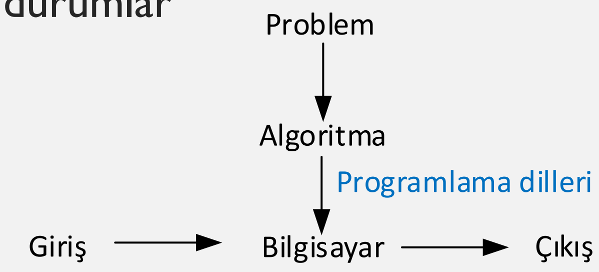
ALGORİTMA ANALİZİ VE TASARIMI

ÇÖZÜLEBİLİR PROBLEM NEDİR?

- Eğer bir problem için algoritma tasarlanabiliyorsa «çözülebilir problemdir» denir.

- Algoritma: Bir problemi çözmek için belirsiz olmayan kurallar dizisidir. Algoritma, problemde, uygun giriş için sonlu bir sürede sonuç elde etmelidir. Herhangi bir giriş verisine karşılık, çıkış verisi elde edilmesi gereklidir. Bunun dışındaki durumlar algoritma değildir.


Problem
Algoritma
Programlama dilleri
Giriş
Bilgisayar
Çıkış

Bir algoritma;

-Belirsizlik içermemeli

-Aralıklar uygun belirlenmeli

-Bir çok farklı çözümü olabilir

-Aynı algoritma farklı şekilde ifade edilebilir

-Aynı problem için farklı algoritmalar aynı çözümü farklı hızlarda verebilir

ALGORİTMA SÜRECİ

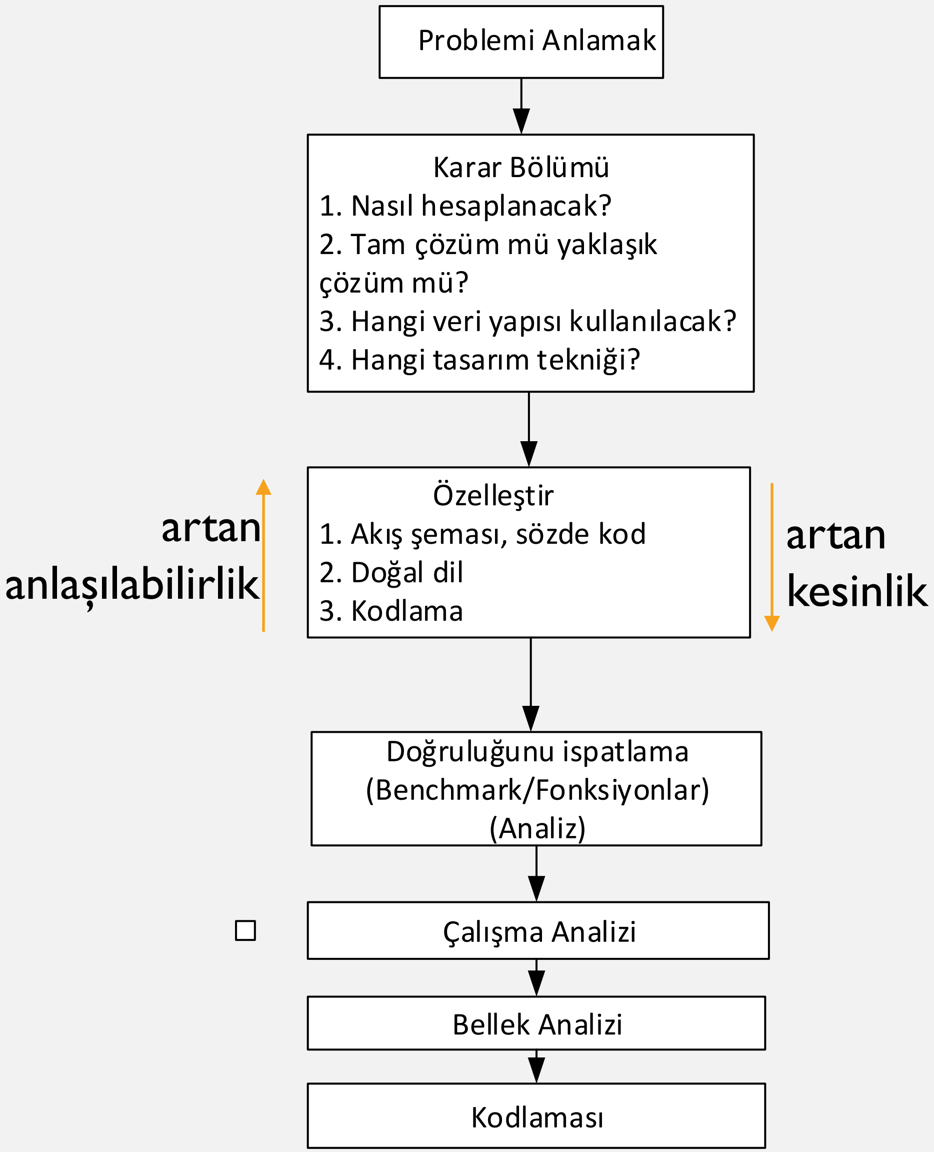
- Tasarım (design)
- Doğruluğunu ispat etme (validation)
  - Belirli bir girdi girme
  - Abstract girdi kümesi girme <a1,a2,a3… an>
  - Ters örnek girme (çok küçük, uç noktada büyük, bağımlı girdi)
- Analiz (analysis)
- Uygulama (implementation)
- Test

ALGORITMA ANALİZİ

- Neden algoritmayı analiz ederiz?
    - Algoritmanın performansını ölçmek için
    - Çözüm olup olmadığını araştırmak için
    - Tek çözüm bu mu?
    - Farklı algoritmalarla karşılaştırmak için (O(n))
    - Daha iyisi mümkün mü? Olabileceklerin en iyisi mi?

- Özelliklerinin analizi
    - Algoritmanın çalışma zamanı
    - Hafızada kapladığı alan

ALGORİTMİK PROBLEM ÇÖZMENİN ESASLARI



Problemi Anlamak

Karar Bölümü
1. Nasıl hesaplanacak?
2. Tam çözüm mü yaklaşık çözüm mü?
3. Hangi veri yapısı kullanılacak?
4. Hangi tasarım tekniği?

Özelleştir
1. Akış şeması, sözde kod
2. Doğal dil
3. Kodlama

artan anlaşılabilirlik

artan kesinlik

Doğruluğunu ispatlama (Benchmark/Fonksiyonlar) (Analiz)

☐

Çalışma Analizi

Bellek Analizi

Kodlaması

ÖNEMLİ PROBLEM TİPLERİ

- Sorting
- Searching
- String Processing
- Graph Problems
- Combinational Problems
- Geometric Problems
- Numerical Problems
- Heuristic Problems

TASARIM TEKNİKLERİ

- 1. Brute Force (Kaba Kuvvet)
- 2. Divide and Conquer (Böl ve Yönet)
- 3. Decrease and Conquer (Azalt ve Yönet)
- 4. Transform and Conquer (Dönüştür ve Yönet)
- 5. Dinamik Yaklaşım (Dinamik Programlama)
- 6. Greedy Approach (Açgözlü Yaklaşım)

ÖRNEK

- Ortak çarpanların en büyüğünü bulma problemi (pozitif tamsayı)

gcd (m,n) ->greatest common divisor-OBEB

- Euclid's algorithm

gcd(m,n) = gcd(n, m mod n) eşitliğine dayanır.

Algoritma:

1. Eger n=0 ise, m değerini cevap olarak döndür ve dur. Değilse 2. adıma git
2. m'yi n'ye böl, kalanı r'ye ata.
3. n'yi m'ye ata. r'yi n'ye ata. 1. adıma git.

Sözde kod (Pseudo code):

```
while n != 0 do
        r <- m mod n
        m <- n
        n <- r
return m
```

ÖRNEK DEVAM

Ardışıl hesaplama algoritmasıyla;

- 1. m ve n'den küçük olanı t'ye ata
- 2. m'yi t'ye böl. Eğer bölümden kalan 0 ise 3. adıma git. Değilse 4. adıma git.
- 3. n'yi t'ye böl. Eğer bölümden kalan 0 ise t değerini sonuç olarak döndür ve dur. Değilse 4. adıma git.
- 4. t'yi 1 azalt ve 2. adıma git

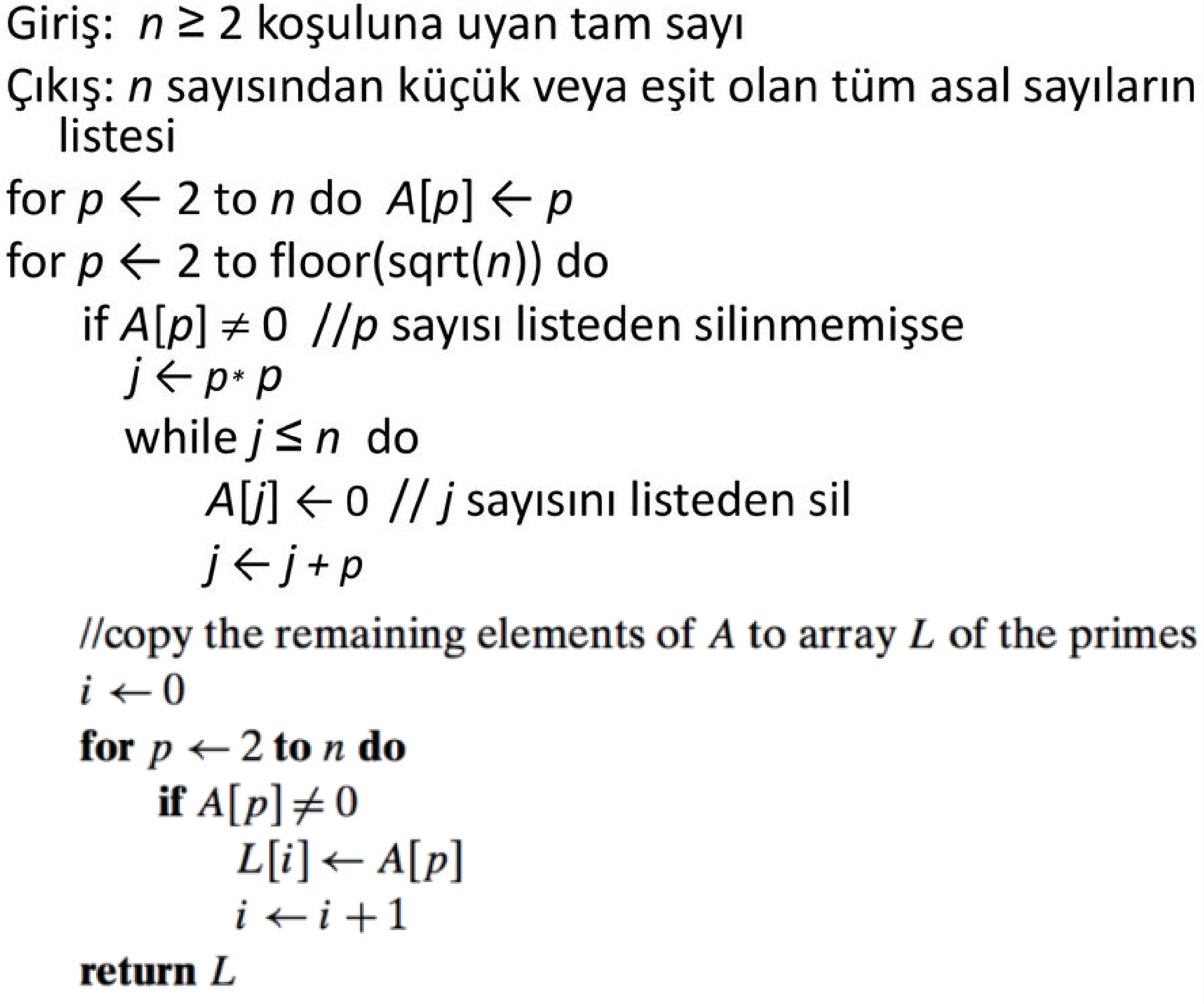
İlköğretim bilgileri ile hesaplama algoritması;

1. m'nin asal çarpanlarını bul
2. n'nin asal çarpanlarını bul
3. 1. ve 2. adımdaki ortak asal çarpanları belirle
4. Ortak çarpanların çarpımını hesapla. Bu değeri döndür.

Bu yöntem karmaşık ve zor. 1. ve 2. adımlar açık değil ve belirsiz. Asal sayı listesi gerekiyor. 3. adım da açık değil.

- Asal sayıları bulmak için Eratosthenes’in Eleği(Kalburu) kullanılır:

Pseudo Code



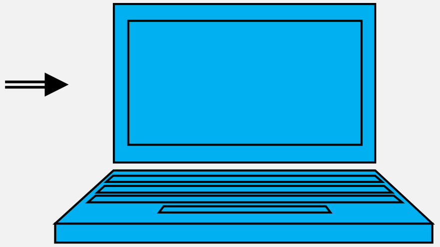
```
Giriş:  $n \geq 2$ koşuluna uyan tam sayı
Çıkış: $n$ sayısından küçük veya eşit olan tüm asal sayıların listesi
for $p \leftarrow 2$ to $n$ do  $A[p] \leftarrow p$
for $p \leftarrow 2$ to floor(sqrt($n$)) do
    if $A[p] \neq 0$  //$p$ sayısı listeden silinmemişse
        $j \leftarrow p * p$
        while $j \leq n$  do
            $A[j] \leftarrow 0$  // $j$ sayısını listeden sil
            $j \leftarrow j + p$
    //copy the remaining elements of $A$ to array $L$ of the primes
    $i \leftarrow 0$
    for $p \leftarrow 2$ to $n$ do
        if $A[p] \neq 0$
            $L[i] \leftarrow A[p]$
            $i \leftarrow i + 1$
    return $L$
```

ÇALIŞMA ZAMANI ANALİZİ

Algoritma 1 $T_1(N)=1000N$

Algoritma 2 $T_2(N)=N^2$

N giriş verisi ⟹



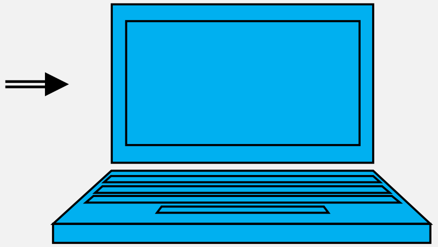
⟹ *Çalışma zamanı $T_1(n)$*

Algoritma 1

N giriş verisi

100010101010001111100011000111010
101010101010100100010101010001000
00000000011110101000111010

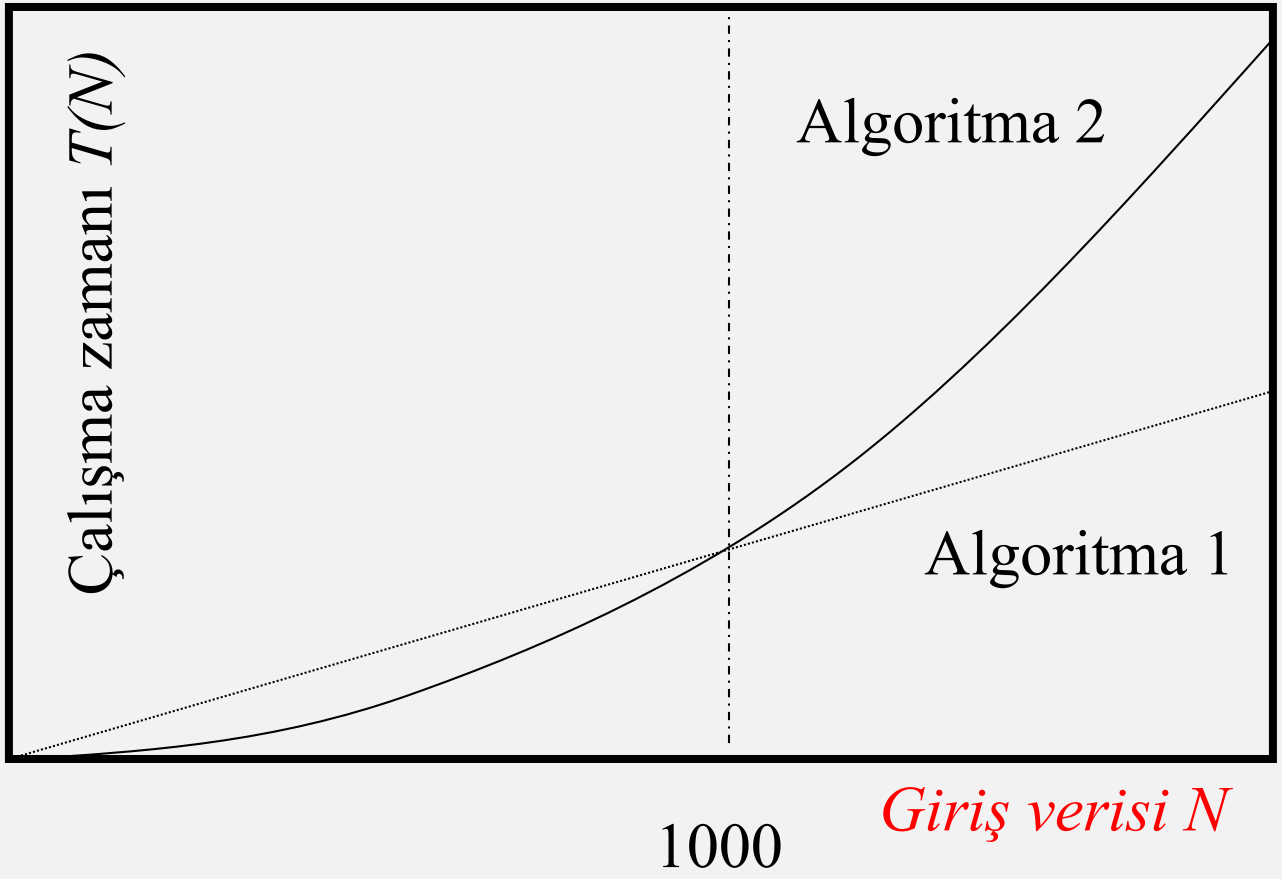
⟹



⟹ *Çalışma zamanı$T_2(n)$*

Algoritma 2

ÇALIŞMA ZAMANI ANALİZİ



ÇALIŞMA ZAMANI ANALİZİ

N	T1	T2
10	10^{-2} sec	10^{-4} sec
100	10^{-1} sec	10^{-2} sec
1000	1 sec	1 sec
10000	10 sec	100 sec
100000	100 sec	10000 sec

N değerinin 1000’den küçük olduğu durumlarda iki algoritma arasındaki çalışma zamanı ihmal edilebilir büyüklüktedir.

ALGORITHMIK PERFORMANS

Algoritmik performansın iki yönü vardır:

- Zaman (Time)
  - Komutlar zaman alır.
  - Algoritmanın çalışması nasıl hızlanır?
  - Çalışma zamanının etkileyen unsurlar nelerdir?
- Bellek (Space)
  - Kullanılan veri yapısı bellek kullanımını etkiler
  - Kullanılan veri yapısı algoritma çalışma zamanına etki eder.
  - Kullanılan veri yapısı algoritma tasarım fikrini etkiler

➢Genel olarak «zaman» üzerine odaklanılır

- Bir algoritma için gerekli zaman nasıl tahmin edilebilir?
- Gereken bu zaman nasıl azaltılabilir?

ALGORİTMANIN İŞLETİM ZAMANI

- Algoritmadaki her işlemin bir maliyeti vardır.

➔ Her işlem belirli bir zaman alır.

`count = count + 1;` ➔ belirli bir miktar zaman alır ama sabittir.

Bir dizi işlem:

`count = count + 1;` maliyet: c_1

`sum = sum + count;` maliyet: c_2

➔ toplam maliyet = $c_1 + c_2$

ALGORİTMANIN İŞLETİM ZAMANI

Örnek:: Basit If-Deyimi:

```
                                    maliyet  Tekrar
 if (n < 0)            c1                      1
    absval = -n        c2                      1
 else
      absval = n;      c3                      1
```

Toplam maliyet <= c1 + max(c2,c3)

ALGORİTMANIN İŞLETİM ZAMANI

Örnek:Temel Döngü

	maliyet	**Tekrar**
`i = 1;`	c1	
1		
`sum = 0;`	c2	
1		
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	
n		
`    sum = sum + i;`	c5	n
`}`		

Toplam maliyet = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ Bu algoritma için gerekli zaman n veri sayısı ile orantılıdır.

ALGORİTMANIN İŞLETİM ZAMANI

Örnek: İç içe Döngü

maliyet **Tekrar**

```
i=1;                        c1             1
sum = 0;                           c2             1
while (i <= n) {            c3             n+1
     j=1;                          c4             n
     while (j <= n) {              c5
n*(n+1)
         sum = sum + i;     c6             n*n
         j = j + 1;                c7             n*n
   }
   i = i +1;                c8             n
}
```

Toplam maliyet=c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ Bu algoritma için gerekli zaman n^2 ile orantılıdır.

ÇALIŞMA ZAMANI TAHMİNİ İÇİN GENEL KURALLAR

- **Döngüler:** Bir döngünün çalışma zamanı, en fazla döngü içindeki ifadelerin tekrar sayısı kadardır.
- Yuvalanmış Döngüler: En içteki döngüye kadar ifade ve işlem içeren yuvalanmış döngülerin çalışma zamanı, bütün iç döngülerin boyutlarının çarpımı kadardır.
- **Ardışık Deyimler:** Bu ardışık deyimlerin çalışma zamanı toplam çalışma zamanına eklenir.
- **If/Else**: Test sonucuna göre işletilecek olan S1 ve S2 deyimlerinden çalışma zamanı daha büyük olandır.

ALGORITMA BÜYÜME HIZLARI

- Algoritmanın çalışma zamanı *problem girdi boyutunun* bir fonksiyonu olarak ölçülür.
  - Problem girdi boyutu uygulamaya göre değişir. Sıralama algoritması için eleman sayısı, hanoi kuleleri için disk sayısı olur.
- Örnek olarak problem girdi boyutu n olsun
  - A Algoritması **$5*n^2$** zaman gerektirsin.
  - B Algoritması **$7*n$** zaman gerektirsin.
- Giriş veri sayısına bağlı olarak A algoritması **n^2** ile B algoritması ise n ile orantılı olarak büyümektedir.
- Bir algoritmanın oransal zaman gereksinimine büyüme hızı (***growth rate)***.

Neden büyüme hızlarında ortak terim gereklidir?

Algoritmaları büyüme hızları ile karşılaştırmak

Veri artışına göre çalışma zamanı arasındaki ilişkinin tespiti

Her zaman geçerli bir gösterim olması

BÜYÜME HIZLARI – ASİMTOTİK YAKLAŞIM

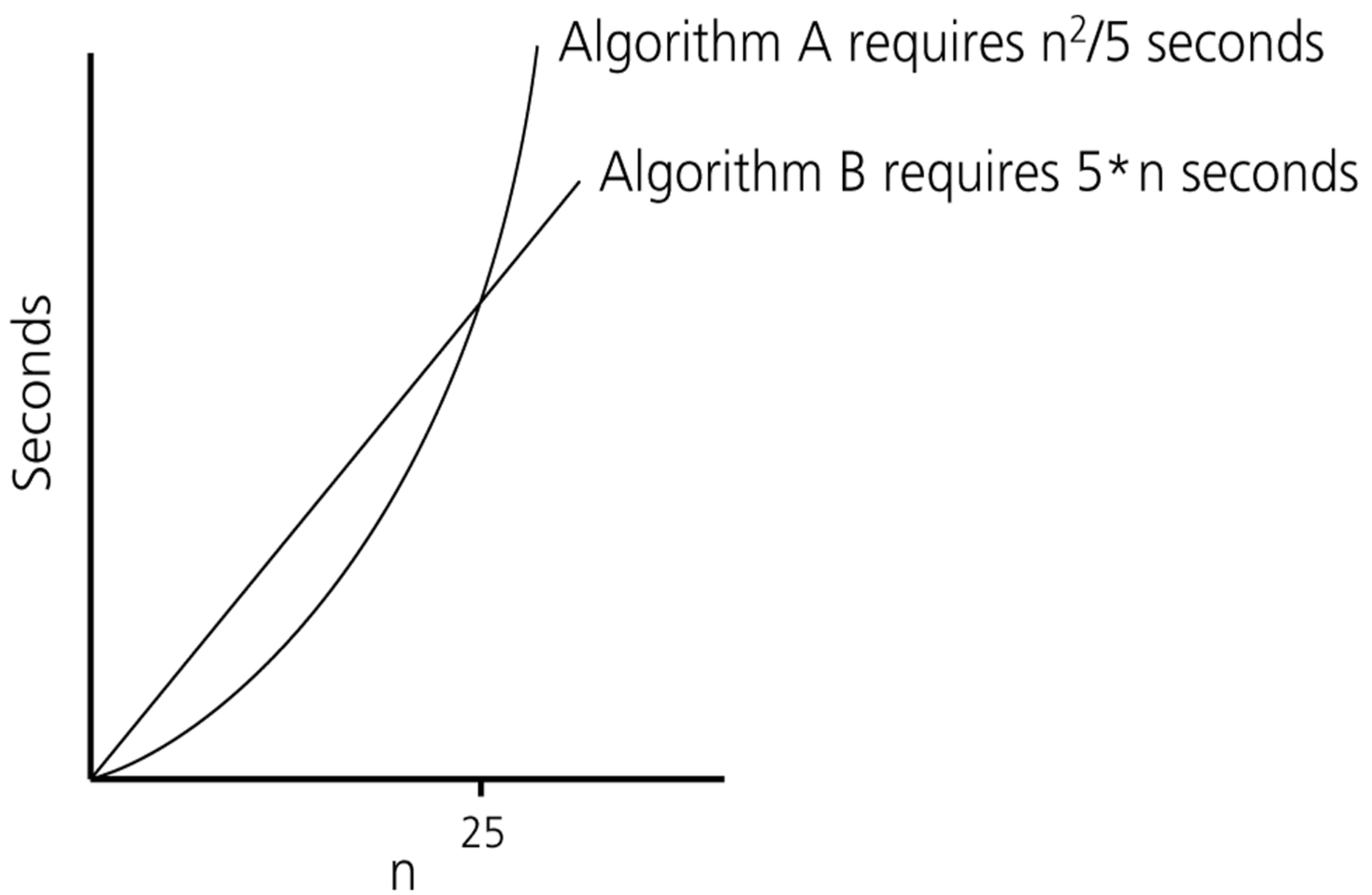
- A algoritması cn^2 davranıı sergiliyorsa. «n»nin çok yüksek değerlerinde c ihmal edilebilir. B algoritması dn^3 davranışı sergiliyorsa, «n» çok büyük değerlerde d ihmal edilebilir.

$f(x)= n^2logn + 10n^2 + n$

Çalışma zamanı: n^2logn

$$\left.\begin{array}{l} \text{A -> } 5n^2+3n-7 \\ \text{B -> } 9n^2 -6n +100 \end{array}\right\} \text{A ve B algoritmalarının davranışı asimtotik gösterilişe göre kareseldir.}$$

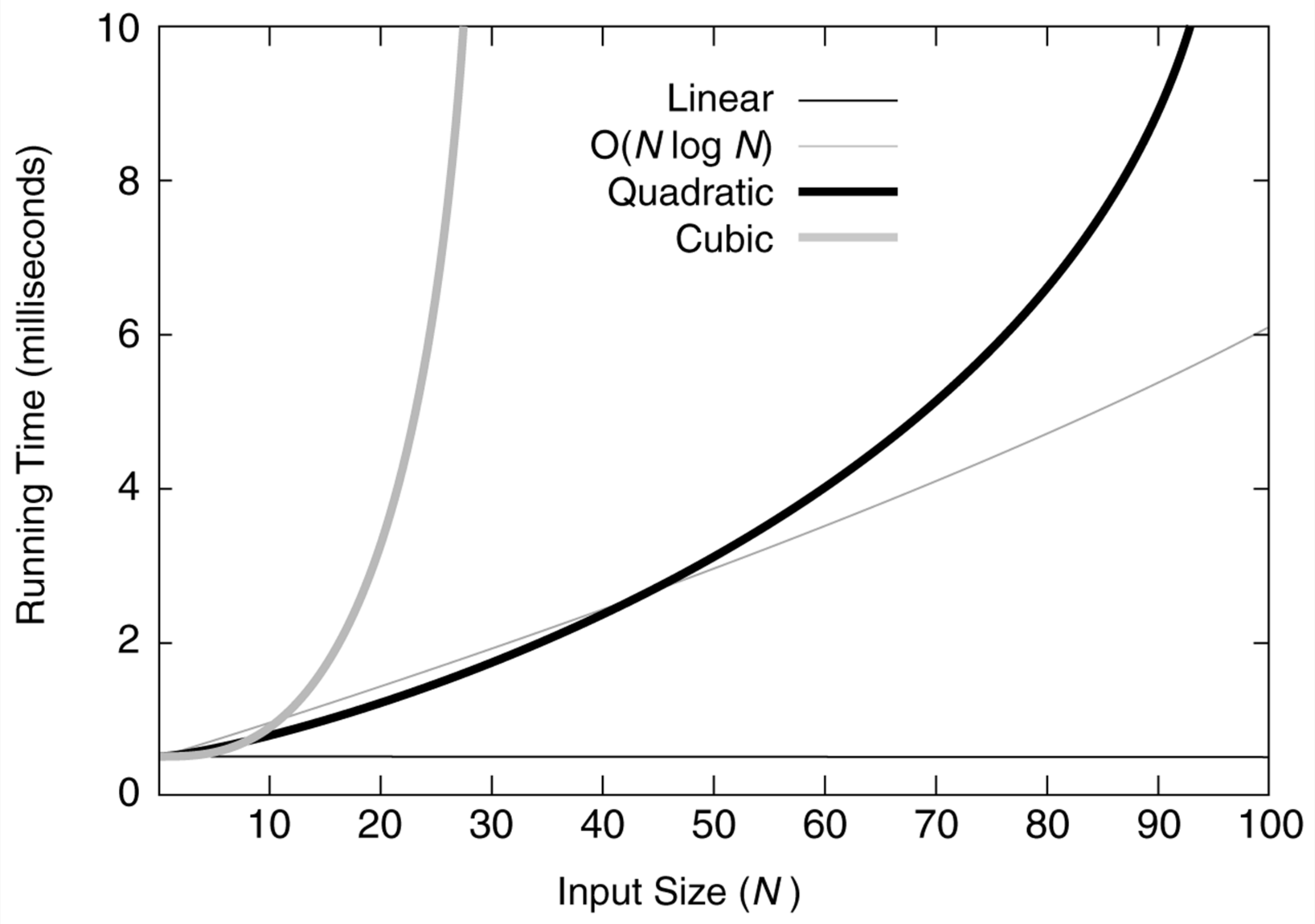
ALGORITMA BÜYÜME HIZLARI



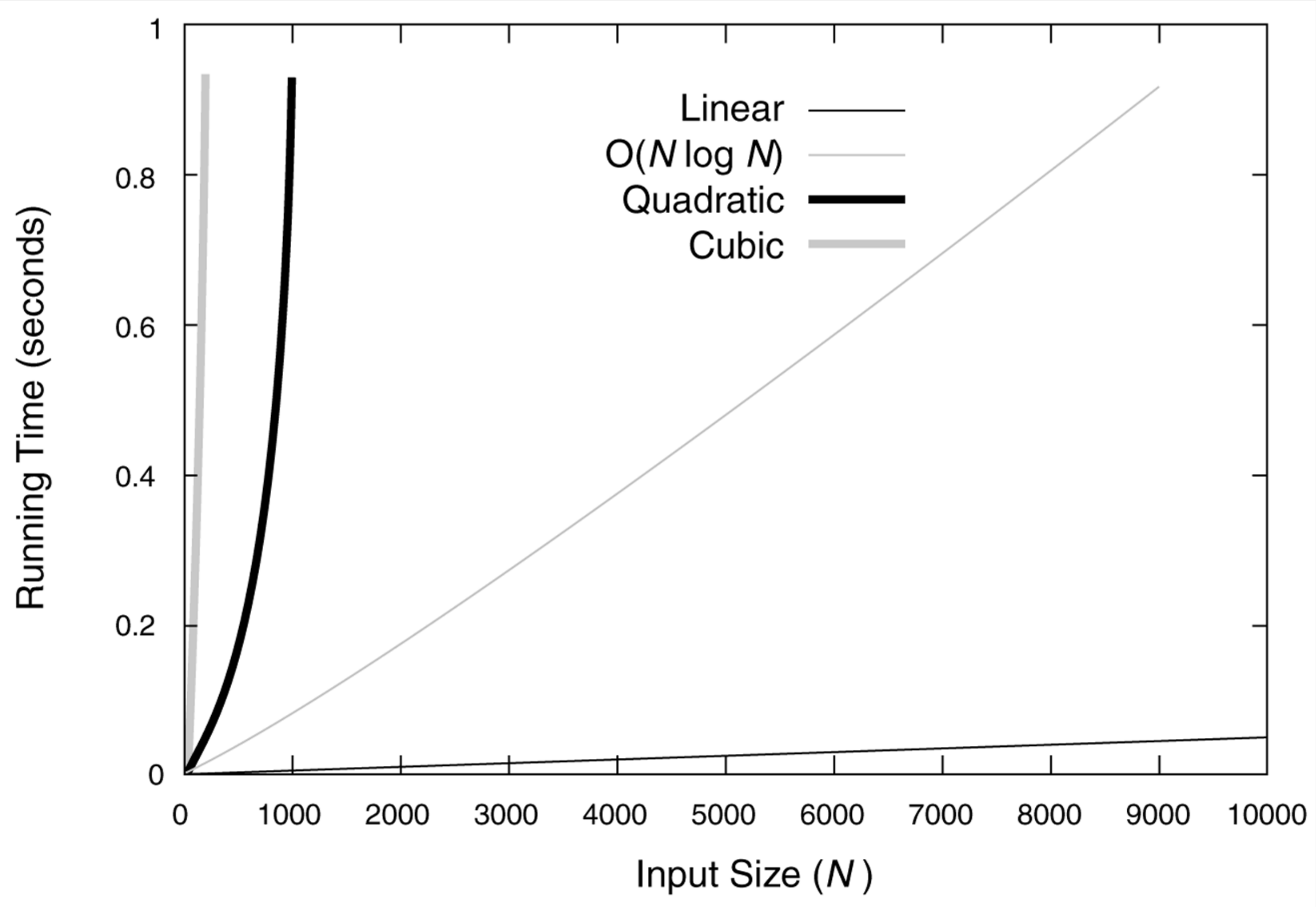
YAYGIN BÜYÜME HIZLARI

Fonksiyon	Büyüme hızı adı
c	Constant
$log\ N$	Logarithmic
$log^2 N$	Log-squared
N	Linear
$N\ log\ N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Küçük veriler için çalışma zamanları



Ortalama veriler için çalışma zamanları



$\log^k n, cn, n\log n, cn^2, cn^3$

daha hızlı

- <, >, >=, <=, +, -, /, * temel işlemlerdir ve aynı yükte oldukları kabul edilir. Bu işlemlere ek olarak atama, pointer güncelleme, düğüm güncelleme, bağlı listelerde dolaşmak da temel işlem kabul edilir.

BIG O GÖSTERİMİ

- Bir A algoritması *f(n) ile orantılı zaman gerektiriyorsa* A Algoritmasına f(n) mertebesindedir denilir ve O(f(n)) ile gösterilir**.**
- **f(n)'e** algoritmanın **growth-rate fonksiyonu** denir**.**
- Bu gösterime **Big O notation** adı verilir.
- A algoritması $\mathbf{n^2}$, ile orantılı zaman gerektiriyorsa $\mathbf{O(n^2)}$**.**
- A algoritması **n** ile orantılı zaman gerektiriyorsa **O(n)** ile ifade edilir.

ALGORİTMA DERECESİ NEDİR?

Tanım:

Öyle bir k ve n_0 sabitleri vardır ki A algoritması $n \geq n_0$ boyutunda bir problemi çözmek için k*f(n) den daha fazla zamana ihtiyaç duymaz ise A algoritmasının mertebesi O(f(n)) ile gösterilir.

ALGORITMANIN DERECESİ (MERTEBESİ)

- Eğer bir algoritma n elemanlı bir problem için $n^2 – 3*n+10$ saniye gerektiriyorsa ve öyle bir k ve n_0 değerleri vardır ki;

  bütün $n \geq n_0$ için $k*n^2 > n^2 – 3*n+10$

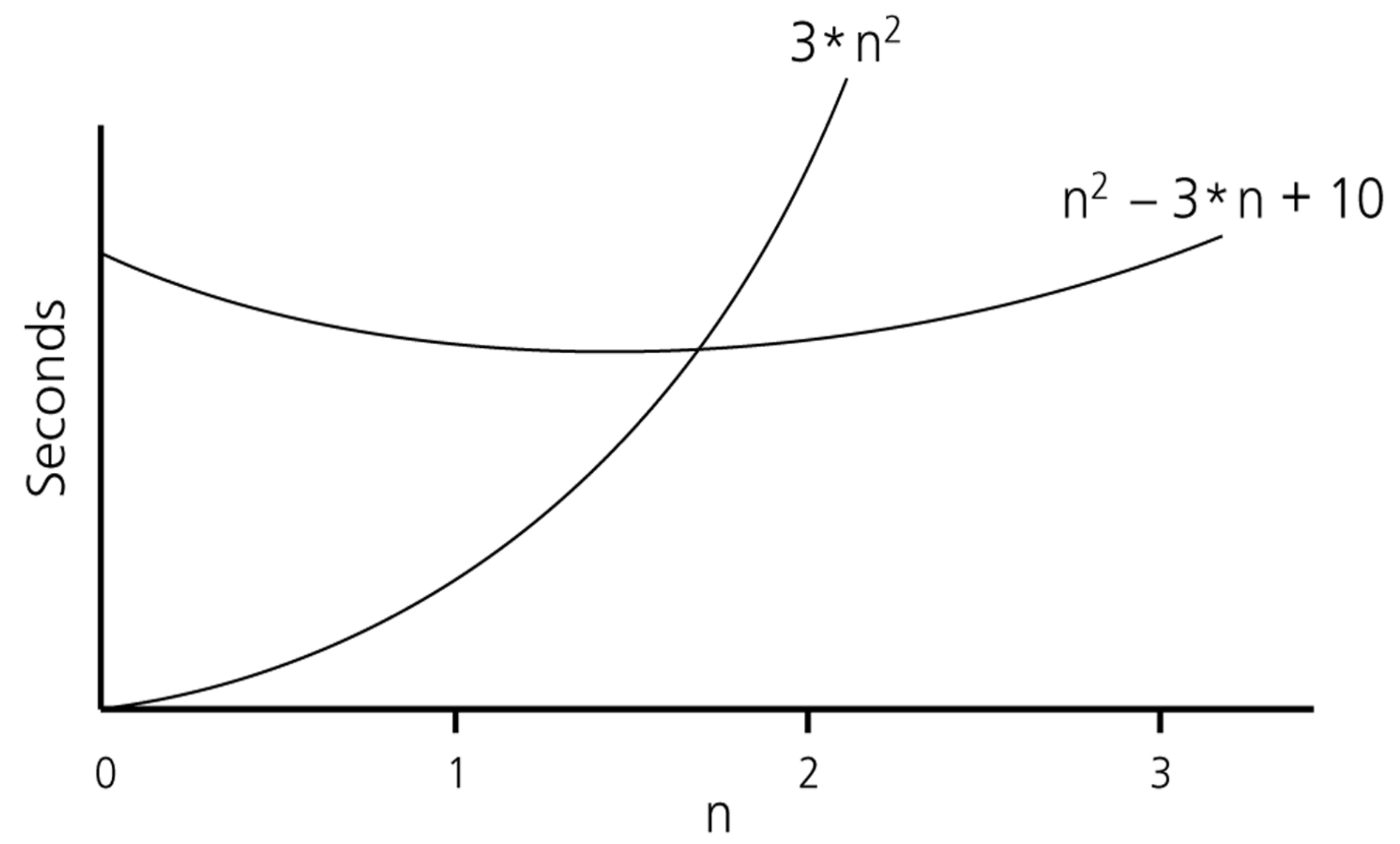
  ve algoritmanın mertebesi n^2 olur. (Gerçekten k=3 ve $n_0 = 2$)

  Bütün $n \geq 2$ *için* $3*n^2 > n^2 – 3*n+10$ olur.

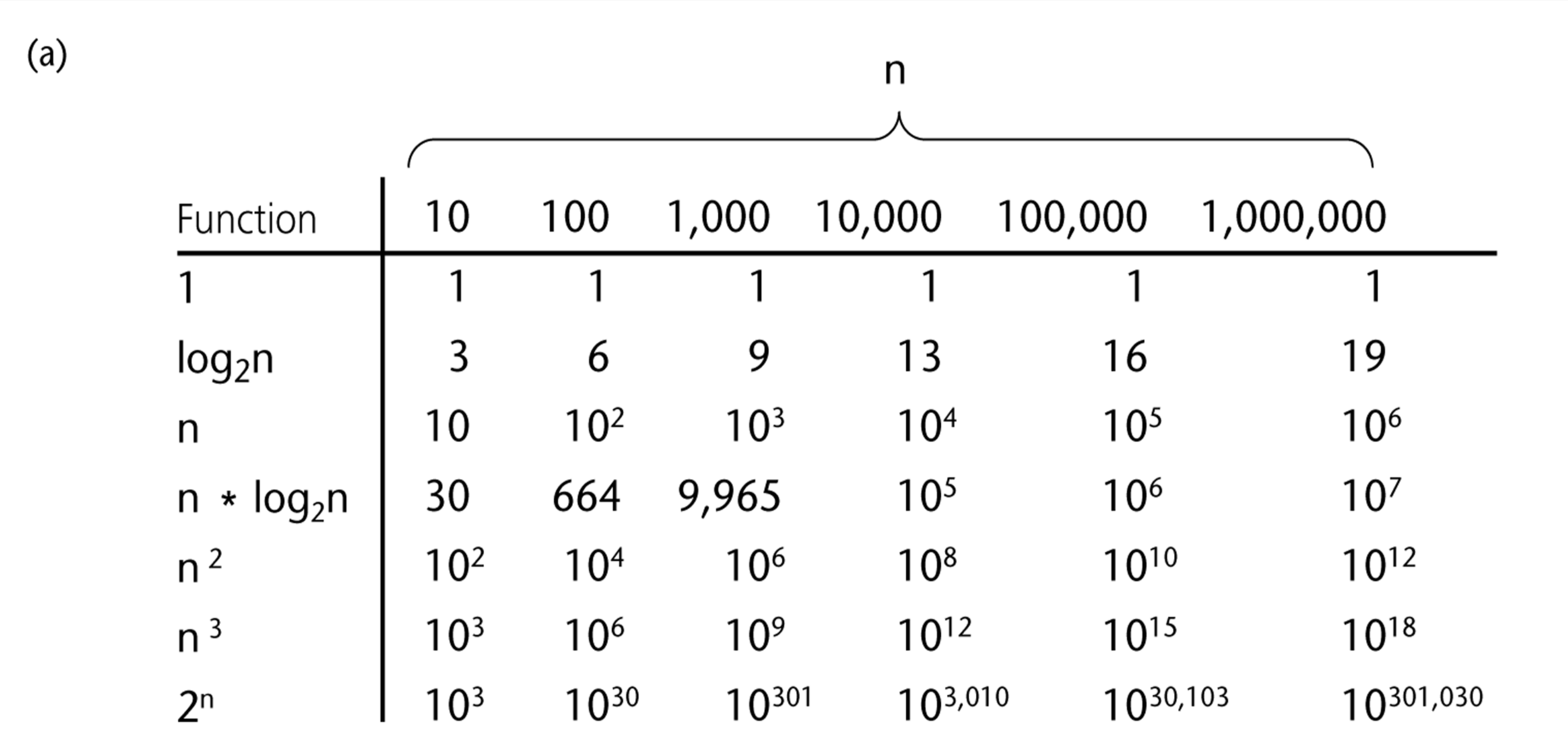
  Yani algoritma ($n \geq n_0$) için, $k*n^2$ den daha fazla zamana ihtiyaç duymaz.

  ve böylece **O(n^2) ile ifade edilir.**

ALGORİTMANIN MERTEBESİ (DERECESİ)


$3*n^2$
$n^2 - 3*n + 10$
Seconds
0
1
2
3
n

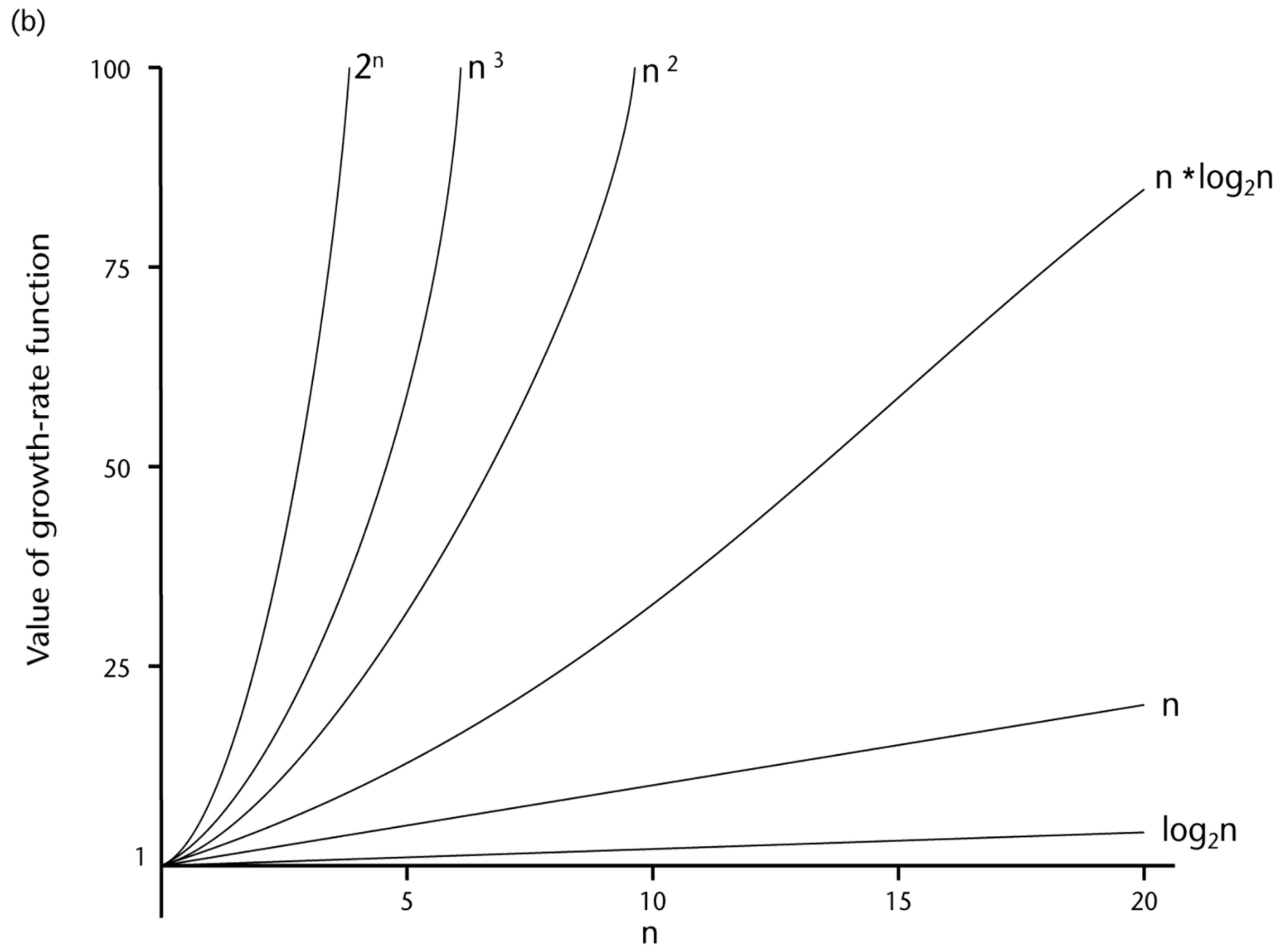
BÜYÜME HIZI FONKSİYONLARININ KARŞILAŞTIRILMASI



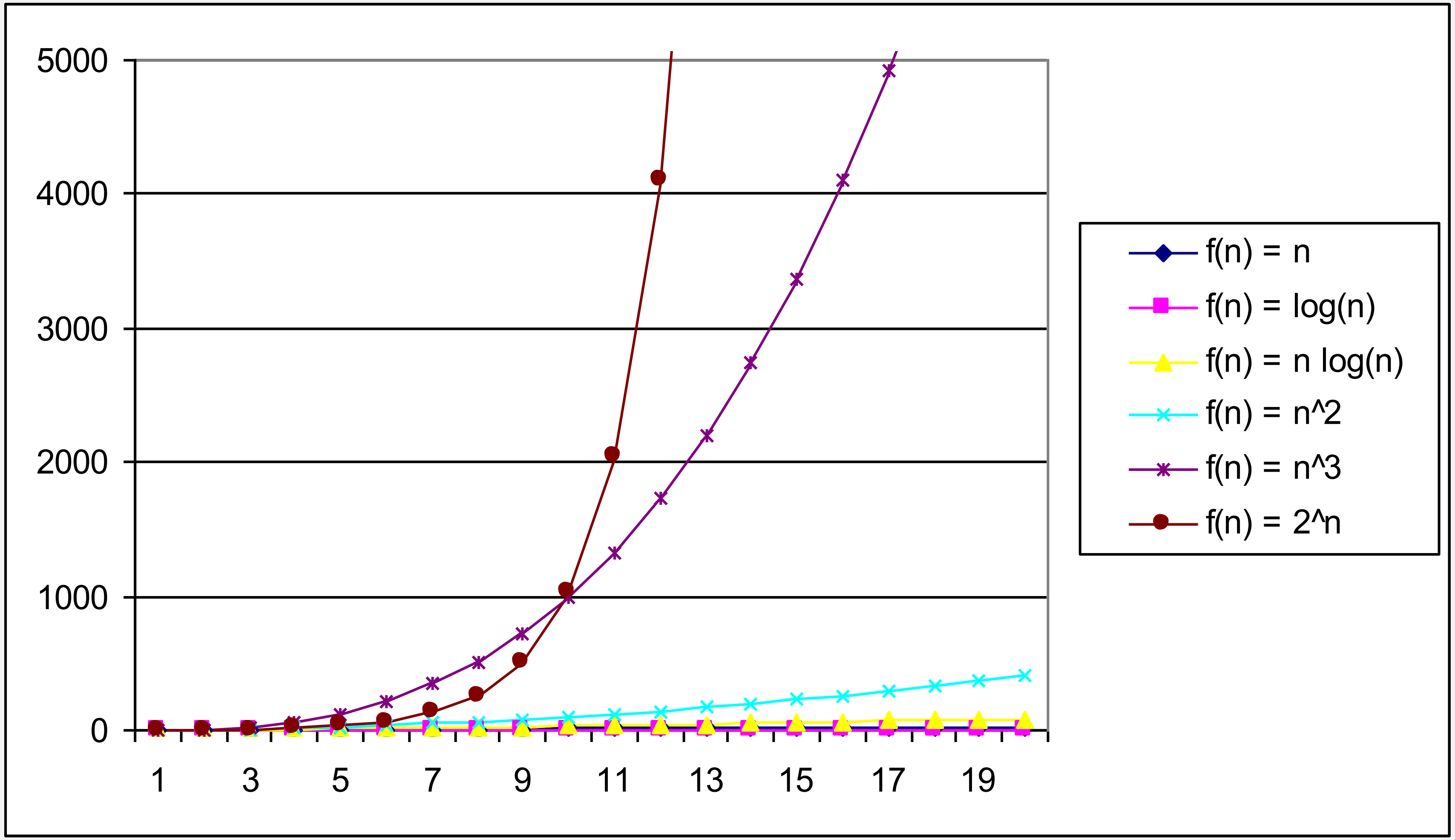
(a)

	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

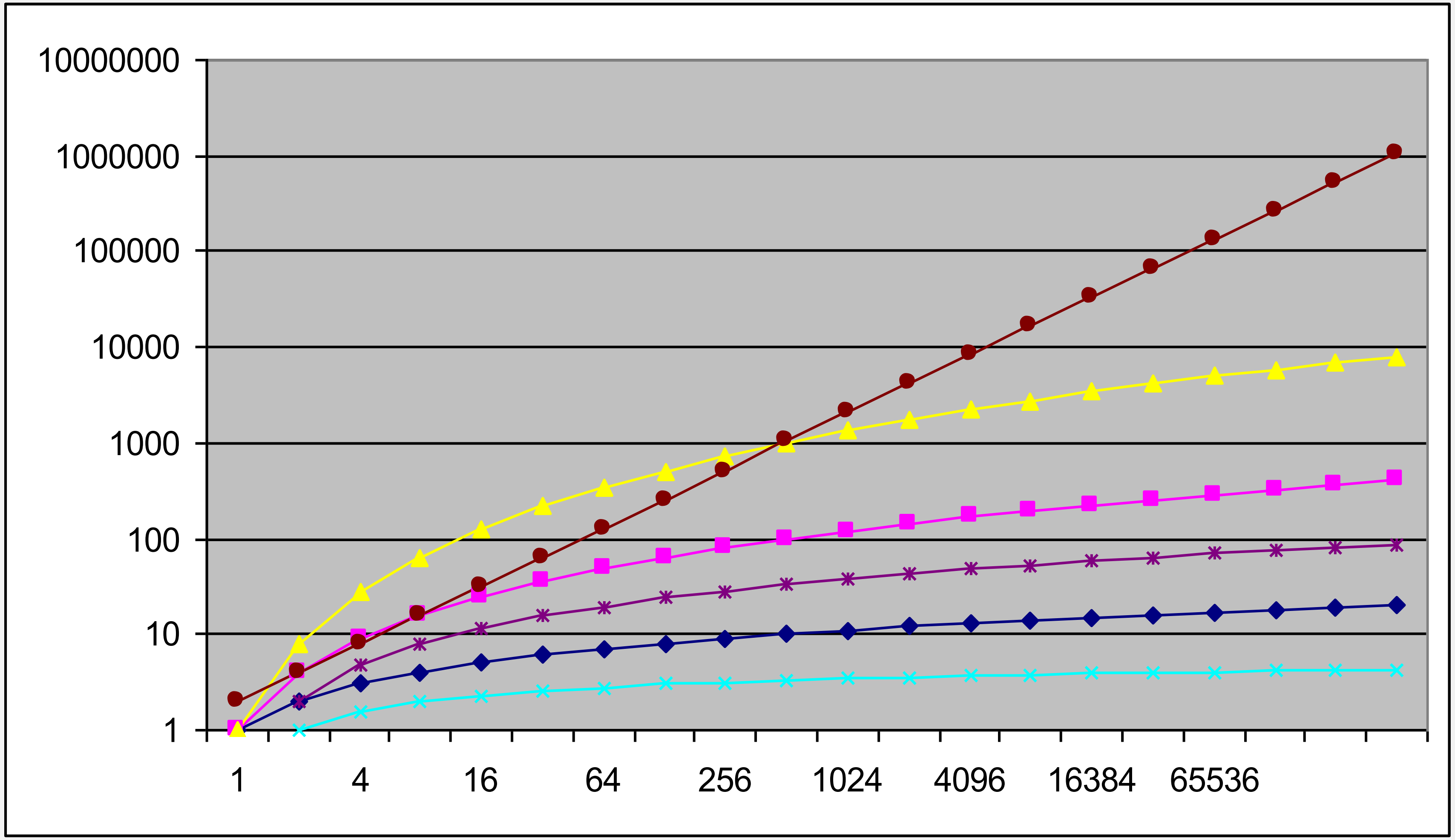
BÜYÜME HIZI FONKSİYONLARININ KARŞILAŞTIRILMASI



KARMAŞIKLIK



KARMAŞIKLIK



BÜYÜME HIZI FONKSİYONLARI

$O(1)$	Zaman gereksinimi **sabit**tir ve problem boyutundan bağımsızdır.
$O(\log_2 n)$	Zaman gereksinimi **logaritmiktir** ve problem boyutuna göre yavaş artar.
$O(n)$	Zaman gereksinimi **doğrusal**dır ve problem girişiyle doğru orantılı artar.
$O(n*\log_2 n)$	Zaman gereksinimi **$n*\log_2 n$** dir ve **doğrusal**dan daha hızlı artar.
$O(n^2)$	Zaman gereksinimi **karesel** olup problem boyutuna göre hızlı bir artış gösterir.
$O(n^3)$	Zaman gereksinimi c**ubic** problem boyutuna göre hızlı bir artış gösterir.
$O(2^n)$	problem girdi boyutu artarken zaman üstel (çok çok hızlı) olarak artar.

BÜYÜME HIZI FONKSİYONLARI

- Bir algoritma 8 elemanlı bir problemi 1 saniyede sonuçlandırıyorsa 16 elemanlı bir problem için ne kadar zaman gerekir.
- Algoritmanın mertebesi:

$O(1)$ ➔ $T(n) = 1$ saniye

$O(\log_2 n)$ ➔ $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ saniye

$O(n)$ ➔ $T(n) = (1*16) / 8 = 2$ saniye

$O(n*\log_2 n)$ ➔ $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ saniye

$O(n^2)$ ➔ $T(n) = (1*16^2) / 8^2 = 4$ saniye

$O(n^3)$ ➔ $T(n) = (1*16^3) / 8^3 = 8$ saniye

$O(2^n)$ ➔ $T(n) = (1*2^{16}) / 2^8 = 2^8$ saniye = 256 saniye

BÜYÜME HIZI FONKSİYONLARININ ÖZELLİKLERİ

1. *Algoritmanın büyüme hızı fonksiyonundaki düşük dereceli terimleri yok sayabiliriz.*
    - $O(n^3+4n^2+3n)$, aynı zamanda $O(n^3)$ olarak ifade edilebilir.
    - Büyüme hızı fonksiyonu olarak sadece en yüksek derece kullanılabilir.

2. *Büyüme hızı fonksiyonundaki en yüksek dereceli terimin sabit çarpanını yok sayabiliriz.*
    - $O(5n^3)$, aynı zamanda $O(n^3)$ ile ifade edilir.

3. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
    - Büyüme hızı fonksiyonları birleştirilebilir.

BAZI EŞİTLİKLER

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

ÖRNEK 1

```
i = 1;
sum = 0;
while (i <= n) {
      i = i + 1;
      sum = sum + i;
}
```

maliyet	**Tekrar**
c1	1
c2	1
c3	n+1
c4	n
c5	n

$$\begin{aligned} T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*c5 \\ &= (c3+c4+c5)*n + (c1+c2+c3) \\ &= a*n + b \end{aligned}$$

➔ Algoritma büyüme hızı: **O(n)**

ÖRNEK2

```
                                                      maliyet            Tekrar
i=1;                                                  c1                   1
sum = 0;                                              c2                   1
while (i <= n) {                                      c3                   n+1
      j=1;                                            c4                   n
      while (j <= n) {                                c5                 n*(n+1)
          sum = sum + i;                              c6                   n*n
          j = j + 1;                                  c7                   n*n
   }
   i = i +1;                                          c8                   n
}
```

$$
\begin{aligned}
T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8 \\
&= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3) \\
&= a*n^2 + b*n + c
\end{aligned}
$$

➔ Algoritmanın büyüme hızı fonksiyonu: **$O(n^2)$**

ÖRNEK 3

	maliyet	**Tekrar**
`for (i=1; i<=n; i++)`	c1	n+1
`for (j=1; j<=i; j++)`	c2	$\sum_{j=1}^{n}(j+1)$
`for (k=1; k<=j; k++)`	c3	$\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)$
`x=x+1;`	c4	$\sum_{j=1}^{n}\sum_{k=1}^{j}k$

$$T(n) = c1*(n+1) + c2*\left(\sum_{j=1}^{n}(j+1)\right) + c3*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)\right) + c4*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}k\right)$$

$$= a*n^3 + b*n^2 + c*n + d$$

➔ Algoritmanın büyüme hızı fonksiyonu: **O(n^3)**

ÖRNEK:SIRALI ARAMA

```
int sequentialSearch(const int a[], int item, int n){
   for (int i = 0; i < n && a[i]!= item; i++);
   if (i == n)
         return -1;
   return i;
}
```

Aranan eleman bulunamadı: ➔ O(n)

Aranan eleman bulundu:

Best-Case: Aranan eleman dizinin ilk elemanı ➔O(1)

Worst-Case: Aranan eleman dizinin son elemanı ➔O(n)

Average-Case: Karşılaştırma sayısı, 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2 + n)/2}{n}$$ ➔ O(n)

İKİLİ ARAMA - BINARY SEARCH

```
int binarySearch(int a[], int size, int x) {
   int low =0;
   int high = size -1;
   int mid;     // mid will be the index of
                       // target when it’s found.
   while (low <= high) {
     mid = (low + high)/2;
     if (a[mid] < x)
        low = mid + 1;
     else if (a[mid] > x)
          high  = mid - 1;
     else
          return mid;
   }
   return -1;
}
```

İKİLİ ARAMA: ANALİZ

- Aranan eleman bulunamadı:
  - Döngüdeki adım sayısı: $\lfloor \log_2 n \rfloor + 1$

    ➔ $O(\log_2 n)$
- Aranan eleman bulundu:
  - ***Best-Case:*** tek adımda bulunur. ➔ $O(1)$
  - ***Worst-Case:*** Adım sayısı: $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$
  - ***Average-Case:*** Adım sayısı $< \log_2 n$ ➔ $O(\log_2 n)$

`0   1   2   3   4   5   6   7` ← 8 elemanlı bir dizi

`3   2   3   1   3   2   3   4` ← # adımlar

Ortalama adım sayısı= $21/8 < \log_2 8$

n	***O(log$_2$n)***
16	4
64	6
256	8
1024 (1KB)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1MB)	20
1,073,741,824 (1GB)	30