ALGORİTMA ANALİZİ VE TASARIMI

BIG O GÖSTERİMİ

- Çalışma zamanının üst sınırını göstermektedir (en kötü durum).
- 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 **$O(n^2)$.**
- A algoritması **n** ile orantılı zaman gerektiriyorsa **O(n)** ile ifade edilir.

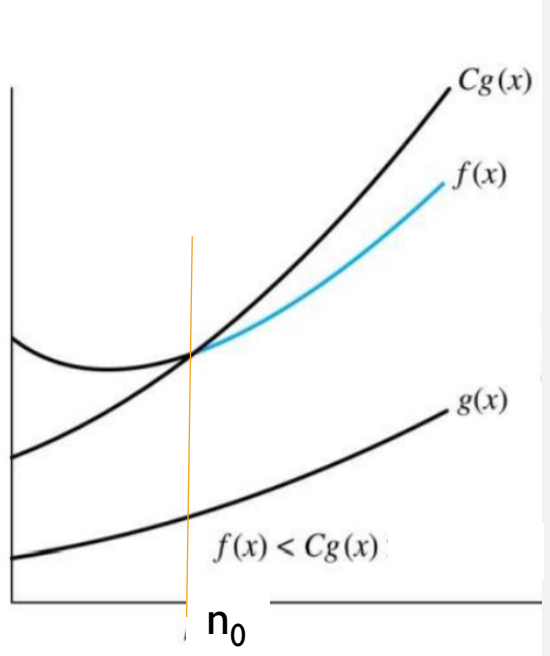
BIG O GÖSTERİMİ

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.

Yani; f(n) ve g(n) iki fonksiyon olsun. Her $n \geq n_0$ ve $c > 0$ için eger $f(n) \leq c.g(n)$, n>0 oluyorsa, f(n)=O(g(n)) yani g(n):f(n) için üst sınırdır denir.

$\lim_{n \to \infty} \frac{f(n)}{g(n)}$ var ve ∞'dan farklı ise $\Rightarrow f(n) = O(g(n))$



BIG O GÖSTERİMİ

- 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;

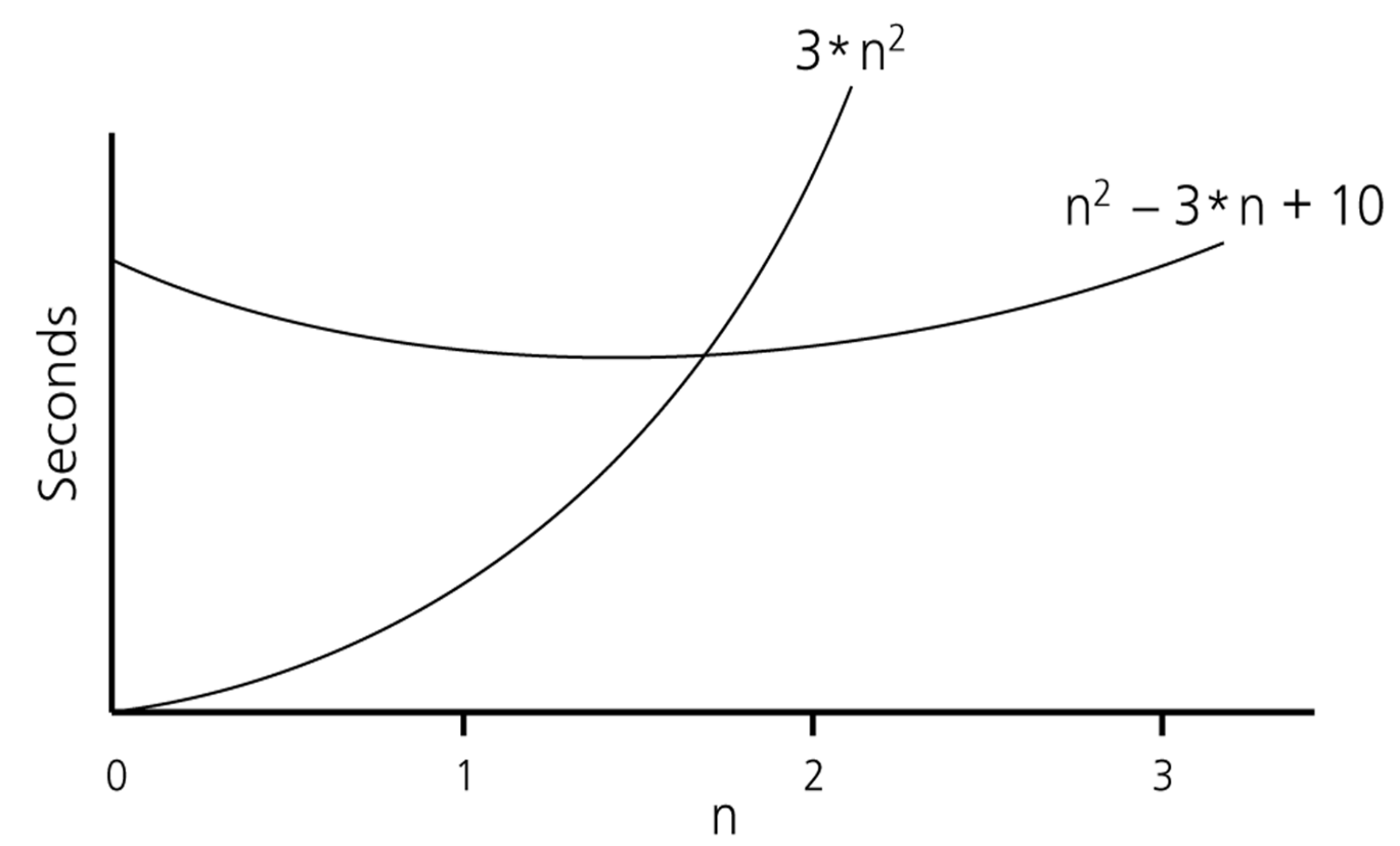
  bütün $n \geq n_0$ için $k*n^2 > n^2–3*n+10$

  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.**


3*n2
n2 – 3*n + 10
Seconds
0
1
2
3
n

BÜYÜK- Ω(BIG-OMEGA) GÖSTERİMİ

- Alt sınır hakkında bilgi verir (en iyi durum).

$\lim_{x \to \infty} \frac{f(x)}{g(x)} \neq 0 \rightarrow f(x) = \Omega(g(x))$

Tanım: ***f*** ve ***g,*** tamsayı kümesinden veya reel sayı kümesinden reel sayılara tanımlanmış olsun. $\mathbb{Z}+ \rightarrow \mathbb{R}$

Eğer, $x > k$ olduğunda $|f(x)| \geq C|g(x)|$ oluyorsa ve bu eşitsizliği sağlayan C ve k gibi sabit sayılar varsa $f(x)=\Omega(g(x))$ olmaktadır.

Big-O ile Big- Ω arasında sıkı bir ilişki vardır.
Ancak ve ancak g(x), O(f(x)) olduğunda f(x), Ω(g(x)) olacaktır.

BÜYÜK- Θ(BIG-THETA) GÖSTERİMİ

- Çalışma zamanı hakkında yaklaşık değil, tam cevap verir. Hem alt hem üst sınır g(x) ise, g(x) tam çözümdür (ortalama durum).

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = C \Rightarrow f(x) = \Theta(g(x)) \text{ olur.}$$

Tanım: ***f*** ve ***g,*** tamsayı kümesinden veya reel sayı kümesinden reel sayılara tanımlanmış olsun. $\mathbb{Z} + \rightarrow \mathbb{R}$

Eğer, f(x), O(g(x)) ve f(x), Ω(g(x)) ise f(x), θ(g(x)) deriz.

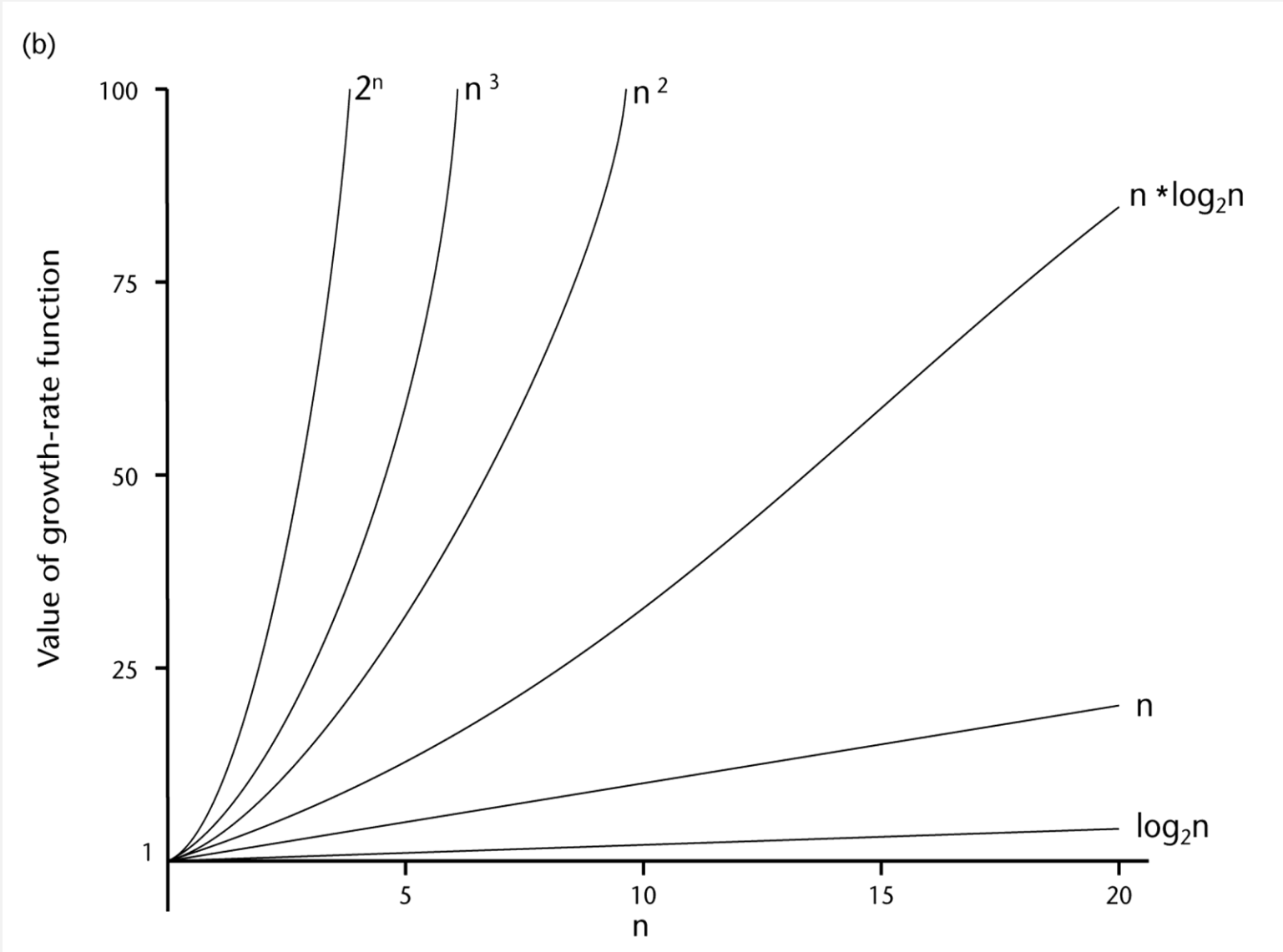
Eğer x>k olduğunda $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ oluyorsa ve bu eşitsizliği sağlayan pozitif C_1 ve C_2 reel sayıları ve bir pozitif ***k*** reel sayısı bulunabiliyorsa bu durumda f(x)'in θ(g(x)) olduğunu gösterebiliriz.

BÜYÜME HIZI FONKSİYONLARININ KARŞILAŞTIRILMASI

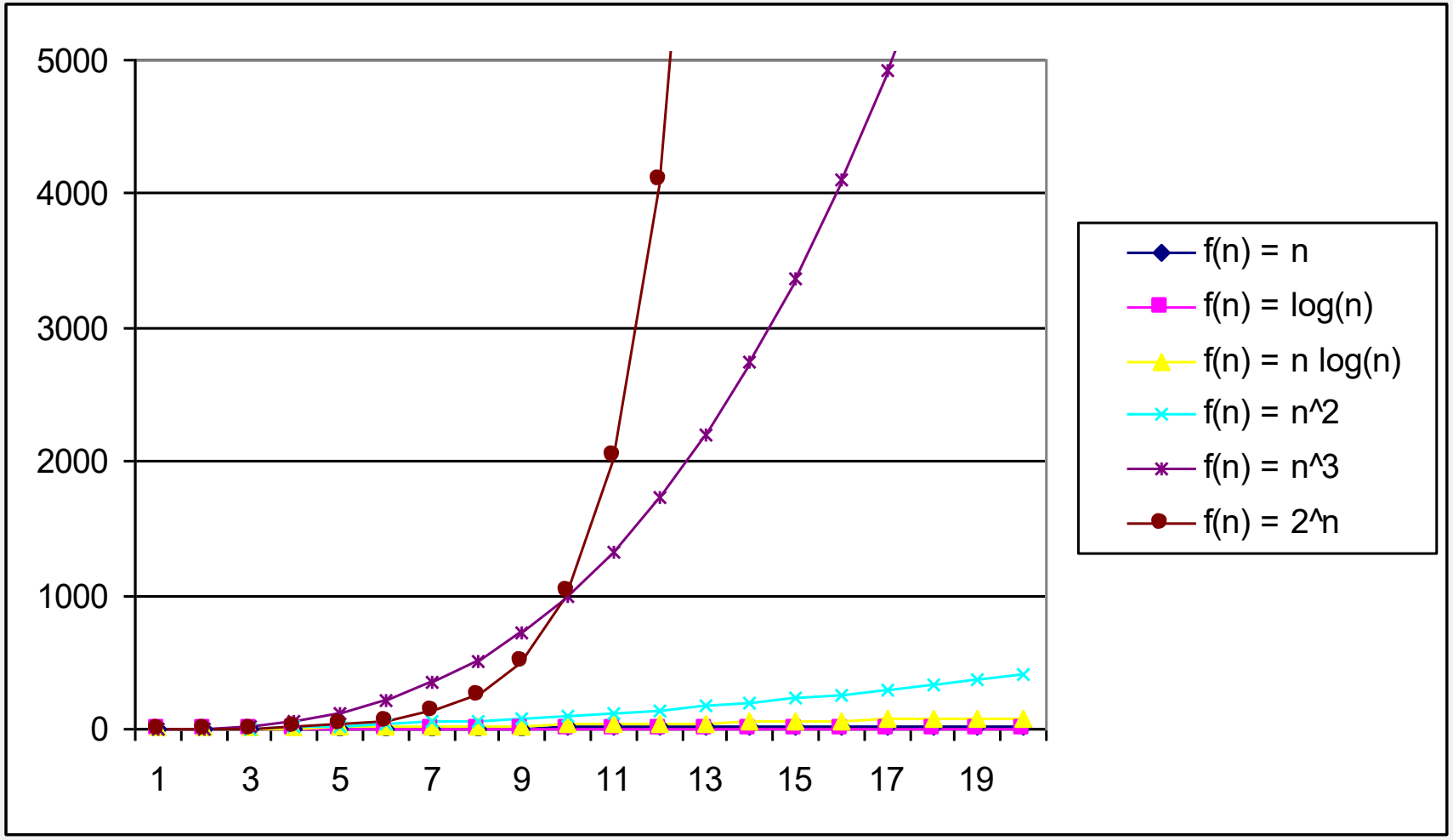
(a)

Function	n = 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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

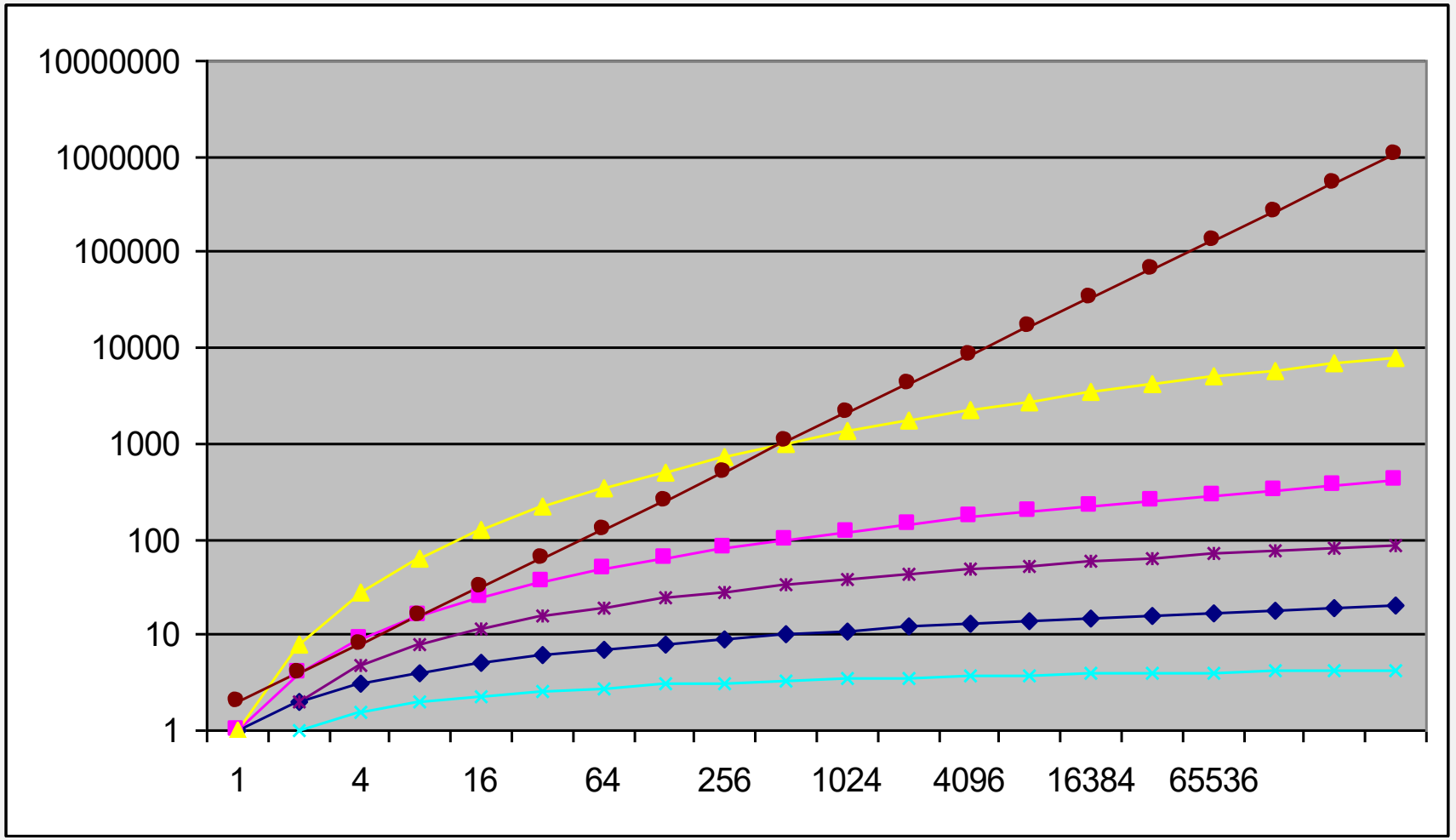
BÜYÜME HIZI FONKSİYONLARININ KARŞILAŞTIRILMASI



KARMAŞIKLIK



KARMAŞIKLIK



BÜYÜME HIZI FONKSİYONLARI

O(1) Zaman gereksinimi **sabit**tir ve problem boyutundan bağımsızdır.

O(log_2n) Zaman gereksinimi **logaritmiktir** ve problem boyutuna göre yavaş artar.

O(n) Zaman gereksinimi **doğrusal**dır ve problem girişiyle doğru orantılı artar.

O(n*log_2n) Zaman gereksinimi **n*log_2n** 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 I

```
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

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

= (c3+c4+c5)*n + (c1+c2+c3)

= a*n + b

➔ 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
`}`		

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

= $(c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$

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

➔ 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

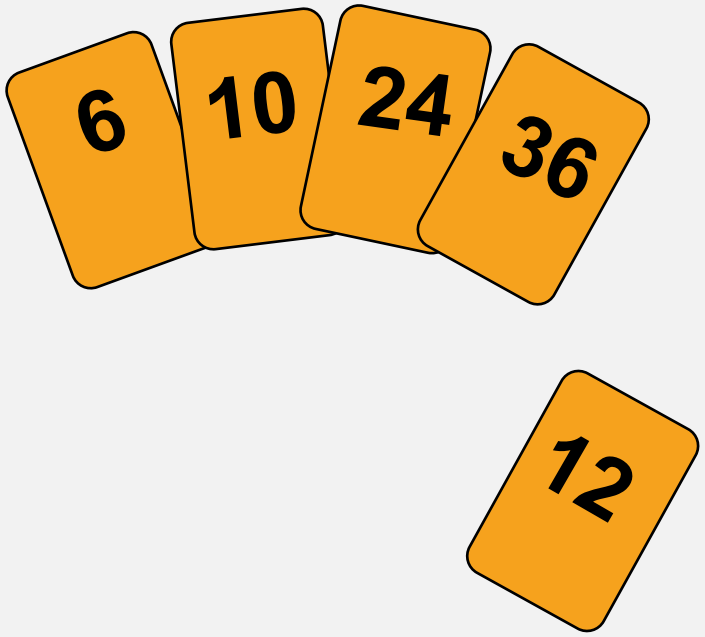
Ortalama adım sayısı= $21/8 < \log_2 8$

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

INSERTION SORT

- Fikir: Oyun kartlarını sıralamaya benzer.
  - Sol elimiz boş ve masadaki kartlar resimleri aşağıda (tersyüz) olarak başlarız.
  - Masadan bir kart alırız ve onu sol elimizde uygun yere yerleştiririz.
    - Elimizdeki kartların herbiriyle sağdan sola karşılaştırırız.
  - Sol elimizde tuttuğumuz kartlar sıralıdır.
    - Bu kartlar orijinal olarak masadaki destenin en üstündeki kartlar idi.

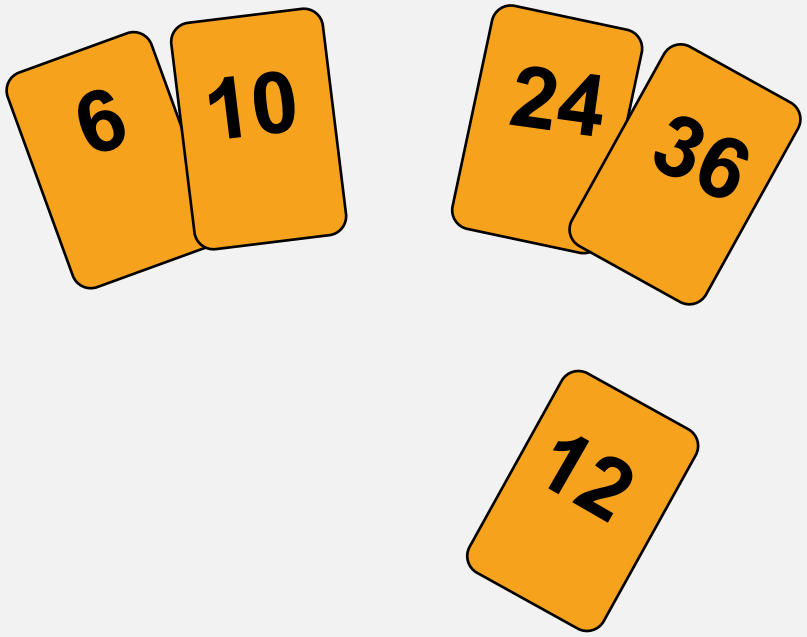
INSERTION SORT



12’yi eklemek için önce 36’yı ve daha sonra 24’ü sağa kaydırmamız gerekmektedir.

INSERTION SORT

INSERTION SORT



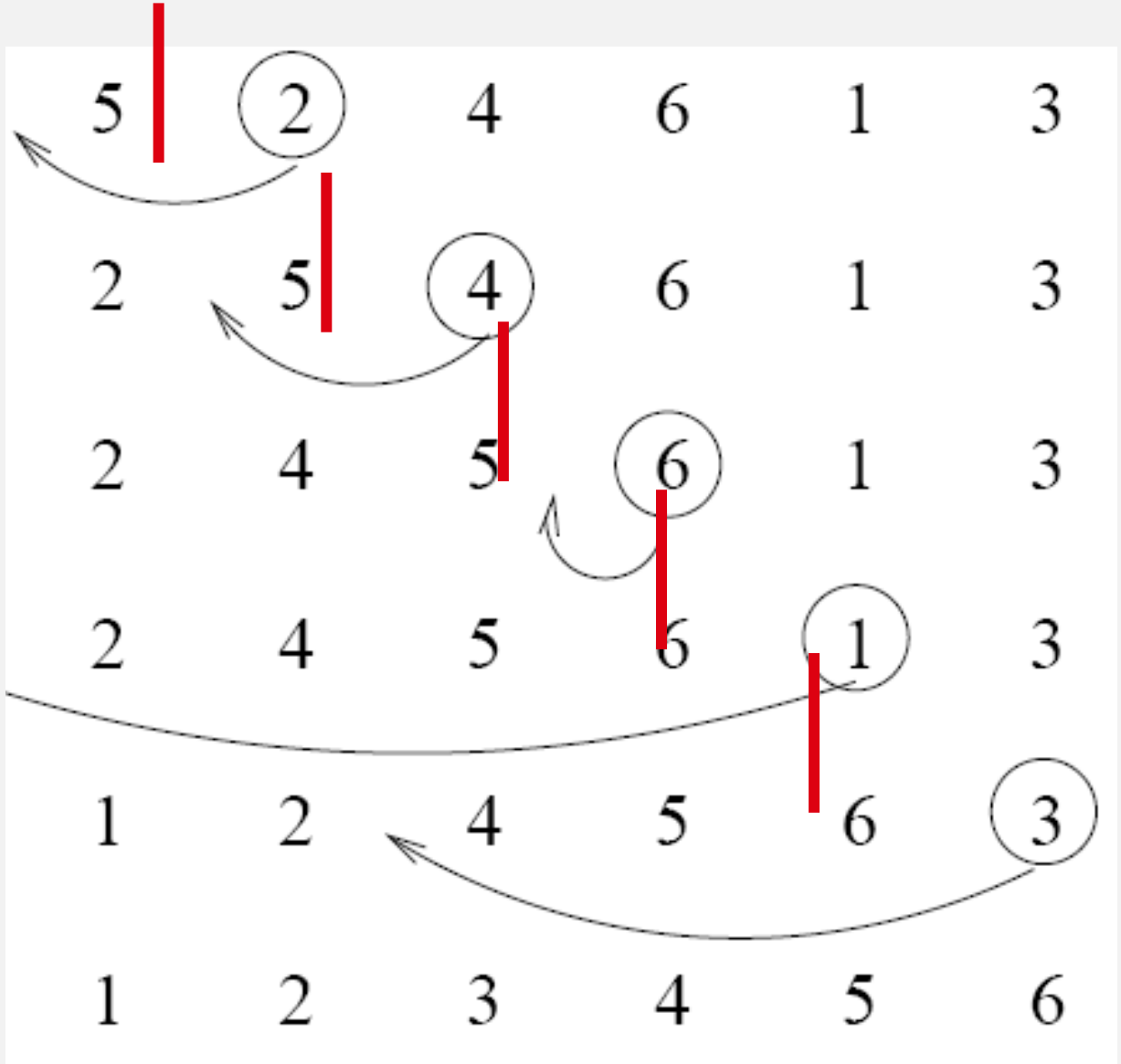
INSERTION SORT

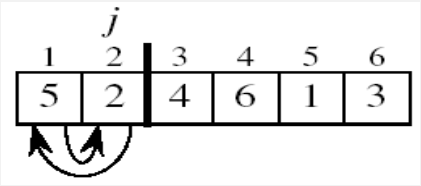
Giriş Dizisi

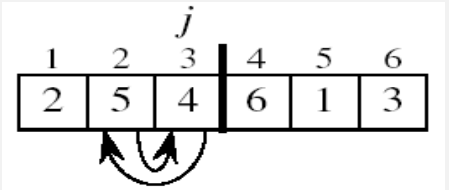
5 2 4 6 1 3

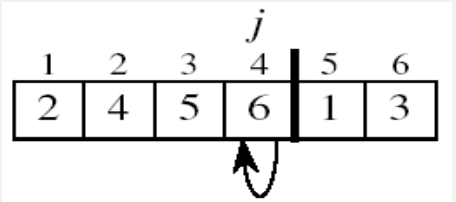
Her adımda,dizi iki alt diziye bölünür.

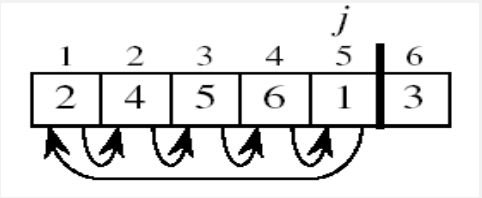
INSERTION SORT

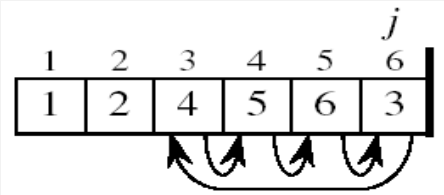




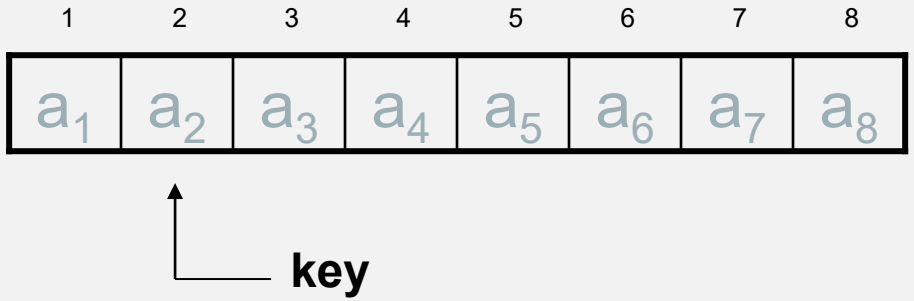








INSERTION-SORT



Alg.: INSERTION-SORT*(A)*

for j ⊳← 2 **to** n //2. elemandan sona kadar devam et

do key ← A[j] //2. eleman yerini aradığımız key olsun

A[j] sıralı A[1 . . j -1] dizisine ekle

i ← j - 1

while i > 0 and A[i] > key //i. eleman key'den küçükse i dizinin başına gelene kadar azalarak devam etsin.

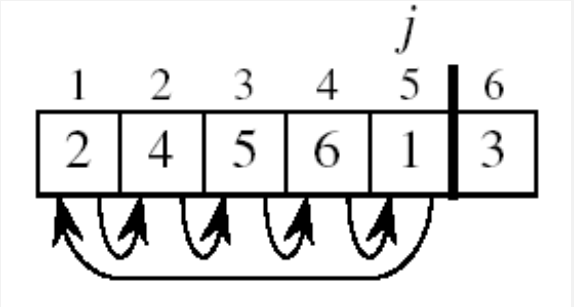
do A[i + 1] ← A[i] //dizide key'in yeni yerini bulma

i ← i - 1

A[i + 1] ← key //yerini bulunca key'i oraya yerleştirme

- Insertion sort elemanları dizi içinde sıralar.

INSERTION SORT DÖNGÜ İÇERİKLERİ



Alg.: INSERTION-SORT*(A)*

for j ← 2 **to** n

do key ← A[j]

A[j]'yi sıralı A[1 . . j -1] dizisine ekle

i ← j - 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i - 1

A[i + 1] ← key

Invariant: **for** döngüsünün başlangıcında A[1 . . j-1] içerisindeki elemanlar sıralıdır.

DÖNGÜLERİN ÇALIŞMASI

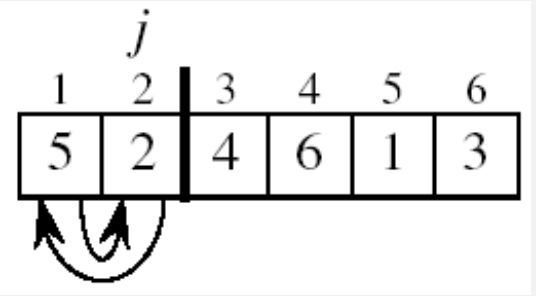
- Döngü değişkenlerinin sınanması «induction» gibi çalışır.
- **Döngü Başlangıcı (base case):**
  - Döngünün ilk iterasyonundan önce doğru
- **Emin olma (inductive step):**
  - Döngünün bir iterasyonundan önce Doğru ise sonraki iterasyonundan önce doğru kalır.
- **Sonlanma (Termination):**
  - Döngü sonlandığında, Değişmezler algoritmanın doğru olduğunu göstermeye yardım eden yararlı özellikler verir.

DÖNGÜ SÜREÇLERİ

- **Başlangıç:**
  - İlk iterasyondan hemen önce,j = 2:

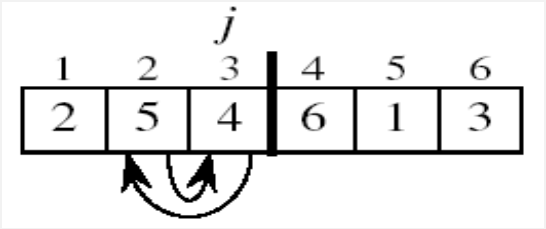
    $A[1 . . j-1] = A[1]$, alt dizisi

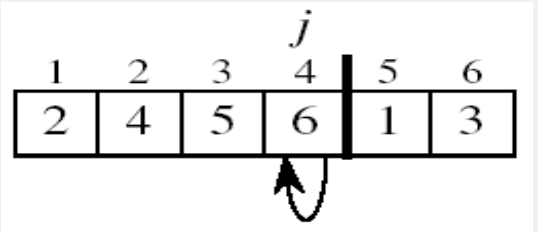
  (1 elemanlı bir dizi doğal olarak sıralıdır.)



DÖNGÜ SÜREÇLERİ

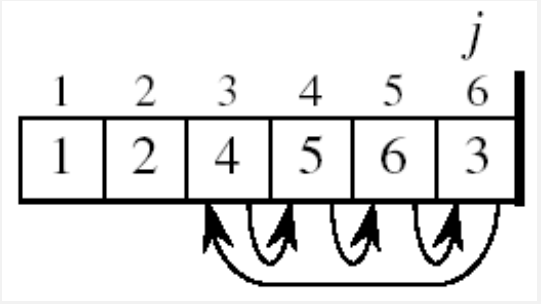
- **Maintenance: (Emin olma)**
  - İç **while döngüsü** , key=A[j] değeri uygun pozisyona yerleşinceye kadar *A*[j -1], *A*[j -2], *A*[j -3],… biçiminde hareket eder
  - Bu noktada, key değeri uygun pozisyona yerleşir.

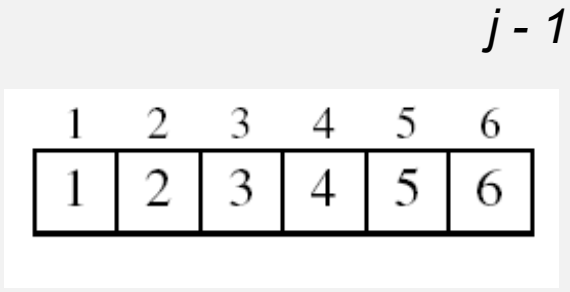




DÖNGÜ SÜREÇLERİ

- **Sonlanma:**
  - Dıştaki **for döngüsü** j = n + 1 $\Rightarrow$ j-1 = n olduğunda biter.
  - n, loop invariant içindeki j-1 ile yer değiştirir.
    - *A*[1 . . n] alt dizisi orjinal *A*[1 . . n] dizisini (sıralı) içerir.





INSERTION SORT

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷A[j]'yi sıralı A[1 . . j -1] dizisine ekle	0	n-1
i ← 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] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: j. İterasyonda çalıştırılan while ifadesi sayısı

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

EN İYİ DURUM ANALİZİ (BEST CASE ANALYSIS)

- Dizi zaten sıralıdır. "**while** i > 0 and A[i] > key"

  - **while** döngüsü testi ilk çalıştığında $A[i] \leq key$ olur ($i=j-1$)

  (küçükten büyüğe zaten sıralı olduğu için key, A[i]'den zaten büyük)

  - $t_j = 1$ (her turda sadece 1 karşılaştırma)

- $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

EN KÖTÜ DURUM ANALİZİ (WORST CASE ANALYSIS)

- Dizi tersten sıralıdır.

"while i > 0 and A[i] > key"

  - **while** döngüsü testinde her zaman A[i] > key olacaktır.
  - Key j'nci pozisyonun solundaki bütün elemanlarla karşılaştırılır

(j. elemanı⇒ karşılaştır j-1 eleman ile ⇒ t_j = j)

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Elimizde olanların özdeşlikleri:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$= an^2 + bn + c$ Karesel bir fonksiyon

- $T(n) = \Theta(n^2)$ =>> Büyüme derecesi:n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

INSERTION SORT ALGORİTMASINDAKİ KARŞILAŞTIRMA VE YER DEĞİŞTİRMELER

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j]'yi sıralı diziye A[1 . . j -1]	0	n-1
i ← 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] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

$\approx n^2/2$ karşılaştırma

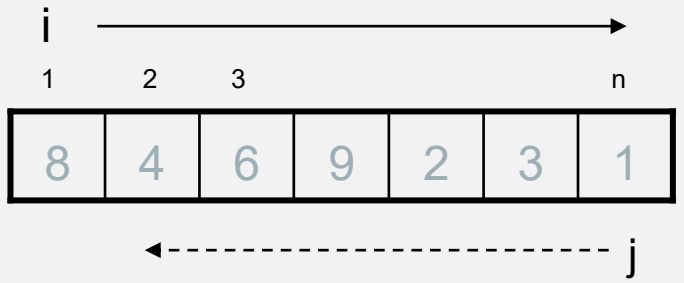
$\approx n^2/2$ yerdeğiştirme

GENEL DEĞERLENDİRME

- Avantaj:
  - Zaten sıralı bir dizi için iyi bir çalışma zamanına sahiptir: $\Theta(n)$
- Dezavantaj:
  - Ortalama ve en kötü durumlarda çalışma zamanı $\Theta(n^2)$ olur.
  - $\approx n^2/2$ karşılaştırma ve yer değiştirme

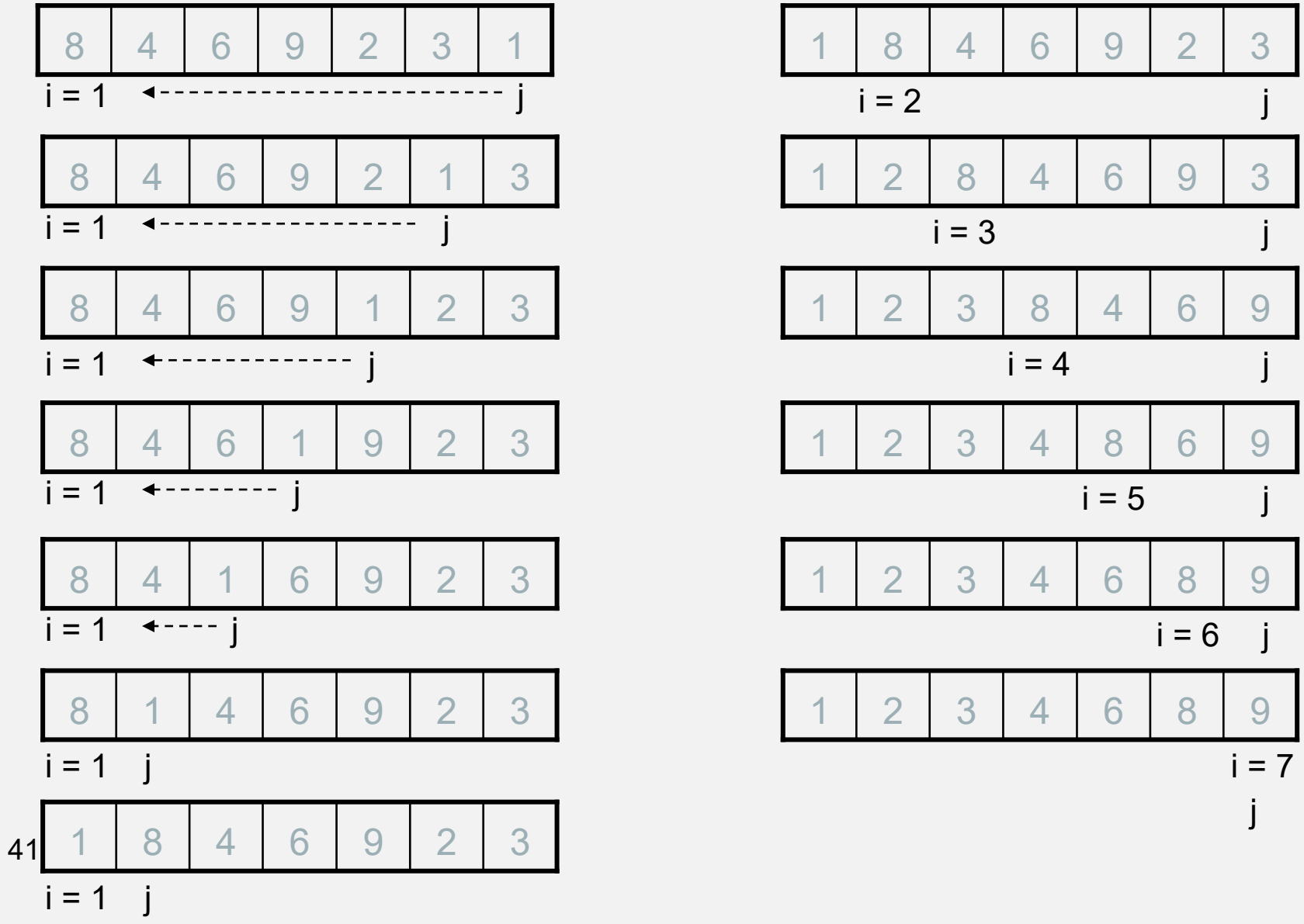
BUBBLE SORT

- İşletim :
  - Tekrar tekrar dizinin elemanları test edilerek dizi taranır.
  - Büyüklük küçüklük durumuna göre komşu elemanla yer değiştirilir.



- İşletimi kolay olmasına rağmen çalışma zamanı kötü $(O(n^2))$ bir algoritma
- İşletimi kolaydır,Fakat Insertion sort algoritmasından daha yavaştır.

ÖRNEK


8 4 6 9 2 3 1
i = 1 j
8 4 6 9 2 1 3
i = 1 j
8 4 6 9 1 2 3
i = 1 j
8 4 6 1 9 2 3
i = 1 j
8 4 1 6 9 2 3
i = 1 j
8 1 4 6 9 2 3
i = 1 j
1 8 4 6 9 2 3
i = 1 j
1 8 4 6 9 2 3
i = 2 j
1 2 8 4 6 9 3
i = 3 j
1 2 3 8 4 6 9
i = 4 j
1 2 3 4 8 6 9
i = 5 j
1 2 3 4 6 8 9
i = 6 j
1 2 3 4 6 8 9
i = 7
j

BUBBLE SORT

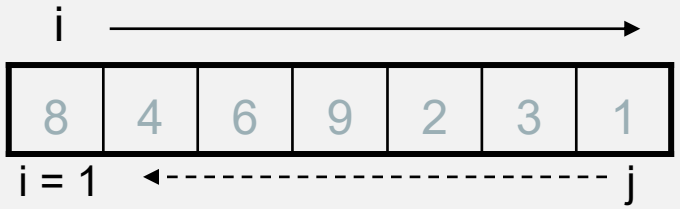
Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

do for j ← length[A] **downto** i + 1

do if A[j] < A[j -1]

then exchange A[j] ↔ A[j-1]



BUBBLE-SORT ALGORİTMASININ ÇALIŞMA ZAMANI (RUNNING TIME) ANALİZİ

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A] c_1

do for j ← length[A] **downto** i + 1 c_2

Karşılaştırmalar: ≈ $n^2/2$ **do if** A[j] < A[j -1] c_3

Yerdeğiştirme: ≈ $n^2/2$ **then** exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

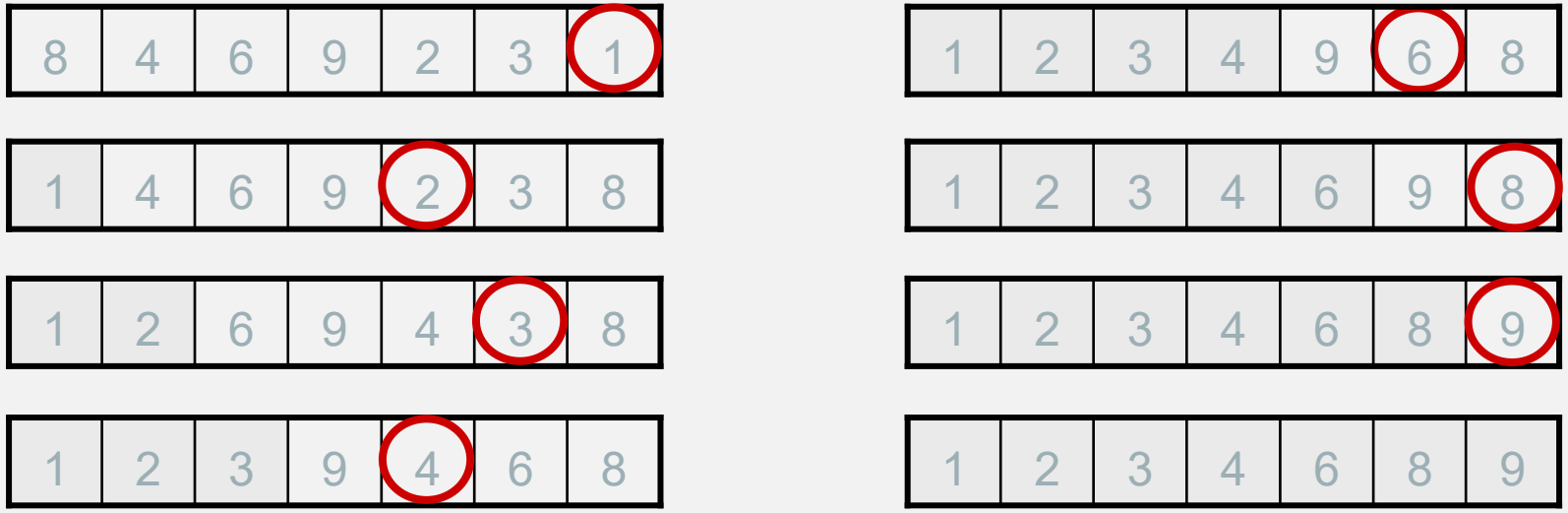
$$\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Böylece, $T(n) = \Theta(n^2)$

SELECTION SORT

- İşletim:
  - Dizideki en küçük elemanı bul
  - En küçük elemanı dizinin ilk elemanı ile yer değiştir.
  - Dizinin ikinci en küçük elemanını bul ve bunu dizinin ikinci indisindeki elemanla yer değiştir.
  - Dizi sıralanıncaya kadar bu işlemlere devam et.
- Dezavantaj:
  - Çalışma zamanı, dizideki elemanların düzenine çok az bağlı, eleman sayısını artması durumunda çok maliyetli.

ÖRNEK:


8 4 6 9 2 3 1
1 4 6 9 2 3 8
1 2 6 9 4 3 8
1 2 3 9 4 6 8
1 2 3 4 9 6 8
1 2 3 4 6 9 8
1 2 3 4 6 8 9
1 2 3 4 6 8 9

SELECTION SORT

Alg.: SELECTION-SORT*(A)*

n ← length[A]

for j ← 1 **to** n – 1 //dizinin başından sonunda kdr git

do ek ← j //ek=1.eleman olsun

for i ← j + 1 **to** n //sonraki elemanları teker teker kontrol et, ek'den küçükse ek olarak ata

do if A[i] < A[ek]

then ek ← i

exchange A[j] ↔ A[ek] //bu türde kontrol bitince, dizinin ilk indisine ek yerleştir

SELECTION SORT ALGORİTMASININ ANALİZİ

Alg.: SELECTION-SORT(*A*)

	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do ek ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[ek]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then ek ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[ek]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$