15/9/22

# TIME COMPLEXITY

## ⇒ What is Time Complexity?

Let us consider an example of two computers and a certain algorithm that they have to perform.

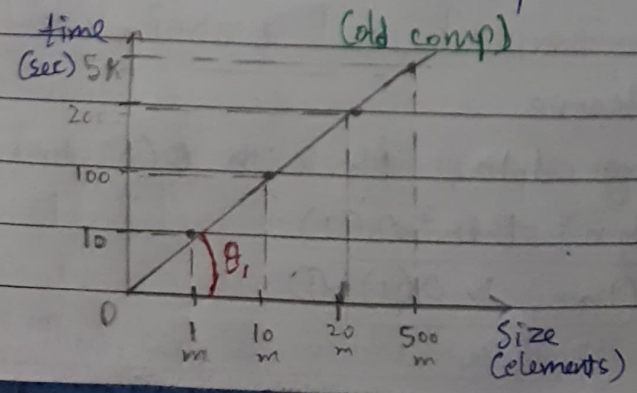
| | Old Computer (very slow) | M1 Macbook (very fast) |
|---|---|---|
| dataset : | 1,000,000 elements in an array | 1,000,000 elements in an array |
| Algo : | Linear search for target that doesn't exist | Linear search for target that doesn't exist |
| Time Taken : | 10 seconds | 1 second |

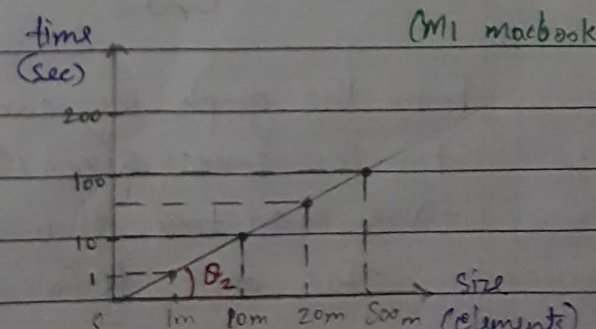
# So now the question is, which computer has better time complexity?

#* The answer is Neither because both computers have SAME time complexity.

* So the conclusion is **Time Complexity ≠ Time Taken**

Now to understand what actually time complexity is, let us make graphs for above example:



From the graphs, we can observe that
Even though the time taken at corresponding sizes is different, the Relationship between the size and the time is SAME i.e LINEAR

# * Thus, Time Complexity is basically a mathematical function that tells us "how the time is going to grow as the input grows".

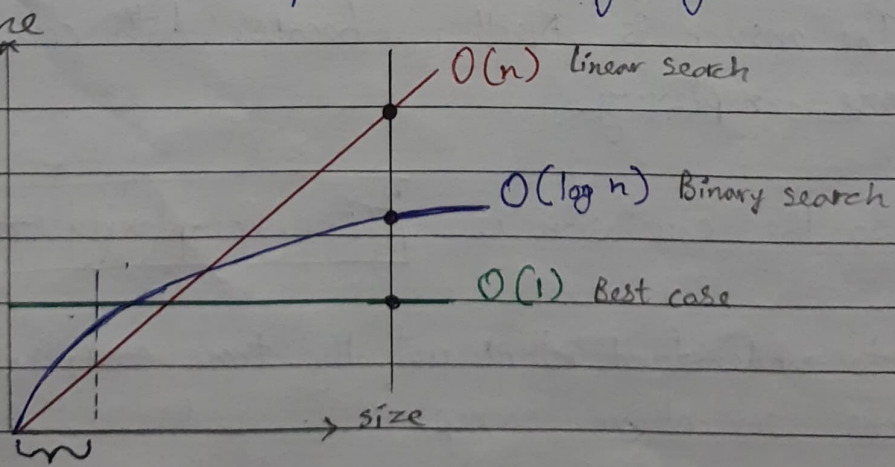
* Here, the time is growing linearly as the size is growing in both cases therefore both computers have linear time complexity i.e $O(n)$.

---

⇒ **Why** time complexity?

Time Complexity provides a way to compare different algorithms, to solve the same problem and to choose the most efficient one of them for the situation.

Let us consider an example of searching algorithms:




From the graph we can observe,
For the same size of large data, Algo with $O(n)$ time complexity takes most time then $O(\log n)$ then $O(1)$

Thus, $\boxed{O(n) > O(\log n) > O(1)}$ -①

and using this comparison we can opt for the best algo.

⇒ What to consider when thinking about complexity?

0> We don't care about ACTUAL value of time taken.

1> Always consider the worst case complexity.

2> Always look at complexity for large/infinite data..

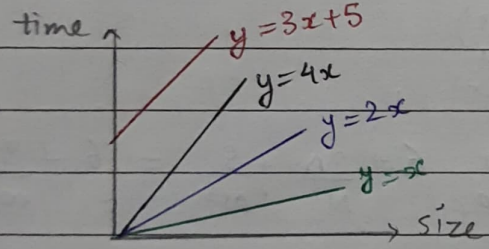
→ Why?

* In above graph, we can see that eq① doesn't hold true for the starting portion of the graph.
* But for that portion, the size of data is already very small thus the time taken will also be small regardless of the complexity
* Also, the eq① holds true indefinitely after that portion.
* Thus, small data is of no concern when considering complexity.

3> Ignore all constants.

→ Why?

Let us consider an example,




* Even though the actual value of time taken is different, each of them are growing linearly.

* Since, from rule 0, we don't care about Actual time taken, thus instead of writing $O(3x+5)$, $O(4x)$, $O(2x)$, $O(x)$, we write $O(x)$ for each of them & ignore the constants

4> Always ignore Less Dominating Terms.

→ Why?

* Let say an algo has complexity $O(N^3 + \log N)$

For 1 million size of data, it'll take $((1\text{ million})^3 + \log(1\text{ million}))$

i.e $(10,00,00,00,00,00,00,00,00 + 6)$ → from rule 2, very small, hence ignore

Thus, complexity becomes $O(N^3)$.

16/9/22

## ⇒ How Time Complexity?

Time Complexity of an algo is given by Asymptotic Notations. i.e Big Omega ($\Omega$), Big Theta ($\Theta$) and Big O ($O$)

### BIG O

In simple words, Big O gives the UPPER BOUND of the time complexity of an algo. or It gives us the WORST CASE Time Comp.
eg → $O(n^3)$ means the algo will never exceed $n^3$ time complexity it may take place in $n^3, n^2, n$ or $\log n$ etc time complexity but will never exceed $n^3$.

Mathematically it says, Let $f(n) = O(g(n))$

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

as n approaches $\infty$ (large data)

this value will always be a finite value (upper bound)

eg → $\underbrace{6N^3 + 3N + 5}_{f(n)} = O(\underbrace{N^3}_{g(n)})$

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

$= 6 + 0 + 0$

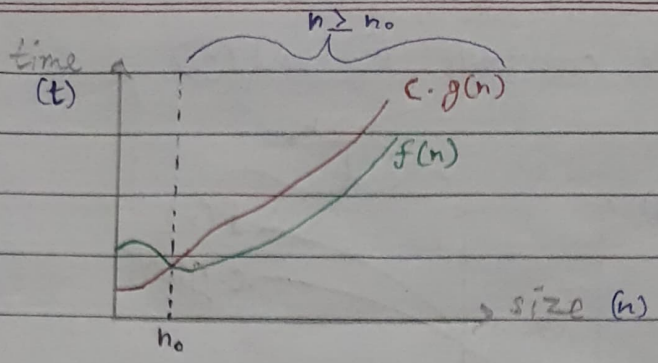
$\therefore \lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{6N^3+3N+5}{N^3} = \boxed{6 < \infty}$

↳ finite value (upper bound)

[Another definition]

⇒ Let us suppose there are two functions $f(n)$ and $g(n)$ then $\boxed{f(n) = O(g(n))}$ if and only if there exist two possible constant ~~such that~~ c and $n_0$ such that $\boxed{f(n) \leq C \cdot g(n)}$ for all $n_0 \leq n$ and for any constant ©.

16/9/22




# Simple words mai, (no) ki koi esi value chuno jiske baad se c·g(n) bada hoga f(n) se tab hoge f(n) = O(g(n)).

eg → $f(n) = n^3 + 1$

$g(n) = n^3$

Let $c = 2$

$\because \quad n^3 + 1 \leq 2n^3$

$f(n) \leq c\,g(n)$

$\therefore \quad n^3 + 1 = O(n^3)$

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

## BIG OMEGA ($\Omega$)

In simple words, Big Omega is the opposite of Big O i.e it gives the LOWER BOUND of the time complexity of an algo.
or It gives the BEST CASE time complexity.

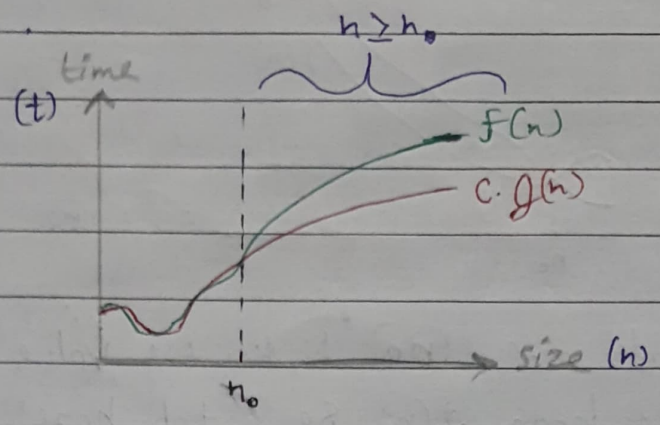
eg → $\Omega(n^3)$ means that the algo will take place in atleast/minimum $n^3$ time complexity, it may be $n^3$, $n^4$, $2^n$ etc but it will atleast be $n^3$.

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

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

[Other definition]
A function f(n) is said to be $\Omega(g(n))$ if and only if there

exists two possible constants (c) and ($n_0$) such that $c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.




eg → $f(n) = n^3+1$, $g(n) = n^0$, $c=1$

$\because \quad 1 n^0 \leq n^3+1$

$c\, g(n) = f(n)$

$\therefore \quad n^3+1 = \Omega(1)$

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

## BIG THETA ($\Theta$) Better picture of runtime, "order of"

In simple terms, Big Theta is the combination of Big O and Big $\Omega$ i.e it gives BOTH Upper Bound & Lower Bound of time complexity

or It gives the AVERAGE CASE time complexity

eg → $\Theta(n^3)$ means that an algo has both upper and lower bound of $(n^3)$ i.e $O(n^3)$ & $\Omega(n^3)$

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

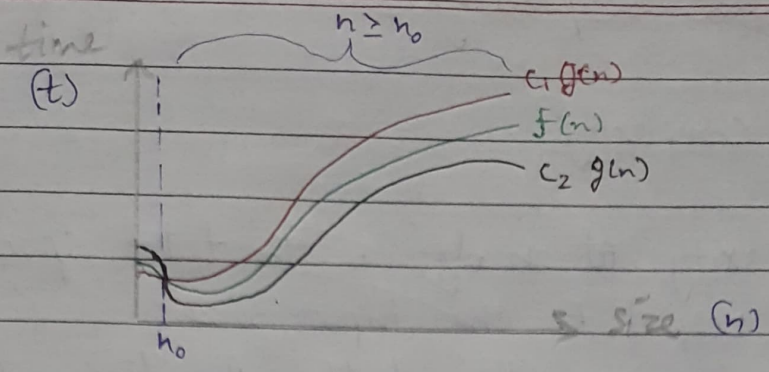
when, $$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

[Other definition]

A function $f(n)$ is said to be $\Theta(g(n))$ if and only if there exists three possible constants ($c_1$)($c_2$) ($n_0$) such that

$$c_2 g(n) \leq f(n) \leq c_1 g(n)$$

for all $n \geq n_0$




## Little (o) Oh

In simple words, it also gives the upper bound as Big O does but its a stronger statement because for Big O, $f(n) \leq cg(n)$ but for Little o, $f(n) < cg(n)$.

↳ growth of f is strictly slower than g

Mathematically, $f(n) = o(g(n))$

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

eg → $f(n) = n^2$, $g(n) = n^3$

then, $\lim_{n \to \infty} \frac{n^2}{n^3} = \lim_{n \to \infty} \frac{1}{n} = \frac{1}{\infty} = \boxed{0}$

$\therefore \quad n^2 = o(n^3)$

## Little (ω) Omega

In simple word, little ω also gives the lower bound as Big Ω but its is a stronger statement because for Big Ω $f(n) \geq cg(n)$ but for Little ω, $f(n) > cg(n)$

↳ growth of f is strictly faster than g

Mathematically, $f(n) = \omega(g(n))$

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

eg → $f(n) = n^3$, $g(n) = n^2$ then $\lim_{n \to \infty} \frac{n^3}{n^2} = \lim_{n \to \infty} n = \boxed{\infty}$

24/9/22

Date
Page 8

# Space Complexity

→ What is Space Complexity?

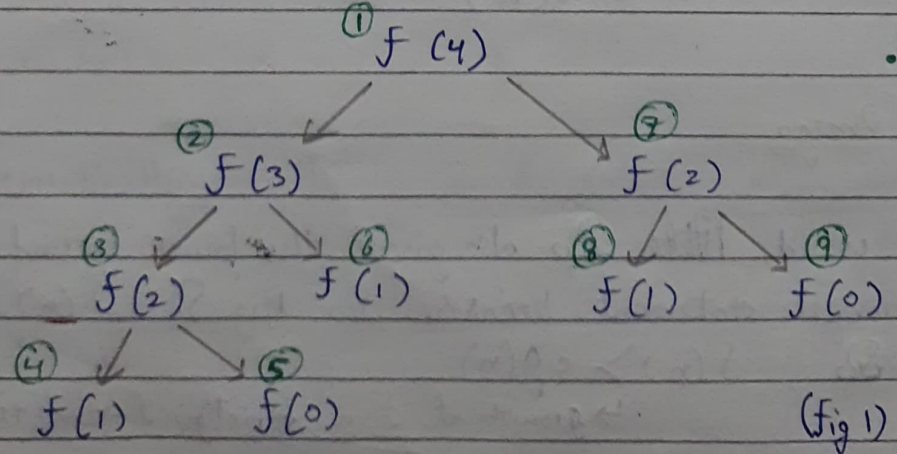
* Space Complexity of an algo is the total space taken by the algo with respect to the input size.
* Space complexity includes both Auxiliary space and space used by input.
* Auxiliary Space is the extra space or temporary space used by an algo.

* But sometimes, like in the case of standard sorting algorithms, Auxiliary space is a better criteria for comparison than Space complexity.

eg → Merge sort, Insertion sort & Heap sort all have space complexity O(n) but Merge sort uses O(n) auxiliary space while Insertion, heap sort use O(1) auxiliary space

⇒ Space complexity in recursive Algorithms

Let us consider the example of finding 4th fibonacci number;



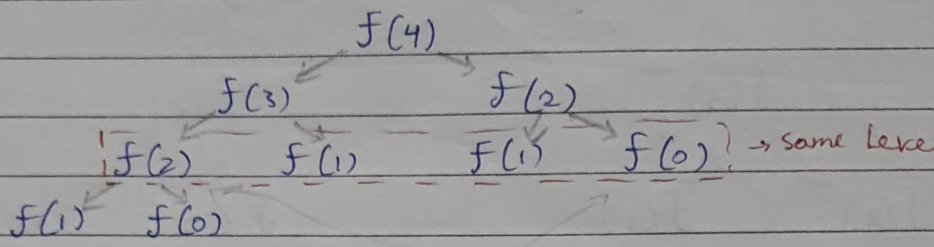

• flow of function calls

(fig 1)

In recursive algo like this, the function calls are actually stored in stack, they take some memory in stack thus the space complexity can not be O(1).

24/9/22

So the question now is, At any point, is it possible to have more than one function call, at the same level, in the stack at same time?



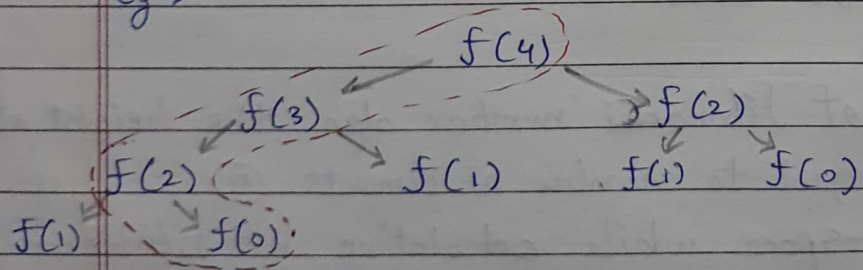

Nope! Because f(0) will not even execute till f(2) is finished as seen in the flow of function calls (fig 1).

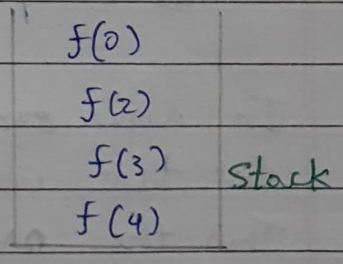
Thus the point to remember is :-

# At any point in time, No two function calls at the same level of recursion will be in the stack at the same time.

# Only function calls that are interlinked will be in the stack at the same time

eg →



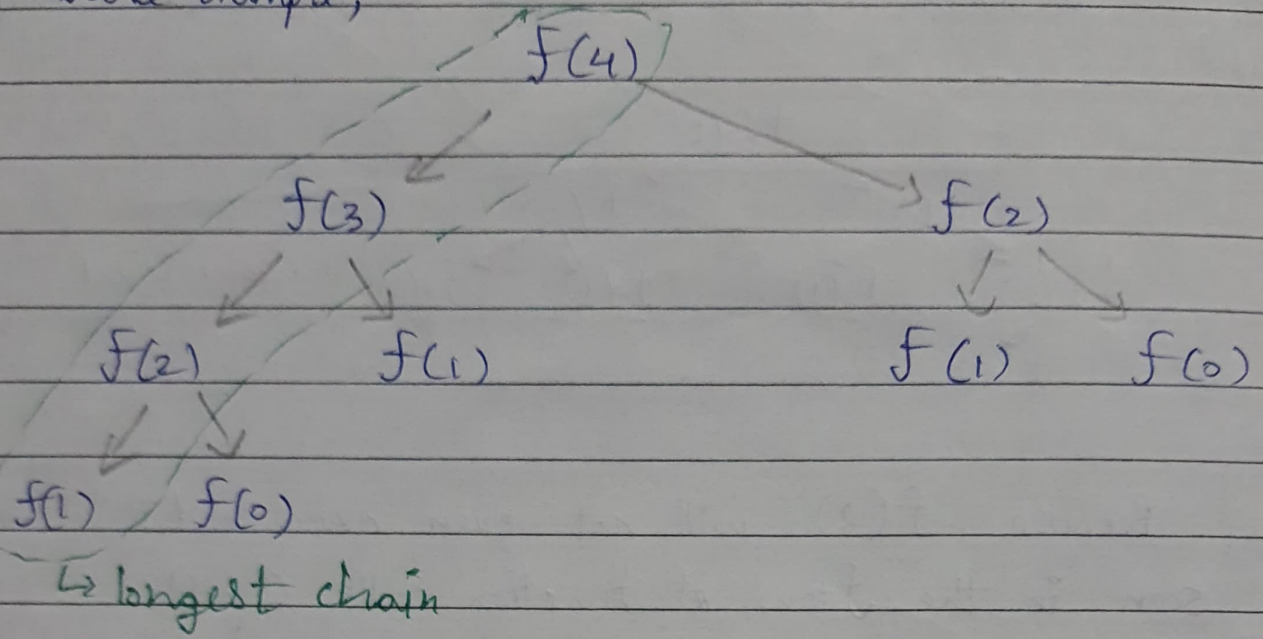

* This interlinking should be in one direction / one flow.

So, what is the maximum space that can be taken at any time? Is it possible to have all 9 function calls in stack at same time?

No, the maximum number of function calls to be in the stack at same time will be the number of function calls present

in the longest chain, starting from root till leaf.

In above example,




Thus, it is possible to have maximum of 4 function calls in the stack, in this case, at a time.

Therefore, we can say:

# Space complexity in recursive algo. is given by the ~~tog~~ longest chain in the tree or the height of the tree or the path of the tree

In above example of fibonacci number algo, the height of the tree will be equal to number of elements (n)
Thus the auxiliary space while calculating $n^{th}$ fibonacci number is $O(n)$.