

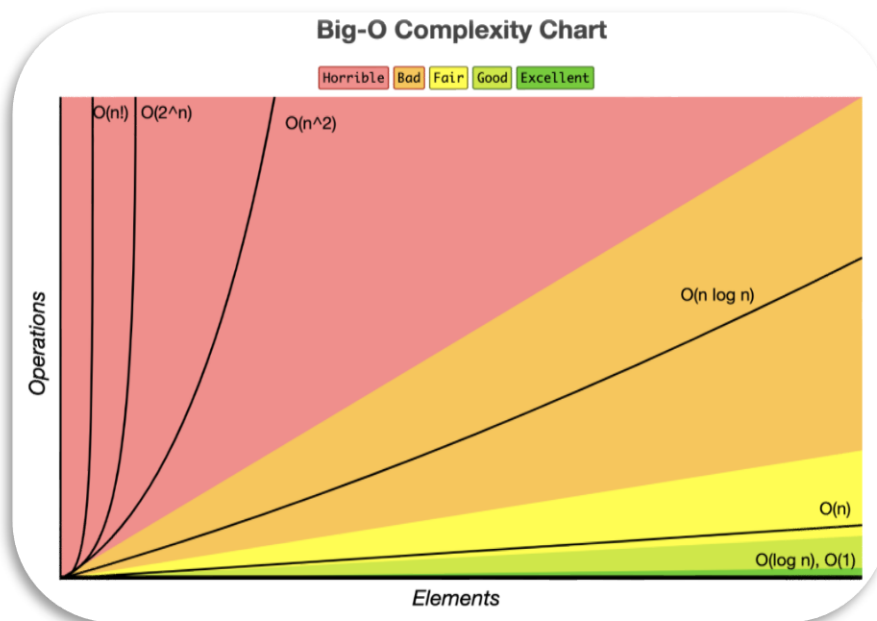


Islamic University of Gaza
Faculty of Engineering
Computer Engineering
Department



ECOM 3001 Data Structures and Algorithms Lab [Fall 2024]

Lab "03" Topic: Analysis of Algorithms



Prepared by:
Eng. Waseem Alfarram

Supervised by:
Dr. Ahmad Mahdi

Lab Objectives:

- To go more deeper in analyzing different algorithms and compare between them in both **Time Complexity** and **Space Complexity**.
- To learn how to calculate Running Time for your algorithm for different cases: **best**, **worst** and **average** cases.
- To recognize **Primitive Operations** and how do we count them in different algorithms so we can estimate the running time for the algorithm.
- To recognize the **Big-Oh** Notation (Asymptotic Notation) and how it works.
- To recognize other asymptotic notations like the **Big-Omega** and the **Big-Theta**.

Comparing Algorithms:

- Given 2 or more algorithms to solve the same problem, how do we select the better one?
- Some criteria for selecting an algorithm:
 1. Is it easy to implement, understand, modify?
 2. How long does it take to run it to completion?
 3. How much of computer memory does it use?

Software engineering is primarily concerned with the first criteria but in this course, we are interested in the **second** and **third** criteria!

- Main Criteria we are going to focus on when selecting an algorithm are:
 1. **Time Complexity:** The amount of time that an algorithm needs to run to completion.
 2. **Space Complexity:** The amount of memory an algorithm needs to run.

In our case: the better algorithm is the one which runs **faster** (has a smaller time complexity) and consumes **less memory** (has a smaller space complexity).

Running Time Calculations:

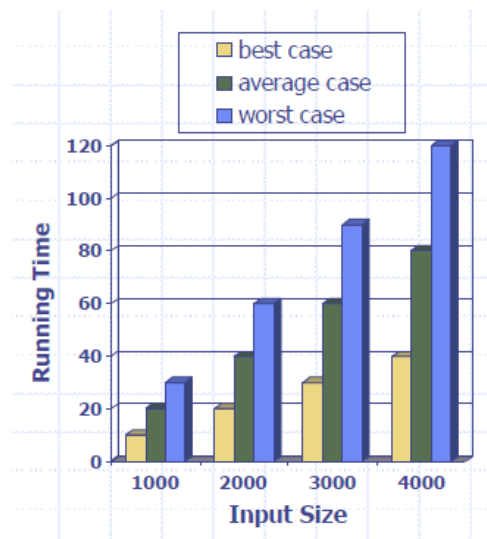
- Most algorithms transform input objects into output objects. The running time of an algorithm typically grows with the input size (analyze running time as **a function of input size**).



- Even on inputs of the same size, running time can be very different. Example: algorithm that finds the first prime number in an array by scanning it left to right.

Analyzing running time in the:

- Best Case.
- Worst Case.
- Average Case.



Average case time is often difficult to determine. So, we will focus on the worst-case running time because:

- It is easier to analyze.
- Crucial to applications such as games, finance and robotics.

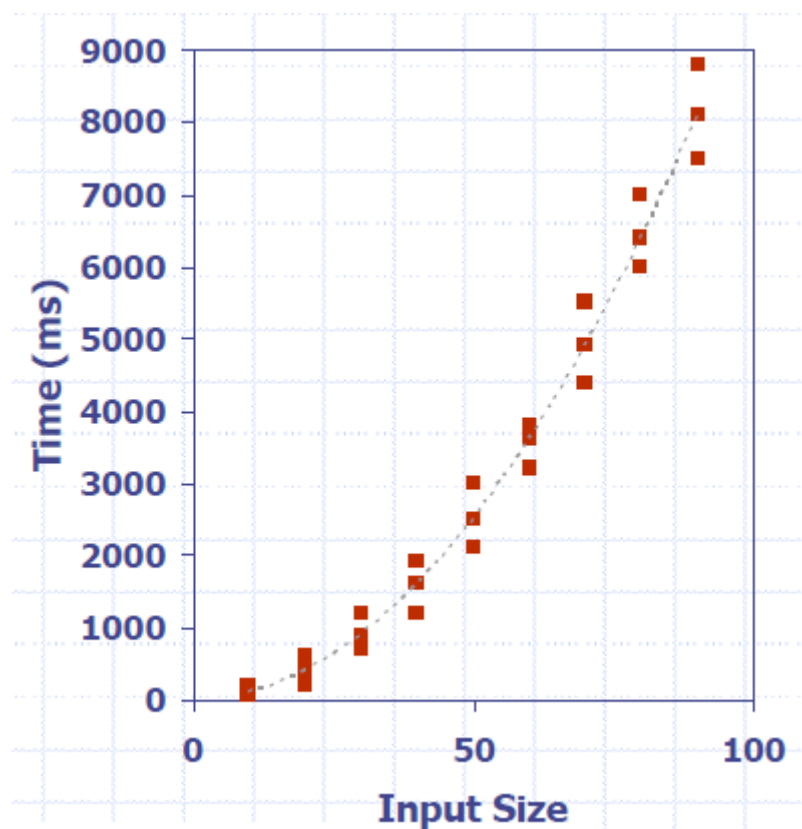
So, for now we agree on two points:

1. Function of input size
2. Focus on the worst case.

➤ Experimental Studies:

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition, noting the time needed.
- Plot the results.

```
1 long startTime = System.currentTimeMillis();           // record the starting time
2 /* (run the algorithm) */
3 long endTime = System.currentTimeMillis();           // record the ending time
4 long elapsed = endTime - startTime;                  // compute the elapsed time
```



➤ Limitations of Experiments:

- It is necessary to implement the algorithm, which may be difficult.
- Results may **not be indicative** of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

➤ Theoretical Analysis:

- Uses a high-level description of the algorithm instead of an implementation.
- Characterizes running time as a function of the input size (n).
- Takes into account all possible inputs.
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment.

➤ The Random-Access Machine (RAM) Model:

A RAM consists of:

- A CPU.
- A potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character.
- Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations:

➤ Example of primitive operations are:

• Evaluating an expression	$x^2 + e^y$
• Assigning a value to a variable	<code>cnt ← cnt + 1</code>
• Indexing into an array	<code>A[5]</code>
• Calling a method	<code>mySort(A, n)</code>
• Returning from a method	<code>return(cnt)</code>

➤ The growth rate of our algorithm is not affected by:

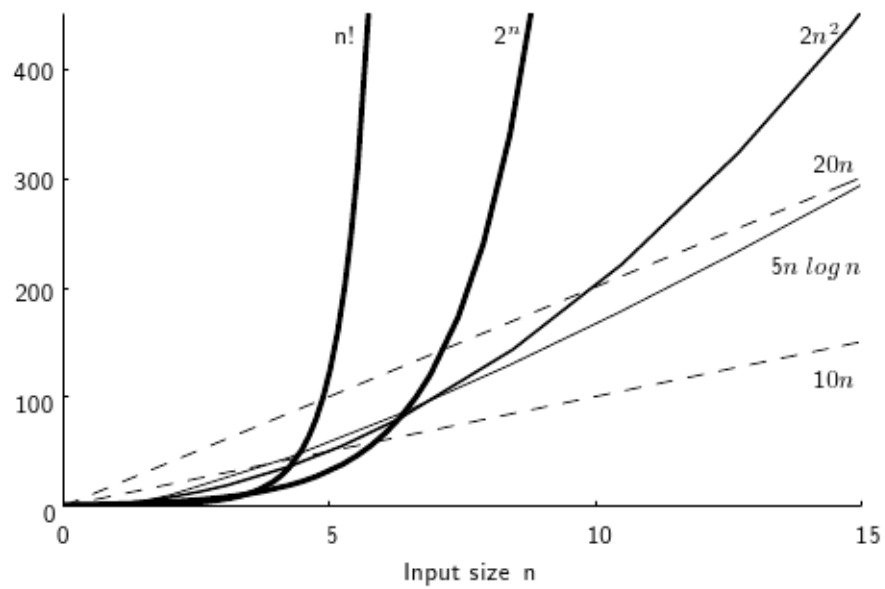
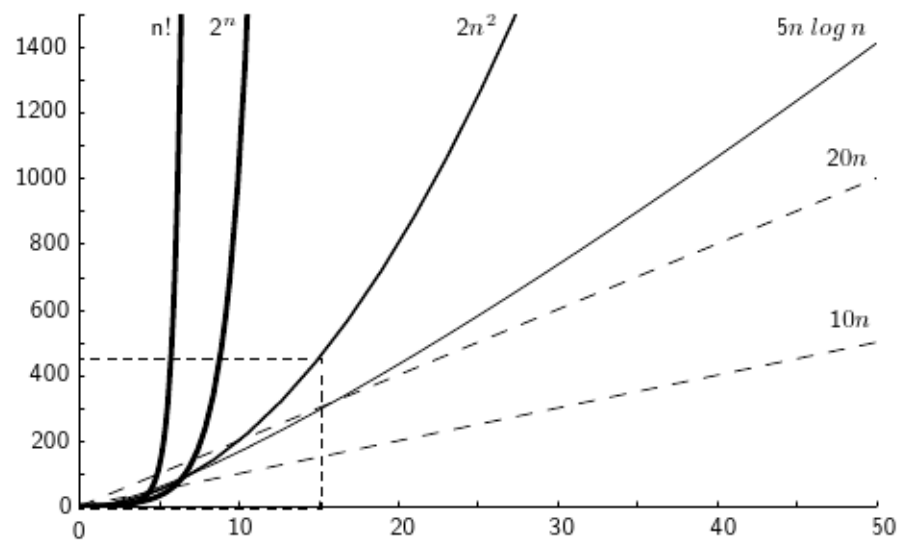
- Constant factors.
- Lower-order terms.

Example:

- $10^2 n + 10^5$ is a **linear function**.
- $10^5 n^2 + 10^8 n$ is a **quadratic function**.

➤ Seven functions that often appear in algorithm analysis:
(From faster to slower)

- Constant ≈ 1 (Fastest – grows the slowest)
- Logarithmic $\approx \log n$
- Linear $\approx n$
- N-Log-N $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Exponential $\approx 2^n$
- Factorial $\approx n!$ (Slowest – grows the fastest)



(Growth Rate for different algorithms)

Big-Oh and Growth Rate:

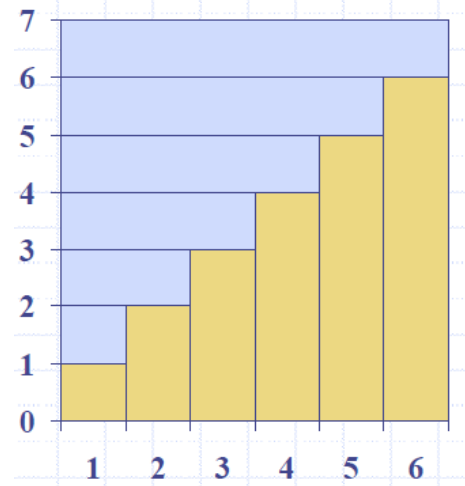
- The big-Oh notation gives an upper bound or the growth rate of a function.
- The statement “**f(n) is O(g(n))**” means that the growth rate f(n) is **no more** than the growth rate of g(n).
- We can use the big-Oh notation to rank functions according to their growth rate.

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

➤ Example 1: (Computing Prefix Averages)

The i -th prefix average of an array X is average of the first $(i + 1)$ elements of X :

$$A[i] = (X[0] + X[1] + \dots + X[i]) / (i + 1)$$



Solution One: (Quadratic Time)

- The running time of `prefixAverage1` is $O(1 + 2 + \dots + n)$.
- The sum of the first n integers is $n(n+1) / 2$
(There is a simple visual proof of this fact)
- Thus, algorithm `prefixAverage1` runs in $O(n^2)$ time.

Algorithm `prefixAverages1(X, n)`

Input array X of n integers

Output array A of prefix averages of X

$A \leftarrow$ new array of n integers

for $i \leftarrow 0$ **to** $n - 1$ **do**

$s \leftarrow X[0]$

for $j \leftarrow 1$ **to** i **do**

$s \leftarrow s + X[j]$

$A[i] \leftarrow s / (i + 1)$

return A

#operations

n

n

n

$1 + 2 + \dots + (n - 1)$

$1 + 2 + \dots + (n - 1)$

n

1

Solution Two: (Linear Time)

- This algorithm **prefixAverage2** runs in **$O(n)$** time.

Algorithm *prefixAverages2*(*X*, *n*)

Input array *X* of *n* integers

Output array *A* of prefix averages of *X*

A $\leftarrow$ new array of *n* integers

s $\leftarrow$ 0

for *i* $\leftarrow$ 0 **to** *n* - 1 **do**

s $\leftarrow$ *s* + *X*[*i*]

A[*i*] $\leftarrow$ *s* / (*i* + 1)

return *A*

#operations

n

1

n

n

n

1

➤ **Example 2: (Summing Triple Array)**

This algorithm **SumTripleArray** runs in $O(n^3)$ time.

Algorithm SumTripleArray(X, n)

Input triple array $X[][][]$ of n by n by n integers

Output sum of elements of X

$s \leftarrow 0$

for $i \leftarrow 0$ **to** $n - 1$ **do**

for $j \leftarrow 0$ **to** $n - 1$ **do**

for $k \leftarrow 0$ **to** $n - 1$ **do**

$s \leftarrow s + X[i][j][k]$

return s

#operations

1

n

$n + n + \dots + n = n^2$

$n^2 + n^2 + \dots + n^2 = n^3$

$n^2 + n^2 + \dots + n^2 = n^3$

1

➤ **Example 3: (Unique Elements)**

This algorithm **UniqueElements** runs in $O(n^2)$ time.

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

//Determines whether all the elements in a given array are distinct

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

//Output: Returns “true” if all the elements in A are distinct

// and “false” otherwise

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

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

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

return true

Intuition for Asymptotic Notation:

- **Big-Oh:**

$f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal** to $g(n)$.

- **Big-Omega:**

$f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal** to $g(n)$.

- **Big-Theta:**

$f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal** to $g(n)$.

Exercises:

➤ Exercise 1:

Given an array and a value, write a function that searches for the index of that value in the array. (Consider the array is sorted increasingly).

Write two solutions. First solution with $O(n)$ time complexity and a second solution with $O(\log n)$ time complexity.

```
public static int searchForTarget (int[] arr, int target)
```

➤ Exercise 2:

The Fibonacci Series is a special kind of sequence that starts with 0 and 1, and every number after those two is the sum of the two preceding numbers.

The Fibonacci series goes like this: **0, 1, 1, 2, 3, 5, 8, 13, 21, ...** and so on.

Write a function that gets the **Nth number** in the Fibonacci series using two solutions. The first solution should be of $O(2^n)$ time and space complexity and the second solution should be of $O(n)$ time and space complexity.

```
public static int fibonacci (int num)
```

➤ Exercise 3:

Write a function that executes the factorial for a specific number. For example, **$n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$** .

First, write a solution with $O(n)$ time and space complexity. Then, in the second solution, enhance its space complexity to be $O(1)$ only.

```
public static long factorial (int num)
```

End of Lab 