



CSE- 207

Algorithms

Lecture: 03

Fahad Ahmed

Lecturer, Dept. of CSE

E-mail: fahadahmed@uap-bd.edu

Algorithm: Complexity Analysis

A SIMPLE PROBLEM

- Assume you are asked to write an algorithm to **find the GCD of 2 integers.**
 - **Algorithm 1:** Using Prime factor
 - Calculate the common prime factors of the 2 numbers and multiply the factors.
 - **Algorithm 2** (Euclidean method): Using Divisor

ALGORITHM 1: USING PRIME FACTOR

- Calculate Prime Factors of number n1
- Calculate Prime Factors of number n2
- Identify the common factors
- Multiply the factors and return

- Example: $n1 = 8$, $n2 = 36$
 - $8 = 2 \times 2 \times 2$
 - $36 = 2 \times 2 \times 3 \times 3$
 - Common factors 2,2
 - $\text{GCD} = 2 \times 2 = 4$

ALGORITHM 2(EUCLIDEAN METHOD)

- Algorithm
 - While n_1 is not divided by n_2
 - Calculate remainder $r = n_1 \bmod n_2$
 - $n_1 = n_2$
 - $n_2 = r$
 - Return n_2
- Example: $n_2 = 8, n_1 = 36$
 - Iteration 1
 - n_1 is not divided by n_2
 - $r = 36 \% 8 = 4$
 - Iteration 2
 - $n_1(8)$ is divided by $n_2(4)$
 - Return 4 as gcd.

**WHICH ONE WILL YOU
CHOOSE FOR YOUR
PROGRAM?**



ALGORITHM 1: USING PRIME FACTOR

○ Example: $n1 = 8$, $n2 = 36$

- $8 = 2 \times 2 \times 2$ *Needs ~3 divisions & 3 checks for prime*
- $36 = 2 \times 2 \times 3 \times 3$ *Needs ~4 divisions & 4 checks for prime*
- Common factors 2,2 *Needs ~3 checks for common prime*
- $\text{GCD} = 2 \times 2 = 4$ *Needs 1 multiplication*

Total ~18 operations

ALGORITHM 2(EUCLIDEAN METHOD)

- Example: $n2 = 8$, $n1 = 36$
 - Iteration 1
 - $n1$ is not divided by $n2$
 - $r = 36 \% 8 = 4$*Needs 1 check and 1 division*
 - Iteration 2
 - $n1(8)$ is divided by $n2(4)$
 - Return 4 as gcd.*Needs 1 check*

Total ~3 operations

ALGORITHM ANALYSIS

ALGORITHM ANALYSIS

- *Analyzing an algorithm is*

- *predicting the resources* that the algorithm requires.
 - Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but **most often it is computational time and space** that we want to measure.

Question: Why do we need analysis?

Question: How do we measure time and space?



ALGORITHM ANALYSIS

○ Why do we need analysis:

- Generally, from **several** candidate algorithms -> **identify a most efficient one**.
- Such analysis may indicate **more than one viable** candidate,
- but we can often **discard several inferior** algorithms in the process.

LET'S TAKE A LOOK AT A PROBLEM

- Assume there are 50 students in your section and your teacher is **storing the ID** of the students who have enrolled for CSE 207. You are asked to find if a particular student has enrolled in the course or not.

Enrolled Students (Count = 21)

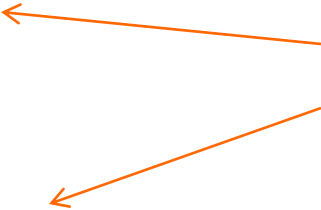
1,3, 4,6, 7, 10,15, 17,18, 19,21, 22, 25,30, 33,34, 35,44, 46,49

SOLUTIONS?

- **Solution 1:**

```
for (i =0; i<21; i++)  
    if array[i] == key (id we are looking for)  
        return true;  
return false;
```

Maximum how many times
are we searching?



- **Solution 2:**

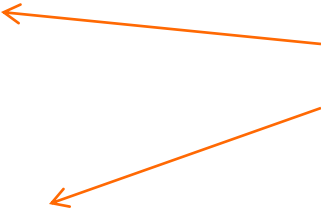
- for (i =0; i<21; i++)
 mid = mid point of the array
 if array[mid] == key
 return mid
 else if key<array[mid]
 search in lower half of the array (upto mid-1)
 else
 search in the upper half (index starting from mid+ 1)

MOVE SPECIFIC TO GENERALIZED SOLUTIONS

- **Solution 1:**

```
for (i =0; i<n; i++)  
    if array[i] == key (id we are looking for)  
        return true;  
return false;
```

Maximum how many times
are we searching?



- **Solution 2:**

- for (i =0; i<n; i++)
 mid = mid point of the array
 if array[mid] == key
 return mid
 else if key<array[mid]
 search in lower half of the array (upto mid-1)
 else
 search in the upper half (index starting from mid+ 1)

ANALYSIS OF ALGORITHMS

It is highly required to use a method to compare the solutions in order to judge **which one is more optimal**.

An algorithm is said to be efficient and fast, if it takes **less time** to execute and consumes **less memory** space.

The performance of an algorithm is measured on the basis of following properties:

- Space Complexity
- Time Complexity

ANALYSIS OF ALGORITHMS

Space Complexity

It is the **amount of memory space** required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Time Complexity

Time Complexity is a way to represent the **amount of time required by the program to run till its completion**. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes its execution in the minimum time possible.

ANALYSIS OF THE ALGORITHM

(FIND HOW MUCH TIME IT WILL TAKE TO RUN.)

THE CLASSICAL PROBLEM THAT MAPS TO THIS SCENARIO.

- Find if a specific integer appear in an array.

```
for i = 0 to array length - 1
```

```
    If X = array[i] then
```

```
        return true
```

```
return false
```

ANALYSIS OF ALGORITHM – RAM MODEL (RANDOM-ACCESS MACHINE)

- Instructions are executed one after another, with **no concurrent operations**.
- Each such instruction **takes a constant amount** of time.
- The **total time** an algorithm takes is the summation of time taken by each instruction.

ANALYSIS OF ALGORITHM – RAM MODEL (**RANDOM-ACCESS MACHINE**)

- The RAM model contains **instructions** commonly found in real computers:
 - Arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
 - Data movement (load, store, copy), and
 - Control (conditional and unconditional branch, subroutine call and return).

RAM MODEL – TIME COMPLEXITY

Line#	Instruction	Cost	# Times this line executes
1	for i = 0 to array length-1	c1	22
2	if X = array[i] then	c2	21
3	return i	c3	1
4	return -1	c4	0

- $T = 22c1 + 21c2 + c3$

RAM MODEL – TIME COMPLEXITY

Line#	Instruction	Cost	# Times this line executes
1	for i = 0 to array length-1	c1	n+1
2	if X = array[i] then	c2	n
3	return i	c3	1
4	return -1	c4	0

- For array size = n,
- Total time $T(n) = c1*(n+1) + c2*n + c3$
 - $= (n+1)c1 + nc2 + c3$
 - $= (c1+c2)n + (c1+c3)$
 - $= cn + d$ [Linear to size of the input]
 - $= O(n)$ [Order of n]

RAM MODEL – TIME COMPLEXITY

Line#	Instruction	Cost	Best	Worst
1	for i = 0 to array length - 1	c1	c1	(n+1)c1
2	if X = array[i] then	c2	c2	nc2
3	return i	c3	1	0
4	return -1	c4	0	1

○ So

- $T_{\text{best}} = c1 + c2 + c3 \rightarrow$ **constant time**
- $T_{\text{worst}} = c1 * (n+1) + c2 * n + c3 \rightarrow$ **Linear function of n**
 $= (c1+c2)*n + (c1+c3)$
 $= an+b$
- $T_{\text{average}} = an/2+b = a1n+b \rightarrow$ **Liner function of n.**

RAM MODEL – TIME COMPLEXITY

- Now think about some scenario
 - The value we are looking for is at the beginning of the Array
 - Known as **Best Case** as minimum time is required to execute.
 - The value we are looking for is not in the Array
 - Known as **Worst Case** as maximum time is required to execute.
 - **Average case:**
 - the amount of some computational resource (typically time) used by the algorithm, averaged over all possible inputs.
 - Similar to **worst case for input size = $\frac{1}{2}$ of original input.**

TYPES OF ANALYSIS

There are three types of analysis that we perform on a particular algorithm.

- **Best case**
 - **Lower bound.**
 - Minimum number of steps/operations to execute an algorithm.
 - Measure the minimum time required to run an algorithm.
 - Not a good measure of Algorithm's performance.

TYPES OF ANALYSIS

- **Worst case**
 - **Upper Bound**
 - Maximum no. of operations/time required to execute
 - Main focus
 - Reduce risks as it gives the highest time of algorithm execution
- **Average case**
 - the amount of some computational resource (typically time) used by the algorithm, **averaged over all possible inputs.**
 - Difficult to determine
 - Typically follow the same curve as worst

SO, WHAT IS ALGORITHM ANALYSIS?

- To analyze an algorithm means:
 - **developing** a formula for predicting *how fast* an algorithm is, based on the **size of the input (time complexity)**, and/or
 - **developing** a formula for predicting *how much memory* an algorithm requires, based on **the size of the input (space complexity)**
- Usually **time** is our **biggest concern**

Time Complexity Analysis



HOW TO MEASURE THE TIME TO RUN AN ALGORITHM?

- One **naïve** way is to implement the algorithm and run it in a machine.
- This time depends on
 - the speed of the computer,
 - the programming language,
 - the compiler that translates the program to machine code.
 - the program itself.
 - And many other factors
- So, you may get different time for the same algorithm.
- Hence, **not a good tool to compare different algorithm.**
- To overcome these issues, need to **model Time complexity.**

TIME COMPLEXITY

- The **best**, **worst**, and **average** case time complexities for any given algorithm are **numerical functions over the size of possible problem instances**.
- However, it is **very difficult** to work **precisely** with these functions,
 - Depends on specific input size.
 - Not a smooth curve
 - Require too much detail
 - So, need more **simplification** or **abstraction**.

ASYMPTOTIC ANALYSIS

- We ignore **too much details** steps such as
 - Initialization cost
 - Implementation of specific operation.
- Rather we **focus on**
 - how **the time change** if input doubles/triples
 - Or **how many more operations** do we need for that change.

ASYMPTOTIC ANALYSIS

- Asymptotic notation are primarily used to describe the running times of algorithms
- The **Running time** of Algorithm is defined as : the time needed by an algorithm in order to deliver its output when presented with legal input .
- In Asymptotic notation, algorithm is treated as a function.
- Let us consider asymptotic notations that are well suited to characterizing running times no matter what the input.

ASYMPTOTIC ANALYSIS

- There are 3 standard asymptotic notations
 1. **Big-O Notation (O)** – Big O notation specifically describes worst case scenario.
 2. **Omega Notation (Ω)** – Omega(Ω) notation specifically describes best case scenario.
 3. **Theta Notation (θ)** – This notation represents the average complexity of an algorithm.

ASYMPTOTIC NOTATION

○ *Big Theta*

- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

○ *Big Oh*

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e., $n \geq n_0$ for some constant n_0).

○ *Big Omega*

- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.

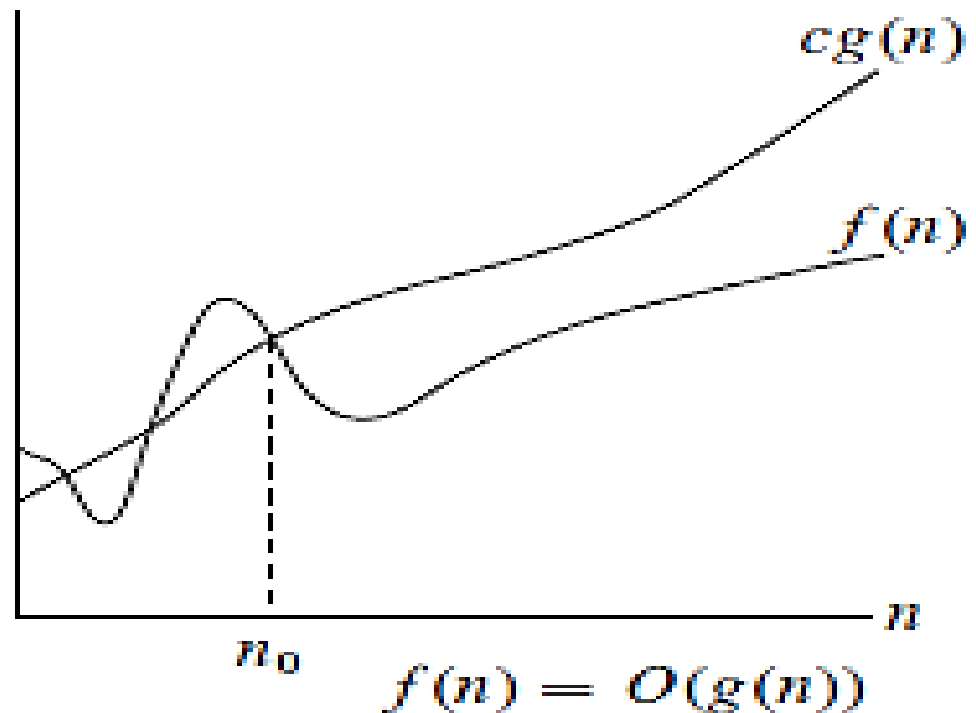
ASYMPTOTIC ANALYSIS

- Classifying functions into **different category**.
- Formally, given functions f and g of a variable n , we define a binary relation
 - $f \sim g$ (as $n \rightarrow \infty$)
- **f** and **g** grows the same way as their input grows.
- In Asymptotic Analysis,
 - we evaluate the **performance** of an algorithm **in terms of input size**
 - Do not measure the actual running time
 - We calculate, **how does** the time (or space) taken by an algorithm **increases with the input size**.

BIG O NOTATION

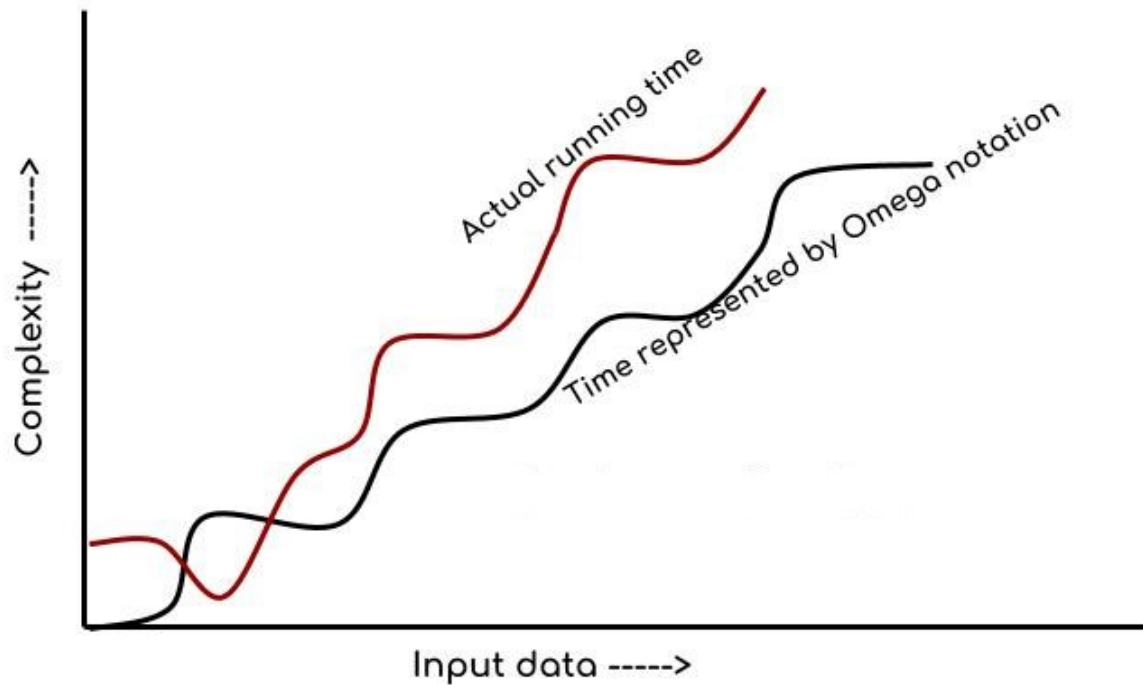
- It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:

$O(g(n)) = \{ f(n) : \text{there exist positive constant } c, n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 \}.$



BIG OMEGA - Ω NOTATION

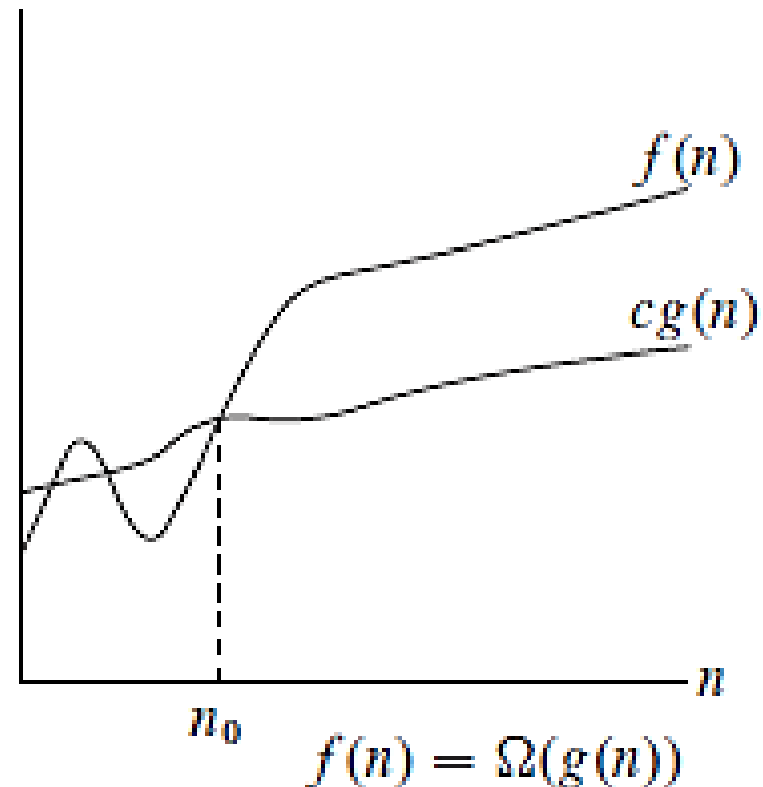
Omega notation specifically describes best case scenario. It represents the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega notation, it means that the **algorithm cannot be completed in less time than this**, it would at least take the time represented by Omega notation or it can take more (when not in best case scenario).



BIG OMEGA - Ω NOTATION

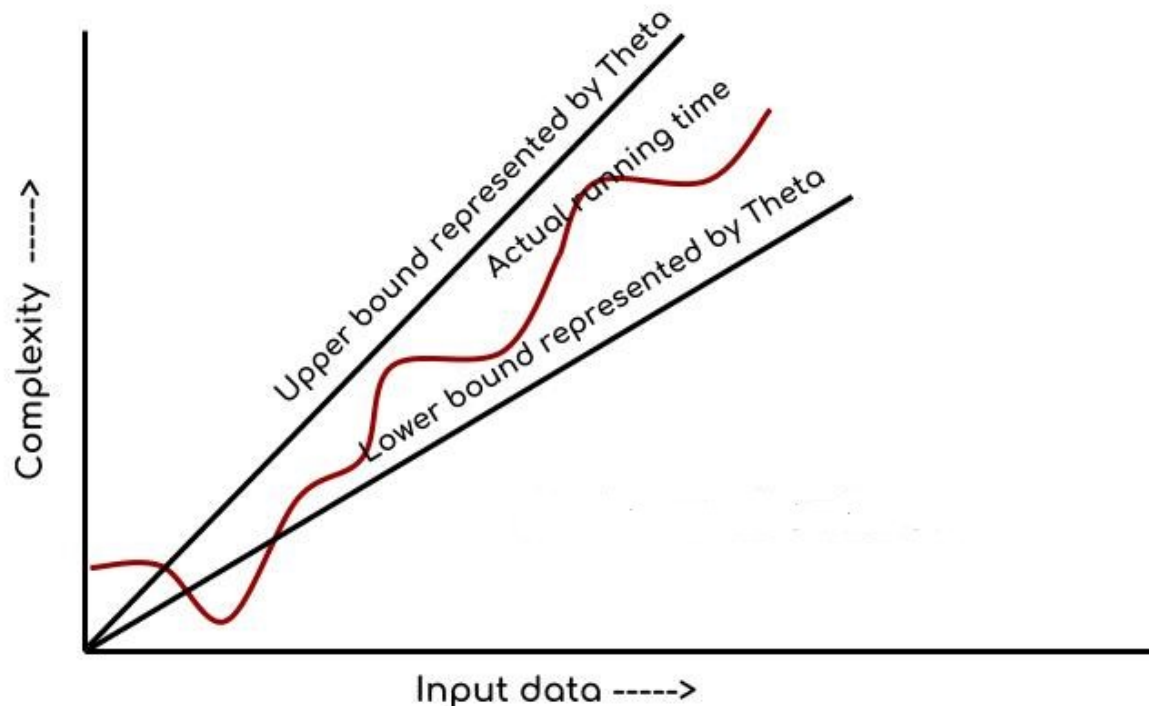
$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$

- Similar to the best case, the Omega notation is the least used notation among all three.



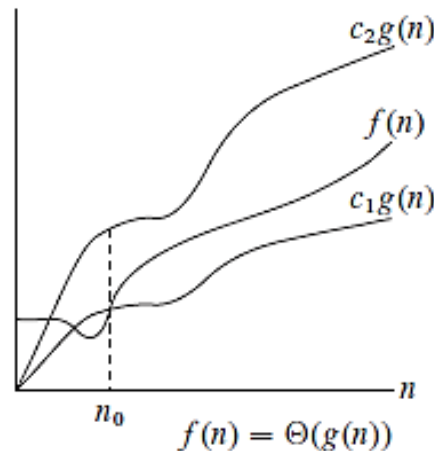
BIG THETA- Θ NOTATION

This notation describes both upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behaviour. In the real case scenario the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.



BIG THETA- Θ NOTATION

- $\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$



- The theta notation bounds a functions from **above** and **below**, so it defines exact **asymptotic** behavior.
- Thus $g(n)$ provides a nice, **tight bound** on $f(n)$.
- Note:
 - The definition of asymptotic is a line that approaches a curve but never touches.*

Time Complexity Analysis: **Big O**

RUNTIME ANALYSIS OF ALGORITHMS

A logarithmic algorithm – $O(\log n)$

Runtime grows logarithmically in proportion to n .

A linear algorithm – $O(n)$

Runtime grows directly in proportion to n .

A superlinear algorithm – $O(n \log n)$

Runtime grows in proportion to n .

A polynomial algorithm – $O(n^c)$

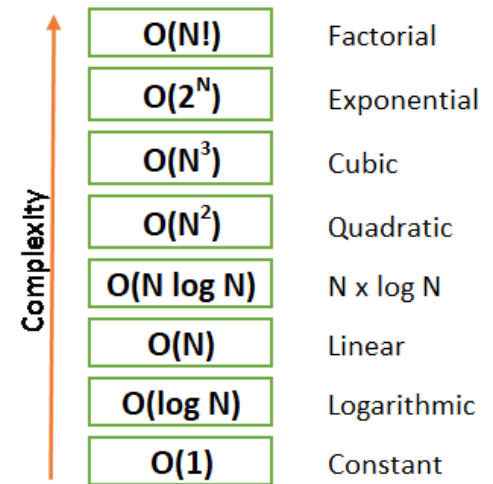
Runtime grows quicker than previous all based on n .

A exponential algorithm – $O(c^n)$

Runtime grows even faster than polynomial algorithm based on n .

A factorial algorithm – $O(n!)$

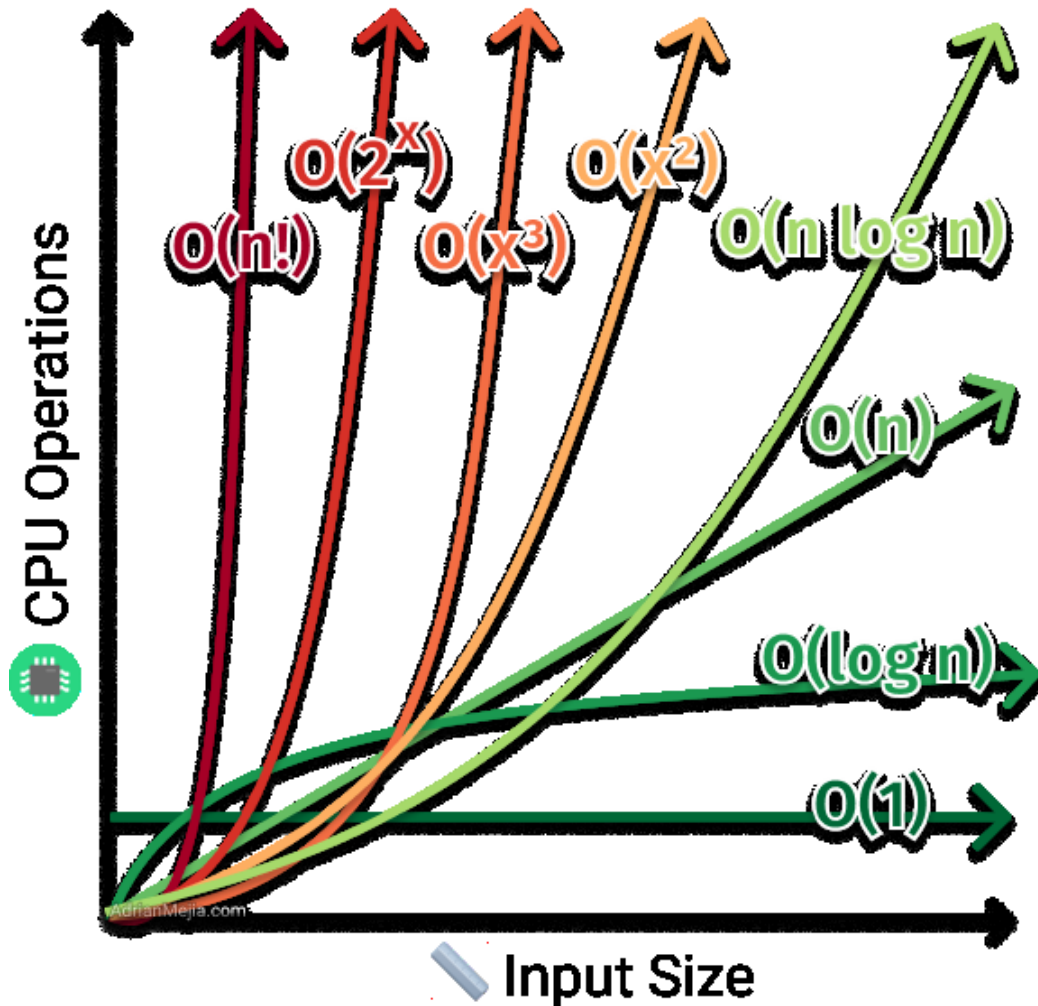
Runtime grows the fastest and becomes quickly unusable for even small values of n .



RUNTIME ANALYSIS OF ALGORITHMS



Time Complexity



$O(n!), O(c^n), O(n^c)$ - Worst

$O(n \log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

RUNTIME ANALYSIS OF ALGORITHMS

	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

RUNTIME ANALYSIS OF ALGORITHMS

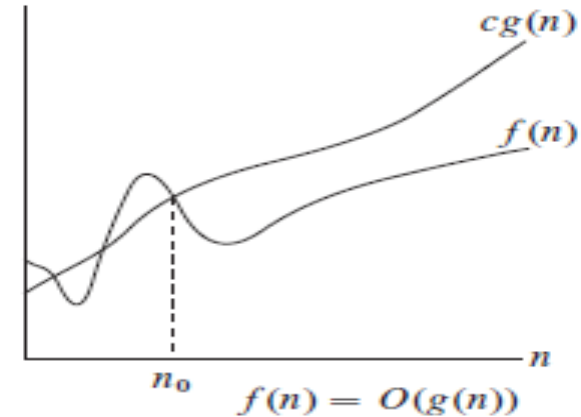
Some of the examples of all those types of algorithms (in worst-case scenarios) are mentioned below:

- *Logarithmic algorithm* – $O(\log n)$ – Binary Search.
- *Linear algorithm* – $O(n)$ – Linear Search.
- *Superlinear algorithm* – $O(n \log n)$ – Heap Sort, Merge Sort.
- *Polynomial algorithm* – $O(n^c)$ – Strassen's Matrix Multiplication, Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort.
- *Exponential algorithm* – $O(c^n)$ – Tower of Hanoi.
- *Factorial algorithm* – $O(n!)$ – Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem.

RUNTIME ANALYSIS OF ALGORITHMS

Example:

- Suppose $f(n) = 2n^2 + 5n + 1$. Is $f(n)$ is $O(n^2)$ or $O(n^3)$?



- Another Example: Suppose $f(n) = 4n^2 + 5n + 3$. Is $f(n)$ is $O(n^2)$?

Solution: $f(n) = 4n^2 + 5n + 3$

$$\leq 4n^2 + 5n + 3, \text{ for all } n > 0$$

$$\leq 4n^2 + 5n^2 + 3n^2, \text{ for all } n > 1$$

$$\leq 12n^2 \text{ for all } n > 1$$

$$4n^2 + 5n + 3 \leq 12n^2$$

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



Thanks to All