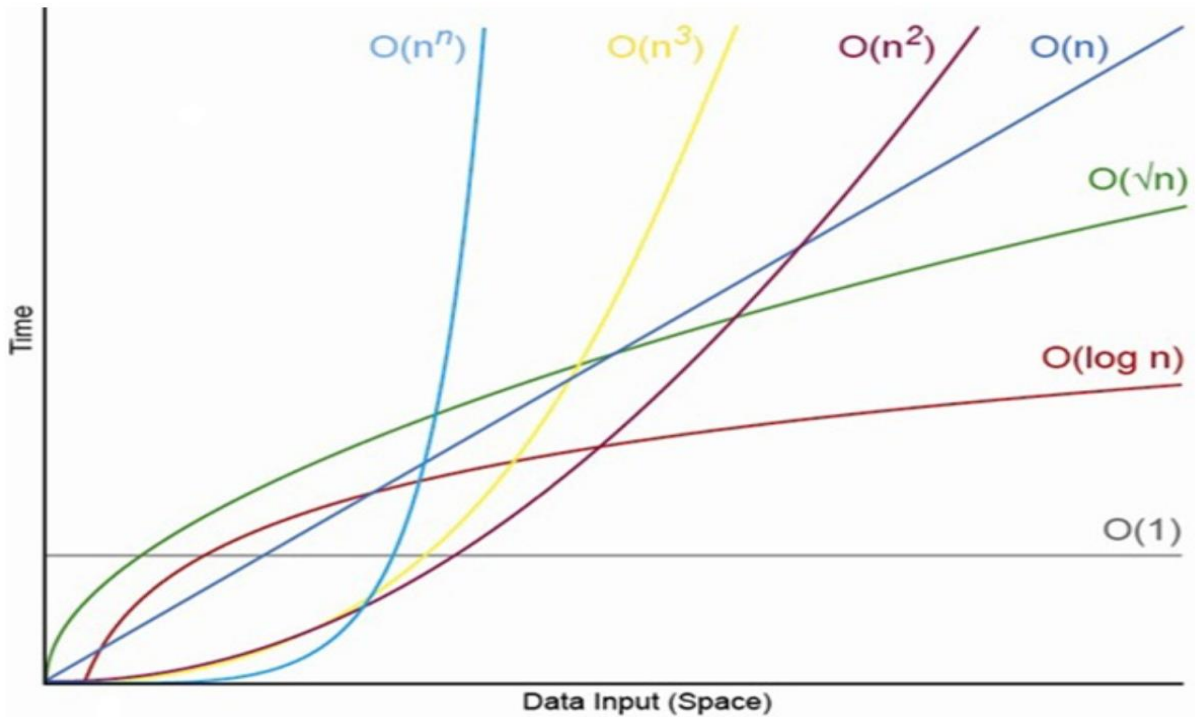


# ALGORİTMALAR

Herhangi bir problemi çözmek için gerekli olan adımlar bütününe “**Algoritma**” denir.

## Algoritmaların Özellikleri

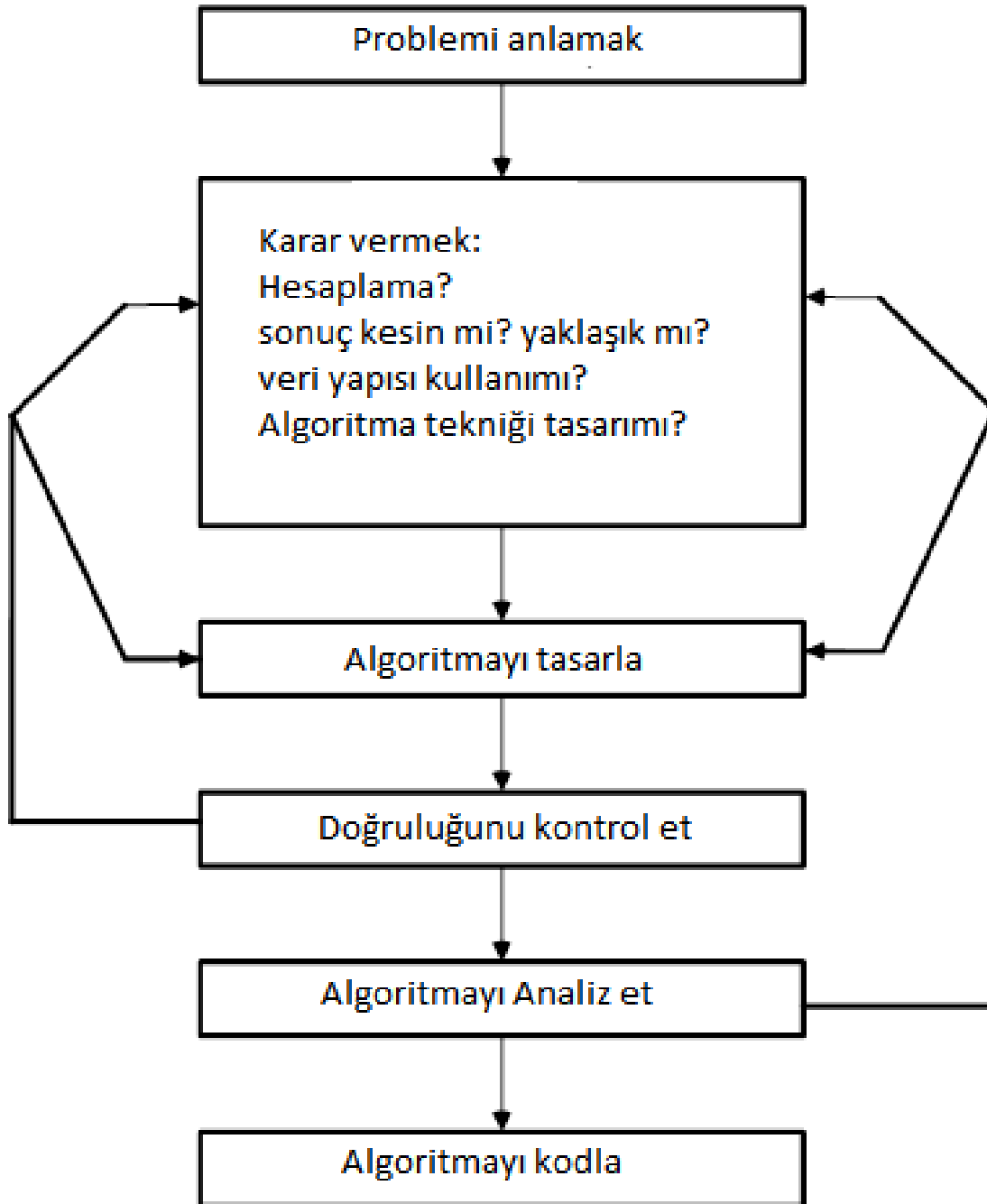
- **Input** Girdi, bir kümedir,
- **Output** Çıktı, bir kümedir (çözümdür)
- **Definiteness** Kesinlik (algoritmanın adımlarının belirli olması),
- **Correctness** Doğruluk (bütün girdiler için algoritmanın tanımlı olması),
- **Finiteness** Sonluluk (çalışma adımlarının sonlu olması),
- **Effectiveness** Verimlilik (Her adımın ve sonuca ulaşan yolun)
- **Generality** Genellenebilirlik (bir problem kümesi için).



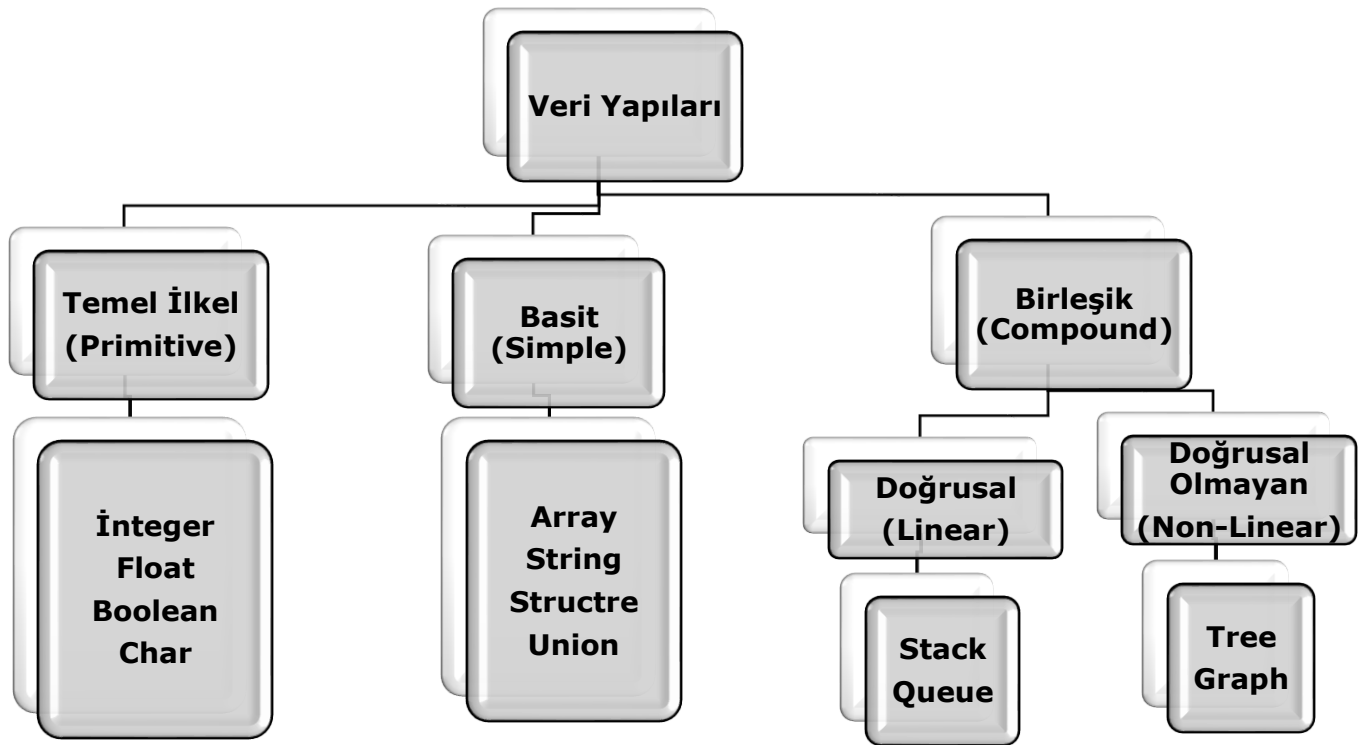
Yatay(x) çizdiğimiz algoritmaya verilen girdilerdir. Dikey(y) çizğimiz bu girdiyi çözmek için ne kadar uğraştığımızı gösterir. İçerisinde gösterilen renkli grafik çizgilerimizin her biri bir algoritmayı temsil etmekte ve;

- Çizgi ne kadar yatay ise algoritmamız o kadar verimli ve hızlıdır.
- Çizgi ne kadar dikey ise algoritmamız o kadar verimsiz ve yavaştır.

# Algoritma Tasarım Süreci



Yazılımın veriye kolay ve hızlıca erişim sağlaması veya değişim yapmasını sağlayan veriyi saklamak, organize etmek ve yönetmek için kullanılan yöntemlere “**Veri Yapıları (Data Structures)**” denir.



Herhangi bir programlama dilinde yazdığımız algoritmayı test ederek veriyi ne kadar sürede ve ne kadar hızda çalıştığını algoritmayı analiz ederek bulabiliriz. **Algoritma analizinde önemli olan zaman ve bellektir.**

Örneğin aynı işi yapan iki farklı algoritmanın farklı zamanlarda sonucu elde ettiğini gözlemleyebiliriz. Her zaman daha iyisini alarak algoritmayı analiz edebiliriz.

**Yürütme Zamanı:** Algoritmanın belirli bir işleme kaç kez gereksinim duyduğunu gösteren bağıntıdır.  
( $n$  elemanlı bir küme için yürütme zamanı  $T(n)$  ile gösterilir.)

### Karmaşıklık (Complexity)

Algoritma analizi çalışmalarının dayandığı karmaşıklık teoremi genel olarak bir bilgisayar programının veya algoritmasının asimptotik değerini bulmayı hedefler. Buradaki amaç, algoritmanın ulaşabileceği en kötü veya en iyi durumu matematiksel olarak modellemektir.

•Şayet girdi küçükse hafıza (Space) ve zaman (time) karmaşıklıkları çok önemli değildir.

Örneğin  $n = 10$ , için ikili veya doğrusal arama kullanılması çok önemli değildir ama  $n = 230$  gibi eleman sayılarında aralarında yıllarca fark olabilir.

Örneğin, aynı problemi çözen  $A$  ve  $B$  gibi iki farklı algoritma olsun.

• $A$  için zaman karmaşıklığı  $5,000n$ , ve  $B$  için  $[1.1^n]$  olsun.

•  $n = 10$  için  $A$  algoritması 50,000 adımda biterken,  $B$  sadece 3, adımda bitmektedir.  $B$  çok daha iyidir denebilir mi?

•  $n = 1000$  için  $A$  5,000,000 adımda biterken  $B$  requires.  $2.5 \cdot 10^{41}$  adım da bitmektedir.

Peki hangisi iyidir?

•  $A$ , büyük veri kümeleri için de kullanışlıyken  $B$ 'nin kullanılamayacağı anlaşılmaktadır.

Karmaşıklığın büyümesi (growth) çok daha önemlidir. Algoritmaların karşılaştırılması için algoritmaların zaman ve hafıza karmaşıklıklarındaki büyüme karşılaştırılır.

### **Bir fonksiyonun büyümesi (Growth)**

• Fonksiyonların büyüme hızını matematikte big-o olarak ifade edilen fonksiyon gösterir.

• Tanım:  $f$  ve  $g$  fonksiyonları reel sayılardan reel sayılara tanımlı iki fonksiyon olsun.

•  $f(x)$  fonksiyonu için  $c$  ve  $k$  sabit olmak üzere  $O(g(x))$  aşağıdaki şekilde tanımlanır

$$|f(x)| \leq c |g(x)| + k$$

•  $x > k$  olmak koşuluyla

• Kısacası, fonksiyonu sabit bir değerle çarpmamız veya bir değerle toplamamız  $x > k$  olmak koşuluyla  $f(x)$  fonksiyonumuzun büyüme hızına etki etmez.

• Karmaşıklık fonksiyonlarının büyümesini karşılaştırırken  $f(x)$  ve  $g(x)$  fonksiyonları her zaman pozitif kabul edilir

• Dolayısıyla gösterim sadeleştirilebilir

$$f(x) \leq C * g(x) + k, x > k \text{ şartı ile.}$$

• Şayet  $f(x)$ 'in  $O(g(x))$  olduğunu göstermek istiyorsak bunu sağlayan bir  $(C, k)$  ikilisi bulmamız yeterlidir.

## Örnek

$f(x) = x^2 + 2x + 1$  fonksiyonunun büyüme fonksiyonunun  $x^2$  olduğunu (yani  $O(x^2)$  olduğunu) gösteriniz

•  $x > 1$  için (bu aynı zamanda  $x > k$ ):

$$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2$$

$$> x^2 + 2x + 1 \leq 4x^2$$

• Öyleyse  $C = 4$  ve  $k = 1$  için:

•  $f(x) \leq Cx^2 + k, x > k$  durumu sağlanır.

→  $f(x)$  is  $O(x^2)$ .

“Bizim için önemli olan fonksiyonun nasıl ve ne kadar hızlı büyüdüğüdür. Fonksiyonu hızlandıran ve büyüten durum ise fonksiyonun üstel derecesidir.”

Şayet  $f(x)$  için  $O(x^2)$ 'dir deniyorsa aynı zamanda  $O(x^3)$ 'tür de denebilir mi?

• Evet.  $x^3$  fonksiyonu  $x^2$ 'den daha hızlı büyür dolayısıyla  $x^3$  fonksiyonu da  $f(x)$ 'den daha hızlı büyüyecektir.

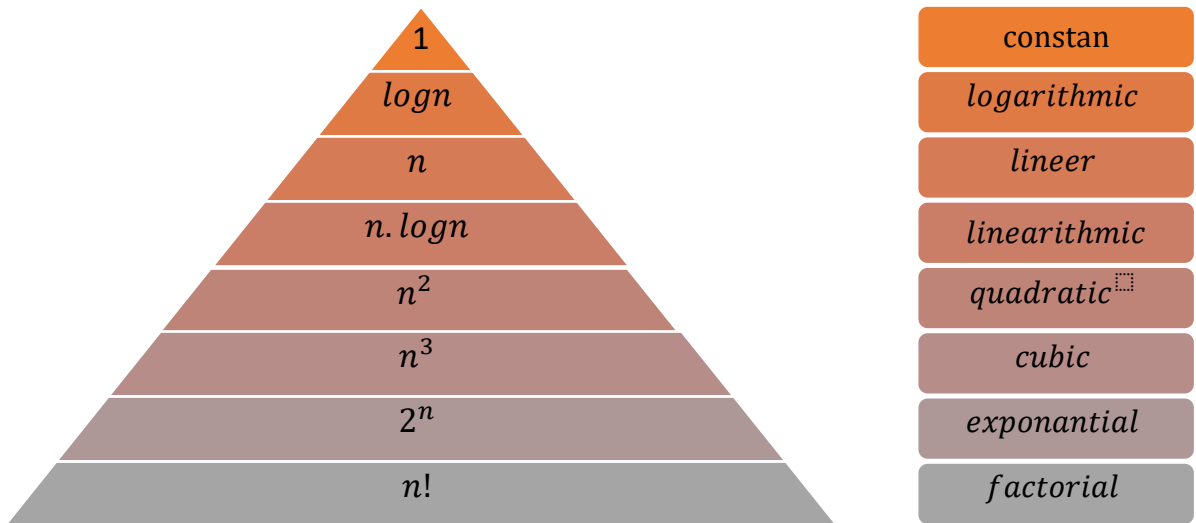
• Ancak karşılaştırma yapabilmek için her zaman bulunabilecek en küçük büyüme fonksiyonunu bulmak isteriz.

## Fonksiyonların Büyümesi

• Sık karşılaşılan fonksiyonlar:

•  $n \cdot \log n, 1, 2^n, n^2, n!, n, n^3, \log n$

En yavaştan hızlıya doğru sıralanmış hali



- Fonksiyon büyüdüğüçe, karmaşıklık büyür, hız ve verim azalır.
- Fonksiyon küçüldükçe karmaşıklık azalır, hız ve verim artar

## Çözümleme türleri

**En kötü durum (Worst-case):** (genellikle)

- $T(n)$  =  $n$  boyutlu bir girişte algoritmanın maksimum süresi

**Ortalama durum (Average case):** (bazen)

- $T(n)$  =  $n$  boyutlu her girişte algoritmanın beklenen süresi.
- Girişlerin istatistiksel dağılımı için varsayım gerekli.

**En iyi durum (Best case):** (gerçek dışı)

- Bir giriş yapısında hızlı çalışan yavaş bir algoritma ile hile yapmak.

### Örnek 1

- Aşağıdaki algoritma ne işe yarar?
- **Procedure** gizemli( $a_1, a_2, \dots, a_n$ : integers)
- $m := 0$
- **for**  $i := 1$  to  $n-1$
- **for**  $j := i+1$  to  $n$
- **if**  $|a_i - a_j| > m$  **then**  $m := |a_i - a_j|$
- { $m$  verilen girdideki herhangi iki sayı arasındaki en uzak mesafeyi verir}
- Karşılaştırma:  $n-1 + n-2 + n-3 + \dots + 1$
- $= (n-1)n/2 = 0.5n^2 - 0.5n$
- Zaman karmaşıklığı  $O(n^2)$ .

### Örnek 2

- Aynı problemi çözen farklı bir algoritma
- **procedure** max\_diff( $a_1, a_2, \dots, a_n$ : integers)
- $\min := a_1$
- $\max := a_1$
- **for**  $i := 2$  to  $n$
- **if**  $a_i < \min$  **then**  $\min := a_i$
- **else if**  $a_i > \max$  **then**  $\max := a_i$
- $m := \max - \min$
- Karşılaştırma Sayıları:  $2n - 2$
- Zaman Karmaşıklığı  $O(n)$ .

<b>Örnek:</b> Basit bir döngü	<b>Maliyet</b>	<b>Tekrar</b>
<i>i</i> = 1;	<i>c</i> 1	1
<i>sum</i> = 0;	<i>c</i> 2	1
<i>while</i> ( <i>i</i> <= <i>n</i> ) {	<i>c</i> 3	<i>n</i> + 1
<i>i</i> = <i>i</i> + 1;	<i>c</i> 4	<i>n</i>
<i>sum</i> = <i>sum</i> + <i>i</i> ;	<i>c</i> 5	<i>n</i>
}		

Örneğin *n* değerine 2 atarsak;

*i*=1'den küçüktür while döngüsü 1 kere döner

*i*=2 küçük eşittir while 1 kere döner

*i*=3 while döngüsüne girer ama

- Toplam maliyet =  $c_1 + c_2 + (n + 1) * c_3 + n * c_4 + n * c_5 = 3n + 3$
- $T(n) = 3n + 3$  ,  $T(n) = O(n)$
- Bu algoritma için gerekli zaman *n* ile doğru orantılıdır.

<b>Örnek:</b> İç içe döngü	<b>Maliyet</b>	<b>Tekrar</b>
<i>i</i> = 1;	<i>c</i> 1	1
<i>sum</i> = 0; : 0;	<i>c</i> 2	1
<i>while</i> ( <i>i</i> <= <i>n</i> ) {	<i>c</i> 3	<i>n</i> + 1
<i>j</i> = 1;	<i>c</i> 4	<i>n</i>
<i>while</i> ( <i>j</i> <= <i>n</i> ) {	<i>c</i> 5	$n * (n + 1)$
<i>sum</i> = <i>sum</i> + <i>i</i> ;	<i>c</i> 6	$n * n$
<i>j</i> = <i>j</i> + 1; }	<i>c</i> 7	$n * n$
<i>i</i> = <i>i</i> + 1;	<i>c</i> 8	<i>n</i>
}		

- Toplam maliyet=  $c_1 + c_2 + (n + 1) * c_3 + n * c_4 + n * (n + 1) * c_5 + n * n * c_6 + n * n * c_7 + n * c_8$
- $T(n) = 3n^2 + 4n + 2$  ,  $T(n) = O(n^2)$
- Bu algoritma için gerekli zaman  $n^2$  ile doğru orantılıdır.

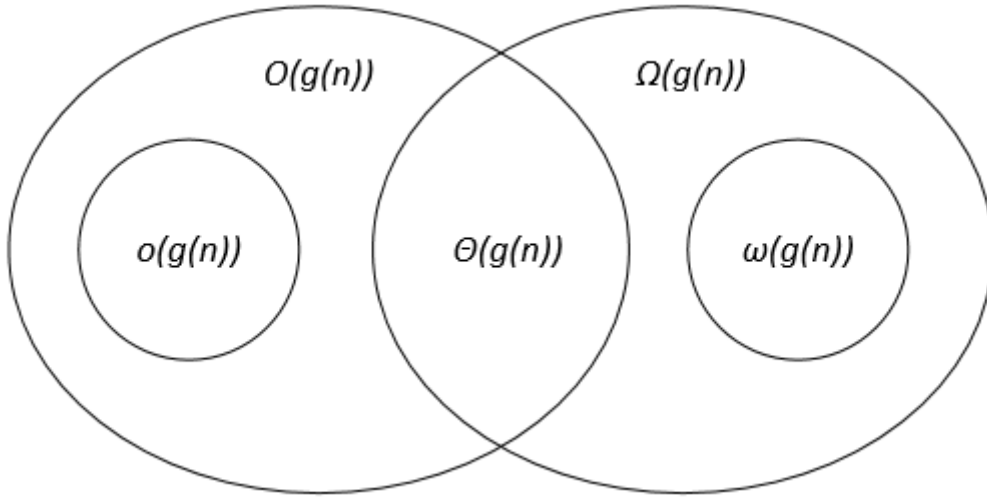
## Asymptotic Notasyon ve Temel Verimlilik Sınıfları

En önemlisi:  $n \rightarrow \infty$ 'a giderken algoritmanın performansı hangi sınırlarda bunu anlayabilmek.

### Asimptotik Büyüme Dereceleri

Fonksiyonların büyüme hızlarını karşılaştırmak için kullanılan, sabit çarpanları ve küçük girdi boyutlarını yok sayan, bir yöntem.

- $O(g(n))$ :  $g(n)$  fonksiyonundan daha hızlı büyümeyen  $f(n)$  fonksiyonlarını kapsar
- $\Theta(g(n))$ :  $g(n)$  fonksiyonları ile aynı derecede büyüyen  $f(n)$  fonksiyonlarını gösterir.
- $\Omega(g(n))$ : en az  $g(n)$  fonksiyonları kadar hızda büyüyen  $f(n)$  fonksiyonlarını belirtmek için kullanılır.



### Big O Formal Tanımı

**Tanım:**  $f(n) \in O(g(n))$  ise,  $f(n)$  fonksiyonunun büyüme derecesi,  $g(n)$ 'in büyüme sabit bir sayı ile çarpımının büyüme derecesinden küçüktür.

$$f(n) \leq c g(n), \forall n \geq n_0$$

Eşitsizliğini sağlayan pozitif bir sabit  $c$  ve pozitif bir tamsayı  $n_0$  vardır



## Örnekler:

$$T(n) = 3n + 8 = O(n) \text{'dir.} \quad (3n + 8 \in O(n))$$
$$3n + 8 \leq c * n$$

$$n \geq 1 \text{ için } 3n + 8 \leq 3n + 8n \leq 11n \quad (c = 11, n_0 = 1)$$

$$n \geq 4 \text{ için } 3n + 8 \leq 5n \quad (c = 5, n_0 = 4)$$

Diğer  $c$  ve  $n_0$  değerleri bulunabilir.

$$T(n) = 3n + 8 = O(n^2) \text{'dir.} \quad (3n + 8 \in O(n^2))$$
$$3n + 8 \leq c * n^2$$

$$n \geq 5 \text{ için } 3n + 8 \leq c * n^2 \quad (c = 1, n_0 = 5)$$

$$n \geq 3 \text{ için } 3n + 8 \leq c * n^2 \quad (c = 2, n_0 = 3)$$

Diğer  $c$  ve  $n_0$  değerleri bulunabilir.

$$T(n) = 2n^2 = O(n^3) \text{'dir.} \quad (2n^2 \in O(n^3))$$
$$2n^2 \leq c * n^3$$

$$n \geq 2 \text{ için } 2n^2 \leq c * n^3 \quad (c = 1, n_0 = 2)$$

Diğer  $c$  ve  $n_0$  değerleri bulunabilir.

$$T(n) = \frac{1}{2}n^2 + 3n = O(n^2) \text{'dir.} \quad (2n^2 \in O(n^3))$$
$$\frac{1}{2}n^2 + 3n \leq c * n^2$$

$$n \geq 6 \text{ için } \frac{1}{2}n^2 + 3n \leq c * n^2 \quad (c = 1, n_0 = 6)$$

Diğer  $c$  ve  $n_0$  değerleri bulunabilir.

Çözüm kümesini sağlayan kaç tane  $n_0$  ve  $c$  çifti olduğu önemli değildir.

Tek bir çift olması notasyonun doğruluğu için yeterlidir.

## Big O'nun Özellikleri

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$  iff  $g(n) \in \Omega(f(n))$
- If  $f(n) \in O(g(n))$  and  $g(n) \in O(h(n))$ , then  $f(n) \in O(h(n))$

**$a \leq b$  eşitsizliğine benzer bir şekilde;**

- If  $f_1(n) \in O(g_1(n))$  and  $f_2(n) \in O(g_2(n))$ , then

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

## $\Omega$ Formal Tanımı

**Tanım:**  $f(n) \in \Omega(g(n))$  ise,  $f(n)$  fonksiyonunun büyüme derecesi,  $g(n)$ 'in sabit bir sayı ile çarpımının büyüme derecesinden büyük veya eşittir.

$$f(n) \geq c g(n), \forall n \geq n_0$$

Eşitsizliğini sağlayan pozitif bir sabit  $c$  ve pozitif bir tamsayı  $n_0$  vardır.

### Örnekler:

$$T(n) = 3n + 5 = \Omega(n) \text{ 'dir.} \quad (3n + 5 \in O(n))$$

$$\begin{aligned} f(n) &\geq c g(n) \\ 3n + 5 &\geq c * n \end{aligned}$$

$$n \geq 1 \text{ için } 3n + 5 \geq c * n \quad (c = 3, n_0 = 1)$$

$$T(n) = n = \Omega(\lg n) \text{ 'dir.} \quad (n \in \Omega(\lg n))$$

$$\begin{aligned} f(n) &\geq c g(n) \\ n &\geq c * \lg n \end{aligned} \quad (\log_2 n = \lg n)$$

$$n \geq 2 \text{ için } n \geq c * \log_2 n \quad (c = 1, n_0 = 2)$$

$$T(n) = \sqrt{n} = \Omega(\lg n) \text{ 'dir.} \quad (\sqrt{n} \in \Omega(\lg n))$$

$$\begin{aligned} f(n) &\geq c g(n) \\ \sqrt{n} &\geq c * \lg n \rightarrow n^{\frac{1}{2}} \geq c * \lg n \end{aligned} \quad (\log_2 n = \lg n)$$

$$\sqrt{n} \geq 2 \text{ için } \sqrt{n} \geq c * \lg n \quad (c = 1, n_0 = 2)$$

$$T(n) = n^3 = \Omega(n^2) \text{ 'dir.} \quad (n^3 \in \Omega(n^2))$$

$$\begin{aligned} f(n) &\geq c g(n) \\ n^3 &\geq c * n^2 \end{aligned}$$

$$n^3 \geq 1 \text{ için } n^3 \geq c * n^2 \quad (c = 1, n_0 = 1)$$

## Ⓜ Formal Tanımı

**Tanım:**  $f(n) \in \Theta(g(n))$  ise,  $f(n)$  fonksiyonunun büyüme derecesi,  $g(n)$  fonksiyonunun bir sabit katından yüksek aynı zamanda  $g(n)$  fonksiyonunun bir sabit katından da düşük olmaktadır.

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \quad \forall n \geq n_0$$

Eşitsizliğini sağlayan pozitif sabit  $c_1, c_2$  sayıları ve pozitif bir tamsayı  $n_0$  vardır.

$$f(n) = 5n^2 - 3n \in \Theta(n^2)$$

$$4n^2 < 5n^2 - 3n < 5n^2, \text{ tüm } n \geq 4$$

## Recursive olmayan Algoritmaların Time Efficiency'sini Analiz Etmek için Genel Plan

- Girdi boyutu için düşünülen parametreye karar verme.
- Algoritmanın temel işlemini (basic operation) belirle.
- En kötü(worst), ortalama(average), ve en iyi(best) durumlarına karar ver.
- Toplamda kaç farklı temel işlemin çalıştırıldığını say (toplam formülünü belirle).
- Toplamı kurallara göre basite indirge.

$$f(n) = 2n + 5 \in \Theta(n)$$

$$2n \leq 2n + 5 \leq 3n, \text{ tüm } n \geq 5 \text{ için.}$$

**ALGORITHM** *MaxElement*( $A[0..n-1]$ )

//Determines the value of the largest element in a given array

//Input: An array  $A[0..n-1]$  of real numbers

//Output: The value of the largest element in  $A$

$maxval = A[0]$

for  $i = 1$  to  $n - 1$  do

    if  $A[i] > maxval$

$maxval = A[i];$

return  $maxval;$

### ÇÖZÜM

- $c(n) = \sum_1^{n-1} 1 = n - 1 - 1 + 1 = n - 1 \in \theta(n)$

**ALGORITHM** Unique Elements ( $A[0..n-1]$ )

//Determines whether all the elements in a given array are distinct  
 //Input: An array  $A[0..n-1]$  //Output: Returns "true" if all the elements in  $A$  are distinct

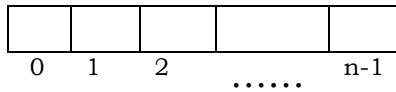
// and "false" otherwise

for  $i = 0$  to  $n - 2$  do;


    for  $j = i + 1$  to  $n - 1$  do


        if  $A[i] = A[j]$  return false;

    return true;

**ÇÖZÜM:**

Input: An Array  $A[0..n-1]$

$i=0$    $n-2$ 'ye kadar gidicek.

$j=1$    $n-1$ 'e kadar gidicek.

- $i=0$  ikeb  $j$ 'deki bütün değerleri gezer,  $j=n-1$  olduğunda döngüden çıkar ve  $i+1$  değerini alır.
- $i=1$  iken  $j=2,3,4 \dots n-1$ 'e kadar giderek döngüyü tamamlar.
- Her döngüde  $A[i] == A[j]$  sorgusu yapılır.

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1) - (i+1) + 1 = \sum_{i=0}^{n-2} n-1-i$$

$$\sum_{i=0}^{n-2} n-1 - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i \quad \sum_{i=0}^{n-2} 1 = n-2+1 = n-1$$

$$\sum_{i=0}^{n-2} i = \frac{(n-2)(n-1)}{2}$$

$$(n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$= \frac{1}{2}n^2 - \frac{1}{2}n \in \theta(n^2)$$

$$\begin{array}{ccc}
 & A & B \\
 \text{row } i & \left[ \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline \end{array} \right] & * \begin{array}{|c|} \hline \\ \hline \\ \hline \\ \hline \\ \hline \end{array} & = \begin{array}{|c|} \hline \\ \hline \\ \hline \\ \hline \\ \hline \end{array} \\
 & & \text{col. } j & \\
 & & & C \\
 & & & \left[ \begin{array}{|c|} \hline C[i,j] \\ \hline \end{array} \right]
 \end{array}$$

**Example 3: Matrix multiplication**

**ALGORITHM** *MatrixMultiplication*( $A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]$ )

//Multiplies two n-by-n matrices by the definition-based algorithm

//Input: Two n-by-n matrices  $A$  and  $B$

//Output: *Matrix*  $C = A B$

for  $i = 0$  to  $n - 1$  do

    for  $j = 0$  to  $n - 1$  do

$C[i, j] = 0.0;$

        for  $k = 0$  to  $n - 1$  do

$C[i, j] = C[i, j] + A[i, k] * B[k, j];$

    return  $C;$

**ÇÖZÜM:**

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

**Algorithm** *GaussianElimination*( $A[0..n-1, 0..n]$ )

//Implements Gaussian elimination of an  $n$ -by- $(n+1)$  matrix  $A$

for  $i = 0$  to  $n-2$  do

    for  $j = i+1$  to  $n-1$  do

        for  $k = i$  to  $n$  do

$A[j,k] = A[j,k] - A[i,k] * A[j,i] / A[i,i]$

Find the efficiency class and a constant factor improvement

**Gaussian-Jordan**

For Each row  $i$  ( $R_i$ ) from 1 to  $n$  ( $\sum_{i=1}^n$ )

    If any row  $j$  below row  $i$  has non zero entries to the right of the first non zero entry in row  $i$

$R_i \leftrightarrow R_j$

$R_i \rightarrow \frac{1}{c}R_i$  where  $c$  = the first non-zero entry of row  $i$  (the pivot).

    For each row  $j > i$  ( $n - i$ )

$R_j \rightarrow R_j - dR_i$  where  $d$  = the entry in row  $j$  which is directly below the pivot in row  $i$ .

    If any 0 rows have appeared exchange them to the bottom of the matrix.

next  $i$  [Matrix is now in REF]

For each non zero row  $i$  ( $R_i$ ) from  $n$  to 1 ( $\sum_{i=1}^n$ )

    For each  $j < i$  ( $n - i$ )

$R_j \rightarrow R_j - bR_i$  where  $b$  = the value in row  $j$  directly above the pivot in row  $i$ .

So a rough calculation counting the number of multiplications yields

$$\begin{aligned} & \sum_{i=1}^n [(n+1) + (n-i)(n+1)] + \sum_{i=1}^n [(n-i)(n+1)] \\ &= \sum_{i=1}^n \sum_{j=1}^n (2n-2i+1)(n+1) \\ &= \sum_{i=1}^n 2n^2 + 3n + 1 - 2(n+1)i \\ &= 2n^3 + 3n^2 + n + n(n+1)^2 \\ &= 3n^3 + 5n^2 + 2n \end{aligned}$$

A similar rough calculation for the number of additions yields  $\sum_{i=1}^n 2(n-i)(n+1) = n^3 - n$

This is only a rough calculation. If we are considering the behaviour of the algorithm for large  $n$ , the highest term will dominate. We can say that the Gauss-Jordan algorithm has order  $n^3$  and complexity  $O(n^3)$ .

## Recurrence Relations(Özyineleme İlişkileri)

- **Exact Solution (Kesin Çözüm)**
  - Forward substitution (İleriye Doğru yerine koyma)
  - Backward substitution (Geriye Doğru Yerine Koyma)
  - Diferansiyel denklemlerde kullanılan yöntemler
- **Asymptotic Solution (Asimptotik Çözüm)**
  - Master theorem

### Recursive algoritmalarda Genel Plan

- Girdi boyutunu belirleyecek parametrelere karar verme.
- Algoritmanın temel işlemini (basic operation) belirle.
- Her recursive çağırmada temel işlem kaç defa çalıştırılıyor belirle.
  - Eğer çalıştırma sayısı aynı boyuttaki farklı girdiler de değişiyorsa;
    - ❖ worst-case, average-case ve best-case durumları ayrı ayrı incelenmelidir.
  - Temel işlem çağrılmasını sayan bir özyineleme ilişkisi oluşturun.
  - Özyinelemeli çağrımları ve bu çağrımlardaki girdi boyutunu belirleyin.
  - Uygun bir initial condition bulun.
- Özyinelemeyi çözün.
  - En azından girdi büyüdükçe işlem adımlarındaki büyümeyi belirleyin.



**Forward Substitution (İleriye Doğru yerine koyma):**

- Initial condition ile başlayın.
- Çözümüne doğru bir kaç terimi hesaplayın.
  - Önceden bulduğunuz terimleri kullanarak.
- Bir desen bulmaya çalışın.
- Kapalı formda bir formül bulmaya çalışın.
- Bulduğunuz Formülü doğrulayın.
  - Tümevarım.
  - Yerine koyma.

**Backward substitution (Geriye Doğru Yerine Koyma):**

- Recurrence denklemi ile başlayın ( $f(n)$  ile)
- $f(n - 1)$  terimini açın
  - Recurrence denklemini kullanarak
- Aynı işlemi birkaç kere tekrarlayın.
  - $for f(n - 2), f(n - 3)$  etc..
- $f(n)$  denkleminin formülünün  $f(n-i)$  cinsinden bulmaya çalışın.
- Desenin doğruluğunu kontrol edin
  - Genellikle tümevarım ile

$n - i$ 'yi initial condition yapan  $i$ 'yi seçin ve kapalı formülü elde ediniz.

•Genellikler özyineleme bulunan fonksiyonlarda görülür.

### Örnek 1 – Faktöriyel Hesaplama

**Hedef:** Herhangi bir doğal tamsayı  $n$  için  $F(n) = n!$  faktöriyel fonksiyonunu hesapla.

$$n! = n * (n - 1) * \dots * 1 = n * (n - 1)! \text{ for } n \geq 1 \text{ and } 0! = 1$$

$$F(n) = F(n - 1) * n \text{ for } n > 0 \text{ } f(0) = 1$$

**Algorithm:**

```
if n = 0 return 1
else return F(n - 1) * n
```

**Analysis:**

$$M(n) = M(n - 1) + 1 \text{ for } n > 0,$$

$$M(0) = 0.$$

**Forward Substitution Tekniği:**

$$M(0) = 0$$

$$M(1) = M(0) + 1 = 1$$

$$M(2) = M(1) + 1 = 2$$

$$M(3) = M(2) + 1 = 3$$

$\vdots$

$$M(n) = n$$

**Backward substitution Tekniği:**

$$M(n) = M(n - 1) + 1$$

$$M(n) = M(n - 2) + 1 + 1$$

$$M(n) = M(n - 3) + 1 + 1 + 1$$

$\vdots$

$$M(n) = M(n - i) + i$$

$$\text{for } n = i$$

$$M(n) = M(0) + n = n$$

## Örnek 2- Towers of Hanoi



**Hedef:** A'çubuğunda bulunan  $n$  adet diski B çubuğu yardımıyla kurallara uyarak C çubuğuna taşımak.

### Algorithm:

- $n - 1$  tane diski A 'dan B'ye C çubuğunu kullanarak aktarın
- En büyük diski A'dan C'ye iletin.
- $n - 1$  tane diski B çubuğundan C'ye A çubuğunu kullanarak iletin.

### Toplam hareket sayısı

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

$$T(1) = 1$$

### Backward substitution Tekniği:

$$\diamond T(n) = 2T(n - 1) + 1$$

$$T(1) = 1 \text{ ise}$$

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

$$T(n - 2) = 2.T(n - 3) + 1$$

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

$$= 4.(2T(n - 3) + 1) + 2 + 1 = 8.T(n - 3) + 4 + 2 + 1$$

$$= 2^i.T(n - i) + \underbrace{1 + 2 + 2^2 \dots 2^{i-1}}_{\sum_{k=0}^{i-1} 2^k = 2^i - 1}$$

$$= 2^i.T(n - i) + 2^i - 1 \quad n-i = 1$$

$$= 2^{n-1}.T(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \in \theta(2^n)$$

## Örnek 2: Counting #bits

**Hedef:** Given a non negative integer number num. For every numbers  $i$  in the range  $0 \leq i \leq \text{num}$  calculate the number of 1's in their binary representation and return them as an array.

**Algorithm:**  $\text{BinRec}(n)$

//Input: A positive decimal integer  $n$

//Output: The number of binary digits in  $n$ 's binary representation

if  $n = 1$  return 1;

else return  $\text{BinRec}([n/2] + 1)$ ;

**Backward substitution Tekniği:**

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$= T\left(\frac{n}{4}\right) + 1 + 1$$

$$= T\left(\frac{n}{8}\right) + 1 + 1 + 1$$

$\vdots$

$$= T\left(\frac{n}{2^i}\right) + i$$

$$= T(1) + \log_2 n$$

$$= 0 + \log_2 n = \log_2 n$$

$$T(n) = \log_2 n \in \theta(\log n)$$

$$T(1) = 0$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1$$

$$\frac{n}{2^i} = 1, 2^i = n$$

$$\log_2 2^i = \log_2 n$$

$$i = \log_2 n$$

**Örnek 3:**  $2T(\sqrt{n}) + 1$

$$T(n) = 2T(n^{\frac{1}{2}}) + 1$$

$$= 2 \cdot \left( 2T(n^{\frac{1}{4}}) + 1 \right) + 1 = 4T(n^{\frac{1}{4}}) + 2 + 1$$

$$= 4 \cdot \left( 2T(n^{\frac{1}{8}}) + 1 \right) + 2 + 1 = 8T(n^{\frac{1}{8}}) + 4 + 2 + 1$$

$\vdots$

$$= 2^i \cdot T(n^{\frac{1}{2^i}}) + 2^i - 1 \longrightarrow$$

$$\begin{aligned} n^{\frac{1}{2^i}} = 2 & \implies \log_n n^{\frac{1}{2^i}} = \log_n 2 \\ \frac{1}{2^i} = \log_n 2 & \implies 2^i = \frac{1}{\log_n 2} \\ \log_2 2^i = \log_2 \log_2 n & \implies i = \log_2 \log_2 n \end{aligned}$$

$$= \log_2 n \cdot T(2) + \log_2 n - 1 \longrightarrow T(2) = 0$$

$$= 0 + \log_2 n - 1 = \log_2 n - 1 \in \theta(\log_2 n)$$

$$T(2) = 0$$

$$T(n^{\frac{1}{2}}) = 2(n^{\frac{1}{4}}) + 1$$

## Fibonacci numbers

**The First 20 Fibonacci numbers are:**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181.

**The Fibonacci recurrence:**

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

- Genel olarak  $2^n$  fonksiyon zorluğuna sahip fonksiyonunun homojen lineer yenileme katsayıları aşağıda ki gibidir:

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

**Fibonacci sayılarını oluşturma fonksiyonu:**

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \phi^{-n})$$

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803 \qquad \phi^{-} = -\frac{1}{\phi} \approx -0.61803$$

**Algorithm:** $F(n)$

//Computes the n'th Fibonacci number revursively by using its definition

//Input: A nonnegative integer n

//Output: The n'th Fibonacci number

*if*  $n \leq 1$  *return*  $n$

*else return*  $F(n-1) + F(n-2)$

## Algoritmalar QUIZ Soru-Çözümler

1. Aşağıdaki fonksiyonların dahil olduğu  $\theta(g(n))$  sınıfını belirtiniz. (Ola-  
bilecek en basit  $g(n)$ 'i kullanın.

- a)  $(n^2 + 1)^{10} \in \theta(n^2)$
- b)  $\sqrt{10n^2 + 7n + 3} \in \theta(n)$
- c)  $2n \log(n + 2)^2 + (n + 2)^2 \log \frac{n}{2} \in \theta(n^2 \cdot \log n)$
- d)  $2^{n+1} + 3^{n-1} \in \theta(3^n)$
- e)  $\log_2 n \in \theta(\log n)$

2. Aşağıdaki algoritma için

### Algoritma 1: Enigma

Function ENIGMA( $A[0..n-1][0..n-1]$ )

for  $i \leftarrow 0$  to  $n-2$  do

for  $j \leftarrow i+1$  to  $n-1$  do

if  $A[i, j] \neq A[j, i]$  then

return false;

return false;

• Giriş olarak  $n \times n$  boyutlu A  
matrisi veriliyor.

a. Bu algoritma neyi hesaplamaktadır?

- Matrisin simetrik olup olmadığını kontrol ediyor

b. Algoritmanın temel işlemi nedir?

- $A[i, j] \neq A[j, i]$ , karşılaştırma.

c. Temel işlem kaç defa tekrar etmektedir?

- $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1) - (i+1) + 1 = \sum_{i=0}^{n-2} n - 1 - i$
- 
- $\sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \cdot (n-1) - \frac{(n-2) \cdot (n-1)}{2} = \frac{n^2 - n}{2}$

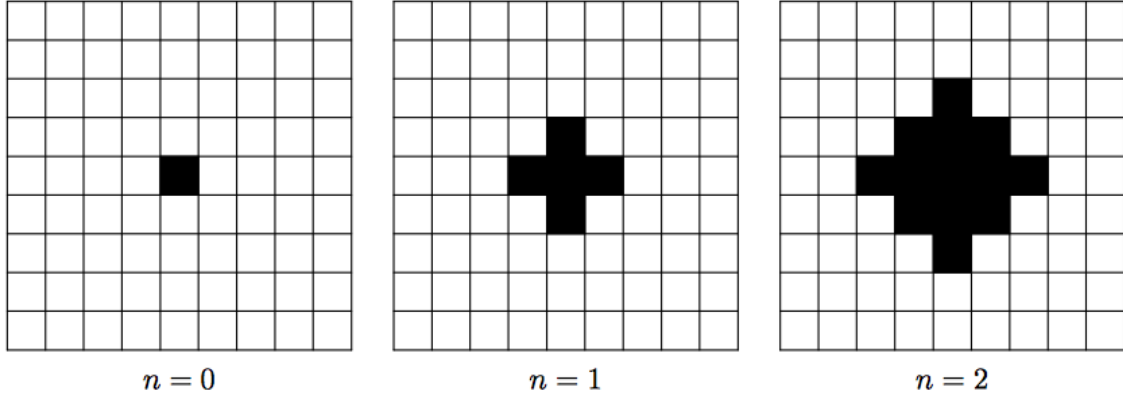
d. Bu algoritmanın verimlilik sınıfı nedir?

- $\frac{n^2 - n}{2} \in \theta(n^2)$

e. Bu iş için bir iyileştirme veya daha iyi bir algortima öneriniz ve verimlilik sınıfını belirtiniz. Eğer önerilemezse neden önerilemeyeceğini ispat etmeye çalışınız.

- Algoritma en verimli haldedir.

3. Tek bir kare ile başlayan ve  $n$  adım boyunca her bir karenin dışarıda kalan kısımlarına birer kare ekleyen algoritmayı düşünün (Von Neumann komşuluğu).  $n$  adım sonunda toplam kaç kare olduğunu hesaplayacak bir yineleme ilişkisi (recurrence relation) kurun, bu ilişkiyi çözün ve algoritmanın verimlilik sınıfını belirtin.



1,5,13,25,41,61.. aralarındaki artış 4,8,12,16,20.. “ $4n$ ” şeklindedir.

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

$$T(0) = 1$$

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

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

$$= T(n - 3) + 4(n - 2) + 4(n - 1) + 4n$$

$= :$

$$= T(n - i) + 4((n - i) + (n - i + 1) \dots (n - 1) + (n))$$

$$= T(n - i) + 4 \sum_{k=0}^i (n - k) = T(n - i) + 4 \left( \sum_{k=0}^i n - \sum_{k=0}^i k \right)$$

$$= T(n - i) + 4n(i + 1) - 4 \left( \frac{i(i + 1)}{2} \right) = T(n - i) + 4n(i + 1) - 2i(i + 1)$$

$t(0) = 1$  için  $i = n$  ise;

$$= T(0) + 4n(n + 1) - 2n(n + 1) = 1 + 2n^2 + 2n$$

$$2n^2 + 2n + 1 \in \theta(n^2)$$



## BRUTE FORCE

*Brute Force* Algoritmaları tam olarak göründükleri gibidir - verimliliği artırmak için gelişmiş teknikler yerine tam bilgi işlem gücüne dayanan ve her olasılığı deneyen bir sorunu çözenin basit yöntemleri. Çok çeşitli problemlere uygulanabilir. Bir problemi çözmek için en kolay yollardan biri. *Brute force* düz mantık bir yaklaşım temelinde, problem bildirimi bulunmaktadır.

---

Çok düşünmeye gerek yok

Sadece sonuca odaklan- JUST do It!

---

- Brute force, olası her rota için toplam mesafeyi hesaplamak ve ardından en kısa olanı seçmektir. Bu, verimli değildir çünkü akıllı algoritmalar aracılığıyla birçok olası rotayı ortadan kaldırmaya neden olur.

**Örnek:** Her biri 0-9 arasında 4 basamaklı küçük bir asma kilidiniz olduğunu hayal edin. Şifreni unuttun ama başka bir asma kilit almak istemiyorsun. Rakamların hiçbirini hatırlayamadığınız için kilidi açmak için kaba kuvvet yöntemini kullanmanız gerekir.

- Böylece tüm sayıları tekrar 0'a ayarlayın ve tek tek deneyin.
- 0001, 0002, 0003, vb. açılana kadar.
- En kötü senaryoda, kombinasyonunuzu bulmak için 104 veya 10.000 deneme gerekir.

**Çeşitler:** Birçok temel ama önemli algoritmik rutinlerde kullanılır.

- $n$  sayıyı toplama
- Bir listedeki en büyük sayıyı bulmak
- Sorting
- Searching
- String matching
- Compression algorithms.

**Sıralama(Sorting) Problemi:** Verilen  $n$  elemanı azalmayan (non-decreasing) bir sıraya yerleştirme. Sıralama algoritmaları, ihtiyacınıza bağlı olarak çeşitli alanlarda kullanılabilir. Bazıları, çok yaygın olarak kullanılanlar:

### Selection Sort

#### Yaklaşım:

- Tüm listeyi en küçük elemanını bulana kadar tara
- İlk eleman ile swap et.
- Listenin ikinci elemanından başlayarak  $n-1$  elemanlı listedeki en küçük elemanı bul.
- İkinci eleman ile ( $n-1$  elemanlı) listenin en küçük elemanını swap et. ..

**Örnek:**

89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89
17	29	34	90	45	68	89
17	29	34	45	90	68	89
17	29	34	45	68	90	89
17	29	34	45	68	89	90

- Algoritmanın  $i$ 'nci dış iterasyonunda
- Son  $n - i$  eleman içerisinde  $A_i$  ile yer değiştirecek olanı arıyoruz.

**Algorithm:**

// The algorithm sorts a given array by selection sort

// Input : An array  $A[0 \dots n - 1]$  of orderable elements

// Output : Array  $A[0 \dots n - 1]$  sorted in ascending order

SelectionSort ( $A[0 \dots n - 1]$ )

for  $i \leftarrow 0$  to  $n - 2$  do

$min \leftarrow i$

    for  $j \leftarrow i + 1$  to  $n - 1$  do

        if  $A[j] < A[min]$   $min \leftarrow j$

    swap  $A[i]$  and  $A[min]$

**Analysis:** Toplamda gerçekleştirilen karşılaştırma sayısı

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$C(n) = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$C(n) = \sum_{i=0}^{n-2} (n-1-i)$$

$$C(n) = \frac{n \cdot (n-1)}{2} = \theta(n^2)$$

## Bubble Sort

### Yaklaşım:

- Listedeki komşu elemanları karşılaştır.
- Eğer sıralı değilse yerini swap et.

**Örnek:** The first two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17

89	↔	45	↔	68	↔	90	↔	29	↔	34	↔	17
45		89	↔	68	↔	90	↔	29	↔	34	↔	17
45		68		89	↔	90	↔	29	↔	34	↔	17
45		68		89		29	↔	90	↔	34	↔	17
45		68		89		29		34	↔	90	↔	17
45		68		89		29		34		17	↔	90
45	↔	68	↔	89	↔	29		34		17		90
45		68		29	↔	89	↔	34		17		90
45		68		29		34	↔	89	↔	17		90
45		68		29		34		17	↔	89		90
etc.												

### Algorithm:

// The algorithm sorts a given array by bubble sort

// Input : An array  $A[0 \dots n - 1]$  of orderable elements

// Output : Array  $A[0 \dots n - 1]$  sorted in ascending order

BubbleSort (  $A[0 \dots n - 1]$  )

for  $i \leftarrow 0$  to  $n - 2$  do

    for  $j \leftarrow 0$  to  $n - 2 - i$

        if  $A[j + 1] < A[j]$  swap  $A[j]$  and  $A[j + 1]$

**Analysis:** Toplam karşılaştırma.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

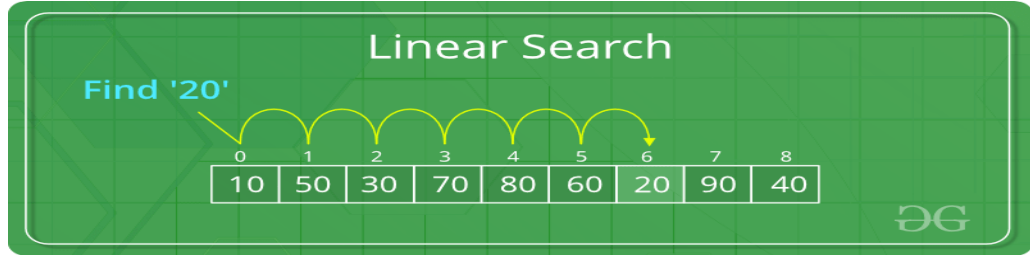
$$C(n) = \sum_{i=0}^{n-2} [(n - 2 - i) - 0 + 1]$$

$$C(n) = \sum_{i=0}^{n-2} (n - 1 - i) = \frac{n(n-1)}{2} \in \theta(n^2)$$

**Search (Arama) Problemi:** Arama Problem, n elemanlı bir listede belirli bir ögeyi (bir key K) bulmaktır.

- K ile bir eşleşme listede bulunur.
- Ya da K listede yoktur

**Sequential Search:** Bir uçtan başladığımız ve istenen öge bulunana kadar listenin her ögesini kontrol ettiğimiz sıralı bir arama algoritmasıdır. Basit bir arama algoritması vardır.



**Yaklaşım:**

- Aranan key ile tüm elemanları sırasıyla karşılaştır.
- Key bulunduğunda çık
- Eğer key listenin hiçbir elemanında bulunmadıysa çık.

**Algorithm :**

*// The algorithm implements sequential search with a search key as a sentinel  
// Input : An array of n elements and a search key  
// Output : The position of the first element in array A[0 .. n - 1] whose  
value is equal to K or - 1 if no such element is found*

*SequentialSearch(A[0..n], K)*

*A[n]  $\leftarrow$  K*

*i  $\leftarrow$  0*

*while A[i]  $\neq$  K do*

*i  $\leftarrow$  i + 1*

*if i < n return i*

*else return - 1*

**Analysis :**

$T(n) = \theta(n)$

## Closest - Pair Problem

$n$  noktanın içerisinde birbirine en yakın olanları bulacağız.

- Problemin iki boyutlu halini düşünelim.
- Problemdeki noktaların Kartezyen koordinatlarının  $(x, y)$  verildiğini düşünelim.
- $P_i = (x_i, y_i)$  ve  $P_j = (x_j, y_j)$  noktaları arasındaki uzaklık standart *Euclidean* uzaklığı ile hesaplanır

$$D(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

“Öklit uzaklığı”

### Yaklaşım:

- Her nokta çifti arasındaki euclidean distance'ını hesapla.
- En küçük uzaklığa sahip olan noktayı bul.

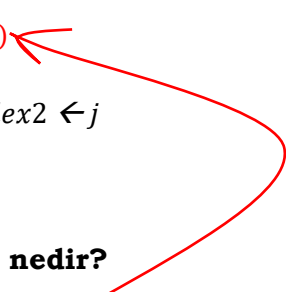
“Aynı nokta çifti için uzaklığı tekrar hesaplamak istemiyoruz.  
Sadece  $(P_i, P_j)$  noktalarını göz önünde bulunduracağız  $i < j$ ”

### Algoritma :

// Input : A list  $P$  of  $(n \geq 2)$  points “ $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$ ”

// Output : Indicates index1 and index2 of the closest pair of points

```
dmin ← ∞
for i ← 1 to n - 1 do
  for j ← i + 1 to n do
     $d \leftarrow \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2)$ 
    if  $d < dmin$ 
       $dmin \leftarrow d$ ;  $index1 \leftarrow i$ ;  $index2 \leftarrow j$ 
return index1, index2
```



### Analiz: Temel işlem (basic operation) nedir?

- Temel işlem noktalar arasındaki **Euclidean** uzaklığını bulmaktır.
- Kare kök algoritması toplama ve çarpma işlemi gibi basit bir işlem değildir.
- Kare kökü almayı yok sayabiliriz.  $(x_i - x_j)^2 + (y_i - y_j)^2$
- Değeri küçük olanın da kare kökü küçük olur.
- Kare kök fonksiyonu **strictly increasing**'dir

### Analiz: Toplam çarpma işlemi

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

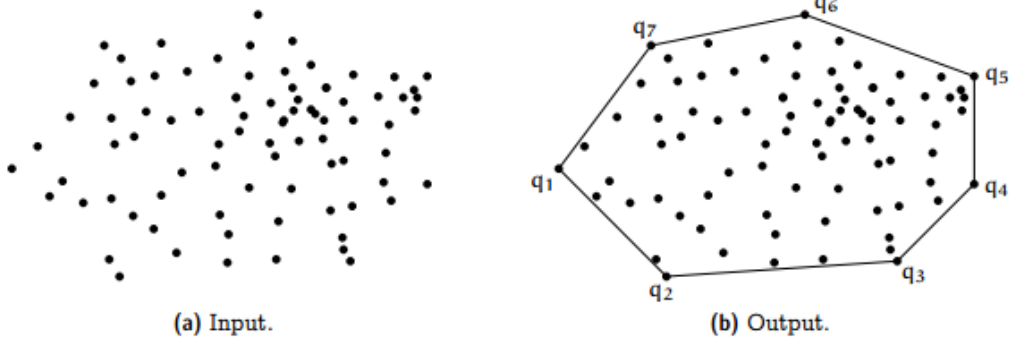
$$C(n) = 2 \sum_{i=1}^{n-1} (n - i)$$

$$C(n) = 2[(n - 1) + (n - 2) + \dots + 1]$$

$$C(n) = (n - 1)n \in \theta(n^2)$$

## Convex-Hull

Çokgenin dışbükey gövdesi, çokgenimizi saran minimum dışbükey kümedir.



### Teorem:

- Herhangi bir S noktalar kümesinin convex hull'ı köşe noktaları bu S'in içerisindeki noktalardan oluşan bir convex çokgendir (polygon).
- Tüm noktalar aynı doğrudaysa S bir doğru parçasıdır.

### Amaç:

- Convex hull problemi verilen S kümesindeki n nokta için bir konveks kabuk oluşturma problemidir.
- Çokgenin köşelerindeki noktaları belirlememiz gerekmektedir “*extreme points*” olarak adlandırılır.
- *Convex kümesinin uç noktası bu küme içerisindeki herhangi bir doğrunun ortasında bulunmayan noktadır.* Örn üçgen için bu uç noktalar köşelerdir.

### Bu problem brute force olarak nasıl çözülebilir?

#### Fikir:

- $P_i$  ve  $P_j$  birleştiren doğru parçası diğer tüm noktaları bir tarafta bırakıyorsa bu doğru parçası convex hull'ın bir kenarıdır.
- Bu testi her düğüm çifti için yaptığınızda convex hull'ın uç noktaları bulunabilmektedir.

### Diğer noktaların doğrunun bir tarafında kaldığını nasıl kontrol edebiliriz?

$(x_1, y_1)$  ve  $(x_2, y_2)$  noktaları arasındaki doğru denklemi:

$$ax + by = c \quad a = y_2 - y_1, \quad b = x_1 - x_2, \quad c = x_1y_2 - y_1x_2$$

- Bu doğru düzlemi aşağıdaki ki ikiye böler
  - Doğrunun bir tarafı  $\rightarrow ax + by > c$
  - Doğrunun diğer tarafı  $\rightarrow ax + by < c$
- Noktaların doğrunun hep bir tarafında kaldığını kontrol edebilmek için bu noktaların hepsinin bir eşizliği sağladığı kontrol edilmelidir.

**Analiz:**

- Tüm  $n(n-1)/2$  düğüm çifti için.
  - $ax + by - c$  nin işareti diğer  $n-2$  nokta için kontrol edilir
  - $T(n) = O(n^3)$
- Bu problem için de daha performanslı çalışan algoritmalar bulunmaktadır.

Ayrıca bkz:

- <https://www.geeksforgeeks.org/convex-hull-set-1-jarvis-algorithm-or-wrapping/>
- <https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/>
- <https://www.youtube.com/watch?v=B2AJoSzf4M&t=268s>

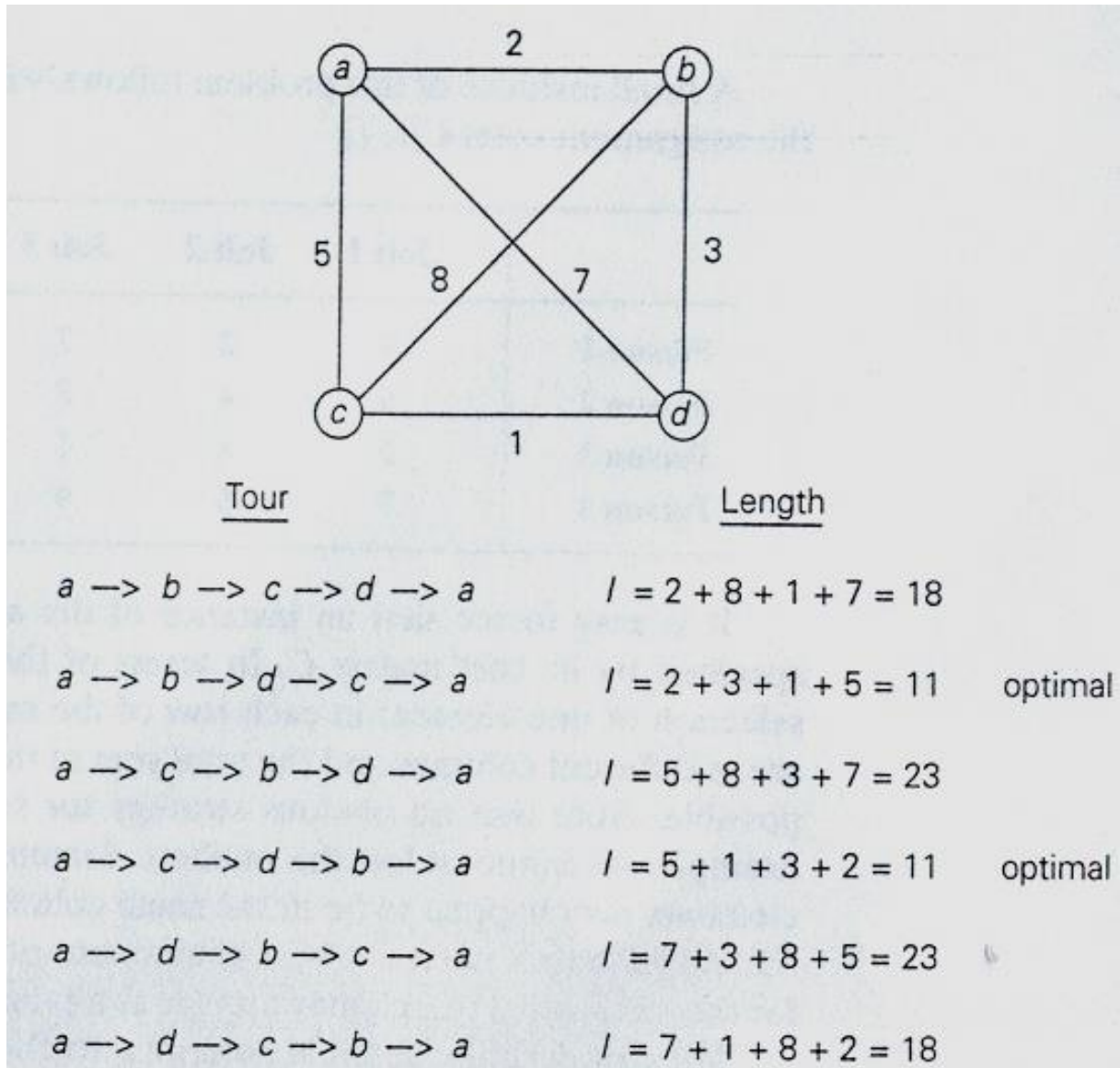
**Exhaustive Search (Kapsamlı Arama):** Kombinasyonel problemlere uygulanan basit bir brute force yaklaşımıdır.

- Problem domain'indeki tüm elemanların oluşturulmasını,
- Problemin kısıtlarını kapsayan elemanların seçilmesini,
- İstenen elemanların bulunmasını
  - Amaç fonksiyonunu optimize eden elemanın bulunmasını gerektirir.

**Traveling Salesman Problemi**

Verilen  $n$  şehir içerisinde minimum turun bulunması öyle ki başlangıç şehrine varmadan tüm şehirlere sadece bir kere uğranması gerekmektedir. Her şehir çifti arasında bir dizi şehir ve mesafe göz önünde bulundurulduğunda, sorun her şehri tam olarak bir kez ziyaret edip başlangıç noktasına dönen mümkün olan en kısa rotayı bulmaktır.

- Problem weighted bir graph ile modellenabilir.
- Vertices (Köşeler) şehirleri temsil eder.
- Kenar ağırlıkları uzaklıkları temsil eder.
- **Exhaustive search** tekniği problemi çözmek için uygulanabilir.



Şekil 1 Seyahat eden satış elemanı sorununun küçük bir örneğine kapsamlı arama ile çözüm

- Kaç yol vardır ?  $(n-1)!$  !
- Yukarda da gözüktüğü gibi yolların yarısı yalnızca yön anlamında farklıdır, ağırlık anlamında elimizde  $\frac{(n-1)!}{2}$  ihtimal vardır.
- Bu problemin çözümünün büyüme hızı o halde, Büyüme hızı:  $\theta(n!)$

#### **Discussion :**

- 150 yıldır araştırmacılar tarafından araştırılmaktadır
- Basit formülasyonu,
- Önemli uygulama alanlarına sahip,
- Diğer kombinyonel problemlerle bağlantısı var.



### Knapsack Problemi:

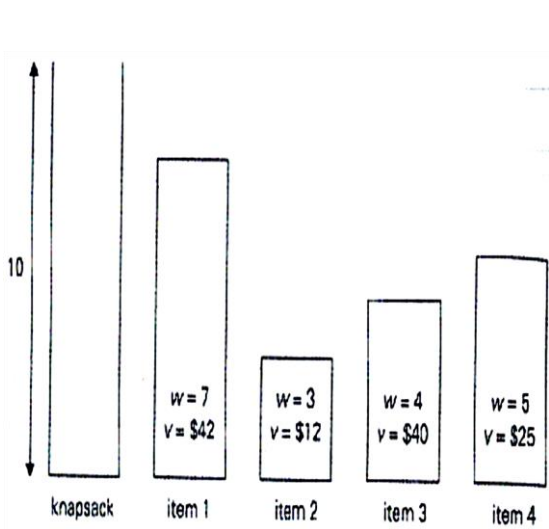
$n$  elemanın ağırlıkları  $w_1, \dots, w_n$  ve değerleri  $v_1, \dots, v_n$  biliniyor bir de  $W$  ağırlığını alabilen bir sırt çantanız var. Sırt çantanızın alabileceği en değerli alt kümeyi bulunuz

### Exhaustive Search Yaklaşımı:

- Verilen  $n$  eleman setinin tüm alt kümelerini düşünün
- Her bir alt kümenin toplam ağırlığını, uygun alt kümeleri tanımlamak için hesaplayın.
- Aralarında en büyük değeri sağlayan alt kümeyi bulun.

### Analysis:

- $n$ -elemanlı bir kümenin tüm alt kümeleri:  $2^n$
- Exhaustive search  $\mathcal{O}(2^n)$  algoritması gerektirmektedir.



Subset	Total weight	Total value
$\emptyset$	0	\$ 0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1, 2}	10	\$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	\$52
{2, 4}	8	\$37
{3, 4}	9	\$65
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Exhaustive Search yaklaşımına göre problemi çözmek istersek, kombinasyon olarak tüm alt grupları bulur, problem sınırimız olan kapasiteye en uygun değeri taşıyan alt grubu seçeriz.

- $n$  elemanlı bir kümenin tüm alt kümeleri  $2^n$  'dir. Problemin çözümünün büyüme hızı o halde,  
Büyüme hızı:  $\Omega(2^n)$

### Assignment Problem (atama problemi)

Bir diğer problem olan Assignment Probleminde n tane insan ve n tane iş olduğu söylenir, her çalışan yalnızca bir işe atabilir ve her işte yalnızca bir çalışan çalışabilir. Her çalışanın bir işte çalışması için belirli bir maliyet vardır. **En düşük maliyete sahip çalışan-iş eşleştirilmesinin** nasıl yapılması gerektiğini sorar.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Yukardaki tabloyu bir matrix olarak düşünürseniz, her satırda ve kolonda yalnızca bir seçim yapılarak en düşük maliyet hesabı yapılacaktır.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18	
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30	
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24	
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26	etc.
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33	
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23	

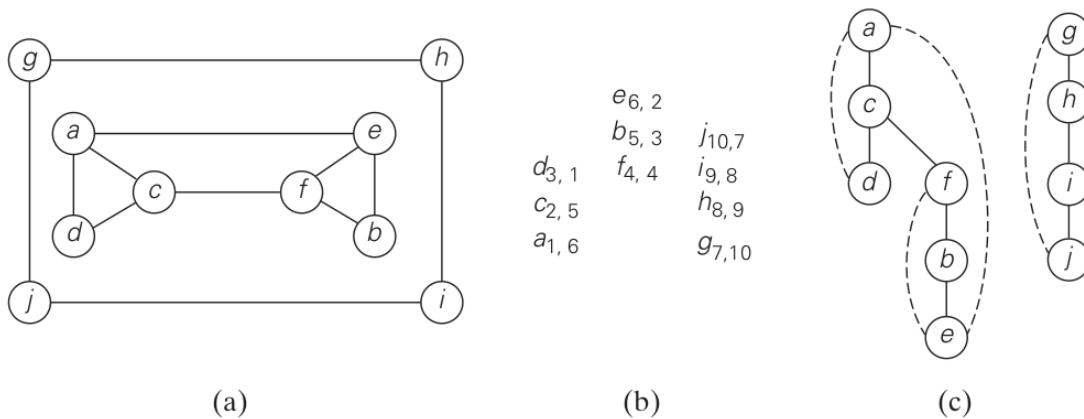
Exhaustive Search yaklaşımı ile problem çözülecek olursa olabilecek tüm kombinasyonel ifadeler bulunur ve en düşük maliyete sahip olan seçilir. Fakat bu yöntemden çok daha kullanışlı olan Hungarian method'unun kullanılması daha mantıklı olacaktır.

## Exhaustive Search ile Graph Traversal Algoritmaları (DFS - BFS)

Exhaustive Search dolaşma yöntemi olan DFS ve BFS'ye de uygulanabilir.

### Depth-first search (DFS)

Derine gidilir. Dolaşma işlemi bir düğümden başlar. Gidilebilecek mümkün komşulardan bir tanesini seçer ve ilerler. Gidilebilecek diğer düğümlere bakmadan önce yeni gidilen düğümün komşuları incelenir ve bir tanesine gidilir. Gidilebilecek komşu kalmayınca geri sarılarak mümkün komşular aranır.



**FIGURE 3.10** Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

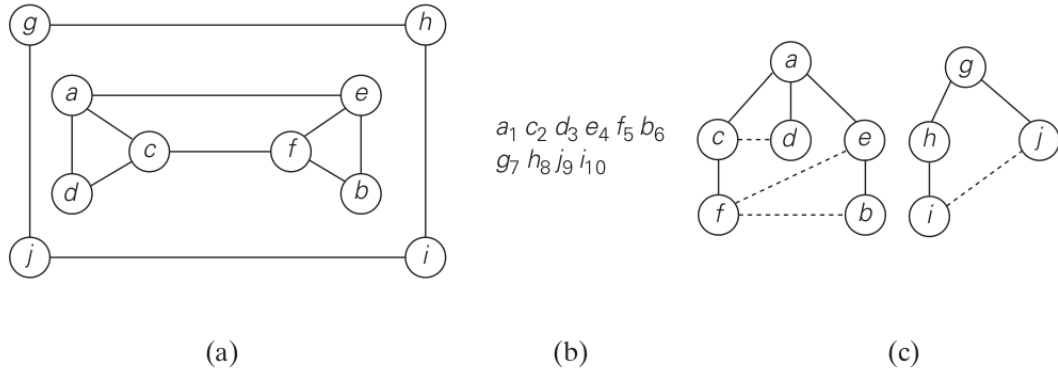
### Pseudocode of DFS

```
def depth_first_search(graph)
    mark each vertex in  $V$  with 0 as a mark of being 'unvisited'
    count = 0
    for each vertex  $v$  in  $V$  do
        if  $v$  is marked with 0
            dfs( $v$ )

def dfs( $v$ )
    count ++
    mark  $v$  with count
    for each vertex  $w$  in  $V$  adjacent to  $v$  do
        if  $w$  is marked with 0
            dfs( $w$ )
```

## Breadth-first search (BFS)

Enine gidilir. Dolaşma işlemi bir düğümden başlar. Gidilebilecek mümkün komşular hepsini sırayla seçer ve ilerler. Gidilebilecek bütün komşular gezildikten sonra ilk komşunun komşuları seçilir.



**FIGURE 3.11** Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

## Pseudocode of BFS:

```
def breadth_first_search(graph)
    mark each vertex in  $V$  with 0 as a mark of being 'unvisited'
    count = 0
    for each vertex  $v$  in  $V$  do
        if  $v$  is marked with 0
            bfs( $v$ )

def bfs( $v$ )
    count ++
    mark  $v$  with count
    initialize a queue with  $v$ 
    while the queue is not empty do
        for each vertex  $w$  in  $V$  adjacent to the front vertex do
            if  $w$  is marked with 0
                count ++
                add  $w$  to the queue
            remove the front vertex from the queue
```

### DFS ve BFS Hakkında Bilgiler

- BFS, DFS ile aynı verimliliğe sahiptir ve aşağıdaki şekilde temsil edilen çizge-lerde uygulanabilir:
  - adjacency Matrisi:  $\Theta(V^2)$
  - adjacency listesi:  $\Theta(|V| + |E|)$
- Vertexlerin queue'ye eklenmesi ve queue'den çıkma sırası bulunmaktadır.
- Uygulamaları: DFS ile aynı , ayrıca bir vertexten diğer vertexlere en kısa yol-ları da bulmaktadır.

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacent matrix	$\Theta( V ^2)$	$\Theta( V ^2)$
Efficiency for adjacent linked lists	$\Theta( V  +  E )$	$\Theta( V  +  E )$