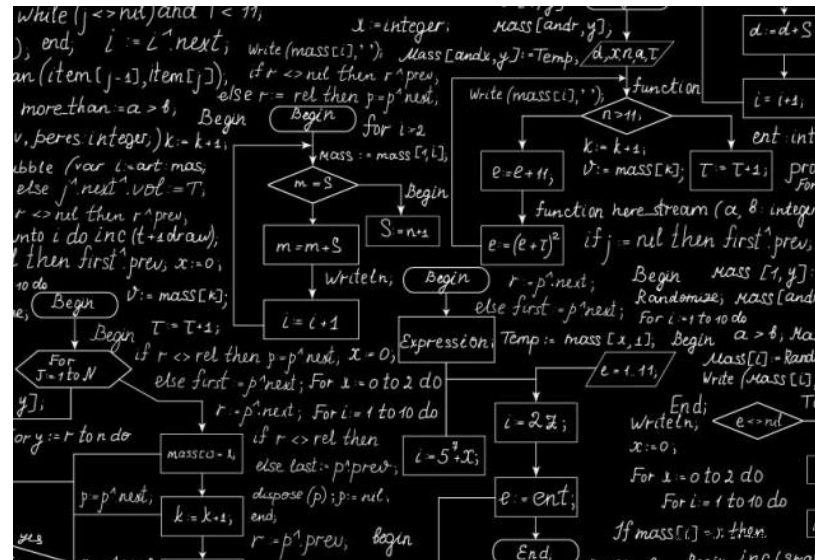
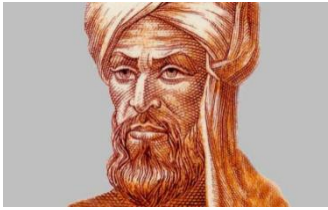


Bölüm 5

Algoritmalar





El Harezmi - Abu Ja'far Mohammed Ibin Musa Al-Khowarizmi (MS 780-850)

- ❑ Algoritma kavramı
- ❑ Temel veri yapıları, temel problemler
- ❑ Fonksiyonların büyümesi, Asimptotik notasyonlar
- ❑ Algoritma karmaşıklığı ve analizi
 - İteratif ve özyinelemeli algoritmaların analizi

Algoritma

Tanım: Algoritma, hesaplama yapmak veya bir problemi çözmek için sonlu (finite) kesin (precise) komutlar/buyruklar/emirler (instructions) kümesidir.

Örnek: Sonlu bir tam sayı dizisinde maksimum değeri bulmak için bir algoritmayı yazınız.

Çözüm: Aşağıdaki işlemler sırasıyla uygulanır

1. Dizinin ilk elemanı geçici maksimum değer olarak seçilir
2. Bir sonraki eleman geçici maksimum ile karşılaştırılır
 - Eğer sayı geçici maksimumdan büyükse, geçici maksimuma bu sayıyı atanır
3. Dizinin son elemanına kadar önceki adım tekrarlanır
4. Algoritma sonlandığında dizideki en büyük sayı bulunur

Algoritma nasıl belirtilir

- ❑ Algoritma adımları Türkçe, İngilizce veya sözde kodla (Pseudocode), akış şeması veya bir programlama dili ile açıklanabilir.
- ❑ Sözde kod, adımların İngilizce açıklaması ile bu adımların bir programlama dili kullanılarak kodlanması arasında ara bir basamaktır.
- ❑ Sözde kod, algoritmayı uygulamak için kullanılan gerçek programlama dilinden bağımsız olarak, algoritma kullanarak bir problemi çözmek için gereken zamanı analiz etmeye yardımcı olur.
- ❑ Programcılar, bir programı kodlarken sözde koddaki algoritmanın açıklamasını kullanabilir.
- ❑ Sözde kod, doğal bir dile doğrudan bağımlı olduğundan Akış Şeması algoritmanın görsel olarak ifade edilmesinde ortak bir dil elde etmeyi sağlar.
- ❑ Sözde kod veya Akış şemasına ihtiyaç olmayan durumlarda, Algoritma herhangi bir programlama diliyle de kodlanabilir.

Algoritmaların Özellikleri

- ❑ **Giriş (Input):** Bir algoritma, belirli bir kümeden giriş değerlerine sahiptir.
- ❑ **Çıkış (Output):** Algoritma, girdi değerlerinden belirli bir kümeden çıktı değerlerini üretir. Çıktı değerleri çözümdür.
- ❑ **Doğruluk (Correctness):** Bir algoritma, her girdi değeri kümesi için doğru çıktı değerlerini üretmelidir.
- ❑ **Sonluluk (Finiteness):** Bir algoritma, herhangi bir girdi için sınırlı sayıda adımdan sonra çıktı üretmelidir.
- ❑ **Etkililik (Effectiveness):** Algoritmanın her adımını doğru ve sınırlı bir süre içinde gerçekleştirmek mümkün olmalıdır.
- ❑ **Genellik (Generality):** Algoritma, istenen formdaki tüm problemler için çalışmalıdır.

Bir dizideki en büyük elemanı bulan algoritma

□ Sözde kod:

```
procedure max( $a_1, a_2, \dots, a_n$ : integers)
  max :=  $a_1$ 
  for  $i := 2$  to  $n$ 
    if  $max < a_i$  then  $max := a_i$ 
  return max{max is the largest element}
```

□ Bu algoritma önceki slayttaki tüm özellikleri sağlıyor mu?

Bazı örnek algoritma problemleri

□ Bu bölümde üç tür problem incelenecektir.

1. *Arama (Searching)* : Bir elemanın arama uzayındaki yerinin bulunması
2. *Sıralama (Sorting)* : Bir listenin elemanlarını artan /azalan sıraya koymak.
3. *Eniyileme (Optimization)* : Tüm olası girdiler üzerinden belirli bir büyüklüğün en iyi değerinin (maksimum/minimum) belirlenmesi. (Düşük maliyet, yüksek verim)

Arama (Searching)

Tanım: Genel arama problemi, ayrı elemanlar ($a_1, a_2, \dots, a_n$) listesinde bir x elemanını bulmak veya listede olmadığını belirlemektir.

- ❑ Bir arama probleminin çözümü, listedeki terimin x 'e eşit olan konumudur ($x = a_i$) veya x listede yoksa 0'dır.
- ❑ Arama sadece bir sayı dizisinde eleman aramak demek değildir!
- ❑ İki farklı arama algoritmasını inceleyeceğiz;
 - doğrusal arama
 - ikili arama.

Doğrusal Arama

- Doğrusal Arama algoritması, aranan sayıyı dizinin elemanlarını baştan başlayarak birer birer incelemesidir

```
procedure linear search( $x$ :integer,  
                         $a_1, a_2, \dots, a_n$ : distinct integers)  
   $i := 1$   
  while ( $i \leq n$  and  $x \neq a_i$ )  
     $i := i + 1$   
  if  $i \leq n$  then  $location := i$   
  else  $location := 0$   
  
  return  $location$  { $location$  is the subscript of the term  
                    that equals  $x$ , or is 0 if  $x$  is not found}
```

İkili Arama

- ❑ İkili arama algoritması, arama uzayındaki orta noktayı dikkate alarak arama gerçekleştiren algoritmadır

procedure binary search(x : integer, $a_1, a_2, \dots, a_n$: increasing integers)

$i := 1$ { i is the left endpoint of interval}

$j := n$ { j is right endpoint of interval}

while $i < j$

$m := \lfloor (i + j)/2 \rfloor$

if $x > a_m$ **then** $i := m + 1$

else $j := m$

if $x = a_i$ **then** $location := i$

else $location := 0$

return $location$ {location is the subscript i of the term a_i equal to x ,
or 0 if x is not found}

Sıralama

- ❑ Bir dizinin öğelerini sıralamak, onları artan sıraya koymaktır (sayısal sıra, alfabetik vb.).
- ❑ Sıralama önemli bir problemdir:
 - Donanım kaynaklarının önemsiz bir yüzdesi, farklı türdeki dizileri, özellikle belirli bir sırayla sunulması gereken büyük bilgi veri tabanlarını içeren uygulamaları (örneğin, müşteriye göre, parça numarası vb.) sıralamak için ayrılmıştır.
 - Çok fazla sayıda Sıralama algoritması geliştirilmiştir. Göreceli avantajları ve dezavantajları kapsamlı bir şekilde incelenmiştir.

Kabarcık (Bubble) Sort

- Kabarcık sıralama, bir dizide birden çok geçiş yapar. Sıra dışı olduğu tespit edilen her öge çifti değiştirilir.

procedure *bubblesort*($a_1, \dots, a_n$: real numbers with $n \geq 2$)

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

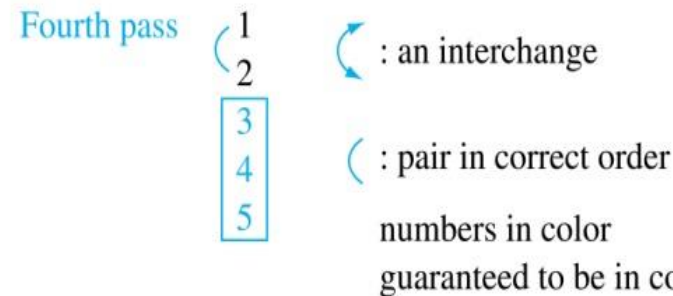
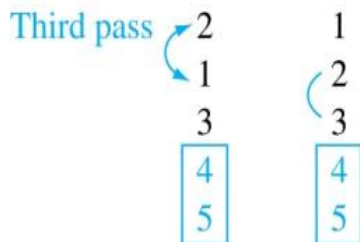
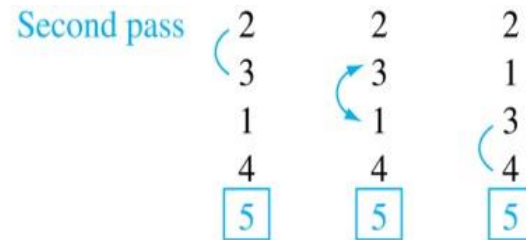
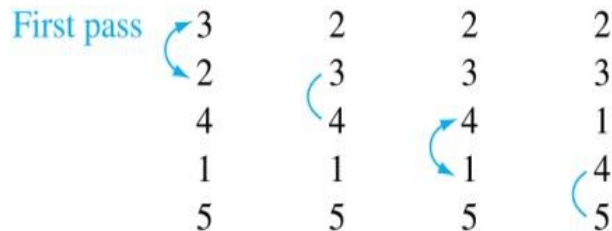
for $j := 1$ to $n - i$

if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

$\{a_1, \dots, a_n$ is now in increasing order $\}$

Bubble Sort

Örnek: 3 2 4 1 5 ile kabarcıklı sıralama adımlarını gösterin



- İlk geçişte en büyük eleman doğru konuma getirildi
- İkinci geçişin sonunda 2. en büyük eleman doğru konuma getirildi
- Sonraki her geçişte, doğru konuma ek bir eleman yerleştirilir

Euclid's Algorithm

Problem: $\gcd(m,n)$, the greatest common divisor : m ve n sayısının OBEB'i

Örnek: $\gcd(60, 24) = 12$, $\gcd(60, 0) = 60$,

Euclidean Algoritması

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

yukarıdaki eşitlik ikinci sayı (n) 0 olana kadar devam eder

Örnek:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12$$

Euclid's Algoritması Sözde Kodu

1. If $n = 0$, return m and stop; otherwise go to 2
2. Divide m by n and assign the value fo the remainder to r
3. Assign the value of n to m and the value of r to n . Go to 1.

while $n \neq 0$ do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

Diğer çözüm $\gcd(m,n)$

Ardışık tamsayı kontrol algoritması

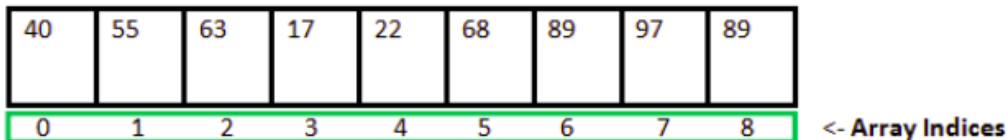
1. İki sayıdan küçük olanı $\{m, n\}$, T değişkenine atayın
2. m 'yi T 'ye bölün. Kalan 0 ise (3)'e gidin; değilse (4)'e gidin
3. n 'yi T 'ye bölün. Kalan 0 ise, T 'yi döndürün ve durun; değilse (4)'e gidin
4. T 'yi 1 azaltın ve (2) gidin

Temel Veri Yapıları

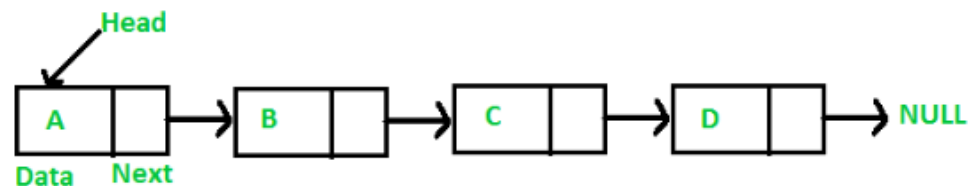
- ❑ Listeler
 - Dizi -Array
 - Bağlantılı listeler -linked list
 - String
- ❑ Yığın - Stack
- ❑ Kuyruk - Queue
- ❑ Priority queue
- ❑ Çizge - Graf
- ❑ Ağaçlar
- ❑ Kümeler

Listeler

- ❑ Dizi-Array – Bağlı Liste- Linked List
 - Diziler belleğe ardışıl olarak yerleştirilir
 - Bağlı listeler belleğin ardışıl olmayan hücrelerine yerleştirilir
 - Birden fazla elemanın saklanması işlenmesi durumunda kullanılır
 - Linked list sıralı erişim, dizi rastgele veya sıralı erişime uygundur



Array Length = 9
First Index = 0
Last Index = 8



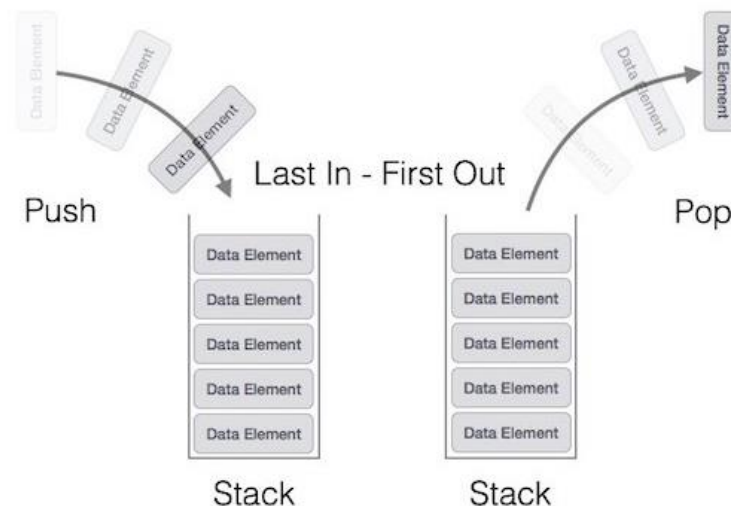
Kuyruk - Queue

- ❑ Özel amaçlı kullanılan listelerdir
- ❑ İlk gelen ilk çıkar / Son gelen son çıkar prensibine göre erişim sağlanır
(FIFO / LILO → First In - First Out / Last In - Last Out)
- ❑ Dizi veya bağlı listeler ile oluşturulabilir
- ❑ Processlerin sıralı işlenmesi, arama algoritması, vb. uygulamaları vardır
- ❑ İki temel işlem yapılır. Ekle (In-Enqueue), Çıkar (Out-Dequeue)



Yığın - Stack

- ❑ Özel amaçlı kullanılan listelerdir
- ❑ İlk gelen son çıkar / Son gelen ilk çıkar prensibine göre erişim sağlanır
(FILO / LIFO → First In-Last Out / Last In-First Out)
- ❑ Dizi veya bağlı listeler ile oluşturulabilir
- ❑ Arama algoritmalarında, matematiksel ifadelerin dönüşümünde kullanılır
- ❑ İki temel işlem yapılır. Ekle (push), çıkar (pop)



Yığın Uygulamaları

- ❑ İşlemci içinde SP (Stack Pointer) başta olmak üzere çok çeşitli uygulamalar vardır. Ders kapsamında aşağıdaki iki uygulamaya ağırlık verilecektir.
 - Matematiksel ifadelerin birbirine dönüşümü
 - Matematiksel ifadelerin işlem önceliğine uygun olarak değerlendirilmesi

Matematiksel İfadelerin Yazılımı

- Bir matematiksel ifadede OPERATÖR ve OPERAND'ların yazılım sırası farklı notasyonlara göre yapılabilir.
- Sıralama farklı olsa da, sonuç aynıdır.
- Derleyicilerin operatör önceliği vb. konularda bu notasyonların kullanımı önemlidir.

Örnek: Matematiksel ifademiz $(A + B) \times C$ olsun.

Operandlar: A, B ve C

Operatörler: + ve x

Yığın Uygulamaları

- ❑ **Infix notasyonu:** Operatör, iki operandın ortasına bulunur

Örnek: $A + B$, $7 - 9$

- ❑ **Prefix notasyonu:** Operatör iki operandın önünde bulunur PN (Polish Notation)

Örnek: $+ A B$, $- 7 9$

- ❑ **Postfix notasyonu:** Operatör iki operandın sonunda bulunur (Reverse Polish Notation)

Örnek: $A B +$, $7 9 -$

- ❑ Bilgisayar sistemlerinde infix yazılımdaki operatör önceliğinin çözümlenmesinin zorluğu sebebiyle postfix /prefix notasyonu dönüşümü ile işlemlerin sonucu hesaplanır!
- ❑ Bu dönüşümler, **infix** $\leftrightarrow$ **postfix**, **infix** $\leftrightarrow$ **prefix**, **prefix** $\leftrightarrow$ **postfix** olabilir
- ❑ Veilen bir notasyonun sayısal sonucunun hesaplanması da çok sık karşılaşılan uygulamalardandır
- ❑ Dönüşümlerde Stack / İkili Ağaçlar kullanılır

Infix ifadenin değ erlendirmesi

- Operands are real numbers.
- Permitted operators: +, -, *, /, ^ (exponentiation)
- Blanks are permitted in expression.
- Parenthesis are permitted

Example:

Approach: Use [Stacks](#)

$A * (B + C) / D$

We will use two stacks

$2 * (5 + 3) / 4$

- Operand stack: This stack will be used to keep track of numbers.
- Operator stack: This stack will be used to keep operations (+, -, *, /, ^)

Output: 4

Order of precedence of operations–

1. ^ (Exponential)
2. / *
3. + –

Note: brackets () are used to override these rules.

Infix ifadenin değ erlendirmesi

Algorithm:

Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator,
 1. If the operator stack is empty then push it to the operator stack.
 2. Else If the operator stack is not empty,
 - If the character's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.
 - If the character's precedence is less than the precedence of the stack top of the operator stack then do **Process** (as explained above) until character's precedence is less or stack is not empty.
3. If the character is "(", then push it onto the operator stack.
4. If the character is ")", then do **Process** (as explained above) until the corresponding "(" is encountered in operator stack. Now just pop out the "(".

Once the expression iteration is completed and the operator stack is not empty, do **Process** until the operator stack is empty. The values left in the operand stack is our final result.

Örn: Infix değerlendirme

Infix Expression: 2 * (5 *(3+6)) / 15-2				
Token	Action	Operand Stack	Operator Stack	Notes
2	Push it to operand stack	2		
*	Push it to operator stack	2	*	
(	Push it to operator stack	2	(*	
5	Push it to operand stack	5 2	(*	
*	Push it to operator stack	5 2	* (*	
(	Push it to operator stack	5 2	(* (*	
3	Push it to operand stack	3 5 2	(* (*	
+	Push it to operator stack	3 5 2	+ (* (*	
6	Push it to operand stack	6 3 5 2	+ (* (*	
)	Pop 6 and 3 from operand stack	5 2	+ (* (*	Do process until (is popped from operator stack
	Pop + from operator stack	5 2	(* (*	
	Do 6+3 = 9	5 2	(* (*	
	Push 9 to operand stack	9 5 2	(* (*	
	Pop (from operator stack	9 5 2	* (*	
)	Pop 9 and 5 from operand stack	2	* (*	Do process until (is popped from operator stack
	Pop * from operator stack	2	(*	
	Do 9*5 = 45	2	(*	
	Push 45 into operand stack	45 2	(*	
	Pop (from operator stack	45 2	*	
/	Push / into operator stack	45 2	/ *	/ has equal precedence to *
15	Push 15 to operand stack	15 45 2	/ *	
-	Pop 15 and 45 from operand stack	2	/ *	- has lower precedence than / , do the process
	Pop / from operator stack	2	*	
	Do 45/15 = 3	2	*	
	Push 3 into operand stack	3 2	*	
	Pop 3 and 2 from operand stack		*	- has lower precedence than * , do the process
	Pop * from operator stack			
	Do 3*2 = 6			
	Push 6 into operand stack	6		- has equal precedence to +
	Push - into operator stack	6	-	
2	Push 2 into operand stack	2 6	-	
	Pop 2 and 6 from the operand stack			Given expression is iterated, do Process till operator stack is not empty, It will give the final result
	Pop - from operator stack			
	Do 6-2 =4			
	Push 4 to operand stack	4		

Prefix ifadenin değ erlendirmesi

Prefix: + 500 40

Output: 540

Explanation: Infix expression of the above prefix is: $500 + 40$ which resolves to 540

Prefix: - / * 20 * 50 + 3 6 300 2

Output: 28

Explanation: Infix expression of above prefix is: $20 * (50 * (3+6))/300-2$ which resolves to 28

Approach: Use Stack

Algorithm:

Reverse the given expression and Iterate through it, one character at a time

1. If the character is a digit, initialize String temp;
 - while the next character is not a digit
 - do temp = temp + currentDigit
 - convert Reverse temp into Number.
 - push Number to the stack.
2. If the character is an operator,
 - pop the operand from the stack, say it's s1.
 - pop the operand from the stack, say it's s2.
 - perform (**s1 operator s2**) and push it to stack.
3. Once the expression iteration is completed, The stack will have the final result. Pop from the stack and return the result.

Örn: Prefix ifadenin değerlendirilmesi

Prefix Expression : - / * 20 * 50 + 3 6 300 2		
Reversed Prefix Expression: 2 003 6 3 + 05 * 02 * / -		
Token	Action	Stack
2	Reverse 2, Push 2 to stack	[2]
003	Reverse 003, Push 300 to stack	[2, 300]
6	Reverse 6, Push 6 to stack	[2, 300, 6]
3	Reverse 3, Push 3 to stack	[2, 300, 6, 3]
+	Pop 3 from stack	[2, 300, 6]
	Pop 6 from stack	[2, 300]
	Push 3+6 =9 to stack	[2, 300, 9]
05	Reverse 05, Push 50 to stack	[2, 300, 9, 50]
*	Pop 50 from stack	[2, 300, 9]
	Pop 9 from stack	[2, 300]
	Push 50*9=450 to stack	[2, 300, 450]
02	Reverse 02, Push 20 to stack	[2, 300, 450, 20]
*	Pop 20 from stack	[2, 300, 450]
	Pop 450 from stack	[2, 300]
	Push 20*450=9000 to stack	[2, 300, 9000]
/	Pop 9000 from stack	[2, 300]
	Pop 300 from stack	[2]
	Push 9000/300=30 to stack	[2, 30]
-	Pop 30 from stack	[2]
	Pop 2 from stack	[]
	Push 30-2=28 to stack	[28]
Result : 28		

Postfix ifadenin değerlendirilmesi

Input: Postfix expression: **A B +**
Output: Infix expression- **(A + B)**

Input: Postfix expression: **ABC/-AK/L-***
Output: Infix expression: **((A-(B/C))*((A/K)-L))**

Algorithm:

Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator,
 1. pop an operand from the stack, say it's s1.
 2. pop an operand from the stack, say it's s2.
 3. perform **(s2 operator s1)** and push it to stack.
3. Once the expression iteration is completed, The stack will have the final result. Pop from the stack and return the result.

Örn:Postfix ifadenin değerlendirmesi

Postfix Expression : 2536+**5/2-		
Token	Action	Stack
2	Push 2 to stack	[2]
5	Push 5 to stack	[2, 5]
3	Push 3 to stack	[2, 5, 3]
6	Push 6 to stack	[2, 5, 3, 6]
+	Pop 6 from stack	[2, 5, 3]
	Pop 3 from stack	[2, 5]
	Push 3+6 =9 to stack	[2, 5, 9]
*	Pop 9 from stack	[2, 5]
	Pop 5 from stack	[2]
	Push 5*9=45 to stack	[2, 45]
*	Pop 45 from stack	[2]
	Pop 2 from stack	[]
	Push 2*45=90 to stack	[90]
5	Push 5 to stack	[90, 5]
/	Pop 5 from stack	[90]
	Pop 90 from stack	[]
	Push 90/5=18 to stack	[18]
2	Push 2 to stack	[18, 2]
-	Pop 2 from stack	[18]
	Pop 18 from stack	[]
	Push 18-2=16 to stack	[16]
Result : 16		

Infix-Postfix Dönüşümü

Input: Infix expression - $A + B$
Output: Postfix expression - $AB+$

Input: Infix expression - $A+B*(C^D-E)$
Output: Postfix expression - $ABCD^E-*+$

Approach: Use Stacks

- **Operator stack:** This stack will be used to keep operations (+, -, *, /, ^)

Order of precedence of operations–

1. ^ (Exponential)
2. / *
3. + –

Note: brackets () are used to override these rules.

Algorithm:

Initialize result as a blank string, iterate through given expression, one character at a time

1. If the character is an operand, add it to the result.
2. If the character is an operator.
 - If the operator stack is empty then push it to the operator stack.
 - Else If the operator stack is not empty,
 - If the operator's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.
 - If the operator's precedence is less than the precedence of the stack top of operator stack then *"pop out an operator from the stack and add it to the result until the stack is empty or operator's precedence is greater than or equal to the precedence of the stack top of operator stack"*. then push the operator to stack.
3. If the character is "(", then push it onto the operator stack.
4. If the character is ")", then *"pop out an operator from the stack and add it to the result until the corresponding "(" is encountered in operator stack. Now just pop out the "("*.

Once the expression iteration is completed and the operator stack is not empty, *"pop out an operator from the stack and add it to the result"* until the operator stack is empty. The result will be our answer, postfix expression.

Örn: Infix-Postfix Dönüşümü

Infix Expression : A+B*(C^D-E)				
Token	Action	Result	Stack	Notes
A	Add A to the result	A		
+	Push + to stack	A	+	
B	Add B to the result	AB	+	
*	Push * to stack	AB	* +	* has higher precedence than +
(	Push (to stack	AB	{ * +	
C	Add C to the result	ABC	{ * +	
^	Push ^ to stack	ABC	^ { * +	
D	Add D to the result	ABCD	^ { * +	
-	Pop ^ from stack and add to result	ABCD^	{ * +	- has lower precedence than ^
	Push - to stack	ABCD^	- { * +	
E	Add E to the result	ABCD^E	- { * +	
)	Pop - from stack and add to result	ABCD^E-	{ * +	Do process until (is popped from stack
	Pop { from stack	ABCD^E-	* +	
	Pop * from stack and add to result	ABCD^E-*	+	Given expression is iterated, do Process till stack is not Empty, It will give the final result
	Pop + from stack and add to result	ABCD^E-*+		
Postfix Expression : ABCD^E-*+				

Infix-Prefix Dönüşümü

Input: Infix expression - $A + B$
Output: Prefix expression- $+AB$

Input: Infix expression - $A+B*(C^D-E)$
Output: Prefix expression- $+A*B-^CDE$

Approach: Use Stack

- Operator stack: This stack will be used to keep operations (+, -, *, /, ^)

Order of precedence of operations–

1. ^ (Exponential)
2. / *
3. + –

Note: brackets () are used to override these rules.

Algorithm:

- Reverse the given infix expression. (Note: do another reversal only for brackets).
- Do Infix to postfix expression and get the result.
- Reverse the result to get the final expression. (prefix expression) .

Infix – Prefix dönüşümü Stack ile doğrudan yapılmaz, önce ifade tersten yazılıp Infix – Postfix dönüşümü ile dolaylı olarak yapılır.

Örn: Infix-Prefix Dönüşümü

Infix Expression : $A+B*(C^D-E)$				
Reverse Infix expression: $)E-D^C(*B+A$				
Reverse brackets: $(E-D^C)*B+A$				
Token	Action	Result	Stack	Notes
(	Push (to stack		(	
E	Add E to the result	E	(	
-	Push - to stack	E	(-	
D	Add D to the result	ED	(-	
^	Push ^ to stack	ED	(- ^	
C	Add C to the result	EDC	(- ^	
)	Pop ^ from stack and add to result	EDC^	(-	Do process until (is popped from stack
	Pop - from stack and add to result	EDC^-	(	
	Pop (from stack	EDC^-		
*	Push * to stack	EDC^-	*	
B	Add B to the result	EDC^-B	*	
+	Pop * from stack and add to result	EDC^-B		- has lower precedence than ^
	Push + to stack	EDC^-B*	+	
A	Add A to the result	EDC^-B*A	+	
	Pop + from stack and add to result	EDC^-B*A+		Given expression is iterated, do Process till stack is not Empty, It will give the final result
Prefix Expression (Reverse Result): $+A*B-^CDE$				

Postfix-Infix Dönüşümü

Input: Postfix expression: **A B +**

Output: Infix expression- **(A + B)**

Input: Postfix expression: **ABC/-AK/L-***

Output: Infix expression: **((A-(B/C))*((A/K)-L))**

Algorithm:

Iterate the given expression from left to right, one character at a time

1. If a character is operand, push it to stack.
2. If a character is an operator,
 1. pop operand from the stack, say it's s1.
 2. pop operand from the stack, say it's s2.
 3. perform **(s2 operator s1)** and push it to stack.
3. Once the expression iteration is completed, initialize the result string and pop out from the stack and add it to the result.
4. Return the result.

Örn: Postfix-Infix Dönüşümü

Postfix Expression : ABC/-AK/L-*			
Token	Action	Stack	Notes
A	Push A to stack	[A]	
B	Push B to stack	[A, B]	
C	Push C to stack	[A, B, C]	
/	Pop C from stack	[A, B]	Pop two operands from stack, C and B. Perform B/C and push (B/C) to stack
	Pop B from stack	[A]	
	Push (B/C) to stack	[A, (B/C)]	
-	Pop (B/C) from stack	[A]	Pop two operands from stack, (B/C) and A. Perform A-(B/C) and push (A-(B/C)) to stack
	Pop A from stack	[]	
	Push (A-(B/C)) to stack	[[A-(B/C)]]	
A	Push A to stack	[[A-(B/C)), A]	
K	Push K to stack	[[A-(B/C)), A, K]	
/	Pop K from stack	[[A-(B/C)), A]	Pop two operands from stack, K and A. Perform A/K and push (A/K) to stack
	Pop A from stack	[[A-(B/C)]]	
	Push (A/K) to stack	[[A-(B/C)), (A/K)]	
L	Push L to stack	[[A-(B/C)), (A/K), L]	
-	Pop L from stack	[[A-(B/C)), (A/K)]	Pop two operands from stack, L and (A/K). Perform (A/K)-L and push ((A/K)-L) to stack
	Pop (A/K) from stack	[[A-(B/C)]]	
	Push ((A/K)-L) to stack	[[A-(B/C)), ((A/K)-L)]	
*	Pop ((A/K)-L) from stack	[[A-(B/C)]]	Pop two operands from stack, (A/K)-L and A-(B/C). Perform (A-(B/C))*((A/K)-L) and push ((A-(B/C))*((A/K)-L)) to stack
	Pop (A-(B/C)) from stack	[]	
	Push ((A-(B/C))*((A/K)-L)) to stack	[[A-(B/C))*((A/K)-L)]]	
Infix Expression: ((A-(B/C))*((A/K)-L))			

Prefix-Infix Dönüşümü

Example:

Input: Prefix expression: + A B

Output: Infix expression- (A + B)

Input: Prefix expression: *-A/BC-/AKL

Output: Infix expression: ((A-(B/C))*((A/K)-L))

Approach: Use [Stacks](#)

Algorithm:

Iterate the given expression from right to left (in reverse order), one character at a time

1. If character is operand, push it to stack.
2. If character is operator,
 1. pop operand from stack, say it's s1.
 2. pop operand from stack, say it's s2.
 3. perform (s1 operator s2) and push it to stack.
3. Once the expression iteration is completed, initialize result string and pop out from stack and add it to result.
4. Return the result.

Örn: Prefix-Infix Dönüşümü

Prefix Expression : *-A/BC-/AKL			
Iterate right to left			
Token	Action	Stack	Notes
L	Push L to stack	[L]	
K	Push K to stack	[L, K]	
A	Push A to stack	[L, K, A]	
/	Pop A from stack	[L, K]	Pop two operands from stack, A and K. Perform A/K and push (A/K) to stack
	Pop K from stack	[L]	
	Push (A/K) to stack	[L, (A/K)]	
-	Pop (A/K) from stack	[L]	Pop two operands from stack, (A/K) and L. Perform (A/K)-L and push ((A/K)-L) to stack
	Pop L from stack	[]	
	Push ((A/K)-L) to stack	(((A/K)-L))	
C	Push C to stack	(((A/K)-L), C)	
B	Push B to stack	(((A/K)-L), C, B)	
/	Pop B from stack	(((A/K)-L), C)	Pop two operands from stack, B and C. Perform B/C and push (B/C) to stack
	Pop C from stack	(((A/K)-L))	
	Push (B/C) to stack	(((A/K)-L), (B/C))	
A	Push A to stack	(((A/K)-L), (B/C), A)	
-	Pop A from stack	(((A/K)-L), (B/C))	Pop two operands from stack, A and (B/C). Perform A-(B/C) and push (A-(B/C))to stack
	Pop (B/C) from stack	(((A/K)-L))	
	Push (A-(B/C)) to stack	(((A/K)-L), (A-(B/C)))	
*	Pop (A-(B/C)) from stack	(((A/K)-L))	Pop two operands from stack, (A-(B/C) and ((A/K)-L). Perform (A-(B/C))*((A/K)-L) and push ((A-(B/C))*((A/K)-L)) to stack
	Pop ((A/K)-L) from stack	[]	
	Push ((A-(B/C))*((A/K)-L)) to stack	(((A-(B/C))*((A/K)-L)))	
Infix Expression: ((A-(B/C))*((A/K)-L))			

Algoritma Analizi



Algoritma Analizi

- ❑ İyi bir algoritmanın kriteri nedir
 - Hız - Süre : time efficiency
 - Bellek : space efficiency

- ❑ Daha iyi bir algoritma var mı?
 - Alt sınırlar
 - optimallik

Algoritma analizi

□ Konular

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

□ Yaklaşımlar

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

Bazı önemli problemler

- ❑ Sıralama - Sorting
- ❑ Arama - Searching
- ❑ Metin işleme - String processing
- ❑ Çizge problemleri - Graph problems
- ❑ Kombinasyonel - Combinatorial problems
- ❑ Geometrik - Geometric problems
- ❑ Sayısal - Numerical problems

Kaba Analiz

- Bir giriş verisi seçilir
- Zaman birimini seç (milisaniye)
veya
Yürütülen temel işlem adımlarının sayısı
- Deneyssel veriler ile analiz yapılır

Teorik Analiz

Zaman verimliliği, girdi boyutunun bir fonksiyonu olarak temel işlemin tekrar sayısı belirlenerek analiz edilir.

Temel İşlem (Basic operation): Algoritmanın çalışma süresini en çok etkileyen işlemdir. Sıralamada temel işlemdir. ??

Problemler - Temel işlemler

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

Temel Kavramlar

Yürütme Zamanı (Running Time) $T(n)$

- Algoritmanın işlevini yerine getirebilmesi için kabul edilen işlemlerden kaç adet yürütülmesi gerektiğini gösteren matematiksel bir ifadedir.
- İşlem olarak karşılaştırma sayısı, çevrim sayısı, aritmetik işlem sayısı kabul edilir
- $T(n)$ hesaplanırken hangi işleme göre hesaplandığı bildirilmelidir

Alan Maliyeti (Space Cost)

- Algoritmanın işlevini yerine getirebilmesi için gerekli olan bellek ihtiyacıdır.
- Maliyet programın kodu, veri yapısı alanları ve yığın bellek alanıdır
- Alan maliyeti algoritma rekürsif değilse, veri yapısı alanı çok çok büyük değilse pek hesaplanmaz

Asimptotik İfade

- Bir problemin büyümesinin üst, alt ve ortalama sınırını belirleyen bir ifadedir
- Big O $\rightarrow O$, Big Omega $\rightarrow \Omega$, Big Theta $\rightarrow \theta$, Little O $\rightarrow o$ ve
- Little Omega $\rightarrow \omega$

Alan Karmaşıklığı (Space Complexity)

- Eleman sayısı n 'nin büyük değerleri için bellek alanı gereksiniminin artışını gösteren asimptotik bir ifadedir.

Zaman Karmaşıklığı (Time Complexity)

- Algoritmanın yürütülürken geçecek zamanın artması hakkında bilgi veren asimptotik bir ifadedir.
- $O(g(n))$, $\theta(g(n))$, $\Omega(g(n))$, $o(g(n))$ gösterilir
- Veri kümesinin büyümesi durumunda çalışma süresinin nasıl etkileneceği hakkında bilgi verir

Yürütme Zamanı-Running Time $T(n)$

$T(n) = 2n^2 - 2n + 5$ olabilir.

n , ifadenin bağımsız değişkenidir ve temel işlem sayısına bağlıdır.

$T(n) = 1$	sabit
$T(n) = \log n$	logaritmik
$T(n) = n$	lineer
$T(n) = n^2$	karesel

Örnek

```
float BulOrta (float A[], int n)
```

```
{
```

```
    float ortalama, toplam=0;  $\longrightarrow$  toplam=0 1 işlem
```

```
    int k;
```

```
    for (k=0; k<n; k++)  $\longrightarrow$ 
```

k=0 **1 kere**

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

k++ **n kere**

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

```
        toplam=toplam + A[k];  $\longrightarrow$ 
```

} atama ve toplama 2 işlem } 2n
döngü içerisinde n kere

```
    ortalama = toplam/n;  $\longrightarrow$ 
```

```
    return ortalama;
```

```
}
```

1 işlem

Döngü dışında bölme ve atama **2 işlem**

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

$$T(n) = \mathbf{4n + 6}$$

Örnek

```
float BulEnkucuk (float A[ ])
```

```
{
```

```
    float enkucuk;
```

```
    int k;
```

```
    enkucuk=A[0];  atama 1 işlem
```

```
    for (k=1; k<n; k++) 
```

```
        if (A[k] < enkucuk) 
```

```
            enkucuk=A[k]; 
```

```
    return enkucuk; 1 işlem
```

```
}
```

k=1 **1 kere**

k<n **n kere**

k++ **n-1 kere**

} 1+n+(n-1)=2n

karşılaştırma 1 işlem **n-1 kere**

Bu işlemin kaç kez yürütüleceği belli değil,
en kötü durumda **n-1 kere**

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

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

```

public static int [] selectionsort(int [] A,int n)
{
    int tmp;
    int min;

    for(int i=0; i < n-1; i++) I
    {
        min=i; II

        for(int j=i; j < n; j++) III
        {
            if (A[j] < A[min]){ IV

                min=j; V
            }

        }
        tmp=A[i];
        A[i]=A[min]; VI
        A[min]=tmp;
    }
    return A; VII
}

```

	İşlem	Tekrar	Toplam
I	1,1,1	1,n,n	2n+1
II	1	n	n
III	1,1,1	n, n(n+1)/2, n(n+1)/2	n+2n(n+1)/2
IV	1	n(n+1)/2	n(n+1)/2
V	1	n(n+1)/2	n(n+1)/2
VI	1,1,1	1,1,1	3
VII	1	1	1
		$4n(n+1)/2 + 4n + 5 = 2n^2 + 6n + 5$	

$$T(n) = 2n^2 + 6n + 5$$

Matris Toplamı

for (i=0; i<n; i++) $\longrightarrow 1+(n+1) + n = 2n + 2$
 for (j=0; j<m; j++) $\longrightarrow (1+(m+1) + m) * n = (2m + 2)n$
 C[i][j] = A[i][j] + B[i][j]; $\longrightarrow (2 * m * n)$

$$4mn + 4n + 2$$

m ve n eşit alırsak $T(n) = 4n^2 + 4n + 2$

Fonksiyonların Büyümesi ve Asimptotik Notasyonlar

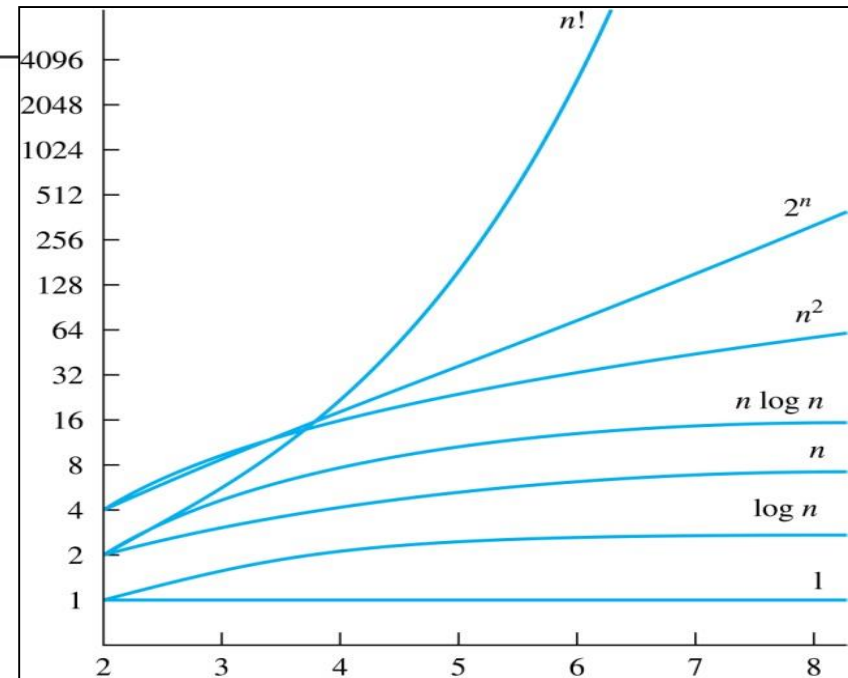
- Hem bilgisayar biliminde hem de matematikte, bir fonksiyonun ne kadar hızlı büyüdüğünü sıklıkla inceleriz
- Bilgisayar biliminde, girdinin boyutu büyüdükçe, bir algoritmanın bir sorunu ne kadar hızlı çözebileceğini anlamak isteriz.
 - Aynı problemi çözmek için iki farklı algoritmanın verimliliğini karşılaştırabiliriz
 - Girdi büyüdükçe belirli bir algoritmayı kullanmanın pratik olup olmadığını da belirleyebiliriz

Karmaşıklık

- ❑ Bir algoritmanın üzerinde işlem yapacağı veri kümesinin eleman sayısının artması durumunda yürütme zaman maliyetinin nasıl değişeceğini gösteren asimptotik bir ifadedir.
 - Yürütme maliyetine, zaman karmaşıklığı
 - Bellek kullanım maliyetine, alan karmaşıklığı
- ❑ Karmaşıklıkta ki amaç, gerçek zaman ve bellek büyüklüğü bilgisi değil, veri kümesi büyüdüğünde maliyet bilgisinin değişimidir.

$n \rightarrow \infty$ bazı önemli fonksiyonların büyüme hızı

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		



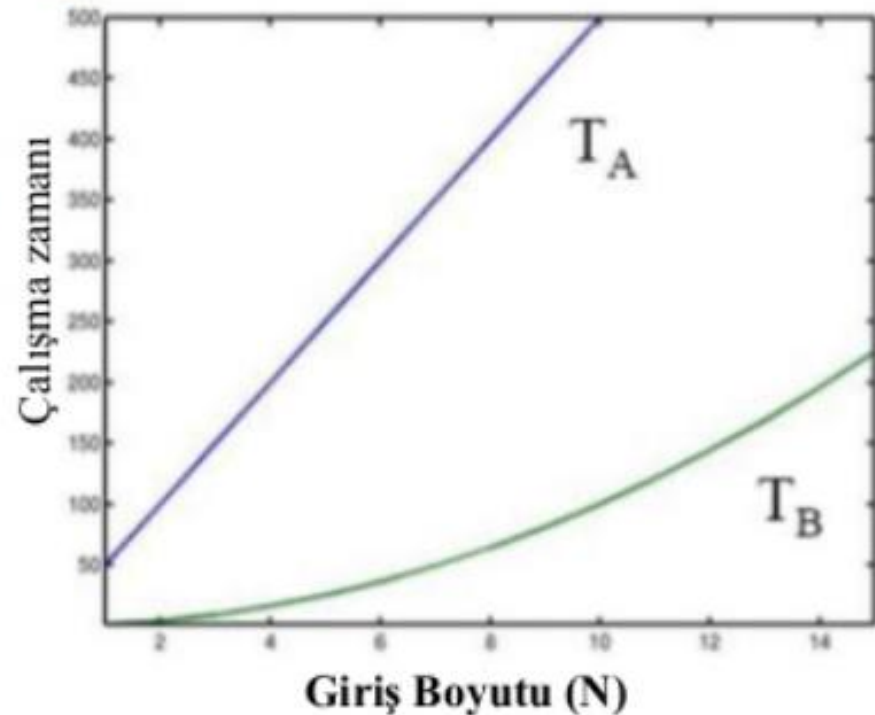
Algoritma Analiz Türleri

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

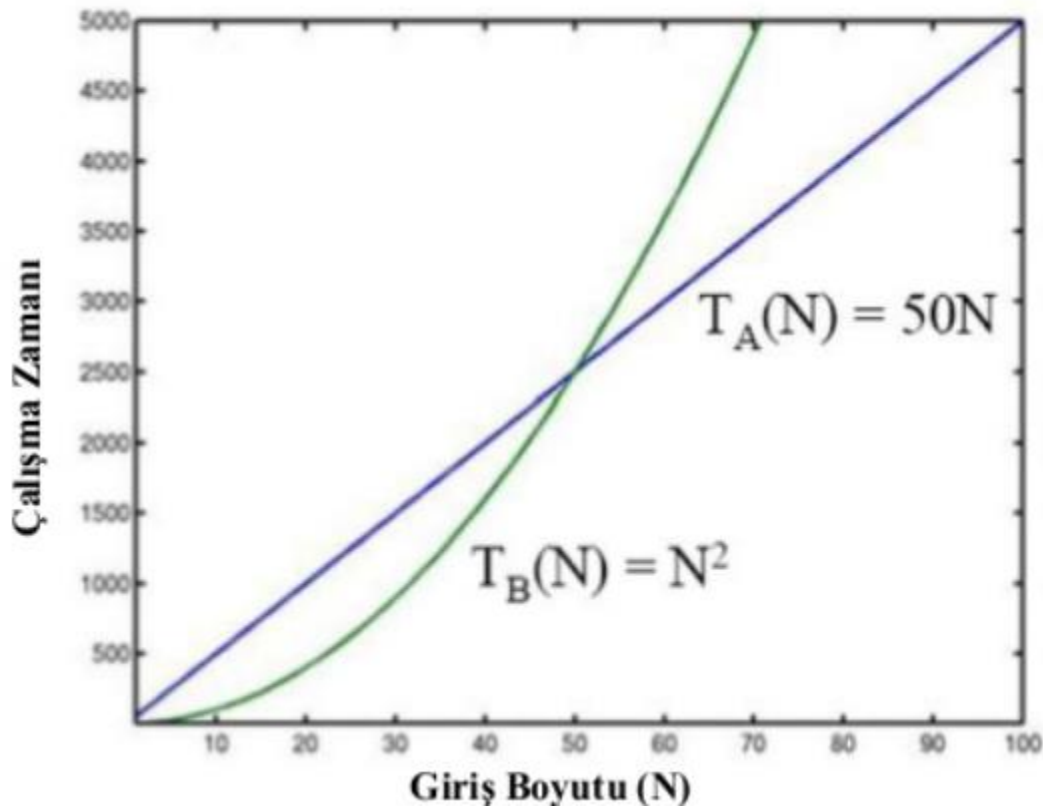
-
- Bir algoritmanın **genelde EN KÖTÜ** durumdaki çalışma zamanına bakılır. **Neden?**
 - En kötü durum çalışma zamanında bir üst sınırdır ve o algoritma için verilen durumdan daha uzun sürmeyeceği garantisi verir.
 - Bazı algoritmalar için en kötü durum oldukça sık rastlanır. Arama algoritmasında,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

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

Hangi algoritmayı seçersiniz?



- N büyüdüğü zaman A ve B nin çalışma zamanı:



**Şimdi hangi
algoritmayı
seçersiniz?**

- Asimptotik notasyon, **eleman sayısı n'nin sonsuza gitmesi durumunda** algoritmanın, **benzer işi yapan algoritmalarla karşılaştırmak** için kullanılır.
- Eleman sayısının *küçük olduğu durumlar* mümkün olabilir fakat bu *birçok uygulama için geçerli değildir*.
- Verilen iki algoritmanın çalışma zamanını $T_1(N)$ ve $T_2(N)$ fonksiyonları şeklinde gösterilir. Hangisinin **daha iyi** olduğunu belirlemek için bir *yol belirlememiz* gerekiyor.
 - Big-O (Big O): Asimptotik üst sınır
 - Big Ω (Big Omega): Asimptotik alt sınır
 - Big Θ (Big Teta): Asimptotik alt ve üst sınır

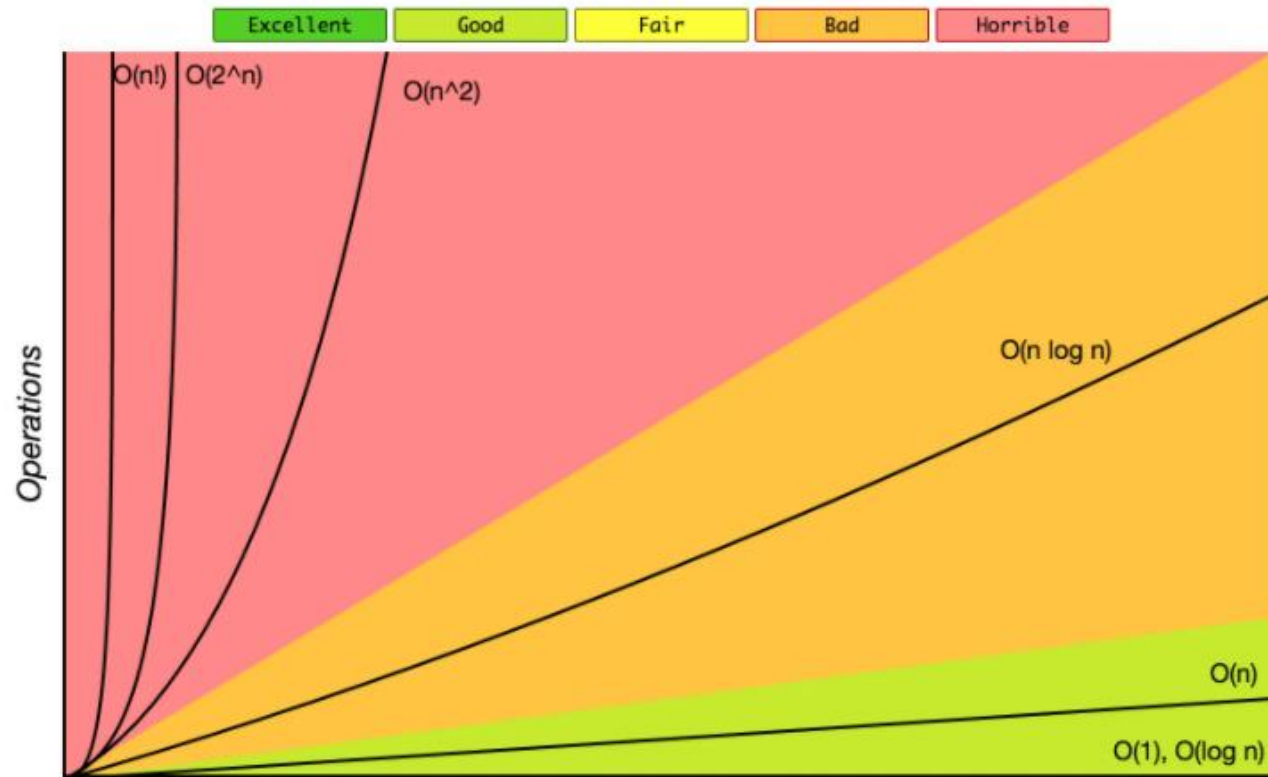
$f(x)$, bir algoritmanın fonksiyon şeklindeki gösterimi ise karmaşıklık $O(f(x))$, $\Omega(f(x))$, ... şeklinde gösterilir.

Büyük- O (Big-O) Gösterimi

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

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

	n=1	n=2	n=4	n=8	n=16	n=32
1	1	1	1	1	1	1
$\log n$	0	1	2	3	4	5
n	1	2	4	8	16	32
$n \log n$	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
$n!$	1	2	24	40320	20.9T	Don't ask!



Örnek

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$

Since when $x > 1$, $x < x^2$ and $1 < x^2$

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

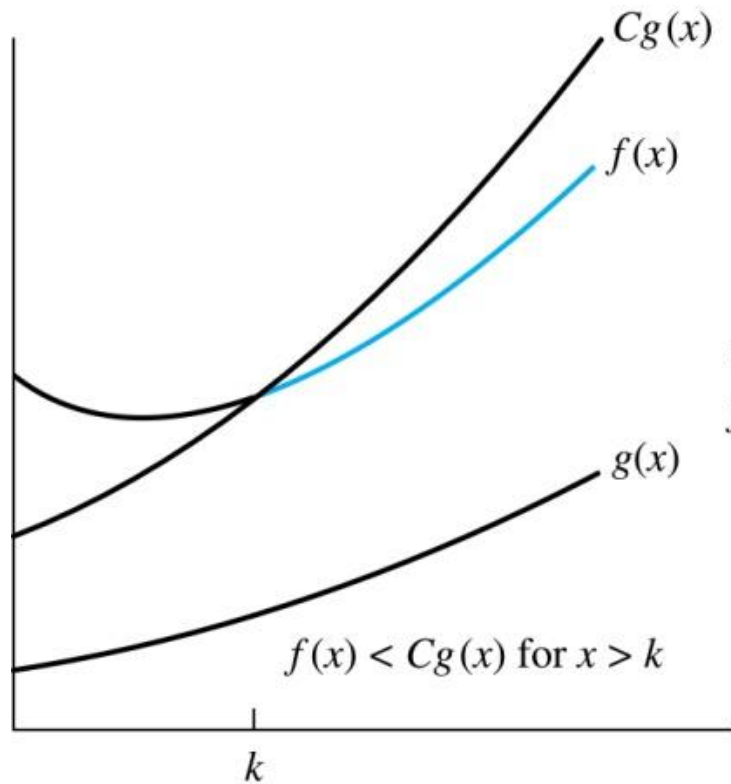
Can take $C = 4$ and $k = 1$ as witnesses to show that

$f(x)$ is $O(x^2)$

Alternatively, when $x > 2$, we have $2x \leq x^2$ and $1 < x^2$.

Hence, $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$
when $x > 2$.

- Can take $C = 3$ and $k = 2$ as witnesses instead.



$f(x)$ is $O(g(x))$

The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

ALGORITHM *SequentialSearch*($A[0..n - 1]$, K)

//Searches for a given value in a given array by sequential search

//Input: An array $A[0..n - 1]$ and a search key K

//Output: The index of the first element of A that matches K

// or -1 if there are no matching elements

$i \leftarrow 0$

while $i < n$ **and** $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

❑ Worst case $O(n)$

❑ Best case $O(1)$

❑ Average case $\frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2n} = \frac{(n+1)}{2}$

Ordering Functions by Order of Growth

- Put the functions below in order so that each function is big-O of the next function on the list.

We solve this exercise by successively finding the function that grows slowest among all those left on the list.

- $f_1(n) = (1.5)^n$
 - $f_2(n) = 8n^3 + 17n^2 + 111$
 - $f_3(n) = (\log n)^2$
 - $f_4(n) = 2^n$
 - $f_5(n) = \log(\log n)$
 - $f_6(n) = n^2(\log n)^3$
 - $f_7(n) = 2^n(n^2 + 1)$
 - $f_8(n) = n^3 + n(\log n)^2$
 - $f_9(n) = 10000$
 - $f_{10}(n) = n!$
- $f_9(n) = 10000$ (constant, does not increase with n)
 - $f_5(n) = \log(\log n)$ (grows slowest of all the others)
 - $f_3(n) = (\log n)^2$ (grows next slowest)
 - $f_6(n) = n^2(\log n)^3$ (next largest, $(\log n)^3$ factor smaller than any power of n)
 - $f_2(n) = 8n^3 + 17n^2 + 111$ (tied with the one below)
 - $f_8(n) = n^3 + n(\log n)^2$ (tied with the one above)
 - $f_1(n) = (1.5)^n$ (next largest, an exponential function)
 - $f_4(n) = 2^n$ (grows faster than one above since $2 > 1.5$)
 - $f_7(n) = 2^n(n^2 + 1)$ (grows faster than above because of the $n^2 + 1$ factor)
 - $f_{10}(n) = n!$ ($n!$ grows faster than c^n for every c)

Büyük- Ω (Big-*Omega*) Gösterimi

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 $f(x) = \Omega(g(x))$ olacaktır.

Example: Show that $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$ where $g(x) = x^3$.

Solution: $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers x .

- Is it also the case that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$?

Büyük- θ (Big-*Theta*) Gösterimi

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 $f(x)$, $\Omega(g(x))$ ise $f(x)$, $\theta(g(x))$ 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 $\theta(g(x))$ olduğunu gösterebiliriz.

Big-Theta Notation

Example: Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$.

Solution:

- $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$,
since $0 \leq 8x \log x \leq 8x^2$.
 - Hence, $3x^2 + 8x \log x$ is $O(x^2)$.
- x^2 is clearly $O(3x^2 + 8x \log x)$
- Hence, $3x^2 + 8x \log x$ is $\Theta(x^2)$.

Big-Theta Notation

When $f(x)$ is $\Theta(g(x))$ it must also be the case that $g(x)$ is $\Theta(f(x))$.

Note that $f(x)$ is $\Theta(g(x))$ if and only if it is the case that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

Sometimes writers are careless and write as if big- O notation has the same meaning as big-Theta.

Birçok algoritma birden fazla alt programdan oluşabilir.

$$\left. \begin{array}{l} f1 \rightarrow O(n^c) \\ f2 \rightarrow O(n^d) \end{array} \right\} \max(O(n^c), O(n^d)) \rightarrow O(n^d) \\ 1 < c < d \text{ ise}$$

$$\left. \begin{array}{l} f1 \rightarrow O(\log n) \\ f2 \rightarrow O(n) \end{array} \right\} \max(O(\log n), O(n)) \rightarrow O(n)$$

$$\left. \begin{array}{l} f1 \rightarrow O(2^n) \\ f2 \rightarrow O(n) \end{array} \right\} \max(O(2^n), O(n)) \rightarrow O(2^n)$$

$$\left. \begin{array}{l} f1 \rightarrow O(n^2) \\ f2 \rightarrow O(n^2) \end{array} \right\} \rightarrow O(n^2)$$

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

n , pozitif bir tamsayı olmak üzere Big-O ?

$$\begin{array}{ccc} 3n \log(n!) & \rightarrow & 3n \quad \text{ve} \quad \log(n!) \\ \downarrow & & \downarrow \\ O(n) & & O(\log n^n) \\ & \searrow & \swarrow \\ & & O(n \log n) \\ & \searrow & \swarrow \\ & & O(n^2 \log n) \end{array}$$

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

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

$$3x^2 \rightarrow O(3x^2) \rightarrow O(x^2)$$

$$\max(O(x \log), O(x^2)) \rightarrow O(x^2)$$

Big-Theta Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ where $a_0, a_1, \dots, a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is of order x^n (or $\Theta(x^n)$).

(The proof is an exercise.)

Example:

The polynomial $f(x) = 8x^5 + 5x^2 + 10$ is order of x^5 (or $\Theta(x^5)$).

The polynomial $f(x) = 8x^{199} + 7x^{100} + x^{99} + 5x^2 + 25$ is order of x^{199} (or $\Theta(x^{199})$).

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

Big-oh

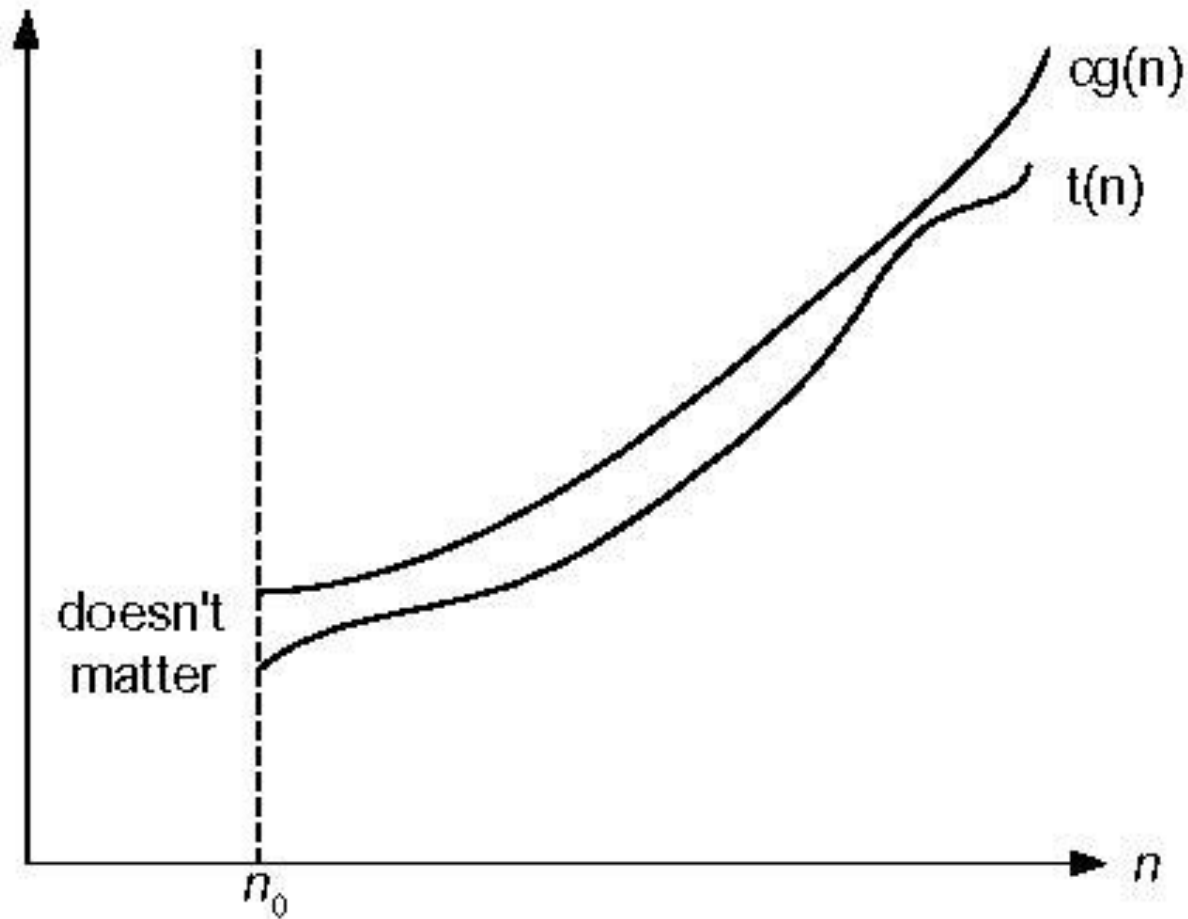


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

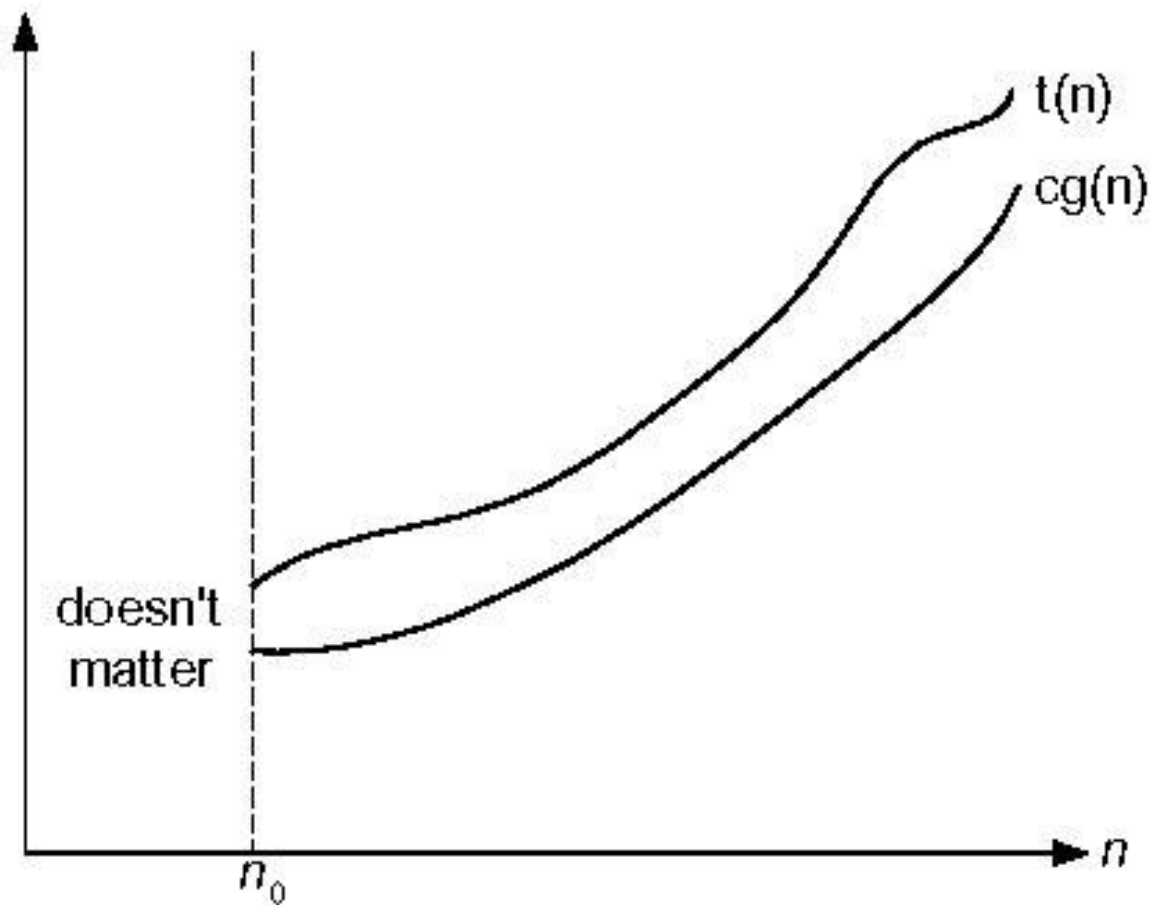


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

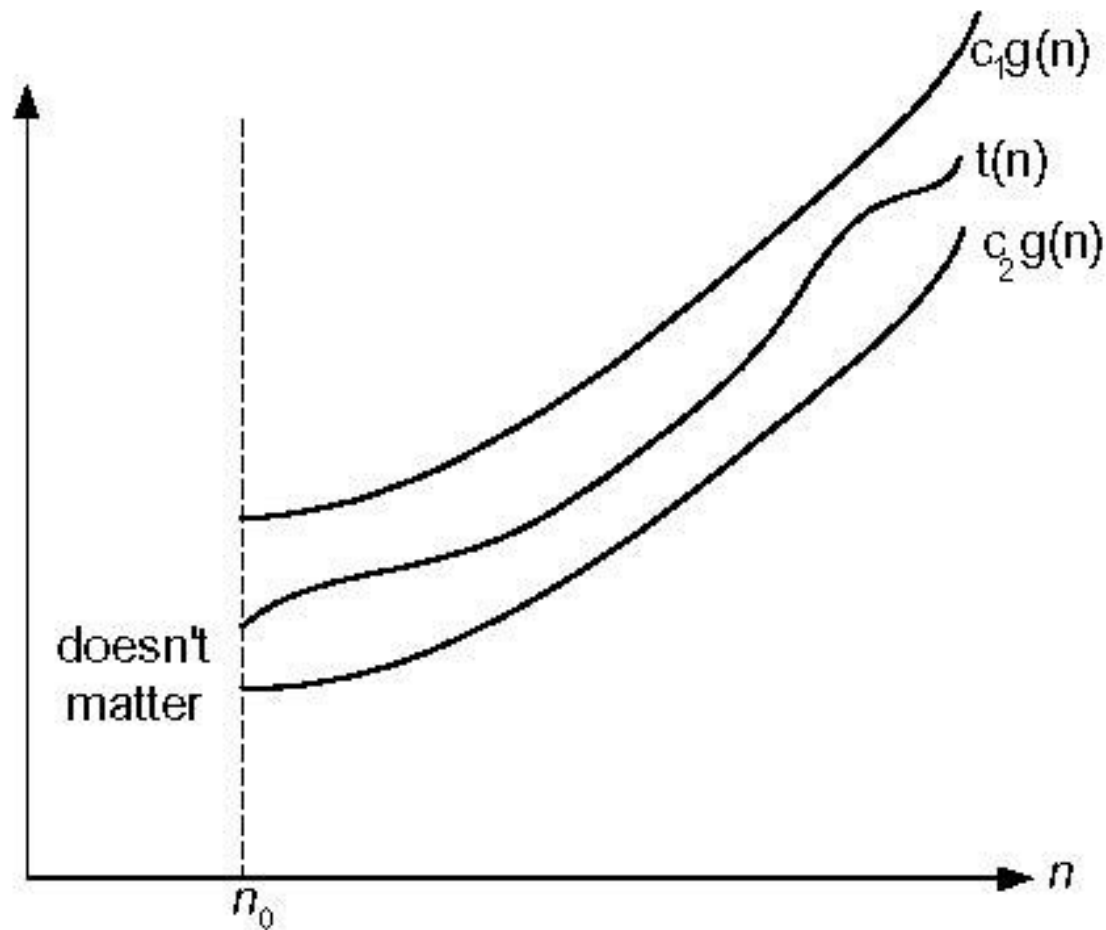


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using limits

$$\lim_{n \rightarrow \infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n), T(n) \in O(g(n)) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n), T(n) \in \Theta(g(n)) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n), T(n) \in \Omega(g(n)) \end{cases}$$

L'Hôpital's rule: If $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$ and the derivatives f' , g' exist, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

Example: $\log n$ vs. n

Example: 2^n vs. $n!$

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)$

* Ex:

$$f(x) = 4x^3 + 3x^2, \quad g(x) = x^4$$

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \frac{4x^3 + 3x^2}{x^4} = \lim_{x \rightarrow \infty} \frac{4x^3}{x^4} + \lim_{x \rightarrow \infty} \frac{3x^2}{x^4}$$

$$\underbrace{\lim_{x \rightarrow \infty} \frac{4}{x}}_{\emptyset} + \underbrace{\lim_{x \rightarrow \infty} \frac{3}{x^2}}_{\emptyset} = \emptyset$$

$L = \emptyset$ aktüjü lön,

$$4x^3 + 3x^2 \in O(x^4)$$

veya

$$4x^3 + 3x^2 \leq C \cdot (x^4) \text{ seçilgen}$$

$C=7$ gibibir sayı verdur. $\therefore O(x^4)$ olur.

Ex | $n > 6$ ve n bir tam sayı iken $3n < n!$ i spotlayın.

Tümevarım olabilir veya Big O kullabilir.

1) $n = \{7, 8, 9, \dots\}$ kısmından;

$n=7$ için

$$3 \cdot 7 < 7! \quad \text{doğru}$$

2) $n=k$ doğru kabul edilir.

$$3k < k! \quad \text{doğru kabul edilir.}$$

3) $n=k+1$

$$3(k+1) < (k+1)! \quad \text{doğruluğu gösterilir,}$$

$$3k+3 < (k+1) \cdot k!$$

$$3k+3 < k \cdot k! + k!$$

$k! > 3k$ $k > 6$ iken
doğru kabul edildi

Burada $k \cdot k! > 3$ ispat edilmeli $k > 6$ iken bu
doğrudur.

$\therefore 3n < n!$ doğrudur.

İteratif ve Özyinelemeli Algoritmaların Analizi



İteratif (nonrecursive) algoritmaların analizi

Genel Adımlar:

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

Selection Sort (Seçmeli Sıralama)

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

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

    /* check the element to be minimum */

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

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

end procedure
```

17	33	14	27	18
0	1	2	3	4

Swap Yapılır. (**i = 0**)

17	33	14	27	18
i = 0	1	2	3	4

14	33	17	27	18
i = 0	1	2	3	4

Swap Yapılır. (**i = 1**)

14	33	17	27	18
----	----	----	----	----

14	17	33	27	18
----	----	----	----	----

Swap Yapılır. (**i = 2**)

14	17	33	27	18
0	1	2	3	4

14	17	18	27	33
0	1	2	3	4

Swap Yapılır. (**i = 3**)

14	17	18	27	33
0	1	2	3	4

```

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

```

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

Total instructions (all iterations of inner loop for all values of j)
 $T(N) = (N-1) + (N-2) + \dots + 2 + 1 =$
 $= [N * (N-1)] / 2 \rightarrow N^2 \text{ order of magnitude}$
 Note that the came from the summation NOT because 'there is an N in the inner loop' (NOT because $N * N$).

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

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

best_case : $O(n^2)$

Öz yinelemeli (Recursive) Algoritmaların Analizi

- ❑ n girdi boyutu (*input size*) belirlenir
- ❑ Algoritmanın temel operasyonu saptanır (*basic operation*)
- ❑ Temel işlemin gerçekleştirilme sayısının aynı boyuttaki farklı girişlere göre değişiklik gösterip göstermediği kontrol edilir (Eğer değişiklik gösterirse, durum analizi ayrı ayrı the worst, average ve best cases yapılmalıdır)
- ❑ Temel işlemin kaç kez yürütüldüğünü ifade eden uygun bir başlangıç koşulu ile bir özyineleme bağıntısı (recurrence relation) kurulur
- ❑ Recurrence relation uygun bir yöntemle çözülür. Çözüm sonucu elde edilen n 'e bağlı fonksiyon $T(n)$, asimptotik notasyonlara göre ifade edilir

Öz yinelemeli (Recursive) bağıntılarının çözümü

- n bağımsız değişkene göre ifade edilen ve n değerinin önceki değerlerini içinde bulunduran fonksiyonlardır

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

- Çözüm yöntemleri
 - Forward Substitution
 - Backward Substitution 
 - Recursive Tree Method
 - Master Theorem
 - Karakteristik Denklem Yöntemi (2. veya yüksek dereceden bağıntılar için)

Forward Substitution

- Recurrence: $T(n) = T(n-1) + 1$, with initial condition $t(1) = 2$
- Look for a pattern:
 - $T(1) = 2$, Initial condition
 - $T(2) = T(1) + 1 = 2 + 1 = 3$
 - $T(3) = T(2) + 1 = 3 + 1 = 4$
 - $T(4) = T(3) + 1 = 4 + 1 = 5$
 - $T(5) = T(4) + 1 = 5 + 1 = 6$
- Guess:
 - $T(n) = n + 1$
- Informal Check:
 - $T(n) = T(n-1) + 1 = [(n-1) + 1] + 1 = n + 1$
 - $T(1) = 1 + 1 = 2$

Örnek : 1'den 5'e kadar olan sayıları toplayalım.

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n = \frac{n(n + 1)}{2}$$



Gauss

1+	2+	3+	...	(n-2)+	(n-1)+	n
n+	(n-1)+	(n-2)+	..	3+	2+	1

$f(n)=n + f(n-1)$ $F(1)=1$ ilk değer

$f(5) = 5 + f(4)$ 10
15  $f(4) = 4 + f(3)$ 6
10  $f(3) = 3 + f(2)$ 3
7  $f(2) = 2 + f(1)$ 1
5  $f(1) = 1$

Örnek : n sayısının faktöriyelini hesaplayalım

factorial(n):

if n is 0

return 1

return n * factorial(n-1)

- factorial(0) sadece 1 karşılaştırma (1 unit of time)
- factorial(n) (**1** karşılaştırma, **1** multiplication, **1** subtraction) (n-1) kez

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

$$T(0) = 1$$

$$\begin{aligned}T(n) &= T(n-1) + 3 \\&= T(n-2) + 6 \\&= T(n-3) + 9 \\&= T(n-4) + 12 \\&= \dots \\&= T(n-k) + 3k\end{aligned}$$

as we know $T(0) = 1$

we need to find the value of k for which $n - k = 0$, $k = n$

$$\begin{aligned}T(n) &= T(0) + 3n, \quad k = n \\&= 1 + 3n\end{aligned}$$

that gives us a time complexity of $O(n)$

Örn: Backward Substitution (guessing and checking)

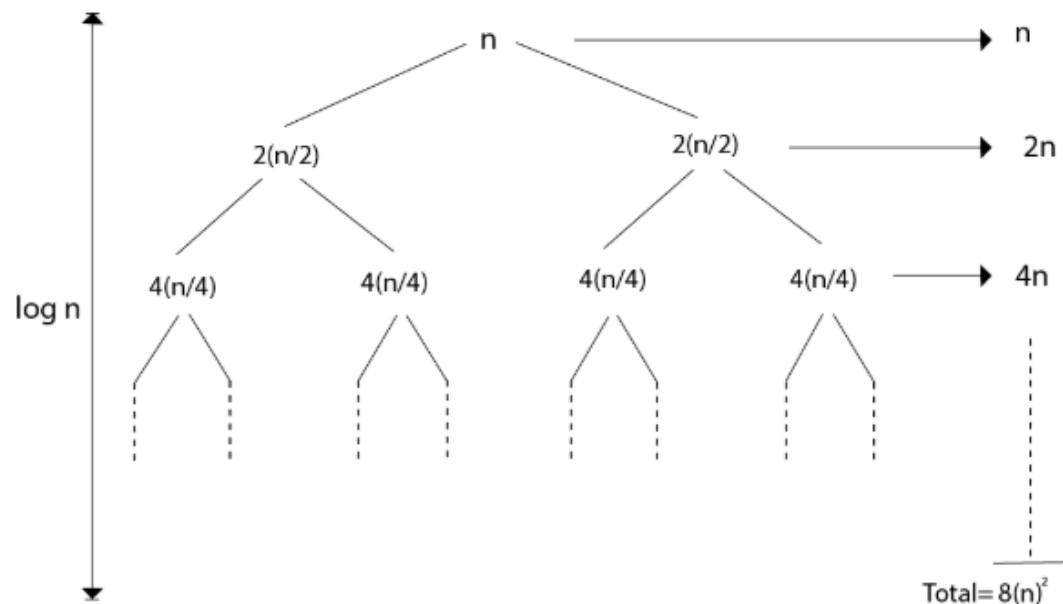
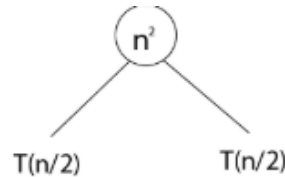
- Look for a pattern (how many 1's in each line?):
 - $T(n) = T(n-1) + 1$
 - $T(n) = [T(n-2)+1] + 1$
 - $T(n) = [[T(n-3)+1]+1] + 1$
 - $T(n) = [[[T(n-4)+1]+1]+1] + 1$
 - ...
 - $T(n) = [...[[T(n-k)+1]+1] ... +1] + 1$ [has k ones]
 - ... [Let $k = n-1$]
 - $T(n) = [...[[T(n-(n-1))+1]+1] ... +1] + 1$ [has $k=n-1$ ones]
 - $T(n) = T(n-(n-1)) + (n - 1)$
 - $T(n) = T(1) + (n-1)$
 - $T(n) = 2 + (n - 1)$
 - $T(n) = n + 1$
- This process leads to this Guess:
 - $T(n) = n + 1$
- Check the guess, by substituting, as in the previous example:
 - $T(n) = T(n-1)+1 = [(n-1)+1] + 1 = n+1$
 - $T(1) = 1+1 = 2$

Recursion Tree Method

- *Özyineleme Ağacı Yöntemi, her seviyede düğümlerin genişletildiği bir ağaç formundaki bir öz yineleme yönteminin grafiksel bir gösterimidir*
- Genel olarak özyinelemede ikinci terimi kök olarak kabul edilir.
- Böl ve Fethet algoritması kullanıldığında faydalıdır.

Örn: Recursion Tree Method

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



We have $n + 2n + 4n + \dots \log_2 n$ times

$$= n (1 + 2 + 4 + \dots \log_2 n \text{ times})$$

$$= n \frac{(2^{\log_2 n} - 1)}{(2 - 1)} = \frac{n(n - 1)}{1} = n^2 - n = \theta(n^2)$$

$$T(n) = \theta(n^2)$$

Master Theorem

- Cookbook way of solving recurrences of this form:
 - $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$
 - $T(1) = c$
 - $n = b^k$, for positive integer k
 - $a \geq 1, b \geq 2, c > 0, d \geq 0$ are constants
- $T(n) \in \begin{cases} \Theta(n^d), & a < b^d \\ \Theta(n^d \lg n), & a = b^d \\ \Theta(n^{\log_b a}), & a > b^d \end{cases}$
- We call these Case 1, Case 2, and Case 3.
- Intuition: Which term of recurrence dominates (ie contributes more)
- In the recurrence we can replace " $T(n)=$ " with " $T(n)\leq$ " or " $T(n)\geq$ " and get O or Ω performance

Örn: Master Theorem

- $T(n) = 16T(n/4) + 5n^3$
 - $a = 16, b = 4, d = 3$
 - $16 < 4^3$ and so Case 1
 - Thus, $T(n) = \Theta(n^d) = \Theta(n^3)$
- $T(n) = 2T(n/2) + n$
 - $a = 2, b = 2, f(n) = n, \log_b a = \lg 2 = 1$
 - Case 2: $f(n) = n = n^1 = \Theta(n^{\log_b a})$
 - Thus, $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$
- $T(n) = 2T(n/2) + n$
 - $a = 2, b = 2, d = 1$
 - $2 = 2^1$ and so Case 2
 - Thus, $T(n) = \Theta(n^1 \lg n)$
- $T(n) = 8T(n/2) + n$
 - $a = 8, b = 2, d = 1$,
 - $8 > 2^1$ and so Case 3
 - Thus, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\lg 8}) = \Theta(n^3)$
- $T(n) = 8T(n/2) + n$
 - $a = 8, b = 2, f(n) = n, \log_b a = \log_2 8 = 3$
 - Case 3: $f(n) = n = O(n^{3-\epsilon})$, with $\epsilon = 1$
 - Thus, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\lg 8}) = \Theta(n^3)$

Rekürans Bağlılıları (Özyineli Bağlılılar)

Bir dizinin içinde kendisinden bir parça bulunuyorsa o diziye rekürans bağıntısı (özyineli dizi) denir.

$$a_n = 5n + 1 \quad \text{(dizi) (sequence)} \quad \rightarrow a_{20} = 101$$

$$\boxed{a_n = 2a_{n-1} + 3} \quad (a_{n-1} \text{ kendisinden parça}) \text{ rekürans bağıntısı}$$

$\downarrow$
 $a_0 = 3 \quad n \geq 1$

$$\left. \begin{array}{l} a_n = a_{n-1} - 2a_{n-2} + 5^n \\ a_n = 3a_{n-1} + 2a_{n-2} \end{array} \right\} \rightarrow a_1 = 3, a_0 = 5, n \geq 2$$

rekürans bağıntısı

Ör: $a_n = 2a_{n-1} - a_{n-2} + 2^{n-1}$, $a_0 = 1, a_1 = 2, n \geq 2$ olarak veriliyor. Buna göre, a_5 nedir?

$$n=2 \Rightarrow a_2 = 2a_1 - a_0 + 2 \Rightarrow a_2 = 4 - 1 + 2 \Rightarrow \boxed{a_2 = 5}$$

$$n=3 \Rightarrow a_3 = 2a_2 - a_1 + 4 \Rightarrow a_3 = 10 - 2 + 4 \Rightarrow \boxed{a_3 = 12}$$

$$n=4 \Rightarrow a_4 = 2a_3 - a_2 + 8 \Rightarrow a_4 = 24 - 5 + 8 \Rightarrow \boxed{a_4 = 27}$$

$$n=5 \Rightarrow a_5 = 2a_4 - a_3 + 16 \Rightarrow a_5 = 54 - 12 + 16 \Rightarrow a_5 = 58 //$$

Ayrık matematikteki hedefimiz

Rekürans Bağıntısı



```
graph TD; A[Rekürans Bağıntısı] --> B[Rekürans bağıntısını çözmek  
a_n = 3a_{n-1} + a_{n-2}  
a_n = 2^n - n + 1]; A --> C[Sözel bir problemi rekürans bağıntısına çevirme]
```

Rekürans bağıntısını çözmek

$$a_n = 3a_{n-1} + a_{n-2}$$

$$a_n = 2^n - n + 1$$

Sözel bir problemi rekürans bağıntısına çevirme

İç bileşenlerden kurtaracak şekilde yazmalıyız

Rekürans Bağıntılarının Sınıflandırılması

a_n 'nin kendisinden sonraki kaç terime bağlı olması rekürans bağıntısının derecesini verir

1) Derece (Order)

$$a_n = 5a_{n-1} + n^2$$

→ 1 first order

$$a_n = 3a_{n-1} - 5a_{n-2}$$

→ 2 second order

$$a_n = 3a_{n-1} + a_{n-3} + 3^n$$

→ 3 third order

$$a_n = 2a_{n-4} + 1$$

→ 4. derece

2) Homojen – homojen değil (homogeneous – nonhomogeneous)

$$a_n = 3a_{n-1} \quad \checkmark$$

$$a_n = a_{n-1} - 2a_{n-2} \quad \checkmark$$

$$a_n = 3a_{n-1} + n \cdot a_{n-2} + 3^n \quad \times$$

$$a_n = a_{n-1} + 2a_{n-3} - 4 \quad \times$$

$$a_n = a_{n-2} + n^3 + n^2 \quad \times$$

3) Lineer, lineer olmayan (Linear, nonlinear)

a_n 'li terimlerin üslerinin alınması veya birbirleri ile çarpılması lineerliği bozar.

$$a_n = 5a_{n-2} + n \cdot a_{n-3} + n^2$$


$$a_n = (a_{n-1})^2 + 3n$$


$$(a_n)^3 = a_{n-1} + a_{n-2}$$


$$a_n = (a_{n-1}) \cdot (a_{n-2}) - 3$$


4) Sabit Katsayılı olan ve olmayan (constant coefficient)

a_n 'li ifadeler n 'li birşey ile çarpım durumunda olursa sabit katsayılı değildir.

$$a_n = 3a_{n-1} + 5a_{n-2} + n - 7$$


$$a_n = n \cdot a_{n-1} + 3a_{n-2} + 5^n$$


$$a_n = 7a_{n-1} + 10a_{n-2}$$


$$a_n = 3^n \cdot a_{n-2} + 1$$


Sabit Katsayılı Homojen Rekürans Bağlıntılarının Çözülmesi (Karakteristik Kök Tekniği)

Karakteristik Kök Tekniği

- İkinci derece ve daha büyük dereceli rekürans bağlıntılarının çözümünde kullanılır.
- Sabit ve homojen rekürans bağlıntılarında kullanılır.
- Homojen olmayan rekürans bağlıntısında ise homojen hale getirmek için kullanılır.

$$a_n = 3a_{n-1} + 4a_{n-2}$$

Sabit Katsayılı Homojen 2. derece
bir rekürans bağıntısı

Homojen olduğundan

$$a_n = a_n^{(h)}$$

homojen olmadığında

$$a_n = a_n^{(h)} + a_n^{(ö)}$$

$$a_{n-2} = 1$$

$$a_{n-1} = r$$

$$a_n = r^2$$

$$r^2 = 3r + 4$$

$$r^2 - 3r - 4 = 0$$

karakteristik denklem

2 tane farklı kökü varsa

Eşit 2 tane kökü varsa

Karakteristik Denklemin Köklerine Göre Homojen Çözümün Yazımı

2. derece

$$1) r_1 \neq r_2 \text{ ise } \Rightarrow a_n^{(h)} = c_1 \cdot (r_1)^n + c_2 \cdot (r_2)^n$$

$$2) r_1 = r_2 \text{ ise } \Rightarrow a_n^{(h)} = c_1 \cdot (r_1)^n + c_2 \cdot (r_2)^n \cdot n$$

$$r=4 \quad r=-1 \\ (r-4)(r+1)=0$$

$$r^2 - 3r - 4 = 0$$

$$a_n = c_1 \cdot 4^n + c_2 \cdot (-1)^n$$

3. derece

$$1) r_1 \neq r_2 \neq r_3 \Rightarrow a_n^{(h)} = c_1 \cdot (r_1)^n + c_2 \cdot (r_2)^n + c_3 \cdot (r_3)^n$$

$$2) r_1 = r_2 = r_3 \Rightarrow a_n^{(h)} = c_1 \cdot (r_1)^n + c_2 \cdot (r_2)^n \cdot n + c_3 \cdot (r_3)^n \cdot n^2$$

$$3) r_1 = r_2 \neq r_3 \Rightarrow a_n^{(h)} = c_1 \cdot (r_1)^n + c_2 \cdot (r_2)^n \cdot n + c_3 \cdot (r_3)^n$$

Soru: $a_n = 3a_{n-1} + 10a_{n-2}$, $a_0 = 5$ ve $a_1 = 8$ olan başlangıç değerleri verilmiş rekürans bağıntısını çözünüz.

1. adım $a_n = a_n^{(h)}$

2. adım $r^2 = 3r + 10$ (karakteristik denklemler)

2. derece $r^2 - 3r - 10 = 0$
 $(r-5)(r+2) = 0$
5 -2

$$a_n^{(h)} = c_1(5)^n + c_2(-2)^n$$

3. adım

$$a_n = c_1 5^n + c_2 (-2)^n$$

$$a_0 = 5 \Rightarrow \boxed{c_1 + c_2 = 5} \quad / 2$$

$$a_1 = 8 \Rightarrow \boxed{5c_1 - 2c_2 = 8}$$

$$7c_1 = 18$$

$$\boxed{c_2 = \frac{17}{7}}$$

$$\boxed{a_n = \frac{18}{7} \cdot 5^n + \frac{17}{7} \cdot (-2)^n}$$

Soru: $a_n = -4a_{n-1} - 4a_{n-2}$, $a_0 = 3$, $a_1 = 5$ başlangıç koşulları ile verilen rekürans bağıntısının çözümünü bulunuz.

1. adım $a_n = a_n^{(h)}$

2. adım $a_{n-2} = 1$

2. derece $a_{n-1} = r \Rightarrow r^2 = -4r - 4 \Rightarrow r^2 + 4r + 4 = 0$
 $a_n = r^2$
 $(r+2)(r+2) = 0$
 $r_1 = -2, r_2 = -2$

$$a_n^{(h)} = c_1 \cdot (-2)^n + c_2 (-2)^n \cdot n$$

3. adım $a_n = c_1 (-2)^n + c_2 (-2)^n$

$a_0 = 3 \Rightarrow \boxed{c_1 = 3}$

$$\boxed{a_n = 3 \cdot (-2)^n - \frac{11}{2} (-2)^n \cdot n}$$

$a_1 = 5 \Rightarrow -2c_1 - 2c_2 = 5 \Rightarrow -2c_2 = 11 \Rightarrow \boxed{c_2 = -\frac{11}{2}}$

Sabit Katsayılı Homojen Olmayan Rekürans Bağlıntılarının Çözülmesi

$$a_n = 3a_{n-1} + 4a_{n-2} + f(n)$$

genel

özel

$$a_n^{(g)} = a_n^{(h)} + a_n^{(ö)}$$

polinom olursa

1

3

$n+2$

n^2

Üstel fonksiyon olursa

$3 \cdot 2^n$

5^n

7^{-n}

Özel çözüm bulunurken $a_n^{(ö)}$ tahmin edilir ve çözüm bu şekilde elde edilir.

Polinom Durumu

Ör: $a_n = 2a_{n-1} + 3a_{n-2} + 5$, $a_0 = 4$, $a_1 = 10$ başlangıç koşulları ile verilen rekürans bağıntısını çözümlü.

1. adım $a_n^{(g)} = a_n^{(h)} + a_n^{(ö)}$

2. adım $a_n^{(h)}$ bulunur.

$$a_n = 2a_{n-1} + 3a_{n-2} \Rightarrow r^2 = 2r + 3$$

$$\begin{aligned} r^2 - 2r - 3 &= 0 \\ (r-3)(r+1) &= 0 \\ r &= 3 \quad r = -1 \end{aligned}$$

$$a_n^{(h)} = C_1 \cdot 3^n + C_2 \cdot (-1)^n$$

$$a_n = 2a_{n-1} + 3a_{n-2} + 5$$

not: polinomun derecesi ne ise a_n nin de dereceside aynı olur

3. adım $a_n^{(0)}$ tahmin edilir.

$$a_n^{(0)} = A$$

$$a_{n-1} = A$$

$$a_{n-2} = A$$

ardından a_{n-1} ve a_{n-2} elde edilip eşitlikte yerine konular, ana a_n tahmindeki katsayıyı bulmaktır.

$$A = 2A + 3A + 5 \Rightarrow -4A = 5 \Rightarrow A = -\frac{5}{4}$$

$$a_n^{(0)} = -\frac{5}{4}$$

4. adım :

$$a_n = c_1 \cdot 3^n + c_2 \cdot (-1)^n - \frac{5}{4}$$

$$a_0 = 4$$

$$a_1 = 10$$

$$c_1 + c_2 - \frac{5}{4} = 4 \Rightarrow c_1 + c_2 = \frac{21}{4}$$

$$3c_1 - c_2 - \frac{5}{4} = 10 \Rightarrow 3c_1 - c_2 = \frac{45}{4}$$

$$\Rightarrow 4c_1 = \frac{33}{2}$$

$$c_1 = \frac{33}{8}$$

$$c_2 = \frac{9}{8}$$

$$a_n = \frac{33}{8} \cdot 3^n + \frac{9}{8} (-1)^n - \frac{5}{4}$$

Ör: $a_n = a_{n-1} + 2a_{n-2} + 3n + 4$, $a_0 = 5$, $a_1 = 7$ başlangıç koşulları ile verilen rekürans bağıntısını çözünüz.

1. adım $a_n^{(g)} = a_n^{(h)} + a_n^{(ö)}$

2. adım $a_n^{(h)} \rightarrow a_n = a_{n-1} + 2a_{n-2} \Rightarrow r^2 = r + 2$
 $r^2 - r - 2 = 0$
 $(r-2)(r+1) = 0$
 $2 \quad -1$

$$a_n^{(h)} = c_1 \cdot 2^n + c_2 \cdot (-1)^n$$

1. derece polinom

$a_n = a_{n-1} + 2a_{n-2} + 3n + 4$

3. adım $a_n^{(ö)} = An + B$
 $a_{n-1} = A(n-1) + B$
 $a_{n-2} = A(n-2) + B$

$\left. \begin{array}{l} a_n^{(ö)} = An + B \\ a_{n-1} = A(n-1) + B \\ a_{n-2} = A(n-2) + B \end{array} \right\} \begin{array}{l} \cancel{An+B} = \cancel{An-A+B} + 2\cancel{An-4A+2B} + 3n+4 \\ \underline{-2An+5A-2B = 3n+4} \end{array}$

$$A = -\frac{3}{2}$$

$$5A - 2B = 4 \Rightarrow -\frac{15}{2} - 2B = 4$$

$$-\frac{23}{2} = 2B \Rightarrow B = -\frac{23}{4}$$

$$a_n^{(ö)} = -\frac{3}{2}n - \frac{23}{4}$$

4. adım

$$a_n = c_1 \cdot 2^n + c_2 \cdot (-1)^n - \frac{3}{2}n - \frac{23}{4}$$

$$a_0 = 5$$

$$a_1 = 7$$

c_1 ve c_2 bulunup en son a_n yazılır.

Sayı Dizilerini Rekürans Bağıntısına Dönüştürme

Bizlere bazen bir sayı dizisi bazen de sözel bir ifade verilerek ondan bir sayı dizi formunda olayı yazmamız beklenir.

4, 7, 13, 25, 49, 97, Sayı dizisi

Rekürans
bağıntısı :

$$a_0 = 4, \quad a_n = 2a_{n-1} - 1, \quad n \geq 1$$

Ör: $\overset{\downarrow}{\underset{\downarrow}{1}}, 3, 7, 17, 41, 99, 239, \dots$ sayı dizisinin rekürans bağıntısını olarak yazınız.

$$\begin{cases} a_0 = 1 \\ a_1 = 3 \end{cases}$$

$$a_n = 2a_{n-1} + a_{n-2}, \quad n \geq 2$$

Homojen olmayan rekürans bağıntısı için bir örnek yazalım



Ör: $1, 3, 6, 13, 27, 56, 115, \dots$ sayı dizisini rekürans bağıntısı haline getiriniz.

$$\begin{aligned} 1+3+2 \\ 3+6+4 \\ 6+13+8 \end{aligned}$$

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 3 \end{aligned}$$

$$a_n = a_{n-1} + a_{n-2} + 2^{n-1}, \quad n \geq 2$$

$$a_2 = 2a_1 + a_0 \quad a_2 = 7$$

$$a_3 = 2a_2 + a_1 \quad a_3 = 17$$

Soru: Sadece 0 ve 1'lerden oluşan n birim uzunluğunda sayı dizileri oluşturuluyor. Buna göre, bu sayı dizilerinden n uzunluğunda olup "00" içerenlerin sayısını veren rekürans bağıntısını başlangıç koşullarını da belirleyerek yazınız.

$n = 0$ birim	$\longrightarrow$	dizi yok	$a_0 = 0$
$n = 1$ "	$\longrightarrow$	0, 1	$a_1 = 0$
$n = 2$ "	$\longrightarrow$	00, 01, 10, 11	$a_2 = 1$
$n = 3$ "	$\longrightarrow$	<u>000</u> , <u>001</u> , 010, <u>100</u> 011, 101, 110, 111	$a_3 = 3$
$n = 4$ "	$\longrightarrow$	0000- 0011- 0111 0001- 0101 1011 0010- 1001- 1101 0100- 0110 1110 1000- 1010 1111 1100-	$a_4 = 8$

0, 0, 1, 3, 8, - - - - -

$$a_n = 2 \cdot a_{n-1} + a_{n-2} + 1 \quad n \geq 2$$

$$a_1 = 0$$

$$a_0 = 0$$

Problemlerin Karmaşıklığı

Çözülebilir Problemler (Tractable)

- ◆ Polinomsal en kötü durum karmaşıklığına sahip bir algoritma kullanarak çözülebilen bir problem **çözülebilir** (tractable) olarak adlandırılmaktadır.
- ◆ Çünkü algoritmanın, nispeten kısa bir zaman içinde makul büyüklükte veriye sahip bir probleme çözüm getireceği beklenmektedir.
- ◆ Bununla birlikte, büyük- O tahminindeki polinomlar yüksek dereceye sahipse (100. derece gibi) veya katsayılar aşırı büyükse, algoritmanın problemi çözmesi oldukça uzun zaman alabilir.
- ◆ Sonuç olarak, polinomsal en kötü durum zaman karmaşıklığına sahip bir algoritma kullanarak çözülebilen bir problemin nispeten küçük veri değerlerinde bile makul zamanda çözülebilmesi garanti edilemez.
- ◆ Pratikte bu tür tahminlerdeki polinomların derece ve katsayıları genellikle küçüktür.

Çözülemez Problemler (Intractable)

- ◆ En kötü durum polinomsal zaman karmaşıklığına sahip bir algoritma kullanılarak çözilemeyen problemler **çözülemez** (intractable) olarak adlandırılmaktadır.
- ◆ En kötü duruma sahip bir problemin çözümü, çok küçük girdi değerleriyle bile her zaman olmasa da genellikle aşırı miktarda zamana ihtiyaç duyabilmektedir.
- ◆ Bununla birlikte, pratikte bazı en kötü durum zaman karmaşıklığına sahip algoritmaların bir problemi, birçok durumda kendisine özgü en kötü durumundan çok daha hızlı çözebildiği durumlar vardır.
- ◆ Muhtemelen çok az sayıdaki durumların makul zamanda çözilememesine izin vermeye istekli olduğumuzda, ortalama durum zaman karmaşıklığı, bir algoritmanın bir problemi ne kadar uzun zamanda çözeceğinin en iyi ölçüsüdür.
- ◆ Endüstride önemli olan birçok problem çözülemez olarak düşünülmektedir ancak uygulamada, aslında günlük yaşamın getirdiği tüm girdi durumları için çözülebilmektedir.
- ◆ Pratik uygulamada karşılaşılan çözülemez problemlerin üstesinden gelmenin bir diğer yolu da problemin kesin çözümünü bulmak yerine yaklaşık çözümler aramaktır.
- ◆ Bu tür yaklaşık çözümler bulmakta hızlı algoritmaların varlığı işe yaramakta ve hatta kesin çözümünden çok farklı olmama olasılığı bulunmaktadır.

Çözümü Bulunmayan Alg. (Unsolvable)

- ◆ Hiçbir algoritmanın onları çözemediği bazı problemler mevcuttur.
- ◆ Bu tür problemler **çözümü bulunamayan (unsolvable)** olarak adlandırılmaktadır
- ◆ Algoritma kullanarak **çözülebilir** problemlerin zıttı olarak ifade edilebilir.
- ◆ Çözümü bulunamayan problemlerin varlığına ilk kanıt, sonlanma (halting) probleminin çözülemez olduğunu gösteren büyük İngiliz matematikçi ve bilgisayar bilimcisi Alan Turing tarafından sağlanmıştır.

P ve NP Problem Sınıfları

- ◆ Çözülebilir problemlerin **P sınıfına (Class P)** ait olduğu söylenir.
 - ◆ Polinomsal zamanda kontrol edilebilen bir çözüm için problemlerin **NP sınıfına (Class NP)** ait olduğu söylenmektedir.
 - ◆ NP kısaltması, *belirli olmayan polinomsal zamanlı (nondeterministic polynomial time)* anlamına gelmektedir.
-
- ❖ Problemlerin herhangi birisi polinomial en kötü-durum zaman algoritmaları ile çözülebildiğinde, NP sınıfındaki tüm problemler de polinomsal en kötü-durum zaman algoritmaları ile çözülebilir. Bu özelliği taşıyan algoritmalara **NP-tam problemler (NP-complete problems)**
 - ❖ **NP-Hard**, polinomsal zamanda bir çözümü olduğunu ispatlayamadığımız karar problemlerinin karmaşıklık sınıfıdır.

Kaynaklar

- Levitin “Introduction to the Design & Analysis of Algorithms,” 3rd ed., Ch. 1 ©2012 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved
- <https://www.javatpoint.com>
- <https://algorithms.tutorialhorizon.com>
- <https://www.tutorialspoint.com/>
- www.buders.com
- <https://algorithms.tutorialhorizon.com>