Veri Yapıları
3.hafta

BÖLÜM - 2

Bu bölümde,

- Algoritma Analizi,
- Çalışma Zamanı Analizi
- Algoritma Analiz Türleri
- Asimptotik Notasyon,
- Big-O Notasyonu,
- Algoritmalar için "Rate of Growth" (Büyüme Hızı)
- Big-O Hesaplama Kuralları
- Big-O Avantajları

konularına değinilecektir.

Algoritma Nedir?

19.yy da İranlı Musaoğlu Horzumlu Mehmet (Alharezmi adını Araplar takmıştır) ***problemlerin çözümü*** *için* ***genel kurallar*** *oluşturdu. Algoritma Alharezmi'nin Latince okunuşudur.*

- Basit tanım: Belirli bir görevi yerine getiren sonlu sayıdaki işlemler dizisidir.
- Geniş tanım: Verilen herhangi bir sorunun çözümüne ulaşmak için uygulanması gerekli adımların hiç bir yoruma yer vermeksizin açık, düzenli ve sıralı bir şekilde **söz ve yazı** ile ifadesidir. Algoritmayı oluşturan adımlar özellikle **basit** ve **açık** olarak ortaya konmalıdır.

Algoritmaların Sahip Olması Gereken Özellikler

- Giriş/çıkış bilgisi
- Sonluluk
- Kesinlik
- Etkinlik
- Başarım ve performans

Algoritma Analizi

- **Aynı problemi** (örneğin sıralama) birçok algoritma ile (**insertion**, **bubble**, **quick** vs) çözmek mümkün olduğu için algoritmalar ***verimlilik*** (kullandıkları hafıza ve işlemi gerçekleştirdikleri zaman) anlamında kıyaslanmalı ve **seçim** buna göre *yapılmalıdır*.
- Bu kıyaslama algoritma analizinde ***çalışma zamanı karşılaştırması*** olarak bilinir.

Algoritma Analizi (devam…)

- Algoritma analizi yapılma nedenleri:
  - Program, kendisinden istenenleri karşılıyor mu?
  - Doğru olarak çalışıyor mu?
  - Nasıl işletileceği ve nasıl kullanılacağı konusunda belgelendirmeye sahip mi? (Açıklama satırları da dahil)
  - TRUE, FALSE mantıksal değerlerini oluşturmada fonksiyonları etkili bir biçimde kullanabiliyor mu?
  - Program kodu okunabilir (*readable*) mi?
  - Program, ana (*primary*) ve ikincil (*secondary*) belleği etkili (*efficent*) bir biçimde kullanabiliyor mu?
  - Programın işletim zamanı (*running-time*) kabul edilebilir mi?

Performans analizi: Makineden bağımsız olarak zaman ve bellek ile ilgili tahminler yapılır.
Performans ölçümü: Bilgisayara bağımlı işletim zamanı elde edilir.

Performans Analizi

- Bellek Karmaşıklığı
  - Bellek karmaşıklığı, bir programın işletimini tamamlaması için ihtiyaç duyduğu bellek miktarıdır. İki bileşeni vardır:
  - **Sabit Bellek Gereksinimi (Fixed-SpaceRequirement)**: Programın girdi ve çıktı büyüklüğüne bağlı değildir. Kodun yüklenmesi için gereken belleği, sabit uzunluklu değişkenler ve sabit değerler için gereken belleği içerir.
- **Değişken Bellek Gereksinimi (Variable-Space Requirement):** I ile sembolize edebileceğimiz bir girdi büyüklüğüne bağlı olarak değişkenlerin gerek duyduğu bellek miktarını içerir. I girdisi üzerinde çalışan P programı için değişken bellek gereksinimi ile $S_P(I)$ gösterilir. Toplam bellek gereksinimi
- $$S_P(I) + c = S_P$$

  olacaktır.

Bellek Karmaşıklığı

```
float abc( float a, float b, float c ){
    return a + b + b*c - (a+b-c)/(a+b) + 4.00;
}
```

```
float sum( float list[], int n ){
    float tempsum = 0;
    int i;
    for( i=0 ; i<n ; i++ )
        tempsum += list[i];
    return tempsum;
}
```

Algoritma analizi

- Konular
  - Doğruluk - correctness
  - Zaman - time efficiency
  - Bellek - space efficiency
  - En iyi çözüm - optimality

- Yaklaşımlar
  - Teorik analiz - theoretical analysis
  - Kaba analiz - empirical analysis (sayaç/süre ölçmek gibi)

Algoritma Analizi (devam…)

- Teorik Analiz
  - Zaman verimliliği, girdi boyutunun bir fonksiyonu olarak temel işlemin tekrar sayısı belirlenerek analiz edilir.
  - *Temel İşlem (Basic operation)*: Algoritmanın çalışma süresini en çok etkileyen işlemdir. Sıralamada .... temel işlemdir. ??
- Kaba Analiz
  - Bir giriş verisi seçilir
  - Zaman birimini seç (milisaniye)

    veya

    Yürütülen temel işlem adımlarının sayısı
  - Deneysel veriler ile analiz yapılır

Problemler - Temel işlemler

Problem	***Giriş verisinin büyüklüğü***	***Temel işlemler***
n elemanlı bir dizide eleman arama	Dizinin eleman sayısı. *n*	Elemanları karşılaştırma
İki matrisin çarpımı	Matris boyutu veya toplam eleman sayısı	İki sayının çarpımı
İnteger bir sayının asal olup olmadığının kontrolü	N sayısının dijit sayısı (ikili gösterilim)	Bölme
Graf problemi	Düğüm / Kenar sayısı	Düğüm ve kenarların gezilmesi

Çalışma Zamanı Analizi

- *Çalışma zamanı analizi* (***karmaşıklık analizi***) *bir algoritmanın* (***artan***) *“**(veri) giriş**” boyutuna bağlı olarak işlem zamanının / süresinin **nasıl arttığını** (değiştiğini) tespit etmek olarak tanımlanır.*
- Algoritmaların işlediği sıklıkla karşılaşılan *“**(veri) giriş**”* türleri:
  - Array (boyuta bağlı)
  - Polinom (derecesine bağlı)
  - Matris (eleman sayısına bağlı)
  - İkilik veri (bit sayısı)
  - Grafik (kenar ve düğüm sayısı)

Çalışma Zamanı Analizi (devam…)

- Çalışma zamanı/karmaşıklık analizi için kullanılan **<u>başlıca yöntemler</u>** aşağıdaki gibidir:

1. **<u>Deneysel Analiz Yöntemi:</u>** Örnek problemlerde *denenmiş bir algoritmadaki* hesaplama <u>deneyimine</u> dayanır. Amacı, ***pratikte*** *algoritmanın nasıl davrandığını tahmin etmektir*. Bilimsel yaklaşımdan çok, *<u>uygulamaya yöneliktir</u>*. Programı yazan programcının **tekniğine**, **kullanılan bilgisayara**, **derleyiciye** ve **programlama diline** bağlı ***değişkenlik*** gösterir.
2. **<u>RAM (Random Access Machine) Modeli ile Komut</u>**
   - **<u>Sayarak Çalışma Zamanı Analiz Yöntemi</u>**

Çalışma Zamanı Analizi (devam…)

- **RAM Modeli:**
  - RAM modeli algoritma gerçekleştirimlerini ölçmek için kullanılan **basit bir yöntemdir**.
  - Genel olarak çalışma zamanı veri giriş boyutu **n'e** *bağlı* **T(n)** ile *ifade edilir*.
  - Her operasyon (+,-, * =, if, call) "bir" zaman biriminde gerçekleşir.
  - Döngüler ve alt rutinler (fonksiyonlar) basit operasyonlar ile farklı değerlendirilirler.
  - RAM modelinde her bellek erişimi yine **"bir"** zaman biriminde gerçekleşir.

Örnek 1: Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N)
{
  int toplam = 0;

  for  (i=0; i < N; i++){
     toplam += A[i];
  } //Bitti-for

  return toplam;
} //Bitti-Topla
```

Bu **fonksiyonun** **yürütme zamanı** ne kadardır?

Örnek 1: Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N)
{
   int topla = 0;

   for (i=0; i < N; i++){
      topla += A[i];
   } //Bitti-for

   return topla;
} //Bitti-Topla
```

İşlem sayısı

1

N

N

1

Toplam: $1 + N + N + 1 = 2N + 2$

- Çalışma zamanı: $T(N) = 2N+2$
  - N dizideki eleman sayısı

Örnek 2: Dizideki bir elemanın aranması

```
int Arama(int A[], int N,
                 int sayi) {
  int i = 0;

  while (i < N){
    if (A[i] == sayi) break;
    i++;
  } //bitti-while

  if (i < N) return i;
  else return -1;
} //bitti-Arama
```

Bu **fonksiyonun** <u>**yürütme zamanı**</u> ne kadardır?

Örnek 2: Dizideki bir elemanın aranması

```
int Arama(int A[], int N,
                      int sayi) {
   int i = 0;

   while (i < N){
      if (A[i] == sayi) break;
      i++;
   } //bitti-while

   if (i < N) return i;
   else return -1;
} //bitti-Arama
```

İşlem sayısı

- `int i = 0;` → 1
- `while (i < N){` → $1 <= L <= N$
- `if (A[i] == sayi) break;` → $1 <= L <= N$
- `i++;` → $0 <= L <= N$
- `if (i < N) return i;` → 1
- `else return -1;` → 1

Toplam: $1+3*L+1+1 = 3L+3$

- Çalışma zamanı: $T(N) = 3N+3$

```
float BulEnkucuk (float A[ ])
{
        float enkucuk;
        int k;
        enkucuk=A[0];
        for (k=1; k<n; k++)
                if (A[k] < enkucuk)
                enkucuk=A[k];
        return enkucuk;
}
```

enkucuk=A[0]; → atama **1 işlem**

for (k=1; k<n; k++) →

k=1 **1 kere**
k<n **n kere**
k++ **n-1 kere**

$$\left.\begin{array}{l} k=1 \\ k<n \\ k{+}{+} \end{array}\right\} 1+n+(n-1)=2n$$

if (A[k] < enkucuk) → karşılaştırma 1 işlem **n-1 kere**

enkucuk=A[k]; → Bu işlemin kaç kez yürütüleceği belli değil, en kötü durumda **n-1 kere**

return enkucuk; **1 işlem**

$$T(n)=1+2n+(n-1)+(n-1)+1=4n$$

$T(n) = \mathbf{4n}$

```
float BulOrta (float A[], int n)
{
        float ortalama, toplam=0;
        int k;
        for (k=0; k<n; k++)
                toplam=toplam + A[k];
        ortalama = toplam/n;
return ortalama;
}
```

toplam=0 **1 işlem**

k=0 **1 kere**
k<n **(n+1) kere**
k++ **n kere**

1+(n+1)+n=2n+2

atama ve toplama 2 işlem
döngü içerisinde n kere

2n

Döngü dışında bölme ve atama **2** işlem

1 işlem

T(n)=1+2n+2+2n+2+1=4n+6

T(n) = **4n + 6**

Algoritma Analiz Türleri

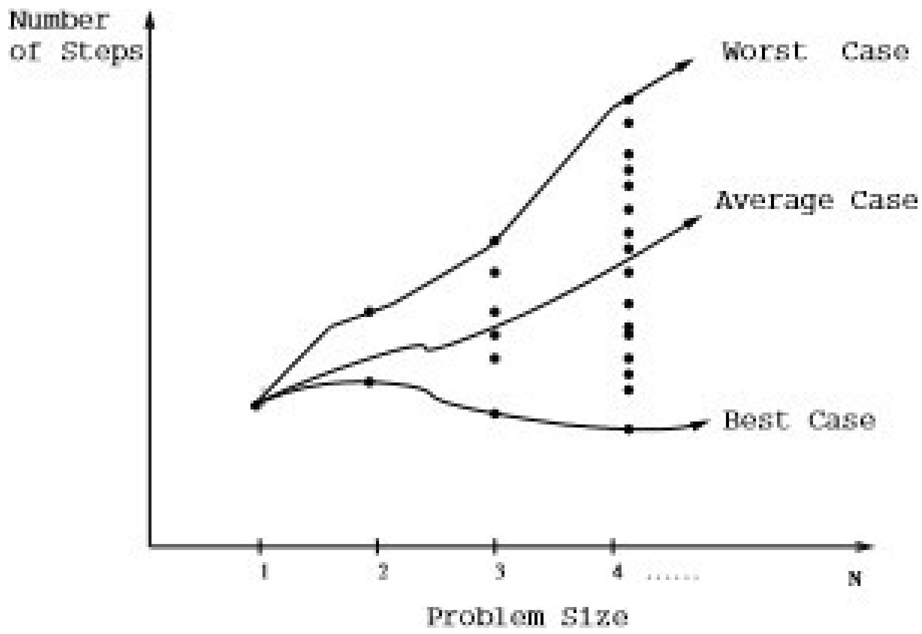
- Bir algoritmanın analizi için o algoritmanın kabaca bir polinom veya diğer zaman karmaşıklıkları cinsinden *ifade edilmesi* gerekir.
- Bu ifade üzerinden veri girişindeki değişime bağlı olarak algoritmanın *best case (en az zaman alan)* ve *worst case (en çok zaman alan)* durumları incelenerek algoritmalar arası kıyaslama yapılabilir. Bu şekilde bir algoritma üç şekilde incelenebilir:
  1. **Worst case (en kötü)**
  2. **Best case (en iyi)**
  3. **Average case (ortalama)**

Algoritma Analiz Türleri (devam…)

- **Worst case (en kötü):** Algoritma çalışmasının en fazla sürede gerçekleştiği analiz türüdür. En kötü durum, çalışma zamanında bir üst sınırdır ve o algoritma için verilen durumdan *"daha uzun sürmeyeceği"* **garantisi** verir. Bazı algoritmalar için en kötü durum ***oldukça sık rastlanır***. Arama algoritmasında, *aranan öğe* genellikle **dizide olmaz** dolayısıyla **döngü N kez çalışır**.
- **Best case (en iyi):** Algoritmanın en kısa sürede ve en az adımda çalıştığı giriş durumu olan analiz türüdür. Çalışma zamanında bir alt sınırdır.
- **Average case (ortalama):** Algoritmanın ortalama sürede ve ortalama adımda *çalıştığı* giriş durumu olan analiz türüdür.

Algoritma Analiz Türleri (devam…)

- Bu incelemeler:
- ***Lower Bound (i) <= Average Bound (ii) <= Upper Bound (iii)*** şeklinde sıralanırlar.
- Grafiksel gösterimi aşağıdaki gibidir:



Algoritma Analiz Türleri (devam...)

- **Örnek 2** için en iyi, en kötü ve ortalama çalışma zamanı nedir?

```
int Arama(int A[], int N,
                  int sayi) {
  int i = 0;

  while (i < N){
    if (A[i] == sayi) break;
    i++;
  } //bitti-while

  if (i < N) return i;
  else return -1;
} //bitti-Arama
```

- En iyi çalışma zamanı
  - Döngü sadece bir kez çalıştı
    T(n) = 6
- Ortalama çalışma zamanı
  - Döngü N/2 kez çalıştı
    T(n)=3*n/2+3 = 1.5n+3
- En kötü çalışma zamanı
  - Döngü N kez çalıştı
    T(n) = 3n+3

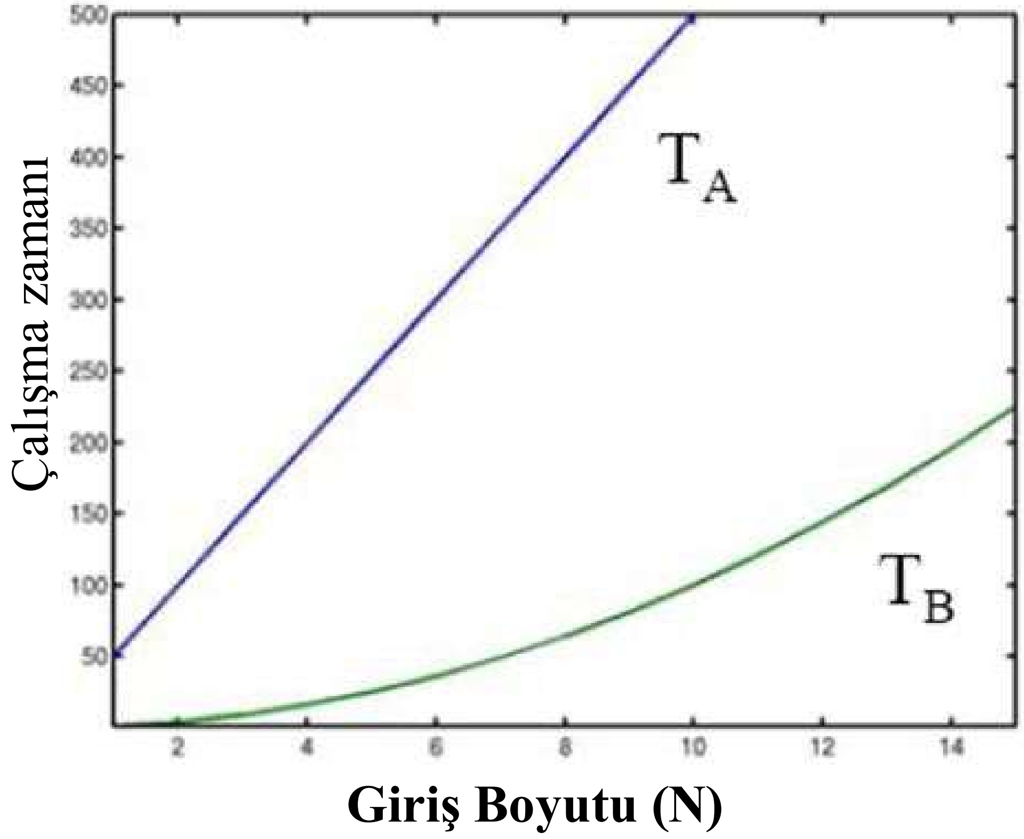
Algoritma En Kötü Durum Analizi

- Bir algoritmanın **genelde** EN KÖTÜ durumdaki çalışma zamanına bakılır. **Neden?**
  - En kötü durum çalışma zamanında bir üst sınırdır ve o algoritma için verilen durumdan daha **uzun sürmeyeceği garantisi** verir.
  - Bazı algoritmalar için *en kötü durum oldukça sık rastlanır*. Arama algoritmasında, aranan öğe genellikle dizide olmaz **dolayısıyla döngü N kez çalışır**.
  - Ortalama çalışma zamanı genellikle en kötü çalışma zamanı kadardır. Arama algoritması için **hem** ortalama hem de en *kötü çalışma zamanı* doğrusal fonksiyondur.

Asimptotik Notasyon

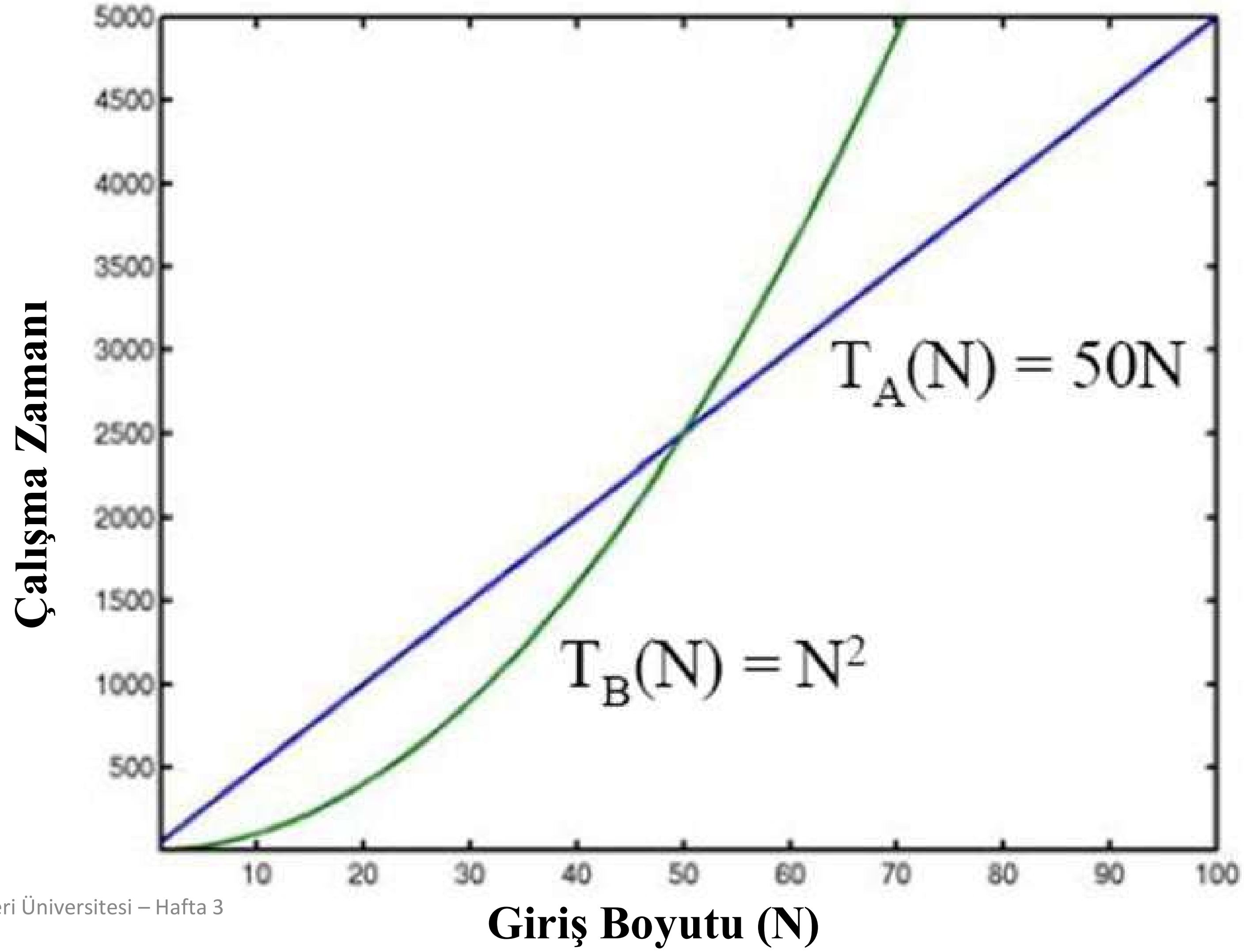
- Bir problemi çözmek için **A ve B** şeklinde iki algoritma verildiğini düşünelim.
- Giriş boyutu N için aşağıda A ve B algoritmalarının çalışma zamanı **T_A ve T_B fonksiyonları** verilmiştir.

Hangi algoritmayı seçersiniz?



Asimptotik Notasyon (devam…)

- N büyüdüğü zaman A ve B nin çalışma zamanı:



Şimdi hangi algoritmayı seçersiniz?

f(x), bir algoritmanın fonksiyon şeklindeki gösterimi ise karmaşıklık O(f(x)), Ω(f(x)), … şeklinde gösterilir.

Asimptotik Notasyon (devam...)

- Asimptotik notasyon, **eleman sayısı n'nin sonsuza gitmesi durumunda** algoritmanın, benzer işi yapan algoritmalarla karşılaştırmak için kullanılır.
- Eleman sayısının *küçük olduğu durumlar* mümkün olabilir fakat bu birçok uygulama için geçerli değildir.
- Verilen iki algoritmanın çalışma zamanını T1(N) ve T2(N) fonksiyonları şeklinde gösterilir. Hangisinin **daha iyi** olduğunu belirlemek için bir *yol belirlememiz* gerekiyor.
  - Big-O (Big O): Asimptotik üst sınır
  - Big Θ (Big Omega): Asimptotik alt sınır
  - Big 0 (Big Teta): Asimptotik alt ve üst sınır

Big-O Notasyonu

- Algoritmanın $f(n)$ şeklinde **ifade edildiğini varsayalım**.
- Algoritma, fonksiyonunun sıkı üst sınırı (tight upper bound) olarak tanımlanır.
- Bir fonksiyonun sıkı üst sınırı genel olarak:

$$f(n) = O(g(n))$$

- şeklinde ifade edilir.
- Bu ifade n’nin artan değerlerinde
  - $f(n)$’nin *üst sınırı* $g(n)$’dir
- şeklinde yorumlanır.

Big-O Notasyonu (devam…)

- **Örneğin:**

  $f(n) = n^4 + 100n^2 + 10n + 50$ algoritma fonksiyonunda

  $g(n)= n^4$ olur.

- "Daha açık bir ifadeyle", n'nin artan değerlerinde f(n) nin **maksimum büyüme oranı**

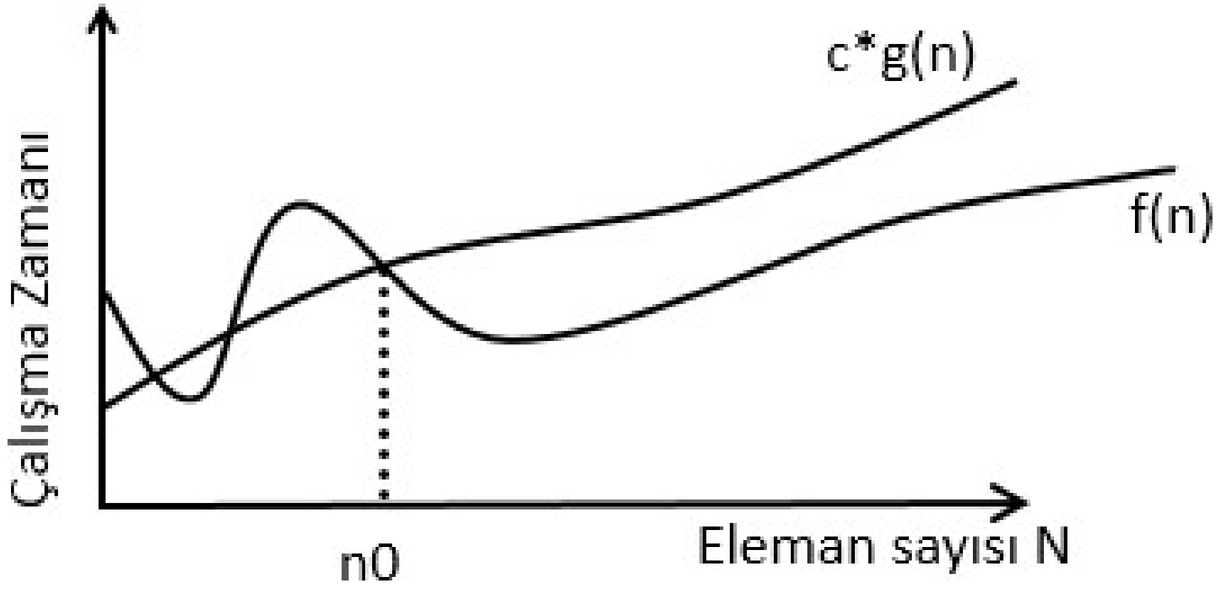
  $g(n) = n^4$

- O-notasyonu gösteriminde bir fonksiyonun *düşük n değerlerindeki* performansı **önemsiz kabul edilir**.

Big-O Notasyonu (devam…)

- $O(g(n)) = \{$
  - f(n): tüm $n >= n_0$ için, $\underline{0 <= f(n) <= cg(n)}$ olmak üzere pozitif c ve n_0 sabitleri bulunsun
- $\}$
- Bu durumda g(n), f(n)'nin asimptotik (n sonsuza giderken) **sıkı üst sınırı** olur.
- n'nin düşük değerleri ve o değerlerdeki değişim dikkate alınmazken, n_o'dan büyük değerler için *algoritmanın büyüme oranı* **değerlendirilir**.

Big-O Notasyonu (devam…)

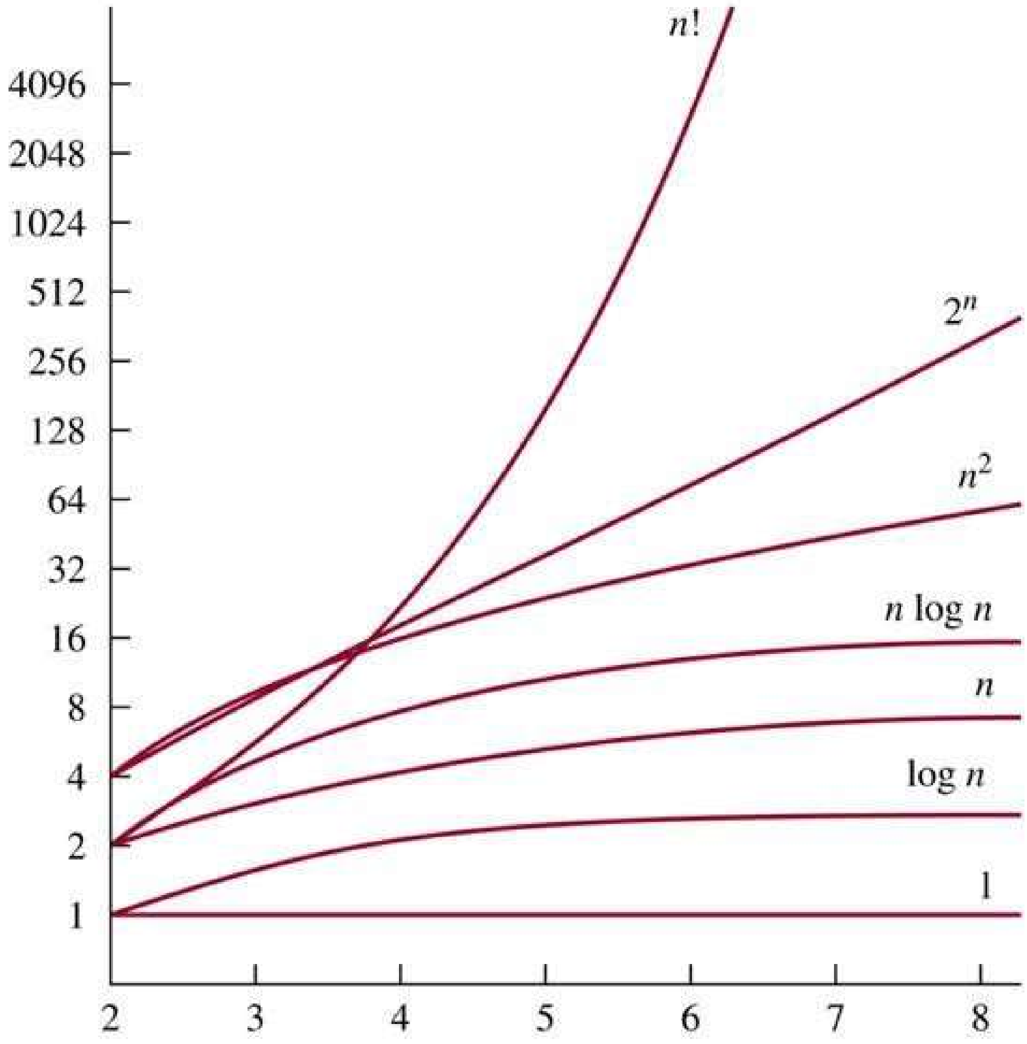


- Dikkat edilirse, n_o'dan büyük değerler için **c*g(n)**,
- f(n) için üst sınırı (asimptot) olarak görülürken,
- **n_o öncesinde** iki fonksiyonun değişimi farklı olabilir.

Büyüme Oranı (Rate of Growth)

Zaman karmaşıklığı	Açıklama	Örnek
O(1)	<u>Sabit:</u> Veri giriş boyutundan bağımsız gerçekleşen işlemler.	Bağlı listeye ilk eleman olarak ekleme yapma
O(log N)	<u>Logaritmik:</u> Problemi küçük veri parçalarına bölen algoritmalarda görülür.	Binary search tree veri yapısı üzerinde arama
O(N)	<u>Lineer – doğrusal:</u> Veri giriş boyutuna bağlı doğrusal artan.	Sıralı olmayan bir dizide bir eleman arama
O(N log N)	<u>Doğrusal çarpanlı logaritmik:</u> Problemi küçük veri parçalarına bölen ve daha sonra bu parçalar üzerinde işlem yapan.	N elemanı böl-parçala-yönet yöntemiyle sıralama. Quick Sort.
$O(N^2)$	Karesel	Bir grafikte iki düğüm arasındaki en kısa yolu bulma veya Buble Sort.
$O(N^3)$	Kübik	Ardarda gerçekleştirilen lineer denklemler
$O(2^N)$	İki tabanında üssel	Hanoi'nin Kuleleri problemi

Büyüme Oranı (Rate of Growth) (devam…)



Big-O Analiz Kuralları

- *f(n)*, *g(n)*, *h(n)*, ve *p(n)* pozitif tamsayılar kümesinden, pozitif reel sayılar kümesine tanımlanmış fonksiyonlar olsun:

1. ***Katsayı Kuralı:*** *f(n)* , *O(g(n))* ise o zaman *kf(n)* yine *O(g(n))* olur. Katsayılar önemsizdir.
2. ***Toplam Kuralı:*** *f(n)*, *O(h(n))* ise ve *g(n), O(p(n))* verilmişse *f(n)+g(n)*, *O(h(n)+p(n))* olur. Üst-sınırlar toplanır.
3. ***Çarpım Kuralı:*** *f(n), O(h(n))* ve *g(n)*, *O(p(n))* için *f(n)g(n)* is *O(h(n)p(n))* olur.
4. ***Polinom Kuralı:*** *f(n), k* dereceli polinom ise *f(n)* için $O(n^k)$ kabul edilir.
5. ***Kuvvetin Log'u Kuralı:*** $log(n^k)$ için *O(log(n))* dir.

Big-O Hesaplama Kuralları

- Programların ve algoritmaların Big-O yaklaşımıyla analizi için aşağıdaki kurallardan faydalanırız:
  1. **Döngüler (Loops)**
  2. **İç içe Döngüler**
  3. **Ardışık deyimler**
  4. **If-then-else deyimleri**
  5. **Logaritmik karmaşıklık**

Kural 1: Döngüler (Loops)

Bir döngünün çalışma zamanı, en çok <u>döngü içindeki deyimlerin</u> çalışma zamanının **iterasyon sayısıyla çarpılması** kadardır.

n defa çalışır

```
for (i=1; i<=n; i++)
{
    m = m + 2;      ← Sabit zaman
}
```

Toplam zaman = sabit c * n = cn = **O(*N*)**

<u>DİKKAT:</u> Eğer bir döngünün n değeri sabit verilmişse.
<u>Örneğin:</u> n = 100 ise değeri O(1)'dir.

Kural 2: İç İçe Döngüler

İçteki analiz yapılır. Toplam zaman <u>bütün döngülerin çalışma sayılarının çarpımına</u> eşittir.

Dış döngü *n* defa çalışır

```
for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        k = k+1;
    }
}
```

iç döngü *n* defa çalışır

Sabit zaman

Toplam zaman = c * n * n * = cn^2 = **O(N^2)**

Kural 3: Ardışık Deyimler

Her deyimin zamanı birbirine eklenir.

Sabit zaman → `x = x +1;`

Sabit zaman → `m = m + 2;`

```
x = x +1;
for (i=1; i<=n; i++) {
    m = m + 2;
}
for (i=1; i<=n; i++)  {
   for (j=1; j<=n; j++) {
      k = k+1;
   }
}
```

İlk döngü: *n* defa çalışır

Dış döngü *n* defa çalışır

iç döngü *n* defa çalışır

Sabit zaman → `k = k+1;`

Toplam zaman = $c_0 + c_1 n + c_2 n^2 = \mathbf{O}(N^2)$

Kural 4: If Then Else Deyimleri

En kötü çalışma zamanı: **test zamanına** *then veya else kısmındaki çalışma zamanının* **hangisi büyükse** o kısım eklenir.

test:
sabit

```
if (depth( ) != otherStack.depth( ) ) {
  return false;
}
else {
  for (int n = 0; n < depth( ); n++) {
  if (!list[n].equals(otherStack.list[n]))
    return false;
  }
}
```

then:
sabit

Diğer if :
sabit+sabit
(else yok)

else:
(sabit +sabit) * n

Toplam zaman = $c_0 + c_1 + (c_2 + c_3) * n = \mathbf{O(}\boldsymbol{N}\mathbf{)}$

Kural 5: Logaritmik Karmaşıklık

Problemin **büyüklüğünü belli oranda(genelde ½) azaltmak** için **sabit bir zaman harcanıyorsa** bu algoritma O(log *N*)'dir.

```
for(i=1; i<=n;)
        i = i*2;
```

- kod parçasında **n döngü sayısı** i = i*2 den dolayı her seferinde yarıya düşer.
- Loop'un k kadar döndüğünü varsayarsak;
  - k adımında $2^i = n$ olur.
  - Her iki tarafın logaritmasını alırsak;
    - ilog2 = logn ve i = log n olur.
  - i'ye bağlı olarak (problemi ikiye bölen değişken!)

Kural 5: Logaritmik Karmaşıklık (devam…)

- Örneğin: **Binary Search (İkili arama)** algoritması kullanılarak bir sözlükte arama:

  - Sözlüğün orta kısmına bakılır.
  - Sözcük ortaya göre sağda mı solda mı kaldığı bulunur?
  - Bu işlem sağ veya solda sözcük bulunana kadar tekrarlanır.

- bu tarz bir algoritmadır. Bu algoritmalar genel olarak "divide and conquer (böl ve yönet)" yaklaşımı ile tasarlanmışlardır. Bu yaklaşımla tasarlanan olan örnek sıralama algoritmaları:
  - **Merge Sort** and **Quick Sort.**

Big-O Avantajları

- Sabitler göz ardı edilirler çünkü
  - Donanım, derleyici, kod optimizasyonu vb. nedenlerden dolayı bir komutun **çalışma süresi** her zaman *farklılık* gösterebilir. Amacımız bu etkenlerden bağımsız olarak algoritmanın ne kadar etkin olduğunu ölçmektir.
  - Sabitlerin atılması analizi **basitleştirir**. $3.2n^2$ veya $3.9n^2$ yerine sadece n^2'ye odaklanırız.
- Algoritmalar arasında kıyaslamayı basit tek bir değere **indirger**.
- Küçük n değerleri göz ardı edilerek sadece büyük n değerlerine *odaklanılır*.

Big-O Avantajları (devam…)

- ***Özetle:*** Donanım, işletim sistemi, derleyici ve algoritma detaylarından bağımsız, sadece büyük n değerlerine odaklanıp, sabitleri göz ardı ederek daha basit bir şekilde algoritmaları analiz etmemize ve karşılaştırmamızı sağlar.

- RAM'den O(n)' dönüşüm:
  - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75 \rightarrow$
  - $n^2 - n \log n + n - n^{2/3} + 1 \rightarrow$ sabitleri atalım
  - $n^2 \rightarrow$ sadece büyük n değerlerini alalım
  - **$O(n^2)$**

$f(n) = 3n \log(n!) + (n^2+3) \log n$

n, pozitif bir tamsayı olmak üzere Big-O ?

$3n \log(n!)$ □ $3n$ ve $\log(n!)$

$O(n)$ $O(\log n^n)$

$O(n \log n)$

$*$

$O(n^2 \log n)$

$(n^2+3) \log n$

$n^2 \log n + 3 \log n$

$O(n^2 \log n)$

max

$O(n^2 \log n)$

$F(x) = (x+1)\log(x^2+1) + 3x^2$ Big-O ?

$$\left.\begin{array}{l} O(x+1) \;\square\; O(x) \\ O(\log(x^2+1)) \;\square\; O(\log\ x^2) \;\square\; O(2\log x) \;\square\; O(\log x) \end{array}\right\} O(x\log x)$$

$$3x^2 \;\square\; O(3x^2) \;\square\; O(x^2)$$

$\max(O(x\log x), O(x^2)) \;\square\;$ $\color{red}{O(x^2)}$

İteratif ve Özyinelemeli Algoritmaların Analizi

İteratif (nonrecursive) algoritmaların analizi

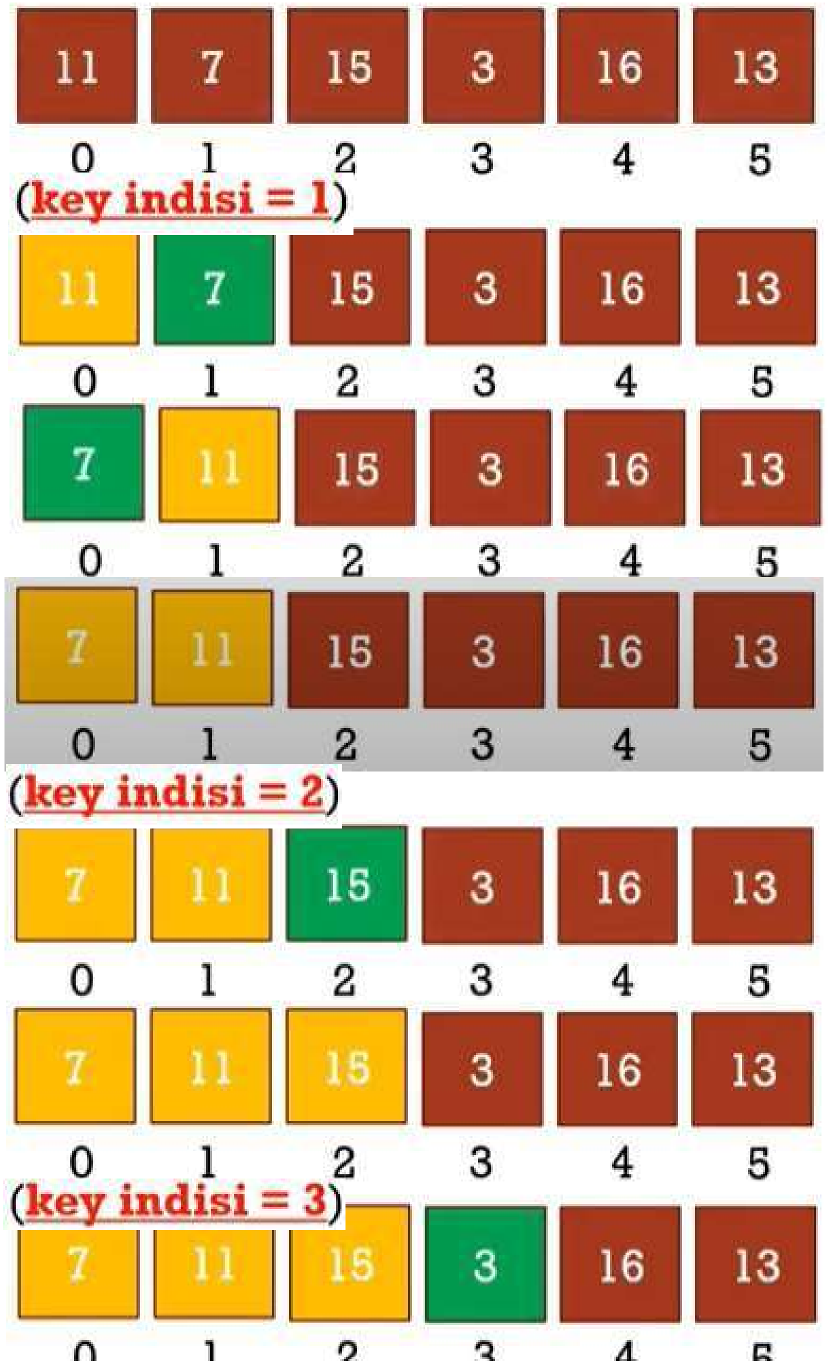
Genel Adımlar:

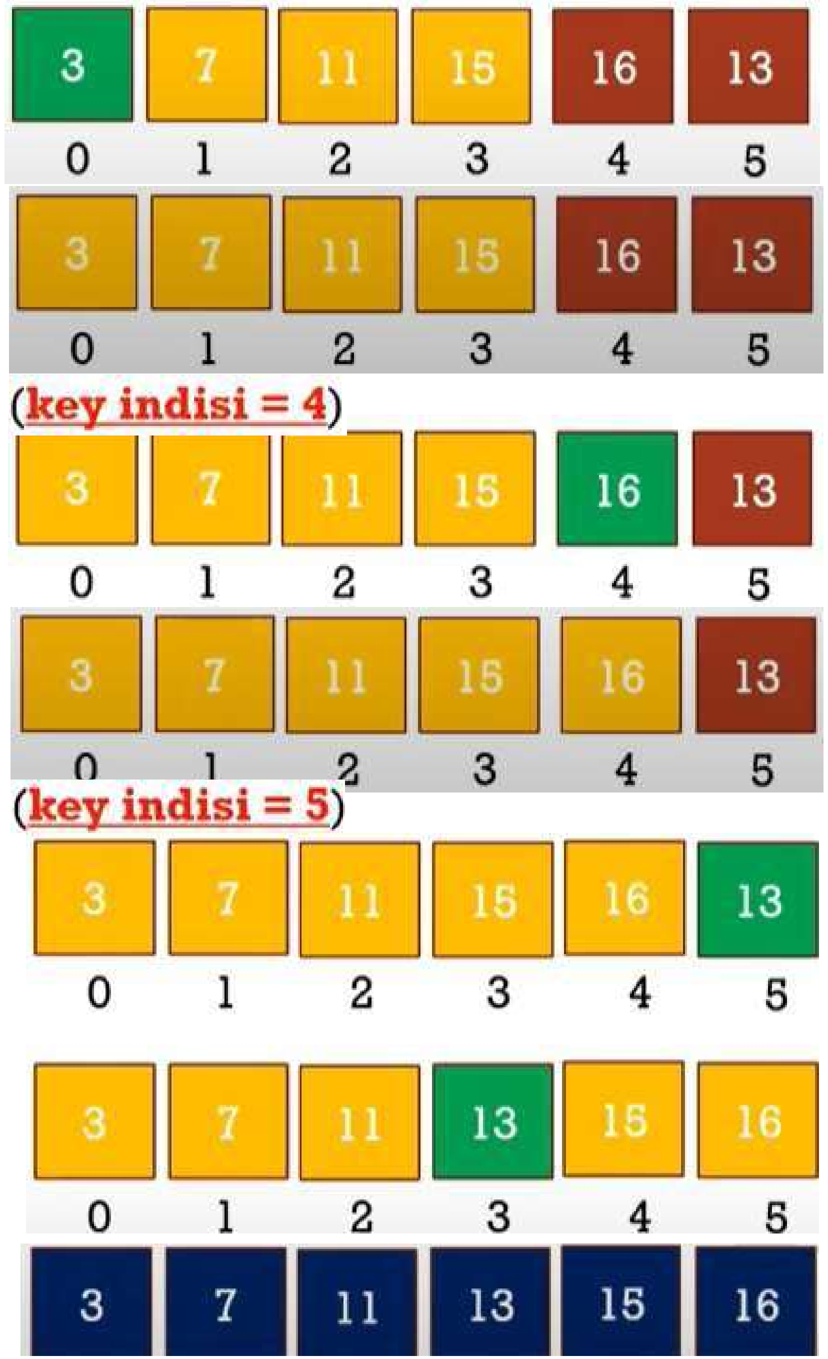
- *n* girdi boyutu *(input size)* belirlenir
- Algoritmanın temel operasyonu saptanır *(basic operation)*
- Durum analizleri (worst, average ve best cases for input of size) n değerine göre belirlenir
- Temel operasyonun kaç kez işletileceğini hesaplamak için toplama işlemi yapılır ve *n* değerine bağlı bir çalışma zamanı fonksiyonu *T(n)* elde edilir
- Toplama sonucu elde edilen *n*'e bağlı fonksiyon *T(n)*, asimptotik notasyonlara göre ifade edilir

Insertion Sort (Sokma Sıralaması)

A *pseudocode* for insertion sort (INSERTION SORT)

```
INSERTION-SORT(A)
for j ← 2 to length [A]
      do key ← A[ j]
        Insert A[j] into the sortted sequence A[1,..., j-1].
        i ← j – 1
        while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                     i ← i – 1
     A[i +1] ← key
```


11 7 15 3 16 13
0 1 2 3 4 5
(key indisi = 1)
11 7 15 3 16 13
0 1 2 3 4 5
7 11 15 3 16 13
0 1 2 3 4 5
7 11 15 3 16 13
0 1 2 3 4 5
(key indisi = 2)
7 11 15 3 16 13
0 1 2 3 4 5
7 11 15 3 16 13
0 1 2 3 4 5
(key indisi = 3)
7 11 15 3 16 13


3 7 11 15 16 13
0 1 2 3 4 5
3 7 11 15 16 13
0 1 2 3 4 5
(key indisi = 4)
3 7 11 15 16 13
0 1 2 3 4 5
3 7 11 15 16 13
0 1 2 3 4 5
(key indisi = 5)
3 7 11 15 16 13
0 1 2 3 4 5
3 7 11 13 15 16
0 1 2 3 4 5
3 7 11 13 15 16

INSERTION - SORT(A)	cost	times
for $j \leftarrow 2$ **to** $length[A]$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
∇ Insert $A[j]$ into the sorted sequence $A[1 \cdot\cdot j-1]$	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ *and* $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Toplam Çalışma süresi

$$T(n) = c_1 + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Best-case : O(n)

Dizi başlangıçta sıralı ise. Yer değiştirme yapılmaz

Average-case : O(n^2)

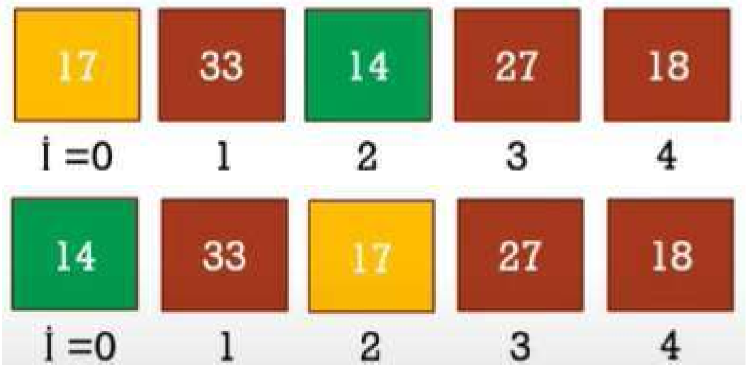
Worst case : O(n^2)

Başlangıçta dizi büyükten küçüğe sıralı ise her eleman için **en başa** kadar karşılaştırma yapılacaktır.

Selection Sort (Seçmeli Sıralama)

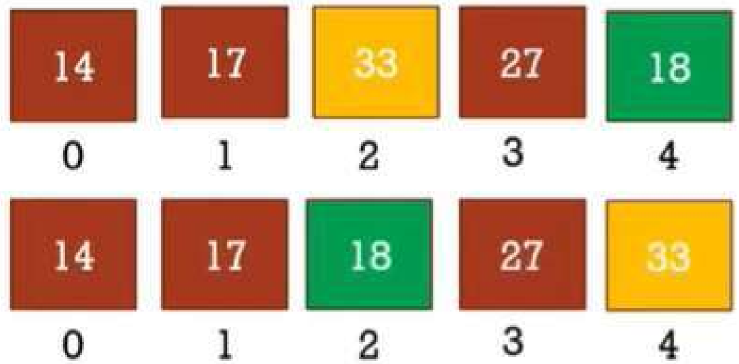
```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Swap Yapılır. (i = 0)



Swap Yapılır. (i = 1)

Swap Yapılır. (i = 2)



Swap Yapılır. (i = 3)

```
void selection(int A[], int N)
{ int i, j, temp;
  for (j = 0; j < N-1; j++)        -- iterations -->  (N-1)
  { int min_idx = j;
    for (i = j+1; i < N; i++)      -- iterations -->  (N-1-j)
      if (A[i] < A[min_idx])                          (depends on j)
            min_idx = i;
    temp = A[min_idx];
    A[min_idx] = A[j];
    A[j] = temp;
  }
}
```

j	Iterations of inner loop = N-1-j
0	N-1
1	N-2
2	N-3
...	...
N-3	2
N-2	1

Total instructions (all iterations of inner loop for all values of j)

$T(N) = (N-1) + (N-2) + \dots 2 + 1 =$

$= [N * (N-1)]/2 \rightarrow$ N^2 order of magnitude

Note that the came from the summation NOT because 'there is an N in the inner loop' (NOT because N * N).

22

worst-case : $O(n^2)$

average-case : $O(n^2)$

best_case : $O(n^2)$

Lab Uygulaması 1

```
int subcalc1(int[] v1)
{
  int sum = 0;
  for (int i=0; i < v1.length; i++)
    sum = sum + v1[i]*v1[i]*v1[i];
  return sum;
}

int subcalc2(int[] v2)
{
  int sum = 0;
  for (int i=0; i < v2.length; i++)
    for (int j=0; j < i; j++)
      sum = sum + v2[i]*v2[j];
  return sum;
}

int calc(int[] v)
{
  return subcalc1(v) + subcalc2(v);
}
```

Yandaki kod bloğunda **calc()** fonksiyonun Big-O cinsinden karmaşıklığını hesaplayınız?

Lab Uygulaması 2

```
int power2(int n)
{
  int prod = 1;
  while (prod < n)
    prod = prod * 2;

  return prod;
}
```

Yandaki kod bloğunda **power2()** fonksiyonun Big-O cinsinden karmaşıklığını hesaplayınız?

Lab Uygulaması 3

- **<u>Tanım:</u>** Verilen bir tamsayı listesi içerisinde/dizisinde *elemanları* ***komşu olmak şartıyla*** hangi (bitişik) ***alt dizi*** <u>en yüksek toplamı</u> verir?

Örneğin:

- { -2, 11,-4, 13, -5, 2 }
- { 1, 2, -5, 4, 7, -2 }
- { 1, 5 ,-3, 4, -2, 1 }

Cevaplar:

- { -2, 11,-4, 13, -5, 2 } → Cevap = 20
- { 1, 2, -5, 4, 7, -2 } → Cevap = 11
- { 1, 5 ,-3, 4, -2, 1 } → Cevap = 7

Çalışma

- Aşağıdaki fonksiyonların karmaşıklıklarını Big O notasyonunda gösteriniz.
  - $f1(n) = 10\ n + 25\ n^2$
  - $f2(n) = 20\ n \log n + 5\ n$
  - $f3(n) = 12\ n \log n + 0.05\ n^2$
  - $f4(n) = n^{1/2} + 3\ n \log n$

İYİ ÇALIŞMALAR…