BLM212 Veri Yapıları

Algorithm Efficiency

Big-O Notation

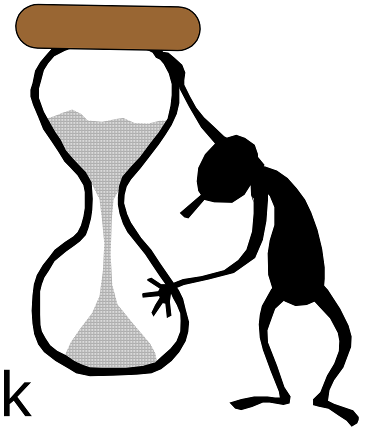
2021-2022 Güz Dönemi

The Time Complexity of an Algorithm

(Bir Algoritmanın Zaman Karmaşıklığı)

Koşma süresinin girişin boyutuna nasıl (ne şekilde) bağlı olduğunu belirtir.

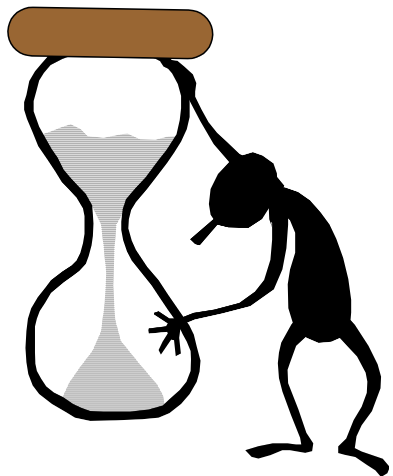
The Time Complexity of an Algorithm



Amaç:

- Bir programın ne kadar süreceğini tahmin etmek
- Makul bir şekilde programa verilebilecek en büyük giriş boyutunu tahmin etmek.
- Farklı algoritmaların verimini karşılaştırmak.
- Kodun en çok (en fazla sayıda) koşulan kısımlarına odaklanmaya yardımcı olmak için.
- Bir uygulamaya uygun olan algoritmayı seçmek için.

The Time Complexity of an Algorithm



Specifies how the running time depends on the size of the input.

(Koşma süresinin girişin boyutuna nasıl (ne şekilde) bağlı olduğunu belirtir.)

A function mapping

“size” of input

↓

“time” T(n) executed .

History of Classifying Problems

Impossible

Halting

Mathematicians' dream

Brute Force
(Infeasible)

Considered Feasible

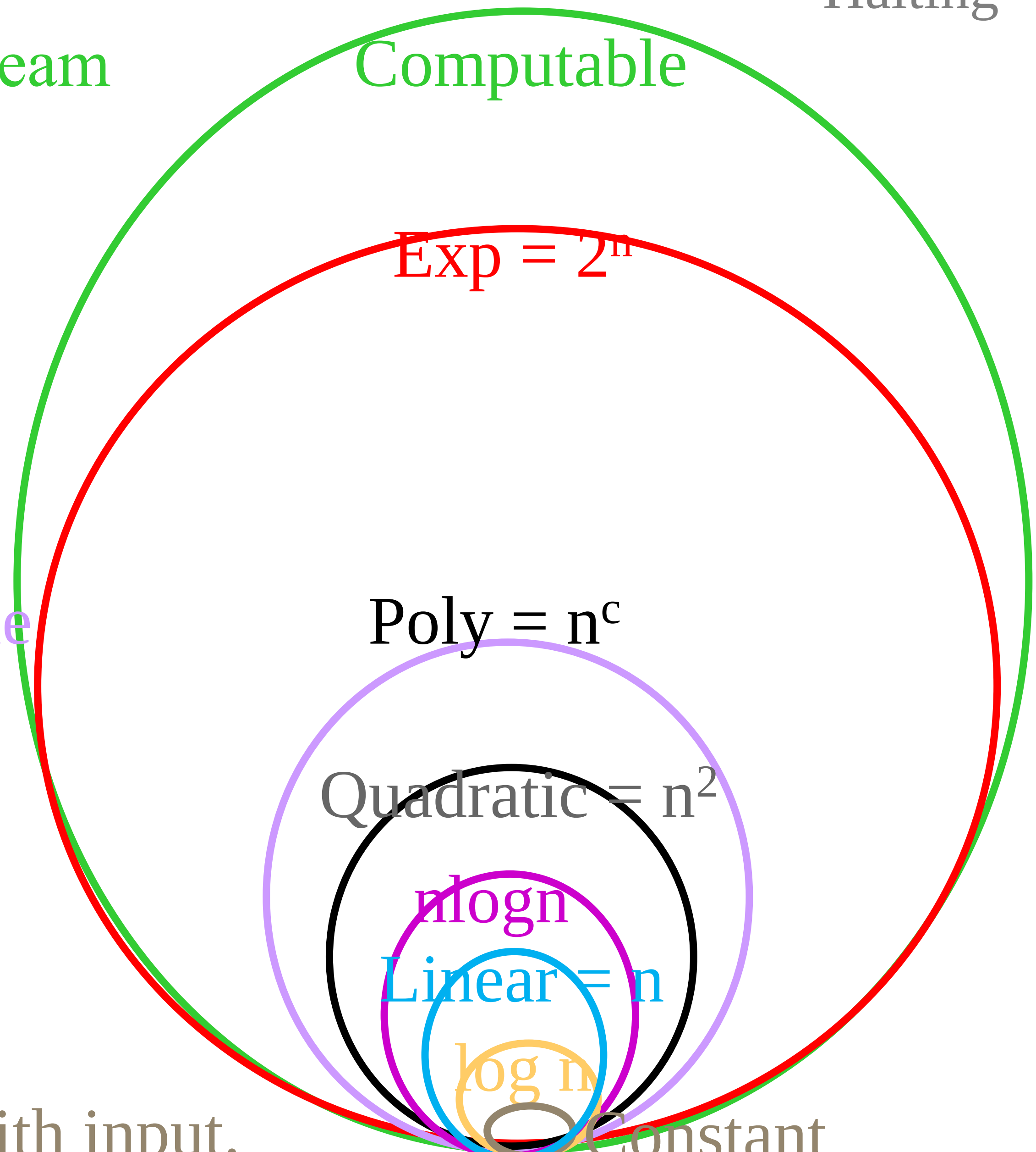
Slow sorting

Fast sorting

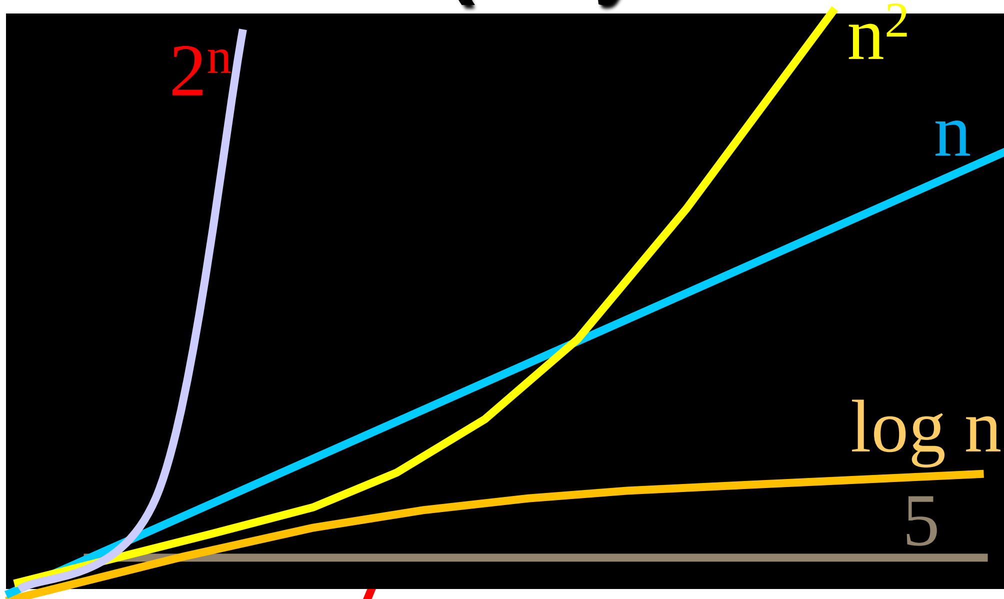
Look at input

Binary Search

Time does not grow with input.



Growth Rates (Büyüme hızları)



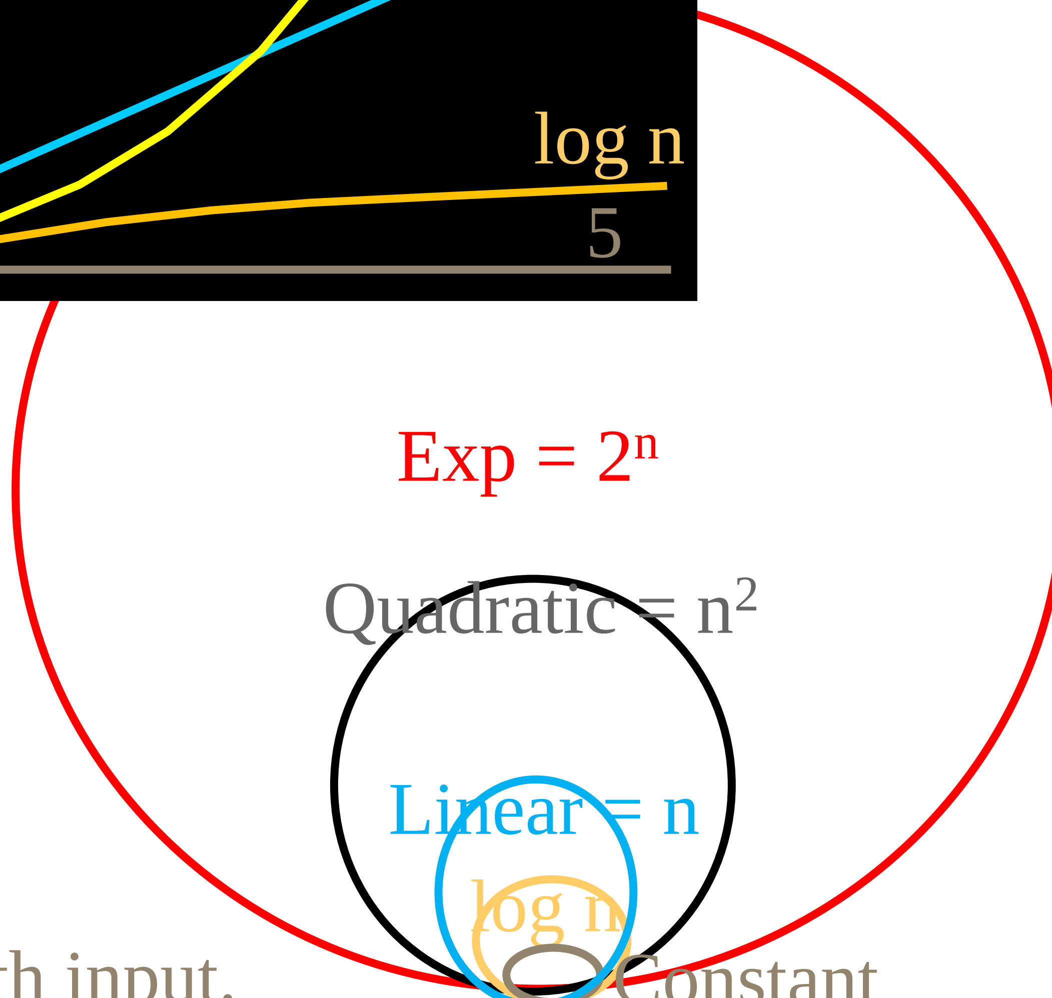
Brute Force
(Infeasible)

Slow sorting

Look at input

Binary Search

Time does not grow with input.



Growth Rates

T(n)	10	100	1,000	10,000	
5	5	5	5	5	atom
log **n**	3	6	9	13	amoeba
$\mathbf{n}^{1/2}$	3	10	31	100	bird
n	10	100	1,000	10,000	human
n log **n**	30	600	9,000	130,000	my father
$\mathbf{n}^2$	100	10,000	10^6	10^8	elephant
$\mathbf{n}^3$	1,000	10^6	10^9	10^{12}	dinosaur
$2^{\mathbf{n}}$	1,024	10^{30}	10^{300}	10^{3000}	the universe

Note: The universe contains approximately 10^{50} particles.
(Evren yaklaşık 10^{50} parçacık içerir.)

Growth Rates

T(n)	n	n+1	2n
5	T	5	5
log **n**	T	T+$^1/_n$	T+1
$\mathbf{n}^{1/2}$	T		
n	T	T+1	2T
n log **n**	T		
$\mathbf{n}^2$	T	? T+2n	? 4T
$\mathbf{n}^3$	T	T+3n^2	8T
$2^{\mathbf{n}}$	T	? 2T	? T^2

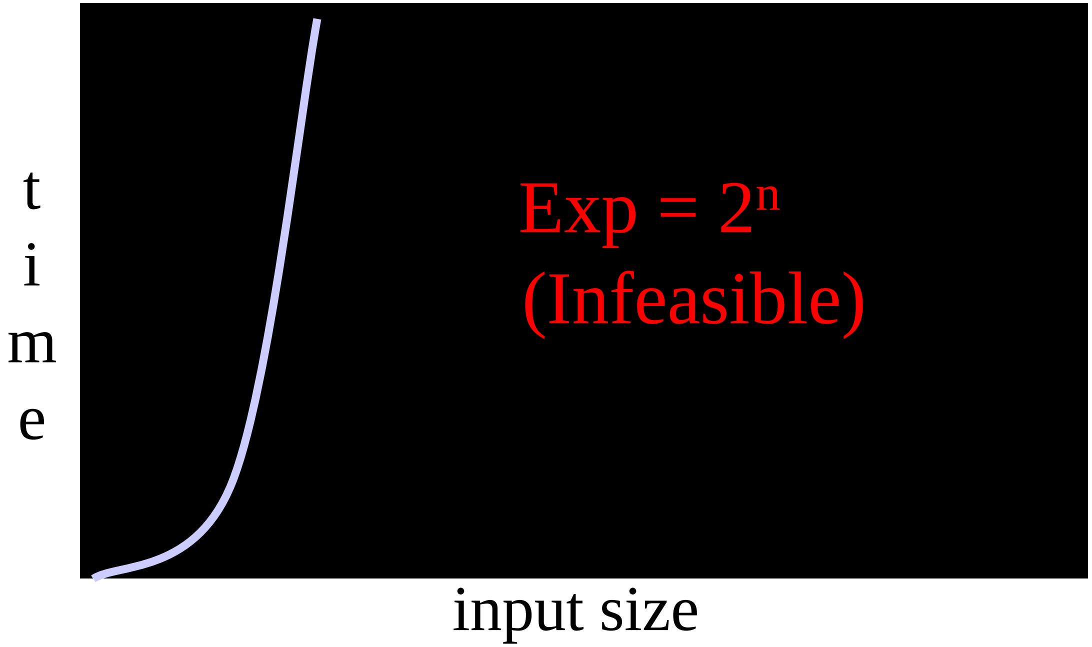
$$(n+1)^2 - n^2 = 2n$$

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

$$\frac{2^{n+1}}{2^n} = 2$$

$$2^{2n} = (2^n)^2$$

Growth Rates

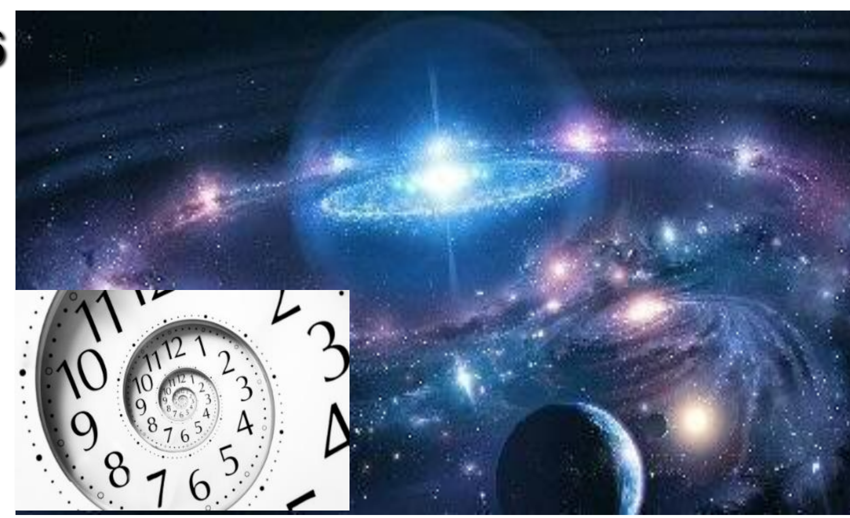


Let input size = n = 60

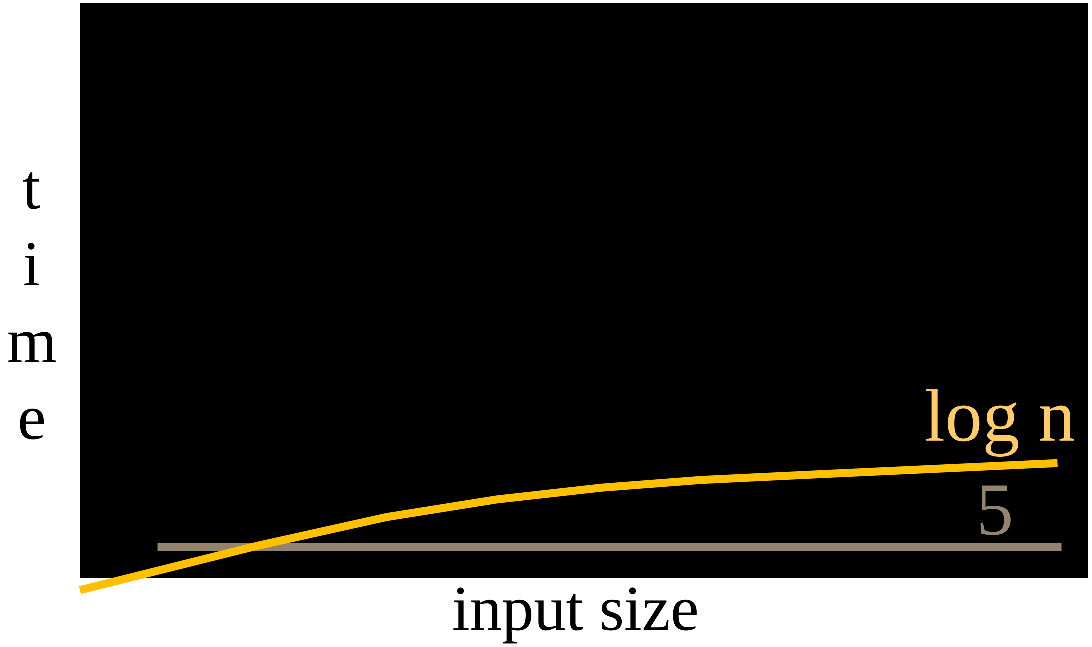
Time $\approx 2^{60} = (2^{10})^6 = (1024)^6$

$\approx (10^3)^6 = 10^{18}$

= Age of universe in seconds



Growth Rates

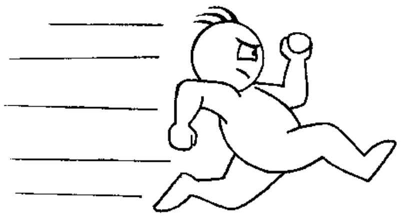


Let input size = n = 2^{60} = 10^{18} = size of universe

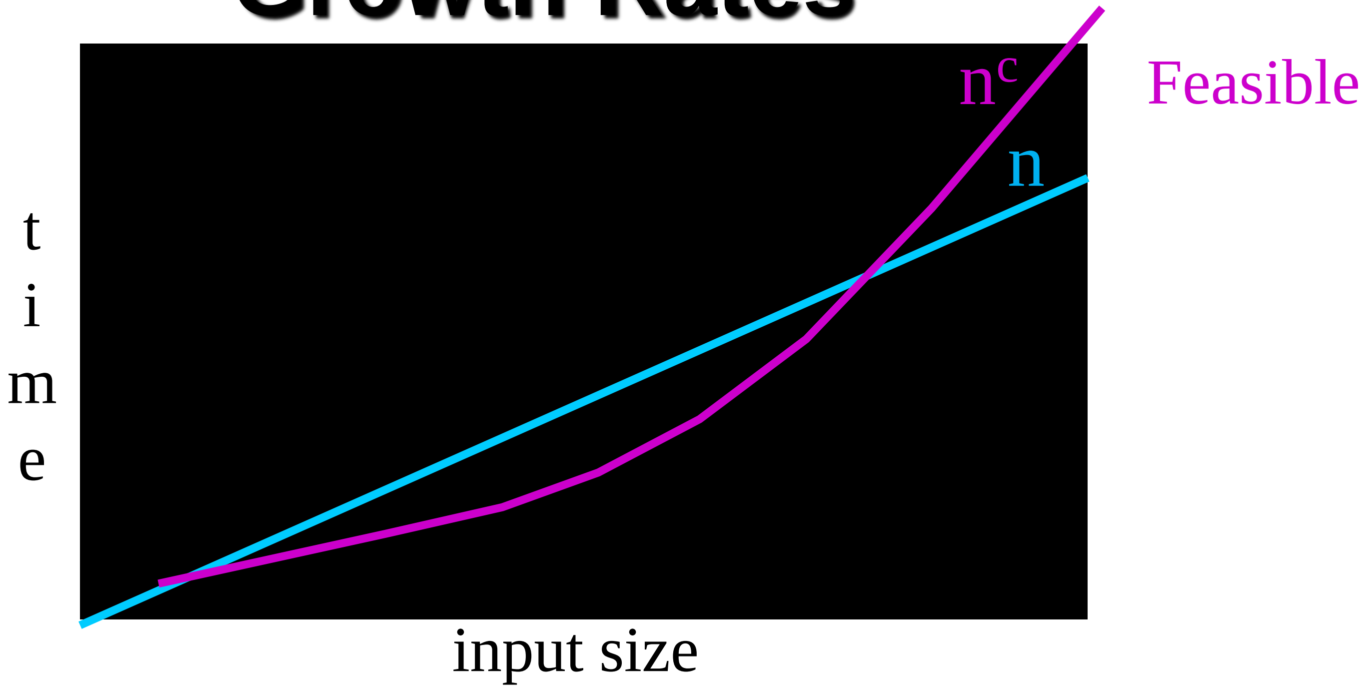
Time = log n = 60

Time grows with input size,

barely!



Growth Rates

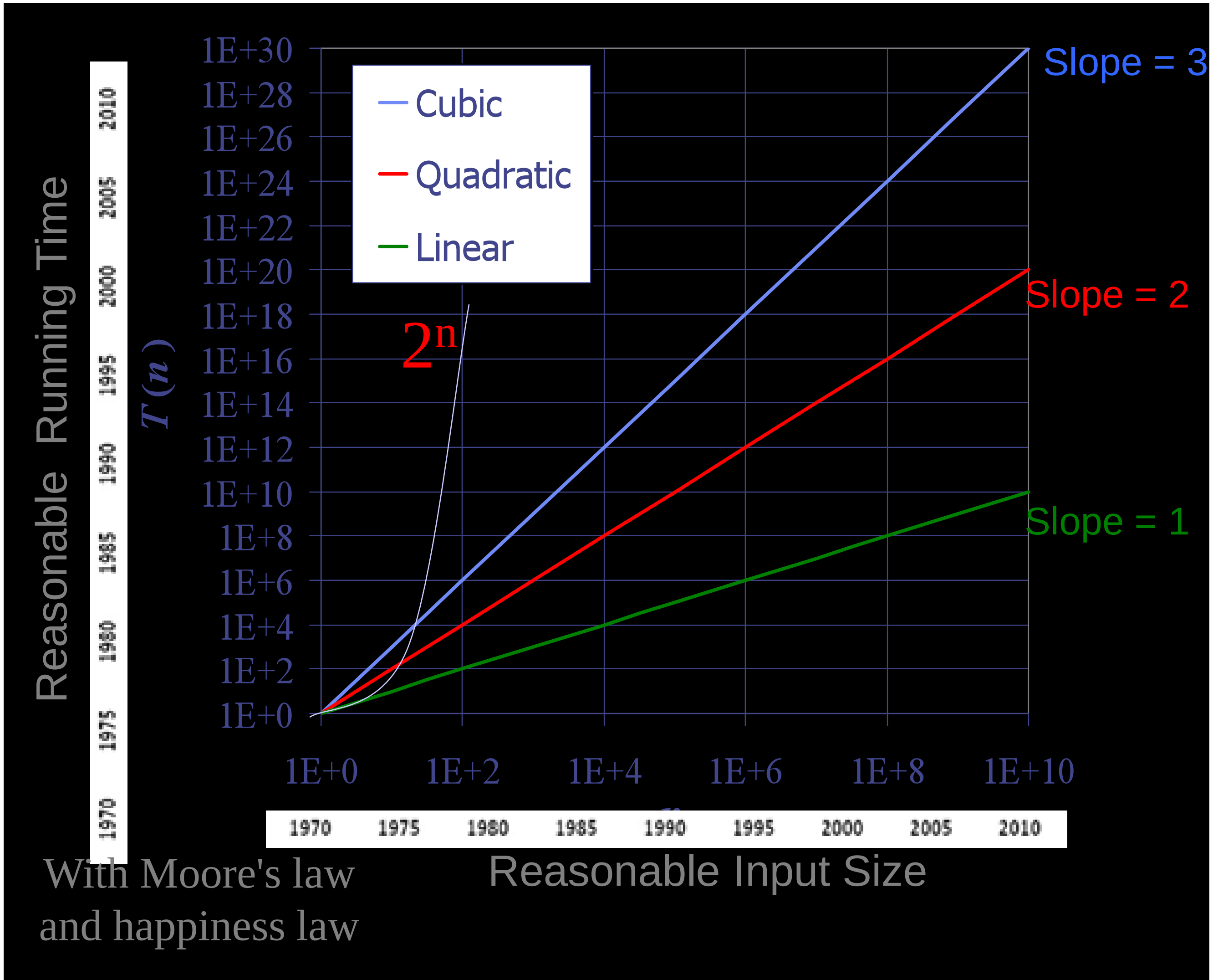


T(n)	10	100	1,000	10,000	
$\mathbf{n}^2$	100	10,000	10^6	10^8	elephant
$\mathbf{n}^3$	1,000	10^6	10^9	10^{12}	dinosaur
$\mathbf{n}^4$	10^4	10^8	10^{12}	10^{16}	manageable

Growth Rates

Plotted on a log-log chart, all polynomials look linear.

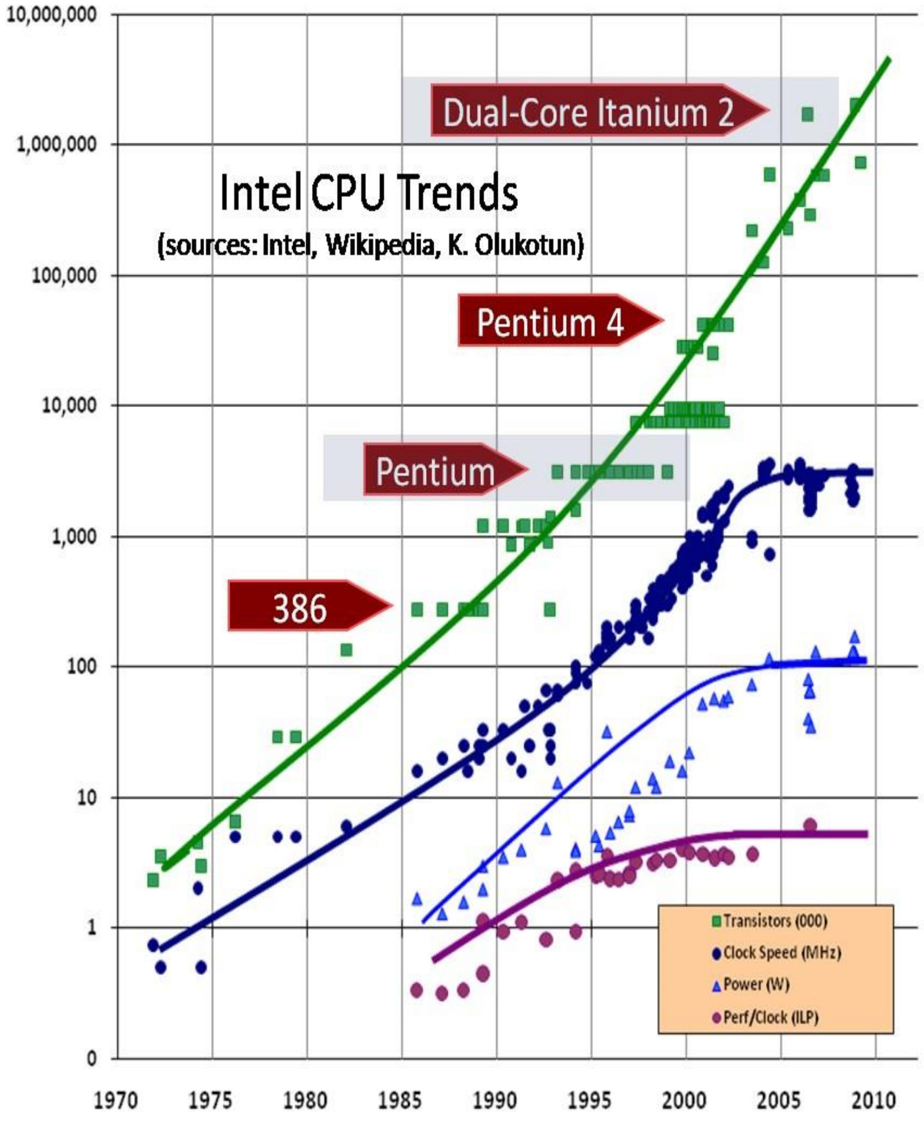
Exponentials are still exponential.



Log Scale

Moore's law: Approximately every two years
in a fixed cost computer
the following have doubled

- # of transistors, memory, speed
- the size of a typical input
- # ops in typical computation



Log Scale

Happiness law:

Happiness goes up by one when resources double.

$$\text{happiness} = \log(\text{ resources })$$

$$\text{resources} = 2^{\text{happiness}}$$

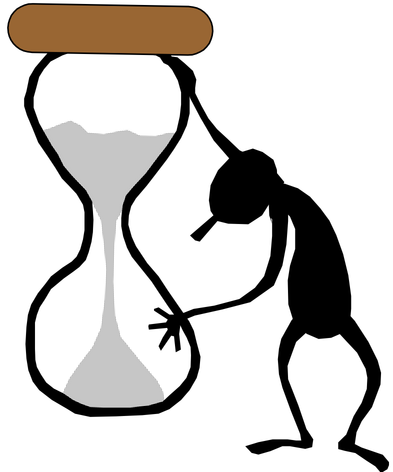
I have $10. I would be happy if I had another $10.

I have $thousand. I would be happy if I had another $thousand.

I have $million. I would be happy if I had another $million.

I have $billon. I would be happy if I had another $billon.

The Time Complexity of an Algorithm



Specifies how the running time depends on the size of the input.

Zamanı tanımlama:

- saniye cinsinden süre (makineye bağlı).
- icra edilen kod satırı sayısı
- belirli bir işlemin icra edilme sayısı (örneğin, toplama (addition))

Hangisi?

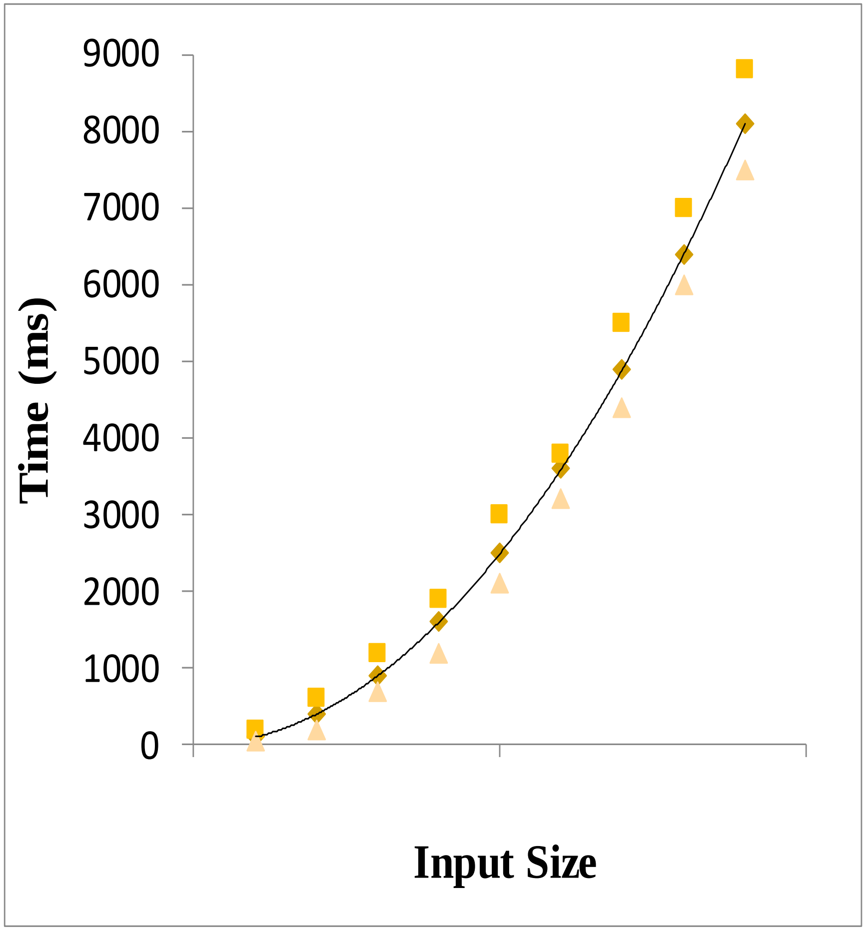
- Bunların hepsi makul zaman tanımlarıdır, çünkü birbirlerine sabit çarpanlar ile bağlılar

Deneysel (**Experimental**) Çalışma

Algoritmayı gerçekleştiren (**implement**) bir program yazılır

Program farklı boyutta girdilerle çalıştırılır

Fiili koşma süresinin tam bir ölçümünü almak için C'de <time.h> kütüphane fonksiyonlarından veya örneğin JAVA'daki system.currentTimeMillis () gibi bir yöntem kullanılır.



Teorik (**Theoretical**) Analiz

- Implementasyon yerine algoritmanın yüksek seviyeli bir tarifini/tanımını kullanır

- Koşma zamanını giriş boyutunun bir fonksiyonu olarak tanımlar, ***n***.

- Tüm olası girişleri dikkate alır

- Bir algoritmanın hızını **donanım / yazılım ortamından bağımsız** değerlendirmemizi sağlar

Efficiency or Complexity?

- Algoritma ders kitaplarında, verimlilik (**efficiency**) ve karmaşıklık (**complexity**) sıklıkla (genel olarak) eş anlamlı olarak kullanılır. Her ikisi de sıklıkla bir algoritmanın koşma süresini (**running time**) ifade etmek için kullanılır.

Time complexity: Bir algoritmanın zaman karmaşıklığı; girdi boyutu ile ilgili olarak ölçülen, algoritma tarafından atılan adım sayısını tanımlar.

Space complexity: Bir algoritmanın alan karmaşıklığı; girdi boyutu ile ilgili olarak ölçülen, algoritmanın koşması için gereken bellek miktarını tanımlar.

Algorithm Efficiency
(Algoritma Verimliliği/Etkinliği)

- Aynı sorunu çözen iki farklı algoritmayı nasıl karşılaştırırsınız?
  - Daha verimli/etkin olanını anlayıp ayırt edebilmelisiniz!
  - Brassard ve Bratley "**algorithmics** " terimini türettiler.
    - «Verimli algoritmalar tasarlamak ve analiz etmek için kullanılan temel tekniklerin sistematik olarak incelenmesi»

"the systematic study of the fundemental techniques used to design and analyze efficient algorithms" – 1988

Algoritma verimliliği nedir?

- Algoritmanın verimliliği (algorithm's efficiency), işlenecek eleman sayısının bir fonksiyonudur.
- Genel format:

$$f(n) = \text{efficiency}$$

n: işlenecek eleman sayısı (giriş boyutu)

Temel kavram

- Aynı sorunu çözen iki farklı algoritmayı karşılaştırırken, bir algoritmanın diğerinden (mesela on kat) daha verimli olduğunu sık sık görürüz.
- Bunun en tipik örneği, meşhur *Fast Fourier Transform (FFT)*'udur.
  - FFT, *NxlogN* adet çarpma ve toplama işlemi yapmayı gerektirirken Fourier Transform algoritması N^2 adet işlem gerektirir.

Temel kavram

- Verimlilik fonksiyonu lineer ise
  - Algoritma doğrusaldır ve döngü (**loop**) veya özyineleme (**recursions**) içermiyor demektir.
  - Verimlilik, algoritmanın içerdiği komut sayısının bir fonksiyonudur.
    - Bu durumda algoritmanın verimliliği **sadece bilgisayarın hızına** bağlıdır.
    - Bilgisayar hızı, genellikle bir programın genel verimliliğini etkileyen bir faktör değildir.

Linear function; contains no loops!

Efficiency = number of instruction in the function + the speed of the computer.

$f(n) = \text{efficiency}$

Temel kavram

- Eğer algoritma döngü veya özyineleme içeriyor ise
  - ➢ Verimlilik fonksiyonu nonlinear'dir.
  - ➢ Verimlilik bakımından büyük ölçüde değişiklik gösterir.
  - ➢ Bu durumda verimlilik fonksiyonu ağırlıklı olarak işlenecek olan **eleman sayısına** bağlıdır.
- Bu nedenle algoritma verimliliği çalışması **döngülere** odaklanır.
  - ❖ özyineleme her zaman bir döngüye dönüştürülebilir.

Linear Loops (*Doğrusal Döngüler*)

- Verimlilik, döngünün gövdesinin kaç kez tekrarlandığına bağlıdır. Doğrusal bir döngüde (linear loop) döngü güncellemesi (**kontrol değişkeni**) ya toplamadır ya da çıkarmadır.
- Örneğin

```
for (i = 0; i < 1000; i++)

    the loop body
```

➢Burada döngü gövdesi 1000 kez tekrarlanır.

➢**Doğrusal döngü için verimlilik**, yineleme sayısıyla doğru orantılıdır:

$$f(n) = n$$

Linear Loops (*Doğrusal Döngüler*)

```
for (i = 0; i < 1000; i=i+2)
        the loop body
```

- Yukarıdaki örnekte döngü gövdesi kaç kere tekrarlanır?
- Cevap: 500 kere
- Tekrar sayısı döngü çarpanının yarısı kadardır
- Bu döngünün **verimliliği**, çarpanın yarısı ile orantılıdır:

$$f(n) = n / 2$$

Linear Loops (*Doğrusal Döngüler*)

for (i = 0; i < 1000; i++) the loop body	for (i = 0; i < 1000; i=i+2) the loop body

- Bu döngü örneklerinden herhangi birini çizecek olursanız, düz bir çizgi elde edersiniz.
  - Bu nedenle **doğrusal döngüler** olarak bilinirler.

Logarithmic Loops *(Logaritmik Döngüler)*

- Bir logaritmik döngüde (logarithmic loop), kontrol değişkeni her tekrarda (iterasyonda) çarpılır veya bölünür
- Örneğin

Multiply loop	*Divide loop*
for (i=1; i<=1000; i*=2)	for (i=1000; i>=1; i /=2)
the loop body	the loop body

➢ **Logaritmik döngü** için **verimlilik** aşağıdaki formülle belirlenir:

$$f(n) = \log n$$

Multiply		Divide	
Iteration	Value of `i`	Iteration	Value of `i`
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

TABLE 1-3 Analysis of Multiply and Divide Loops

f(n) = log n

Döngü yinelenme şartı

multiply	$2^{Iterations} < 1000$
divide	$1000 / 2^{Iterations} >= 1$

Nested Loops *(İç içe Döngüler)*

- Her bir döngünün kaç kez tekrar ettiğini belirlemek gerekir

- Toplam tekrar sayısı

```
Iterations = outer loop iterations x inner loop iterations
```

  – İç içe döngü türleri
    - Linear Logarithmic
    - Quadratic
    - Dependent quadratic

Linear Logarithmic Nested Loop

(*Doğrusal logaritmik iç içe döngü*)

```
for (i=1; i<=10; i++)
for (j=1; j<=10; j *=2)
   the loop body
```

- Bu örnekte dıştaki döngü (**outer loop**) değişkeni artırılırken içteki döngü (**inner loop**) değişkeni çarpılır
- Toplam yineleme/tekrar sayısı, sırasıyla dış ve iç döngülerin yinelenme sayılarının **çarpımına** eşittir. (Bu örnek için <u>10log10</u>).

➢ **Doğrusal logaritmik iç içe döngü** için **verimlilik** aşağıdaki formüle göre belirlenir:

$$f(n) = n \log n$$

Quadratic Nested Loop
(*Karesel iç içe döngü*)

```
for (i=1; i<=10; i++)
for (j=1; j<=10; j ++)
   the loop body
```

- Bu örnekte döngülerin her ikisi de artırır.
- Karesel iç içe döngüdeki toplam yineleme/tekrar sayısı, sırasıyla iç ve dış döngüler için yineleme sayısının çarpımına eşittir. (Bu örnekte 10x10=100).

➢**Karesel iç içe döngü** için **verimlilik** aşağıdaki formüle göre belirlenir:

$$f(n) = n^2$$

Dependent Quadratic Nested Loop (*Bağımlı Karesel iç içe döngü*)

```
for (i=0; i<10; i++)

for (j=0; j<i; j ++)

   the loop body
```

- İç döngünün tekrar sayısı, dış döngüye bağlıdır.
  - İçteki döngü gövdesinin tekrar sayısı: $1 + 2 + 3 + \ldots + 9 + 10 = 55$
  - Bu döngünün ortalamasını hesaplarsak: 55/10=5.5 veya
  - Dıştaki döngünü tekrar sayısının 1 fazlasının yarısıdır: (10+1)/2
  - Yani, $\frac{(n+1)}{2}$

Dependent Quadratic Nested Loop (*Bağımlı Karesel iç içe döngü*)

- İç döngü ile dış döngünün tekrar sayısının çarpılması bize **bağımlı karesel döngü** için aşağıdaki **verimlilik** formülünü verir:

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

Big-O notation

- Genelde veri elemanı *n* için fonksiyonda icra edilen komut/ifade sayısı, eleman sayısının bir fonksiyonudur ve
  - $f(n)$ ile ifade edilir
- Bir fonksiyon için türetilmiş denklem karmaşık olsa bile, denklemdeki bir **baskın** (*dominant*) faktör genellikle sonucun büyüklüğünü (**order of magnitude**) belirler.
- Bu faktör bir **big-O**'dur. *O*(n) olarak ifade edilir.

"On the order of." → **big-O** (Omega)

Big-O notation

- Aşağıdaki adımları izleyerek **big-O** notasyonu f(n)'den türetilebilir:
  1) Her bir terimin katsayısı 1 yapılır.
  2) Fonksiyondaki en büyük terim haricindeki tüm terimler atılır.
     - Terimlerin küçükten büyüğe sıralanması şöyledir:

       log n, n, nlog n, n^2, n^3,…, n^k,…2^n,…, n!

Örneğin,

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

Katsayılar 1 yapılır

$$n^2 + n$$

Küçük terimler atılır

Big-O notation: $O(f(n)) = O(n^2)$

Algoritma verimliliği
kategorileri :7 tane

Standard measures of efficiency

Azalan verimlilik

Low

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

TABLE 1-4 Measures of Efficiency for $n = 10{,}000$

(N=10000 için verimlilik ölçüsü)

Herhangi bir verimlilik ölçüsü, yeterince büyük bir örneklemin dikkate alındığını varsayar.

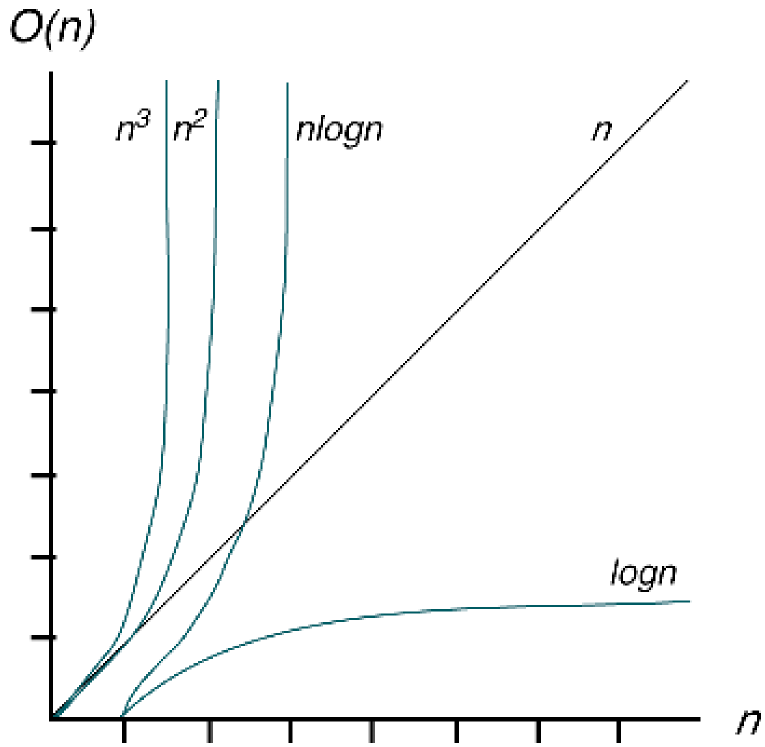


FIGURE 1-14 Plot of Effeciency Measures

Big-O Analiz Örnekleri
İki Matrisi Toplama

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & -3 & 4 \\ 5 & 6 & 2 \end{bmatrix} + \begin{bmatrix} 6 & 1 & 7 \\ 3 & 2 & -1 \\ 4 & 6 & 2 \end{bmatrix} = \begin{bmatrix} 10 & 3 & 8 \\ 3 & -1 & 3 \\ 9 & 12 & 4 \end{bmatrix}$$

FIGURE 1-15 Add Matrices

ALGORITHM 1-3 Add Two Matrices

```
Algorithm addMatrix (matrix1, matrix2, size, matrix3)
Add matrix1 to matrix2 and place results in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns or rows in matrix
   Post matrices added--result in matrix3
1 loop (not end of row)
   1  loop (not end of column)
      1  add matrix1 and matrix2 cells
      2  store sum in matrix3
   2  end loop
2 end loop
end addMatrix
```

Add two matrices

Algorithm **addMatrix** (matrix1, matrix2, size, matrix3)

1. r = 1
2. loop (r<=size)
    1. c = 1
    2. loop (c <= size)
        1. matrix3[r,c] = matrix1[r,c] + matrix2[r,c]
        2. c = c +1
    3. r = r + 1
3. return

end addMatrix

loop2
size times

loop1
size
times

Quadratic loop
(Karesel dögü)

$\boxed{O(\mathbf{size}^2) \Rightarrow O(n^2)}$

Big-O Analiz Örnekleri

İki Matrisi Çarpma

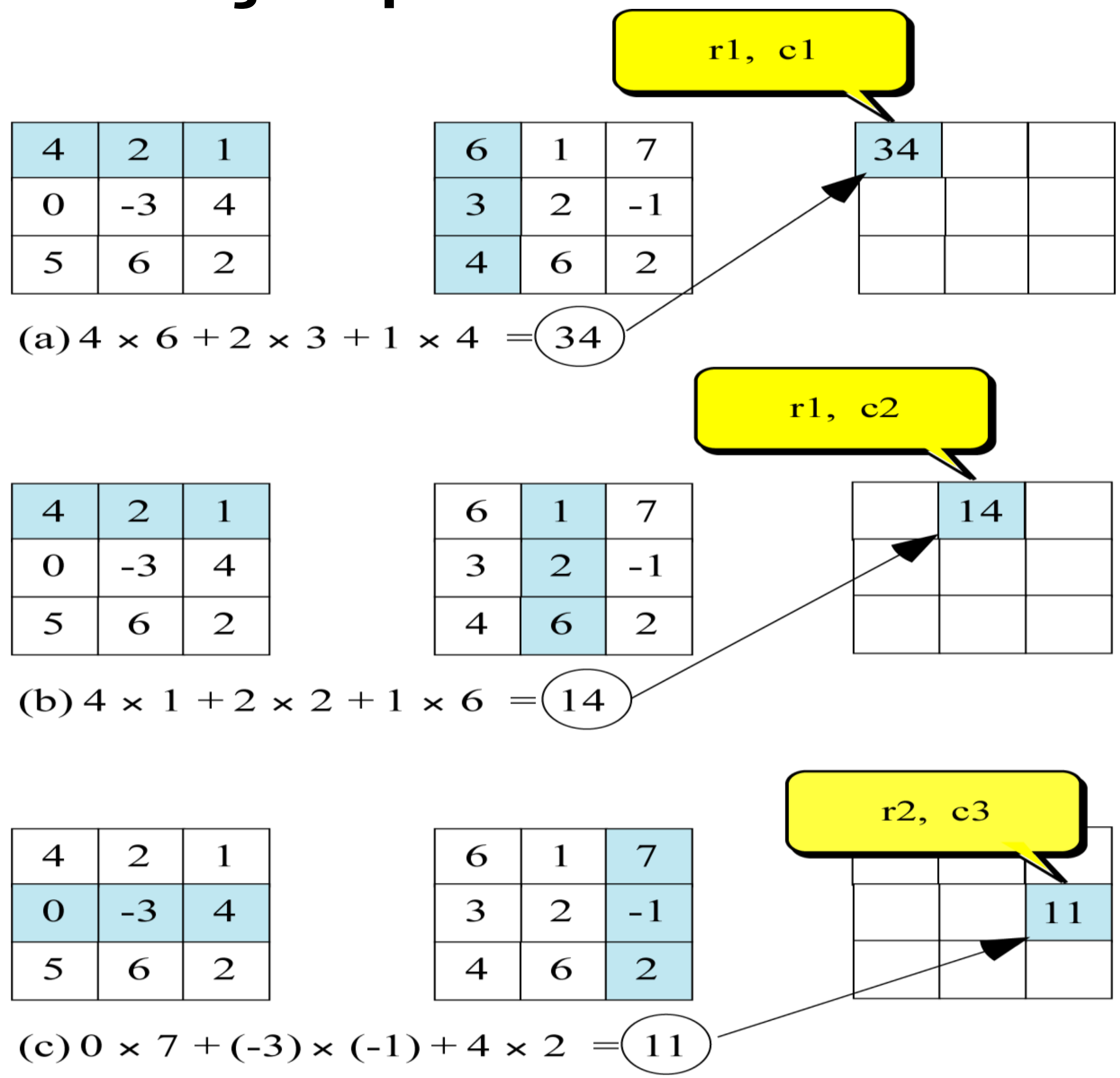


FIGURE 1-16 Multiply Matrices

ALGORITHM 1-4 Multiply Two Matrices

```
Algorithm multiMatrix (matrix1, matrix2, size, matrix3)
Multiply matrix1 by matrix2 and place product in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns and rows in matrix
   Post matrices multiplied--result in matrix3
1 loop (not end of row)
   1  loop (not end of column)
      1  loop (size of row times)
         1  calculate sum of
               (all row cells) * (all column cells)
         2  store sum in matrix3
   2  end loop
2 end loop
3 return
end multiMatrix
```

Multiply two matrices

- 3 tane iç içe (nested) döngü
- Her bir döngü ilk elemandan başlayıp satır(=*sütun*) sayısı kadar tekrar eder.
- **Kübik döngü** (Cubic loop)
  - Algoritma big-O verimliliği: O(**size**^{3}) veya O (**n**$^{\mathbf{3}}$)

Alıştırma-1

- Aşağıdaki verimlilikleri küçükten büyüğe sıralayınız.

  a) $n\log(n)$

  b) $n + n^2 + n^3$

  c) 24

  d) $n^{0.5}$

24 , $n^{0.5}$, $n\log(n)$, $n + n^2 + n^3$

Alıştırma-2

- Eğer XX isimli algoritmanın karmaşıklığı (complexity) $O(n)=n^2$ ise aşağıdaki program parçasının karmaşıklığını nedir?

```
i = 1
loop (i <= n)
 j = 1
 loop (j <  n)
     XX ( ....)
     j = j + 1
 i = i + 1
```

n^2 $n\text{-}1$ n

$$f(n) = n^2.(n\text{-}1).n \quad => O(f(n)) = O(n^4)$$

Alıştırma-2

- Eğer **doIt** isimli algoritmanın (alt programın) 5n verimlilik faktörü (çarpanı) varsa aşağıdaki program parçasının koşma süresi (*run-time*) verimliliği nedir?

```
for (i = 1, i <= n; i++)
    doIt (...)
```

$f(n) = n(5n) = 5n^2$

$\Downarrow$

n^2

$\Downarrow$

$\boxed{O(f(n)) = O(n^2)}$

Alıştırma-3

- Eğer **doIt** isimli algoritmanın verimliliği $O(n)=n^2$ ise, aşağıdaki program parçasının verimliliği nedir?

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

$$\boxed{O(n^2 log_2 n)}$$

Alıştırma-4

- Bir algoritmanın verimliliğinin n^3 olduğu göz önüne alındığında, bu algoritmadaki bir adım **1 nanosaniye** sürüyorsa, algoritmanın 1000 elemanlı bir girişi işlemesi ne kadar sürer?

Süre=(1000^3) * *1 ns*= $10^9 * 10^{-9}$ =*1 saniye*

Alıştırma-5

- Bir algoritma n elemanlı girişleri işlemektedir. n=4096 ise, koşma süresi (**run-time**) 512 milisaniyedir. n=16,384 ise, çalışma süresi 1024 milisaniyedir. Bu algoritmanın verimliliği ve big-O notasyonu nedir? Hesaplayınız.

$$n_1 = 4096 \qquad n_2 = 16384$$
$$f(n_1) = 512 \qquad f(n_2) = 1024$$

$$n_2 = 4 * n_1$$
$$f(n_2) = 2 * f(n_1)$$

n 4 katına çıkarken $f(n)$ sadece 2 katına çıktığı için **verimlilik**=$\sqrt{n} = n^{1/2}$

Big-O notation: $O(n^{1/2})$

Teşekkürler…

Kaynaklar

- Kitaplar
  - **Data Structures: A Pseudocode Approach with C** (2nd Ed.) (Course Technology) Richard F. Gilberg & Behrouz A. Forouzan
- Ders Notları
  - **EECS 2011: Fundamentals of Data Structures by** Jeff Edmonds, York University