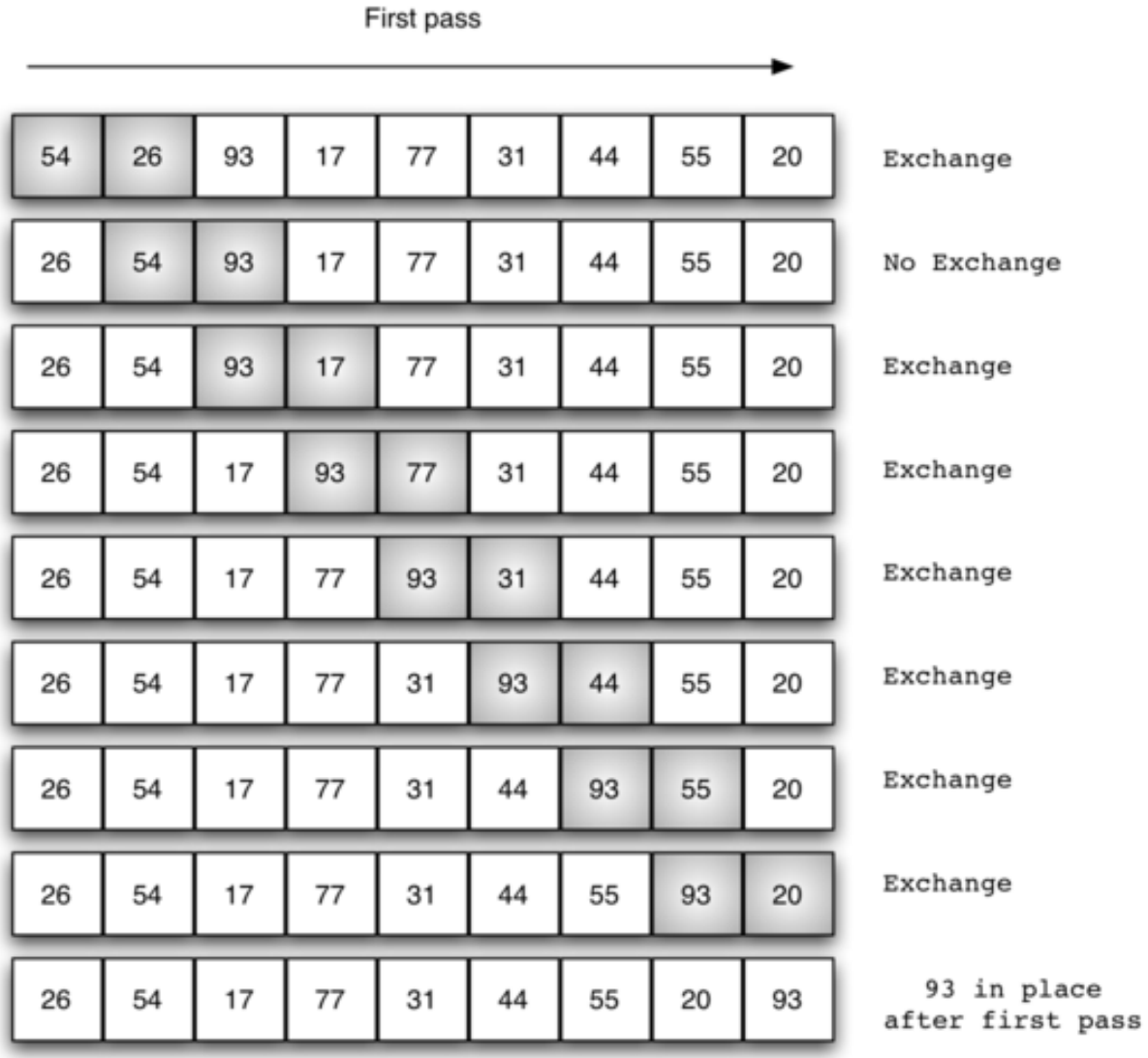



Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*

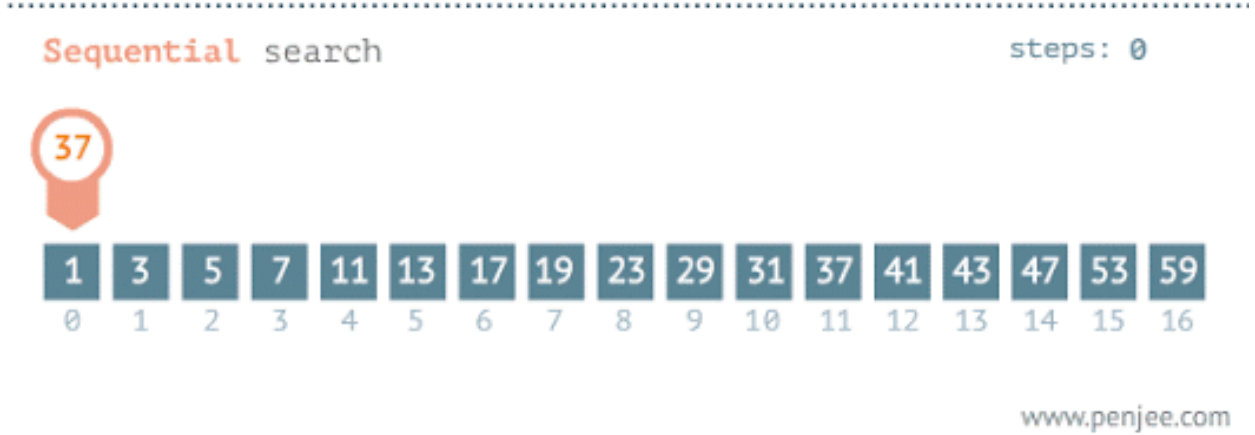
# Insertion Sort (Araya Ekleme Sıralaması)



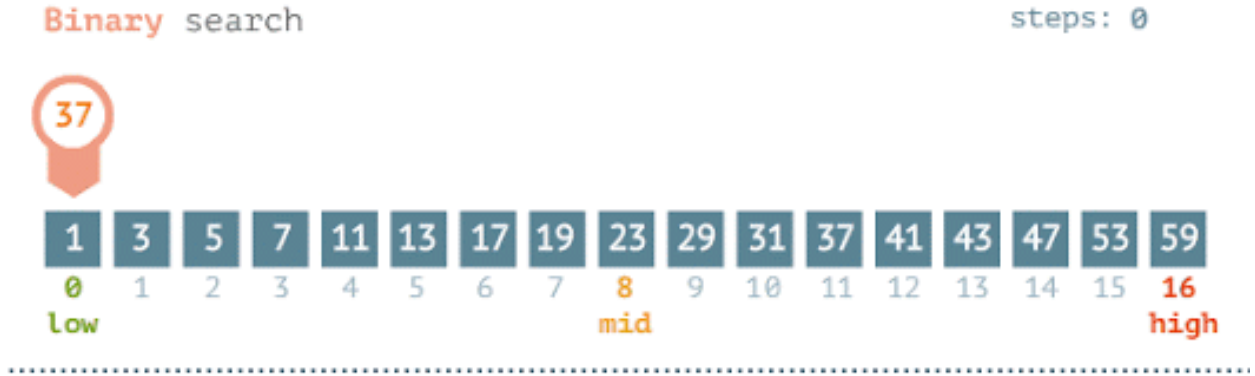
# Bubble Sort (Kabarcık Sıralaması)



# Linear Search (Doğrusal Arama)



# Binary Search (İkili Arama)



# Karşılaştırma