Day 1
19/02/2024

# Time AND Space Complexity

## * what is Time Complexity:

→ Time complexity is a computational concept that describes the amount of computational time that an algorithem takes to complete as a function of length of the input.

→ it provides a theoritical estimate of growth rate of running time of an algorithem as the size of input data increases.

→ Time complexity is often expressed using Big O notation, which characterizes functions according to their growth rates, disregarding constant factors and lower order terms.

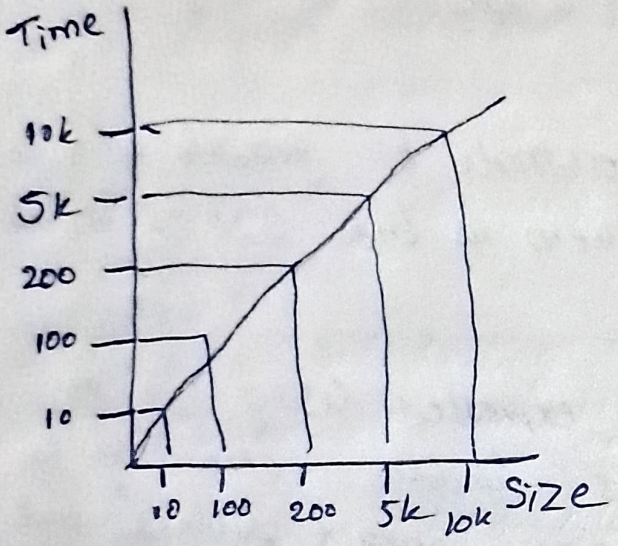
Purpose: Time complexity gives us an idea of the scalability of an algorithem and how efficiently it can handle large data sets.

| Example: | old computer | New m1 macbook (very fast) |
|---|---|---|
| data: 1,000,000 elements in an array | → | " |
| Algorithem: linear search for a Target That does not exist in array | → | " |
| Time Taken : | 10 secs | → Time Taken : 1 sec |

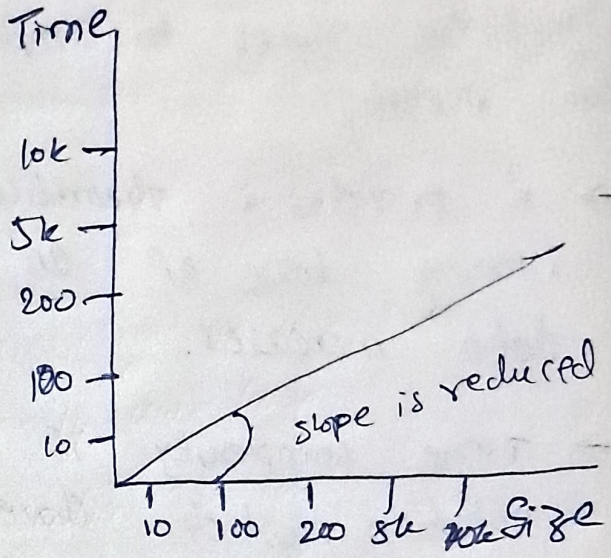
* Both computers have Time complexity ⊖ → confused

# Time Complexity != Time Taken

old computer

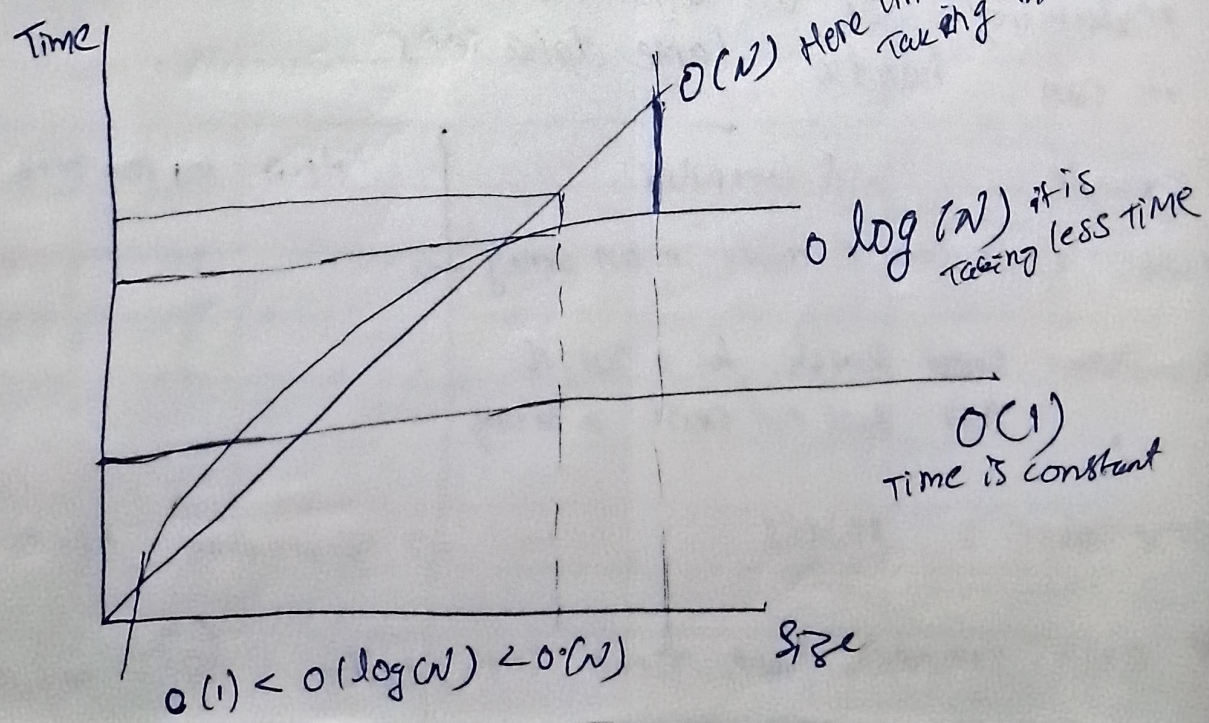



m1 macbook

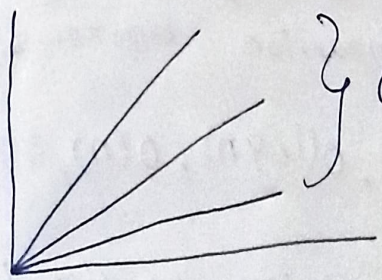



* function that gives us the relationship about how the Time will grow as the input grows

**why?**



what do we need to consider Thinking about Complexity ?

* always look for <u>worst case</u> Complexity
* always look at complexity for large/$\infty$ data
* 

$O(N)$

$y = x$
$y = 2x$
$y = 4x$

→ even through value of actual time is different they are all growing linearly.

→ we dont care about actual time

→ constants doesn't matter, we ignore all constants.

* $O(N^3 + \log N)$

from point number Two

$1\,mil \Rightarrow ((1\,mil)^3 + \log(1\,mil))$

$= 1\,mil^3 + \underbrace{6\,sec}_{\text{very small hence ignore}}$

* Always ignore less dominating terms

Ex: $O(3N^3 + 4N^2 + 5N + 6)$

$= (N^3 + N^2 + N) = O(N^3)$ } considering all above points

***

## Big O Notation:

→ upperbound

→ Big O notation is a mathematical way to describe how the time or space complexity of an algorithm grows relative to the size of the input.

→ it focusses worst case scenario ignoring constants and lower order terms to simplify comparison between algorithms.

→ By expressing complexity as $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$ etc...

→ it helps understanding and predicting an algorithms performance and scalability, especially with large datasets.

→ This makes Big O notation essential for analyzing, comparing and optimizing algorithms in computer science.

$$O(N^3) \longrightarrow \text{upper bound}$$

Ex:

maths representation

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

$$\boxed{\lim_{N \to \infty} \frac{f(N)}{g(N)} < \infty}$$

$$O(N^3) = O(6N^3 + 3N + 5)$$

$g(N)$ - $f(N)$

$$\lim_{N \to \infty} \frac{6N^3 + 3N + 5}{N^3}$$

$$\lim_{N \to \infty} \frac{6N^3}{N^3} + \frac{3N}{N^3} + \frac{5}{N^3}$$

$$\lim_{N \to \infty} 6 + \frac{3}{N^2} + \frac{5}{N^3}$$

$$\cancel{\lim_{N \to \infty}} = 6 + \frac{3}{\infty} + \frac{5}{\infty}$$

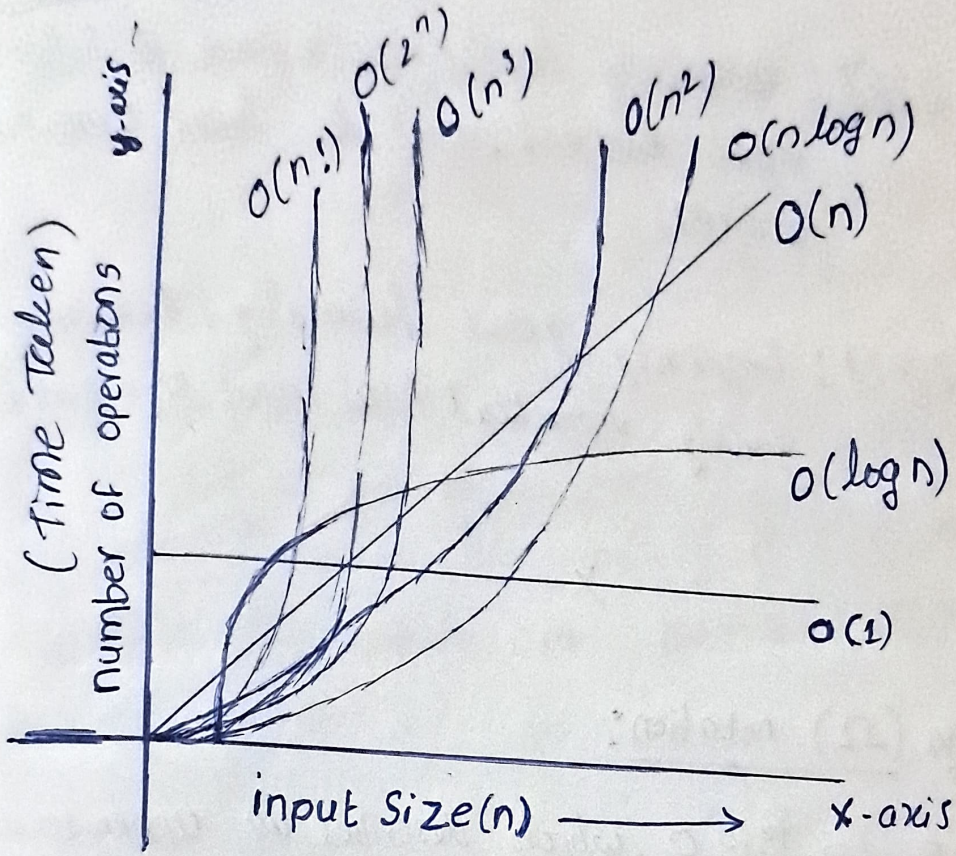
$$= 6 + 0 + 0$$

$$= 6$$

$$6 < \infty$$

6 → finite value
→ upper bound

Common Rate of Growths in Algorthmic Space and Time Complexity:




* constant O(1) : fixed complexity, uneffected by input size
* Linear O(n) : complexity grows directly with input size common for simple iterations
* Logarithmic O(logn) : complexity increases logarithmically with input size. efficient for large data
* Linearithmic O(nlogn) : faster than linear but slower than quadratic. typical for efficient sorting algorithms.
* Quadratic O(n²) : complexity grows quadratically with input size seen in algorithems with nested loops

Cubic $O(n^3)$ : Complexity grows cubically, often impractical for large data

Exponential $O(2^n)$ : Complexity doubles with each additional input element. used for some combinatorial problems

Factorial $O(n!)$ : Complexity grows factorially, becoming quickly impractical as input size increases.

---

## Big omega ($\Omega$) notation:

* opposite to Big O, which describes an upper bound
* omega notation ($\Omega$) is a formal way to express the lower bound of an algorithm's time or space complexity.
* while Big O notation is used to characterize the worst case scenario or the maximum time an algorithem might take
* omega notation ~~make~~ focuses on the best-case scenario or the minimum time an algorithm requires regardless of the input size.

Maths:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

## Theta $\theta$ notation:

* it represents a tight bound on the running time of an algorithm, indicating that an algorithm's running time grows asymptotically at the same rate as it's input size, Both in Best case and worst case scenarios.

* it provides precise measure of an algorithm's efficiency by defining the exact rate growth

* unlike Big O (upper bound) or omega ($\Omega$, lowerbound) notations.

* for example, if an algorithm has a time complexity of $\Theta(n^2)$, it means the time to complete the task grows quadratically with the input size n, neither faster nor slower, for large values of n.

$$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

## little O notation:

* This is also giving upper bound
* But This is a loose upper bound.

| Big O | little O |
| --- | --- |
| $f = O(g)$ | $f = o(g)$ |
| $f \leq g$ | $f < g$ ← strictly lower than G |

This is a stronger statement

math)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Ex: $f = N^2$ $g = N^3$

$$\lim_{N \to \infty} \frac{N^2}{N^3} = \lim_{N \to \infty} \frac{1}{N} = 0$$

## little omega ($\Omega$):

| Big $\Omega$ | little $\Omega$ |
| --- | --- |
| $f = \Omega(g)$ | $f = \Omega(g)$ |
| $f \geq g$ | $f > g$ |

maths

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

Example:

$$\lim_{N \to \infty} \frac{N^3}{N^2}$$

$$\lim_{N \to \infty} N = \infty$$

# Space Complexity:

* Space complexity refers to the amount of memory space required by an algorithm to execute and complete it's function, relative to the size of input.

* it includes both the static space used by the algorithm (space for code, fixed-size variables and constants) and the dynamic space (space for variable-size data structures, dynamic allocations, and recursion stack space).

* understanding space complexity is crucial for Evaluating an algorithm efficiency, especially in environments with limited memory resources.

## Key Components:

1. Fixed Space Requirements

* This includes space for variables, constants, and instruction code. it remains constant irrespective of the input size.

## 2. Variable space Requirements:

This varies with the size of the input and includes:

- → space needed by input data
- → space for dynamic data structures (eg.. lists, stacks, queues)
- → space for function call stacks (especially relevent in recursive algorithms)

## Calculating Space Complexity:

* To determine an algorithm's space complexity, we sum up the fixed space and the variable space required by the algorithm.

* the notation used to express space complexity is similar to the time complexity, often used Big o notation to describe the upperbound of the space needed.