Time & Space Complexity Analysis :-

- what is Complexity / Time Complexity?

→ It's not equal to time taken. ★ (TC != TT)

→ It's a mathematical function that tells us how the time will grow as the input grows.

→ It doesn't give us time.

→ It gives the relation b/w ↑ time with ↑ipt.

Eg: 'A' has 1 Billion elements

→ Performs linear Search (Doesn't find elements)

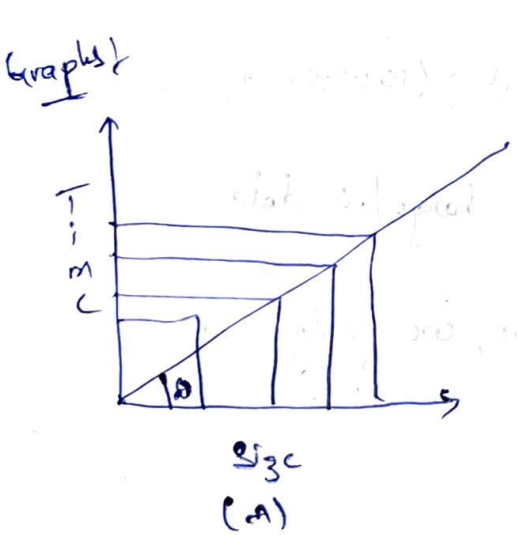
→ Does it in 10 secs

on the other hand, 'B' also has 1 Billion elements

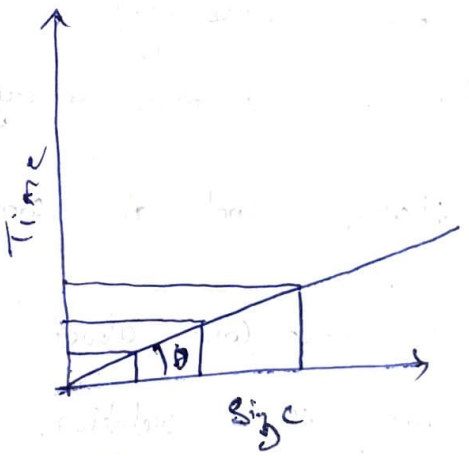
→ Performs Linear Search (Doesn't find)

→ takes 1 Sec

Graphs:



(A)



(B)

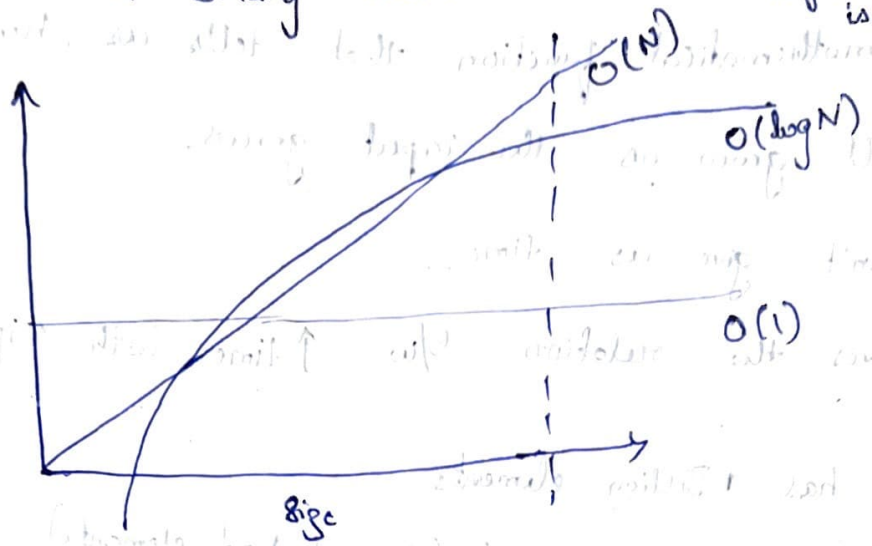
- only teta (θ) value decreased, but the time is linear.
- Both are of time Complexity.

* Why is this relation important? (why we bother)

- In time complexity, always care about large data, the bigger picture / larger data.

Eg:- Linear & Binary Search

shows which Algo/Approach is better



This shows for larger data :- $O(1) < O(\log N) < O(N)$

* What do we need to consider when thinking about Time Complexity?

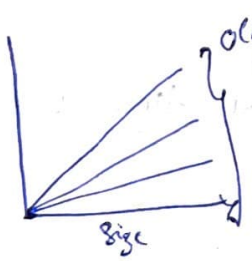
(1) Always look for worst case complexity (Bz it's always more concerning)

Eg:- which is more worrysome of crash? (10 users or 10 mil)

(2) Always look at complexity for large / ∞ data

(3) We dont care about actual time, we only care about the relation.

Eg:-



$O(N) \Rightarrow y = x$
$y = 2x$
$y = 4x$
} different times
But all are growing linearly.

***

- This is why we ignore all constants in time complexity.

**

④ Always ignore the less dominating terms.

Eg: If time Complexity = $O(N^3 + \log N)$

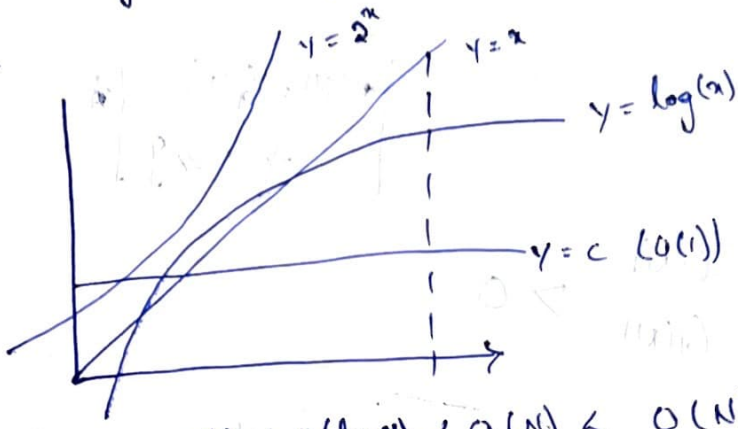
Here, $\log N$ is very vvv small when compared to N^3.

So, No need to bother about it. ignore it.

$\Rightarrow O(N^3)$ //.

- $O(N^3)$ (or) $y = 2^{n(x)}$ is exponential time Complexity and it is very bad, very slow.

Graph:



$O(1) < O(\log N) < O(N) < O(N^2)$

---

* Big-Oh Notation :-

- It says the graph/relation you have is the upper strict Bound.
- Complexity/Graph/relation cannot exceed the time Complexity.

Eg: $O(N^3)$ => Graph/relation maybe solved in N, $\log N$ etc times but cannot exceed or take longer than N^3! (will never be N^4 or etc).

$[f \leq g]$ *

Mathematically: $f(N) = O(g(N)) \Rightarrow$ Finite Value / upper Bound

$$\lim_{n \to \infty} \frac{f(N)}{G(N)} \leq \infty$$

(when value of 'N' reaches ∞) //.

- Algorithm can do better than Big-oh notation but can never exceed it.

* Big-Omega Notation:

- It's basically opposite of Big-oh.
- Meaning it's lower Bound.

Eg: If TC = $\Omega(N^3)$ ⇒ A Algorithm will take minimum of N^3 time, can take more.

- An algorithm can do/take more than it but never less than it.

⇒ $\Omega(N)$

$[f(N) = G(N) \Rightarrow f \gg g]$ *

mathematically,

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

* Everybody cares of Big-oh mostly Eg it's worst Case.

* Big-Theta Notation:

- To specific the running time (Both upper & lower bound combined).

Eg: An algorithm has Both upper & lower bound as N^2.

⇒ $O(N^2)$ & $\Omega(N^2)$

This is redundant, So we combine both

⇒ $\Theta(N^2)$ ⇒ mathematically,

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

[Combining Both Upper & lower Bound]

* Little Oh Notation: $o(N^2)$

- This also gives upper bound but a loose one.
- It tells that it can be strictly less than little oh
- It's a stronger statement.

mathematically, $\lim_{n\to\infty} \frac{f(N)}{G(N)} = 0$ $[f < g]$ *

* Little Omega Notation: $(\omega(\bullet N^2))$

- This also gives lower bound but a loose one.
- It tells that a algorithm can be strictly more than lower bound.
- It's stronger statement (againt Big omega)

$$[f(N) = g(N) \Rightarrow f > g] *$$

mathematically, $\lim_{n\to\infty} \frac{f(N)}{G(N)} = \infty$ //

---

* Space Complexity:-

- It is basically input space and auxilary space.
- Auxilary space is the extra space or temp space taken by the algorithm.

Space complexity = input space + Auxilary Space //

- we always take Auxilary space.

d) Find time complexity of ↓

```
for(i=0; i≤n; ) {
        for(j=1; j≤k; j++) {
                // takes t1 time
        }
        i = i+k;
}
```

First find how many times the loops are going executed.

Inner loop : Runs 'k' times ⇒ kt

Answer : outer loop * inner loop

how many time ol * inner loop

$\Rightarrow O(kt * ol)$

$i \leq N$ & $i += k$, So, $i+xk$ will be last

$1 + xk = N \Rightarrow \quad x = \frac{N-1}{k}$ // (no of times ol runs)

Time complexity : $O\left(kt * \frac{N-1}{k}\right)$

$\Rightarrow O(Nt)$ //.

* Recursive Algorithms :-

- Space complexity is not constant (As func. calls are stored in stack).
- At any particular point of time, As per flow of prog., No func. calls at same level of recursion will be in stack at same time

Trick: only calls that are interlinked will be in stack at same time.

- A link flow is maintained.
- Space complexity is equal to height of the recursion tree.

SC = Height/Path of tree.

---

* Types of Recursion / Recurrence Relation :-

(i) Divide and Conquer :-

- Reduces search space by a factor.
- How to identify? Form :-

$$T(x) = a_1 T(b_1 x + \varepsilon_1(x)) + a_2 T(b_2(x) + \varepsilon_2(x)) + \dots + a_k T(b_k(x) + \varepsilon_k(x)) + g(x)$$

Eg: Binary Search: $T(N) = T(N/2) + c$

$T(x)$: a_1, B_1, $g(x)$, $\varepsilon(x) = 0$

what is $g(x)$ / $g(N)$? ⇒ In simple terms, when you get answer from recursion call plus what you do with that answer.

* How to actually solve for complexity?

(i) Plug and chug. X (waste of time)

(ii) Masters Theorem X

(iii) Akra-Bazzi Formula:

$$T(x) = \Theta\left(x^p + x^p \int_1^x \frac{g(u)}{u^{p+1}}\, du\right)$$

* what is p?

$$a_1 b_1^{\,p} + a_2 b_2^{\,p} + \dots = 1 \quad \text{(or)} \quad \sum_{i=1}^{k} a_i b_i^{\,p} = 1$$

when you can't find value of 'p':

eg: $T(N) = 3T\left(\frac{N}{3}\right) + 4T\left(\frac{N}{4}\right) + N^2$

($a_1 = 3$, $b_1 = 3$, $a_2 = 4$, $b_2 = 4$, $g(N) = N^2$)

Try $p = 1$

$\Rightarrow 3 * \left(\frac{1}{3}\right)^1 + 4\left(\frac{1}{4}\right)^1$ should be $== 1$

But it $== 2$, which is > 1 //.

So we understand from this that we need to increase the value of 'p' (increase denominators) //

So, Try $p = 2$

$3\left(\frac{1}{9}\right) + 4\left(\frac{1}{16}\right) = \frac{7}{12}$ which is less than 1

This means we need to decrease value of 'p' //

So p value is dif. > 1 but < 2.

* PRO TIP: If value of 'p' less than the power of function $g(N)$ / $g(n)$ whatever, then always, the time complexity $G(N)$ or $G(n)$

Logic :- If $P < g(n)$, it implies $g(n)$ is greater or more dominating term so others are ignored. //

Ans = $O(g(n))$ // without even integration.

mathematical Proof :

$$T(n) = \theta\left(n^P + n^P \int_1^n \frac{u^2}{u^{P+1}}\, du\right)$$

$$= \theta\left(n^P + n^P \int_1^n u^{1-P}\, du\right)$$

$$= \theta\left(n^P + n^2\right) \longrightarrow \text{Ans } (G(n))$$

$\because$ w.k.t in above eq. $P < 2$, n^2 which is $G(n)$ is ans.

This is to solve any recurrence relation of Divide and Conquer.

(II) Linear recurrence relation :- Solving them.

Form :- $f(n) = a_1 f(n-1) + a_2 f(n-2) + \ldots + a_n f(n-n)$ FOR HOMOGENEOUS

$$\Rightarrow \sum_{i=1}^{n} a_i f(n-i) = f(n)$$ for a_i, n are fixed

n = order of recurrence

How to solve these? Steps to solve recurrence relations :

i) put $f(n) = \alpha^n$, for α = some constant.

Eg: for fibonacci series,

$$f(n) = f(n-1) + f(n-2)$$

characteristic Equation

$$\Rightarrow \alpha^n = \alpha^{n-1} + \alpha^{n-2} \Rightarrow \alpha^n - \alpha^{n-1} - \alpha^{n-2} = 0$$

dividing both sides by last term α^{n-2}, we get

$$\alpha^2 - \alpha - 1 = 0$$ [Find roots of this Eq]

$$\Rightarrow \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \text{gives roots}$$ //

2) Take roots and you can write them like:

$$f(n) = c_1 \alpha_1^n + c_2 \alpha_2^n \quad \text{— (Eq)}$$

c_1, c_2 ↓ constants; α_1, α_2 ↓ roots

If there were three or four roots, we would add them too.

i.e Add no. of roots * Constants.

(Eg: If 3 roots: $c_1 \alpha_1^n + c_2 \alpha_2^n + c_3 \alpha_3^n$).

Elaborate (Eq).

3) V. Imp't Fact:

No. of roots you have = no. of ans: you already have

Substitute those answers to the elaborated equation and solve to get constants value.

from above example,

$$f(0) = c_1 + c_2 \Rightarrow c_1 = -c_2 \quad \text{— (1)}$$

$$f(1) = 1 \Rightarrow c_1 \left(\frac{1+\sqrt{5}}{2}\right) + c_2 \left(\frac{1-\sqrt{5}}{2}\right)$$

$$\Rightarrow c_1 \left(\frac{1+\sqrt{5}}{2}\right) - c_1 \left(\frac{1-\sqrt{5}}{2}\right)$$

Solving them

$$\Rightarrow c_1 = \frac{1}{\sqrt{5}}, \quad c_2 = -\frac{1}{\sqrt{5}}$$

Put them in elaborated (Eq) // Simple.

from above example,

$$f(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Just put 'n' and get the answer. //.

Solving the final equation for time complexity :-

ignore constants, less dominating terms, so on,

⇒ Time Complexity ans for any Ques/Eq.

for above Example :-

$$\frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

↓ Constants

↳ less dominating

→ Ignore

So, TC of fibonacci = $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ // (Golden Ratio)

General Case :- If equal roots exist, use them as is and same procedure.

- Non-Homogeneous Equations / linear recurrences :-

→ when you have a extra function in equation such as $g(x)$ or $g(n)$ i.e Non-homogeneous.

Eg: $f(n) = a_1 f(n-1) + a_2 (f n-2) + \ldots + a_k (n-k) + g(x)$ ← extra function

How to Solve these ?

1) Replace G(n) [extra func] by '0' (zero) & solve usually.

solve till you get to (Eq) $[f(n) = c_1 \alpha^n \ldots]$

2) Take extra func. on one side and find the particular Sol :-

$$f(n) - \ldots = g(n)$$

now, guess/get the particular solution.

Guess something that's similar to $g(n)$

Eg: If $g(n) = n^2$, Guess a polynomial

replace the left side with guess and solve

3) Add both solutions together and solve, you

Constants, then solve usually.

Simple.

But, how do we guess the particular solution? **

- If $g(n)$ is exponential, guess of same type

  Eg: $G(n) = 2^n$

  Guess: $f(n) = a2^n$

- If $g(n)$ is polynomial, Guess of same degree.

  Eg: $G(n) = n^2 - 1$

  Guess: $an^2 + bn + c = f(n)$

* you can have combinations as well.

→ If your guess fails, keep increasing the degree.

Let's Solve a Example;

a) $f(n) = 2f(n-1) + 2^n$, $f(0) = 1$

Sol: 2^n replaced by '0'.

$\alpha^n = 2\alpha^{n-1} \Rightarrow \alpha^n - 2\alpha^{n-1} \Rightarrow \alpha - 2 = 0$

$\Rightarrow \alpha = 2$ // $\Rightarrow f(n) = c_1 2^n$ — (1)

$\Rightarrow$ Guess: $G(n) = 2^n$

$f(n) = a2^n$

Putting this in eq. where $g(n)$ is on one side.

$\Rightarrow$

$\Rightarrow$

$\Rightarrow$

Substituting a

$\Rightarrow f(n) = n.$

Sum of both eq.'s n

$f(n) = c_1 2^n + n.$

$f(0) = 1 \Rightarrow c_1 + 0$

$\Rightarrow f(n) = 2^n + n.2^n$

Time Complexity =

** you can check them in polynomial polynomial time. These are called

Eg: If $g(n) = n^2$, Guess a polynomial of degree 2.

replace the left side with guess and solve

3) Add both solutions together. and solve, you get constants, then solve usually.

Simple.

But, How do we guess the particular solution? **

- If $g(n)$ is exponential, guess of same type

  Eg: $G(n) = 2^n$

  Guess: $f(n) = a2^n$

- If $g(n)$ is polynomial, Guess of same degree.

  Eg: $G(n) = n^2 - 1$

  Guess: $an^2 + bn + c = f(n)$

* you can have combinations as well.

→ If your guess fails, keep increasing the degree.

Let's Solve a Example;

Q) $f(n) = 2f(n-1) + 2^n$, $f(0) = 1$

Sol: 2^n replaced by '0'

$\alpha^n = 2\alpha^{n-1} \Rightarrow \alpha^n - 2\alpha^{n-1} \Rightarrow \alpha - 2 = 0$

$\Rightarrow \alpha = 2$ // $\Rightarrow f(n) = c_1 2^n$ — (1)

$\Rightarrow$ Guess: $G(n) = 2^n$

$f(n) = a2^n$

Putting this in eq. where $g(n)$ is on one side.

$$f(n) - 2f(n-1) = 2^n$$

$$\Rightarrow a2^n - 2a2^{n-1} + 2^n = a \pm a+1 \quad \times \text{ (wrong Guess)}$$

So increase degree.

$$f(n) = (an+b)2^n$$

$$\Rightarrow (an+b)2^n - 2(a(n-1)+b)2^{n-1} + 2^n$$

$$\Rightarrow (an+b) - a(n-1) - b + 1$$

$$\Rightarrow an - an + a + 1$$

$$\Rightarrow a = 1 \text{ //.}$$

Substituting a in guess & discarding 'b' [we can't find 'b']

$$\Rightarrow f(n) = n.2^n \text{ -- (2)} \rightarrow \text{Particular Solution}$$

Sum of both eq.s now & Eq. (1) + (2)

$$f(n) = c_1 2^n + n.2^n$$

$$f(0) = 1 \Rightarrow c_1 + 0 \Rightarrow c_1 = 1$$

$$\Rightarrow f(n) = 2^n + n.2^n \text{ // Ans}$$

Time Complexity = $O(n.2^n)$ //.

** you can check the problem & solution and verify them in polynomial time but don't know if can be solved in polynomial time. Nobody knows

These are called NP- Completeness, NP- Hard problem