



بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



# Computing Algorithms

**Arab Academy for Science and Technology**

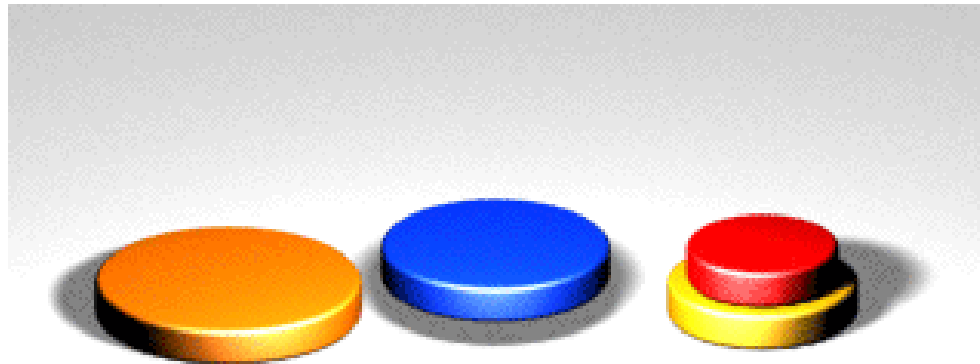
**Prof. Ossama Ismail**



# **Lecture III: Introduction to Algorithms**

## **RECURRENCE RELATIONS**

# Recurrence Relations



$$f_n = f_{n-1} + f_{n-2}$$

# Recurrence Relations



- A Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

$$T(n) = \begin{cases} = c & \text{for } n = 1 \\ = 2 T(n/2) + c n & \text{for } n > 1 \end{cases}$$

- Special techniques are required to analyze the space and time required

# Recurrence Relations



## Example Algorithms and their Recurrence Equations

- Factorial (Every Case):  $T(n) = T(n-1) + 1$ .
- Fibonacci (Every Case):  $T(n) = T(n-1) + T(n-2) + 1$ .
- Binary Search (Worst Case):  $T(n) = T(n/2) + 1$ .
- Quick Sort (Worst Case):  $T(n) = T(n-1) + \Theta(n)$
- Quick Sort (Best Case):  $T(n) = 2T(n/2) + \Theta(n)$
- Merge Sort (Every Case):  $T(n) = 2T(n/2) + \Theta(n)$

# Solving Recurrences



- The substitution method
- The recursion method
- The master method

# The substitution method. 1



**Substitution Method:** This is a straightforward method where you guess a solution and then prove it by induction. It's often useful for simple recurrences.

## Method:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

- Ex:  $T(n) = 2 T(n/2) + n$ . // Sorting

1. We guess that  $T(n) = O(n \lg n)$

$$\begin{aligned} 2. T(n) &\leq 2(c n/2 \lg(n/2)) + n \leq c n \lg(n/2) + n \\ &= c n \lg n - c n \lg 2 + n = c n \lg n - c n + n \\ &\leq c n \lg n \end{aligned}$$

# Recursion method. 2



Sum all the per-level costs to determine the total cost of all levels of the recursion

$$\text{Ex: } T(n) = 3T(n/4) + n$$

$$T(n) = n + 3 T(n/4)$$

$$= n + 3(n/4 + 3T(n/16))$$

$$= n + 3 n/4 + 3 (n/16 + 3T(n/64))$$

$$\leq n + 3 n/4 + 9 n/16 + \dots$$

$$= O(n^2)$$

# The master method. 3



**Master theorem:** Let  $a \geq 1$  and  $b > 1$  be constants,  
let  $f(n)$  be a

function, and let  $T(n)$  be defined by

$$T(n) = aT(n/b) + f(n)$$

Then  $T(n)$  can be bounded asymptotically as follows.

1.If  $T(n) = \Theta(n^{\log_b a})$  for some constant  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

2.If  $f(n) = \Theta(n^{\log_b a})$  then  $T(n) = \Theta(n^{\log_b a} \lg n)$

3.If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$ , and

if  $a f(n/b) \leq c(n)$  for constant  $c < 1$  and all  
sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$ .



# Recurrence - Example 1

## Binary Search

# Example: Binary



```
BinarySearch( sortedArray[], first, last, key)
{ // sortedArray in array of sorted (ascending) values.
  // first, last in lower and upper subscript bounds & key value to search for.
  // returns: index of key, or -insertion_position -1 if key is not in the array.
                                                                    count
if (first <= last)                                                                    1
{ mid = (first + last) / 2;    // compute mid point.                                                                    1
  if (key == sortedArray[mid]) return mid; // found it.                                                                    1
  else if (key < sortedArray[mid])                                                                    1
    // recursively call BinarySearch for the lower part
    BinarySearch(sortedArray, first, mid-1, key); T(n/2)
  else
    // recursively call BinarySearch for the upper part
    BinarySearch(sortedArray, mid+1, last, key); } T(n/2)
}
return -(first + 1); // failed to find key                                                                    1
}
```

# Example 1 : Binary Search



$$T(n) = O(1) + T(n/2)$$

$$T(1) = 1$$

Above equation is example of recurrence relation, this can be solved by direct Substitution.

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

$$= T(n/2^2) + 1 + 1$$

$$= T(n/2^3) + 1 + 1 + 1$$

.....

$$= T(n/2^k) + k = \log n$$

$$// \quad n = n/2^k$$

$$T(n) = O(\log n)$$

# Example

2



**Check that**  $a_n = 2^n + 1$  is a solution to the recurrence relation  $a_n = 2a_{n-1} - 1$  with  $a_1 = 3$

**Solution**

First, it is easy to check the initial condition:  $a_1$  should be  $2^1 + 1 = 3$  according to the given closed formula. Indeed,  $2^1 + 1 = 3$ ,  $2^1 + 1 = 3$ , which is what we want. To check that the proposed solution satisfies the recurrence relation, try plugging it in.

$$\begin{aligned} 2a_{n-1} - 1 &= 2(2^{n-1} + 1) - 1 \\ &= 2^n + 2 - 1 \\ &= 2^n + 1 \\ &= a_n. \end{aligned}$$

That's what the recurrence relation says! □ We have a solution.



# Recurrence - Example 3

## Quick Sort

# EXAMPLE 3: QUICK SORT



## Run time

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1)$$

- In the above case the presence of function of T on both sides of the equation signifies the presence of recurrence relation
- (*SUBSTITUTION METHOD used*) The equations are simplified to produce the final result:  
.....cntd

# EXAMPLE 3: QUICK SORT cont'd



$$T(n) = 2T(n/2) + O(n)$$

$$n)) + n/2) + (n/2^2(2(2 \quad =$$

$$T(n/2^2) + n + n \ 2^2 \quad =$$

$$n + n) + T(n/2^3) + (n/2^2) (2^2 \quad =$$

$$T(n/2^3) + \underline{n + n + n} \ 2^3 \quad =$$

...

$$= 2^k T(n/2^k) + k n \quad // \ n = 2^k$$

$$= n \log n \quad // \ k = \log n$$

# Algorithms



## □ Types of Heuristic algorithms

1. □ Greedy
2. □ Local search

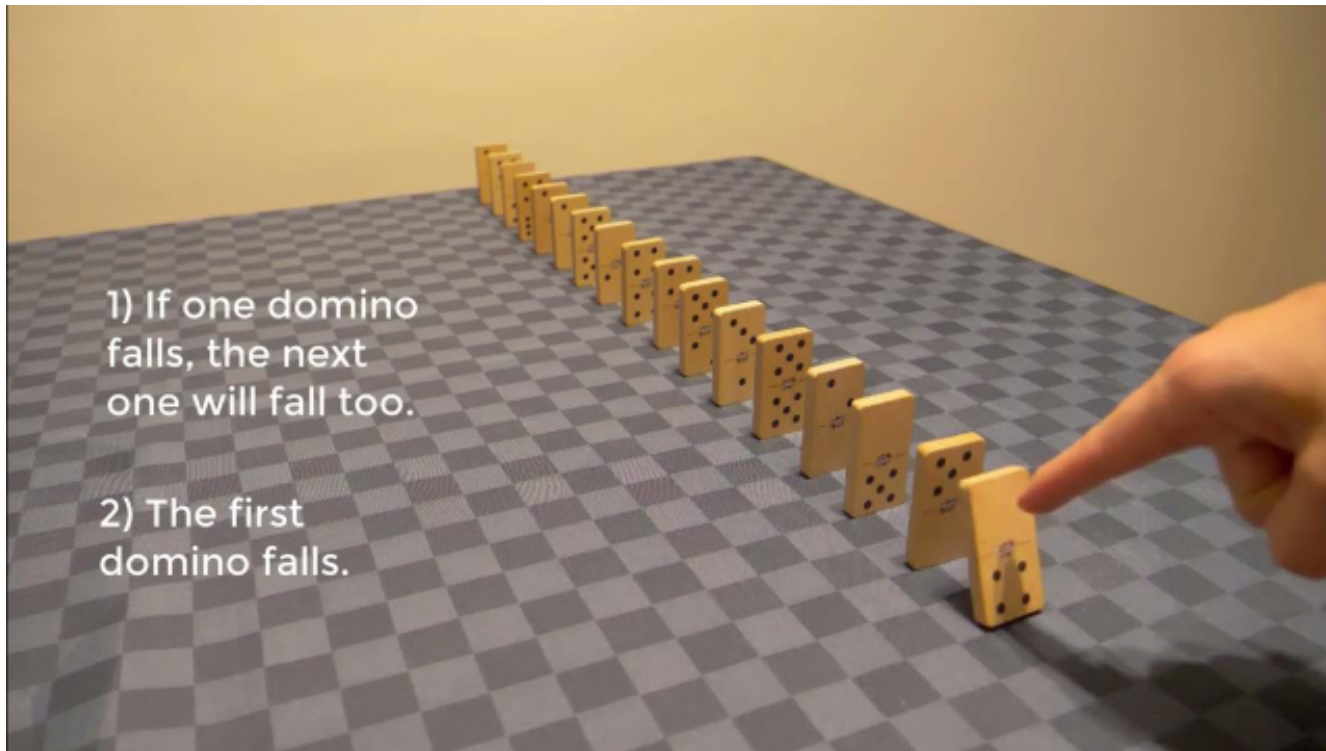
## □ Types of Optimal algorithms

3. □ Exhaustive Search (ES)
4. □ Divide and Conquer (D&C)
5. □ Branch and Bound (B&B)
6. □ Dynamic Programming (DP)

# Proof By



If a domino falls in a chain, the next domino will surely fall too. Since this second domino is falling, the next one in the chain will certainly fall as well. Since this third domino is falling, the fourth will fall too, and then the fifth, and then the sixth, and so on.



# Proof By



**Proof by induction** is a way of proving that a certain statement is true for every positive integer  $n$ . Proof by induction has four steps:

Prove the **base case**: this means proving that the statement is true for the **initial value**, normally  $n=1$  or  $n=0$ .

Assume that the statement is true for the value  $n=k$ . This is called the **inductive hypothesis**.

Prove the **inductive step**: prove that if the assumption that the statement is true for  $n=k$ , it will also be true for  $n=k+1$ .

Write a **conclusion** to explain the proof, saying: "If the statement is true for  $n=k$ , the statement is also true for  $n=k+1$ . Since the statement is true for  $n=1$ , it must also be true for  $n=2$ ,  $n=3$ , and for any other positive integer."

# Proof By



- **Claim:**  $S(n)$  is true for all  $\underline{n \geq k}$
- **Basis:**
  - Show formula is true when  $\underline{n = k}$
- **Inductive hypothesis:**
  - Assume formula is true for an arbitrary  $\underline{n}$
- **Step:**
  - Show that formula is then true for  $\underline{n+1}$

# Proof By



## Gaussian Closed Form

Prove that  $(1 + 2 + 3 + \dots + n) = n(n+1) / 2$

- Basis:
  - If  $n = 0$ , then  $0 = 0(0+1) / 2 = 0$
- Inductive hypothesis:
  - Assume  $1 + 2 + 3 + \dots + n = n(n+1) / 2$
- Step (show true for  $n+1$ ):
  - $1 + 2 + \dots + n + n+1 = (1 + 2 + \dots + n) + (n+1)$
  - $= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2$
  - $= (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2$

# Proof By



## Geometric Closed Form

Prove that  $(a^0 + a^1 + \dots + a^n) = (a^{n+1} - 1)/(a - 1)$  for  $a \neq 1$

- Basis: show that  $a^0 = (a^{0+1} - 1)/(a - 1)$   
$$a^0 = 1 = (a^1 - 1)/(a - 1)$$
- Inductive hypothesis:
  - Assume  $(a^0 + a^1 + \dots + a^n) = (a^{n+1} - 1)/(a - 1)$
- Step (show true for  $n+1$ ):
  - $(a^0 + a^1 + \dots + a^n) = (a^{n+1} - 1)/(a - 1) + a^{n+1}$
  - $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$



# Reading

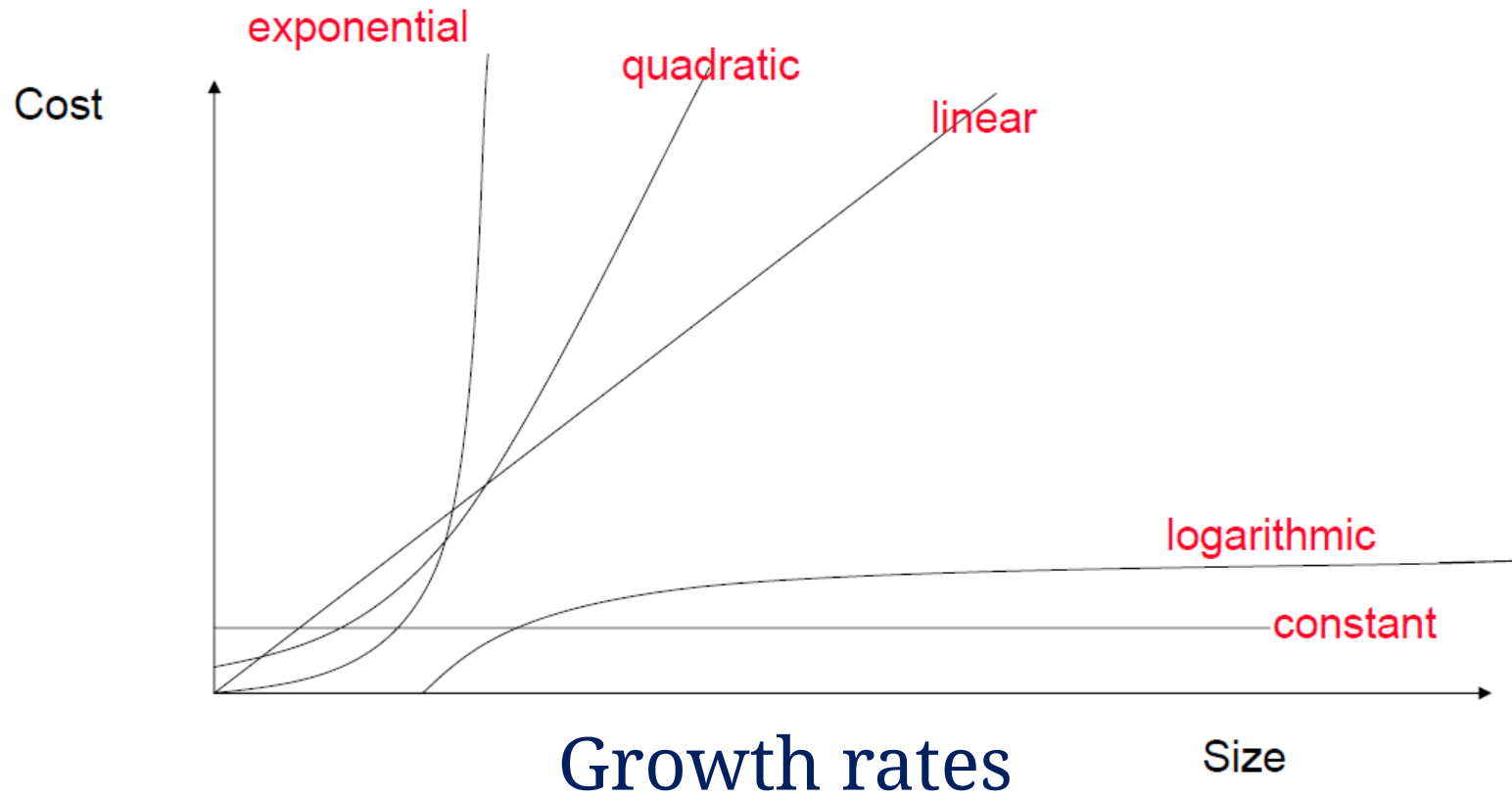
## Chapter two of Your Text Book

# Summary



- Rules for Big-Oh
  - if  $T(n) = O(c * f(n)) \rightarrow$   
$$T(n) = O(f(n))$$
  - if  $T_1(n) = O(f(n))$  and  $T_2(n) = O(g(n)) \rightarrow$   
$$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$
  - if  $T_1(n) = O(f(n))$  and  $T_2(n) = O(g(n)) \rightarrow$   
$$T_1(n) * T_2(n) = O(f(n) * g(n))$$
  - if  $T(n) = a_k n^k + a_{k-1} n^{k-1} + \dots a_1 n + a_0 \rightarrow$   
$$T(n) = O(n^k)$$

# Summary



# Summary



## Classifying functions by their growth rates

Big-oh notion

Say that f is	Mean that f is	Write	if
oh g	no faster than g	$f = O(g)$	$\exists n_0, c > 0 : \forall n > n_0, \frac{f(n)}{g(n)} \leq c \Leftrightarrow f(n) \leq c g(n)$
Small oh g	slower than g	$f = o(g)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
theta g	about as fast as g	$f = \Theta(g)$	$f = O(g)$ and $g = O(f)$
about g	as fast as g	$f \approx g$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$
omega g	no slower than g	$f = \Omega(g)$	$g = O(f)$



# Questions ???