Unit-1 Introduction to Data structure and algorithm

Khushi Lad

What are Algorithms?

- An algorithm may be defined as… a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.
- An algorithm is a set of step-by-step instructions to solve a given problem or achieve a specific goal.

Structure of algorithm

- Input step
- Assignment step
- Decision step
- Repetitive step
- Output step

Properties/Charactersistics of Algorithm

- Finiteness
- Definiteness
- Generality
- Effectiveness
- Input-Output

- **Finiteness:**
- An algorithm must terminate after a finite (limited or fixed) number of steps.
- **Definiteness:**
- The steps of the algorithm must be precisely (exactly or surely) defined or unambiguously (clearly) specified.
- Each of its steps, and their inputs/outputs should be clear and must lead to only one meaning.

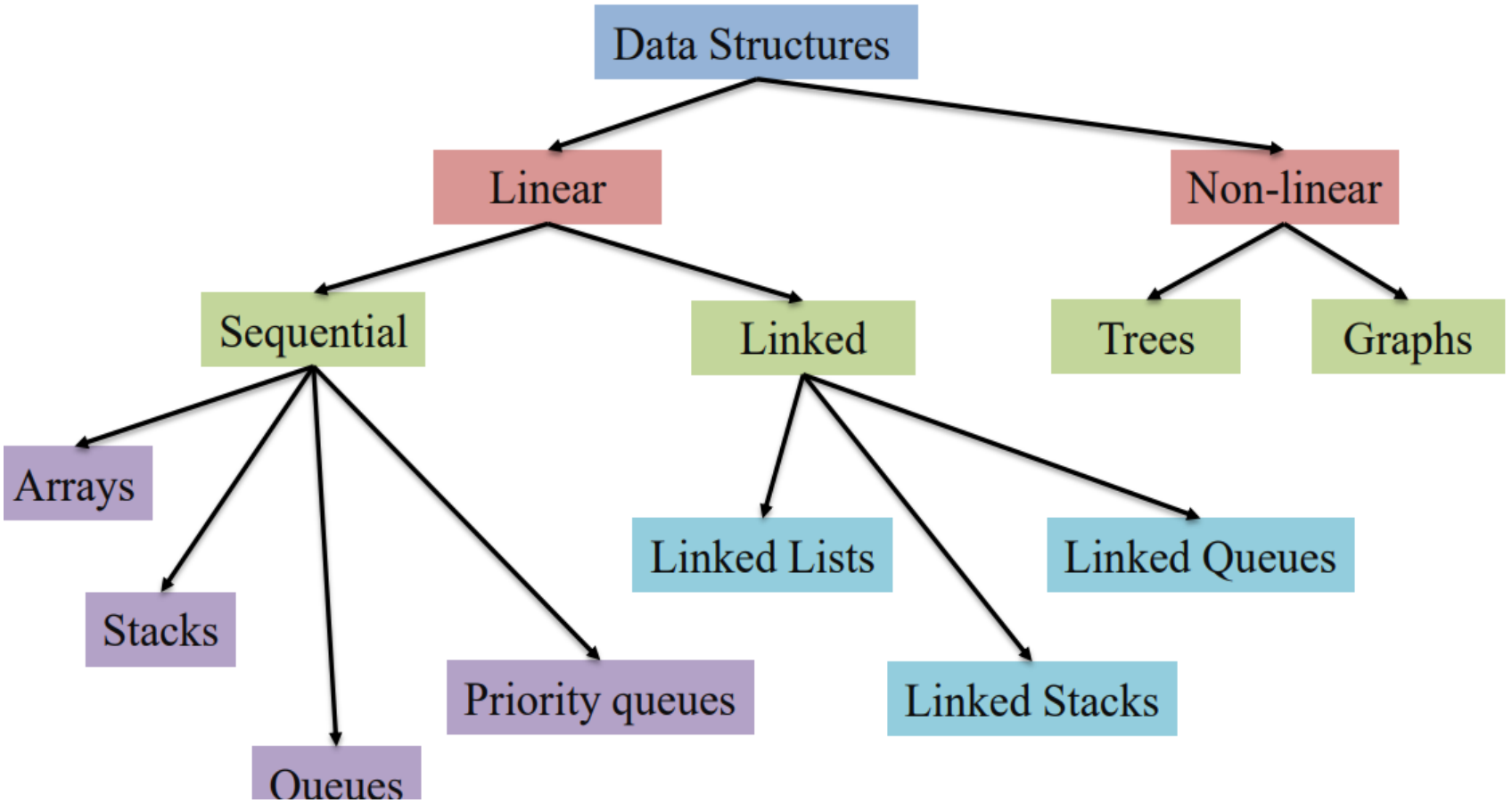
- **Generality:**
- An algorithm must be generic (basic) enough to solve all problems of a particular class.
- **Effectiveness:**
- The operations of the algorithm must be basic (straight-forward) enough that can be put down on pencil and paper.
- Algorithms should not be too complex to warrant (permit or license) in writing another algorithm for the operation!

- **Input-Output:**
- An algorithm must have certain initial (original) and precise (exact or accurate) inputs. i.e. it must have 0 or more well-defined inputs.
- And the outputs that may be generated from the inputs, may be both at its intermediate (in-between) and final steps.

Data Structures Overview

- A Data Structure (DS) is a way of organizing data, that can be used effectively.
- A Data Structure is a group of data of different datatypes, which can be processed as a single unit.
- A **record** is an example of a data structure.
- An **array** is also an example of a data structures.

Classification of Data Structures



Analysis of Algorithms

- In order to solve any problem in computer science, generally we write a program.
- And to write any program in any computer language we write small description (that can be in informal way) in the form of algorithm.
- Now, the algorithms might be of many types, which can solve the problem.

Efficiency of Algorithms

- The performance of algorithms can be measured on the scales of …
- 1. Time
- 2. Space.

- **Time:**
- Previously, it would mean looking for the fastest algorithm, for the problem or that which performs its task in the minimum possible time.
- Therefore,

  the performance measure is termed as time complexity.
- The time complexity of an algorithm or a program is a “ function of the running time of the algorithm or program. ”

- **Space:**
- In the later case, it would mean looking for an algorithm that consumes or needs limited memory space for its execution.
- The performance measure is such case is termed as space complexity.
- The space complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run for the completion.

Analysis of Algorithm

- So, before taking an algorithm and analyzing it, we should get familiar with the terminology used in algorithms.
- There are some of the notation which are used in the algorithms. One of it is…
  => Asymptotic Notations

Asymptotic Notations

- Asymptotic Notations are used to describe the running time of the algorithms or to express the time complexity of the algorithms.

Or

- Asymptotic Notations are the mathematical way to represent the time complexity of an algorithm.

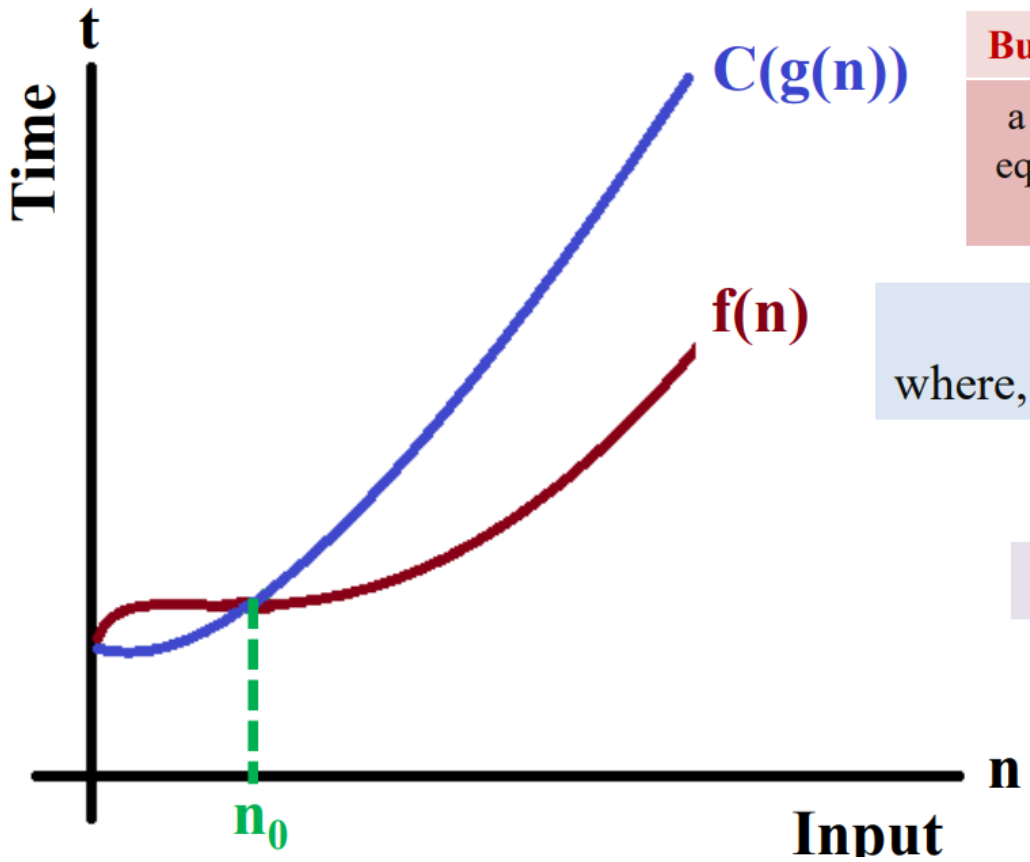
- Apriority analysis employs the following asymptotic notations.
- 1. Big (oh) – O
- 2. Big omega – Ω
- 3. Big Theta – Θ

O means "order at most"
Ω means "order at least"
Θ means "order exactly"

Big (oh) – O

1. Big (oh) --- O

This notation defines an **upper bound** of an algorithm.



But, what is upper bound ???

a value that is greater than or equal to every element of a set of data.

$f(n) \leq C(g(n))$,
where, $C > 0$, $n \geq n_0$ and $n_0 \geq 1$

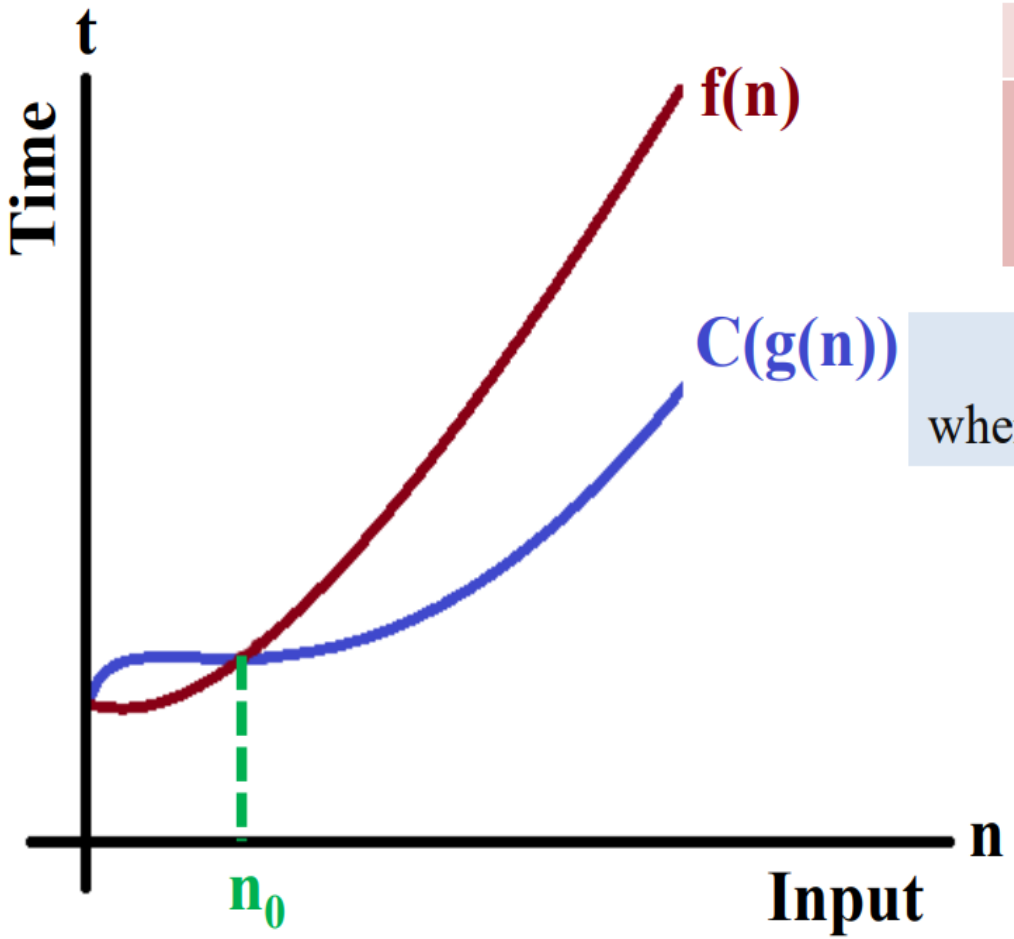
$f(n) = O(g(n))$

- So, you can see in the graph that red color line, if we keep on giving inputs, the time is also increasing. Let us called it as function f(n) (read as f of n).
- You can see in the graph that blue color line, let us called it as function C(g(n)), which is known as upper bound function, that is greater than the function f(n). [i.e. C(g(n)) > f(n)]
- So, after some limit we have n0 (read as n not), that is the green color line in the graph.
- So, If $f(n) \leq C(g(n))$ is satisfied, then we say that f(n) = O(g(n)) (read as f of n is "big oh" of g of n), but it is also f(n) is smaller than g(n).
- Therefore, in **Big O, f(n) will be always less than g(n),** it can be equal to, but **f(n) can never be greater than g(n).**

- For example:
- If $f(n) = 3n + 2$ and $g(n) = n$, then...
- $f(n) = O(g(n))$
- $f(n) \leq C(g(n))$, where $C > 0$, $n0 \geq 1$
- $3n + 2 \leq C\,n$ Lets take $C = 4$
- $3n + 2 \leq 4n$
- $2 \leq 4n-3n$
- $n \geq 2$

Big Omega

2. Big omega --- Ω This notation defines a **lower bound** of an algorithm.



But, what is lower bound ???

A value that is less than or equal to every element of a set of data.

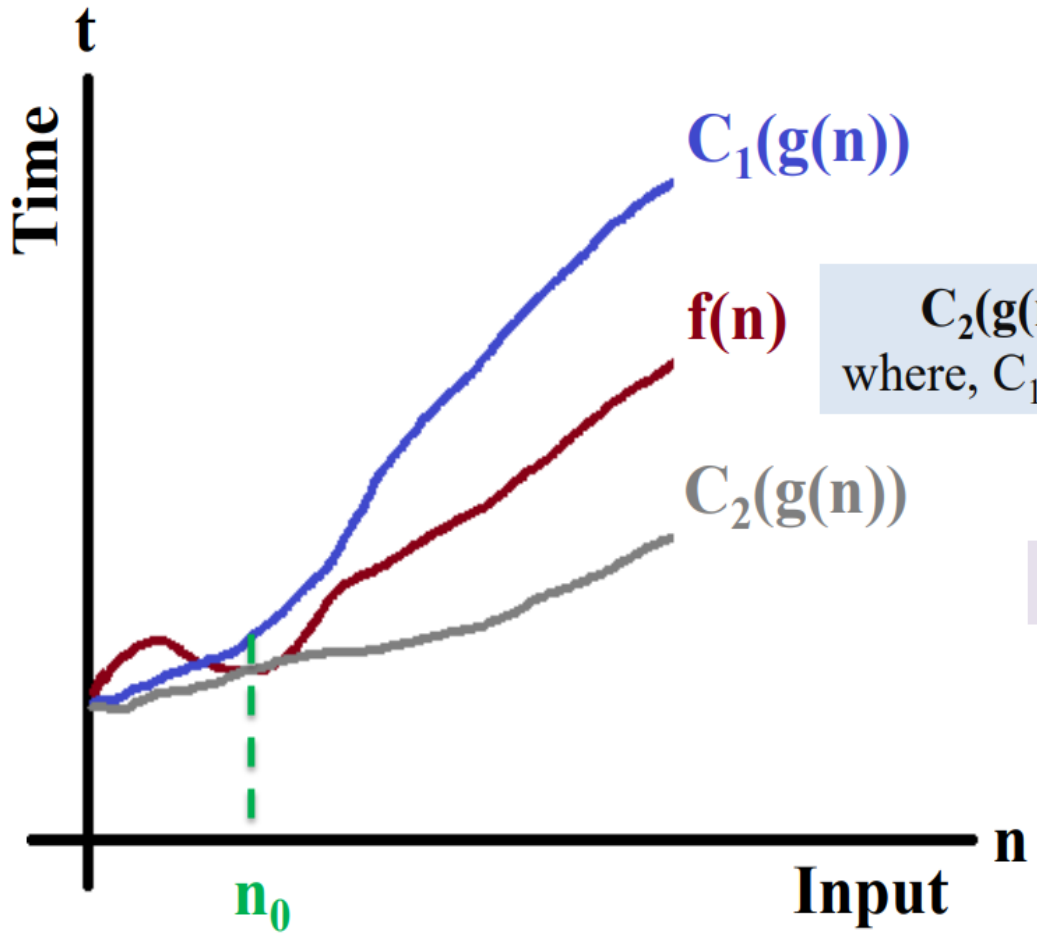
$f(n) \geq C(g(n))$,
where, $C > 0$, $n \geq n_0$ and $n_0 \geq 1$

$f(n) = \Omega(g(n))$

- So, you can see in the graph that red color line, if we keep on giving inputs, the time increases. Let us called it as function f(n).
- You can see in the graph that blue color line, let us called it as function C(g(n)), which is known as lower bound function, that is smaller than the function f(n).
- So, after some limit we have n0 , that is the green color line in the graph.
- So, If $f(n) \geq C(g(n))$ is satisfied, then we say that $f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), but it is also… f(n) is greater than g(n).
- Therefore, in **Big Ω, f(n) will be always greater than g(n),** it can be equal too, **but f(n) can never be smaller than g(n).**

Big Theta

3. Big Theta --- Θ This notation defines a **tight bound** of an algorithm.



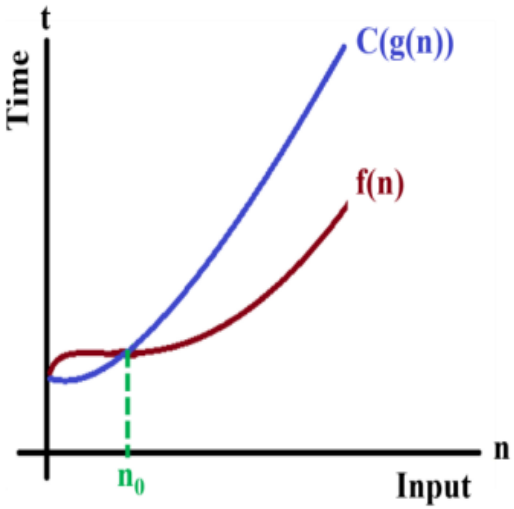
$C_2(g(n)) \leq f(n) \leq C_1(g(n))$,
where, $C_1, C_2 > 0$, $n \geq n_0$ and $n_0 \geq 1$

$f(n) = \Theta(g(n))$

- So, now you can see in the graph that red color line, let us called it as function f(n).
- Next that blue color line, let us called it as function C1 (g(n)), which is greater than function f(n).
- And that grey color line, let us called it as function C2 (g(n)), which is smaller than function f(n).
- So, after some limit we have n0 , that is the green color line in the graph.
- **So, If C2 (g(n)) ≤ f(n) ≤ C1 (g(n)) is satisfied, then we say that… f(n) = Θ(g(n)) (read as f of n is theta of g of n).**

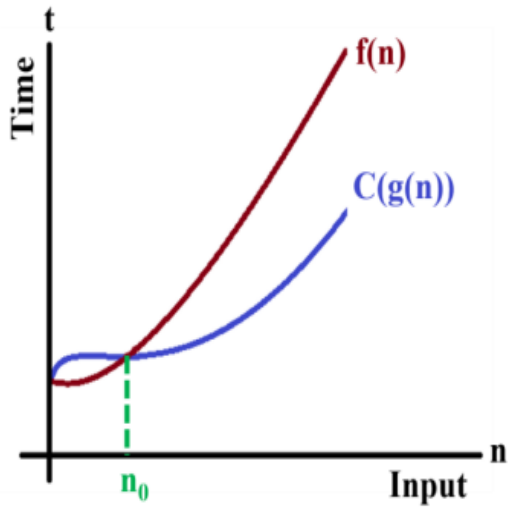
Best, Average, Worst

- The instances or input instances for which the algorithm takes the maximum possible time is called the worse case. And the time complexity in such a case is referred to as the worst case time complexity.
- The input instances for which the algorithm takes the minimum possible time is called the best case. And the time complexity in such a case is referred to as the best case time complexity.
- All other input instances which are neither of the two categorized as the average cases. And the time complexity of the algorithm in such cases is referred to as the average case time complexity.



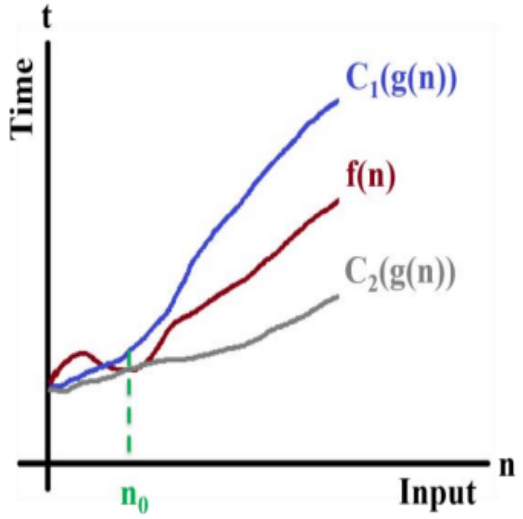
$f(n) = \mathbf{O}(g(n))$

Worst Case



$f(n) = \mathbf{\Omega}(g(n))$

Best Case



$f(n) = \mathbf{\Theta}(g(n))$

Average Case

- Therefore, very often the running time of the algorithm is depended on the input size.
- In such a case, it is fair enough to assume that larger the input size of the problem cases, the larger is its running time. But, such case is not always.
- So, there are problems whose time complexity is not just dependent on the size of the input, but on the nature of the input as well.