Space & Time Complexity

1) What is time complexity?

Ans:

Functions that gives us the relationship about how the time will grow as the input grows

'OR'

As the input grows, times grows is known as time complexity

Now as an ~~Enpamp~~ Example :-

We have two computer

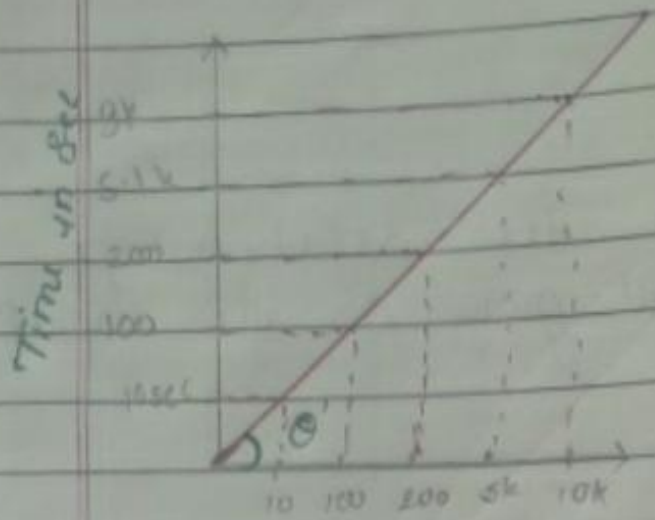
* We run an algo we have :-

	Old computer	M1 macbook (very fast)
data :-	1,000,000 elements in arr.	1000,000 elements in arr.
algo :-	linear search for target that doesn't exist in the array	Linear search -11-
Time taken	10 sec	1 sec.

Q. Which machine has a better time complexity in between these?

Ans :- Both of the machine have the same time complexity.

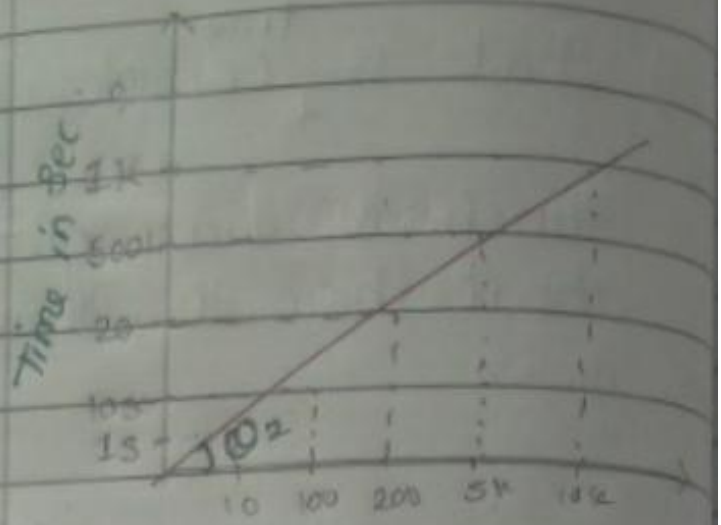
Time complexity != time taken.



Size of an array.

Eg:- old machine

1) Straight line.



Size of an array.

My Mack book.

1) steeper line with less slope.

Even though the time taken is different but the relationship between the size and the time is same "linear".

Over here, time is growing linearly as the size is growing. In both the case though values are different.

Q Why? & Why this relationship is important?

Ans:-

Linear search grows linearly. (N)
Binary search grows with logN

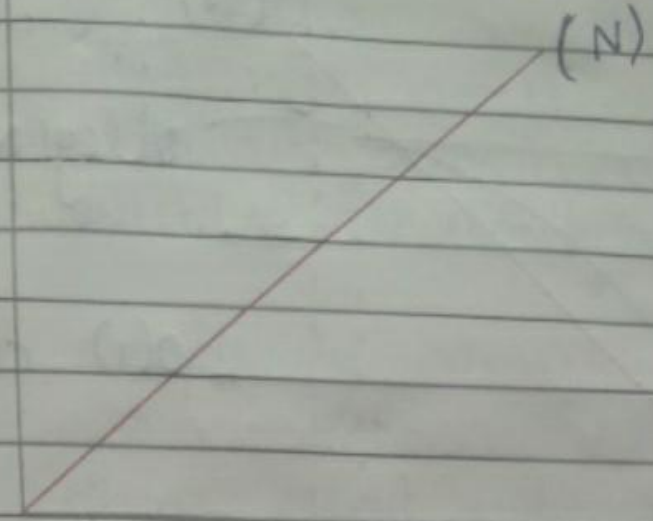


fig: Graph for linear search

Because the time complexity is linear

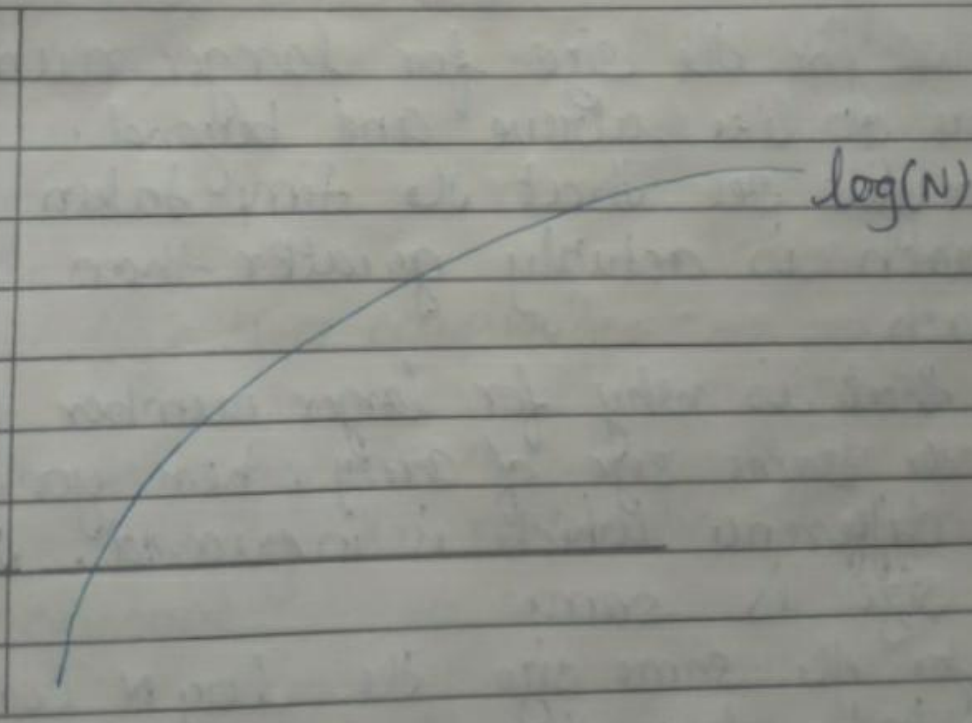
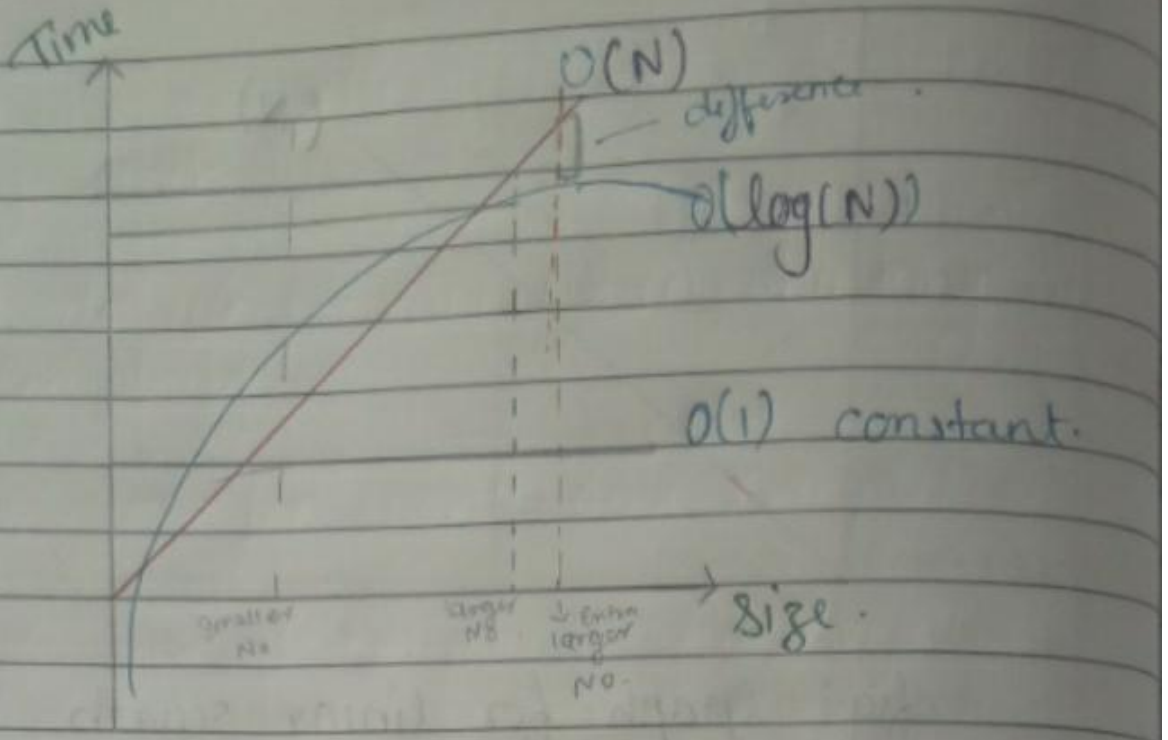


fig: Graph for Binary search.

This is growing 'log N' times.



Q Now let's Notice. Why does it matter?

Ans:

1) If we fix the size for larger numbers it may go like above and beyond.
2) Can you see that the time taken by linear search is actually greater than the binary search.
3) So, that is why for larger number. (fig :↑ graph) for the same size of array. here you can see the difference which is increased. though the the size is same
4) So, for the same size the log N complexity will take less time.
5) And, the linear complexity will take more time.

c) for smaller no. logn will take more time, linear less, O(1) will

Q So, which one is better?

Ans O(logn) is better because it is more efficient. So, that's why it matters.

Now let we talk a constant time complexity. So here does not matter what the size is time will always remain constant and for smaller number we don't care about smaller no.

Note:-

1) In time complexity always look at bigger numbers.
2) Always think about when your data will grow large in size in that case what will happen?

Ans:-

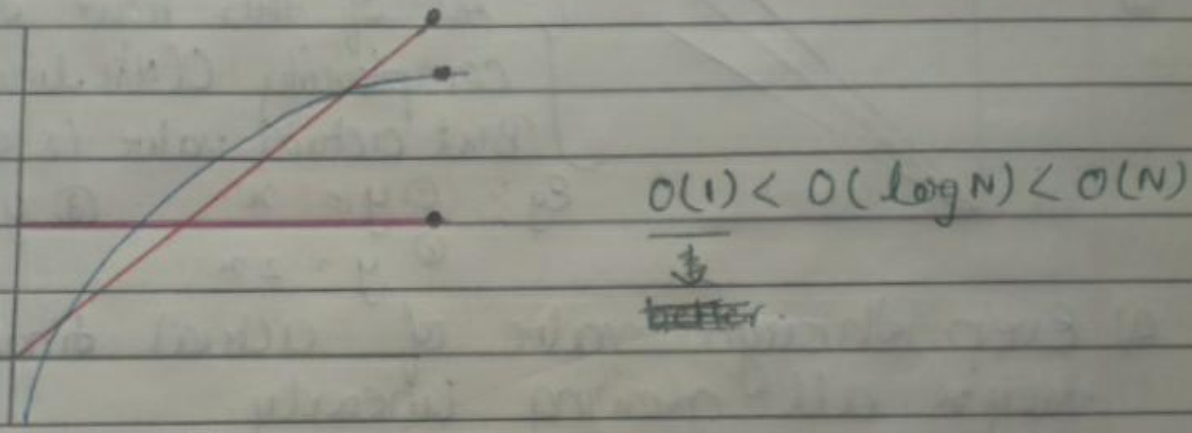


fig:- Time complexity.

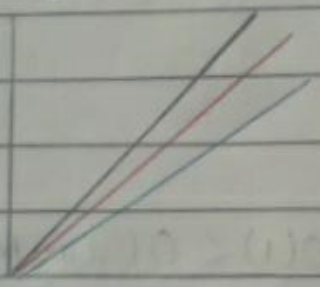
1) Now in the fig we can see that the (Red) linear is taking the most time, than log(n) then constant.

2) Thef Therefore, constant is always better than O(log(n)) & O(N)

* As you can see when the size was fixed for the same amount of data O(N) was taking the most time. then log(N) & last O(1).

Q. Which one is better?
Ans: Binary search O(log n).

* Q. What do we consider when thinking about the complexity?

Ans:
1) Always look for worst case complexity
2) Always look at complexity for large / ∞ data
3) 
All of this have the same complexity O(N) ie linear.
But actual value is different
Eg:- ① $y = x$ ③ $y = 4x$.
② $y = 2x$

a) Even though value of actual time is different they're all growing linearly.

b) "We don't actually care about what the time taken is" because that will vary from machine to machine".
We only care about the relationship of how the time will grow, when the input grows.

Q. Do we really need to worry about these constants then?

Ans: No, we only care about how its growing.

c) This is why, we ignore all constants.

4) Always ignore less dominating terms.

Okay :-

let's say you've complexity of

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

So, from point 2. { Always look at complexity for large / ∞ data }.

∴ if we take 1million ~~time~~ amt of data.

∴ $N = 1\,mil.$

∴ $= ((1\,mil)^3 + \log(1\,mil))$

$= (1\,mil)^3\,sec + \underbrace{6\,sec}$

It is very small

So does this 6 sec as compared to $(1mil)^3$ sec has any significant.

Hence, ignore it.

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

- (Ignore the constants)

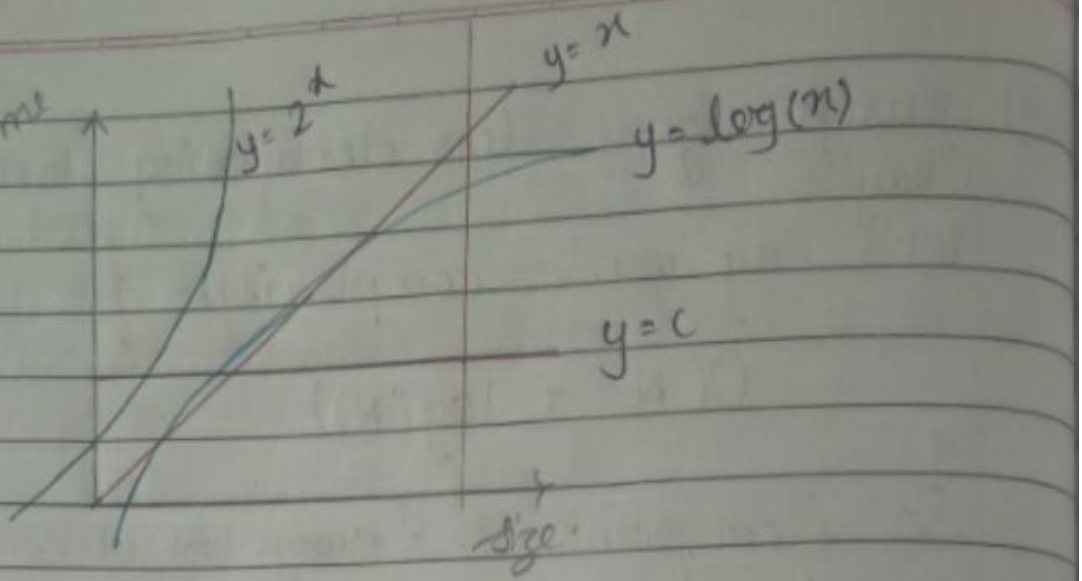
$= N^3 + N^2 + N$

- (Ignore the less dominating term)

$= N^3$

$\therefore O(N^3)$.

So, $O(3N^3 + 4N^2 + 5N + 6) = O(N^3)$.



1) $y = c$ is always be constant so, it will always be less than whatever value I you provide. So, $y = c$ will always be best optimized.

2) $\log(x)$ so, if we take some large amount of data, then
for same amount of day $y = c$ will take less amount of time.
After that $\log(x)$.
then the 'x' which will be little bit more than then as you can see 2^x which is very very poor complexity. (which is exponential complexity).

3) For such a small amount of data, time limit has exceeded alot: (2^{20}), which is not even visible on the graph.
Eg: fibonacci like for such a small amount of data time has exceeded alot, that is already not visible on the graph ∴ this is bad.

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

There other complexity also like $O(n \log n)$, $O(N^2 \log N)$ etc

* Big-Oh Notation : Def^{n} :

Suppose, that an algo has complexity of

$O(N^3)$

① What does it mean in simple words?

Ans :) So, in simple language it means that, this is the upper bound.

2) Meaning, CN^3 : Size of the ~~arrog~~ array will grow as the input grow in an N^3 fashion (eg: Binary & Linear search). But, what does this Big-Oh saying. It's saying that the complexity "the graph & the relationship" it can't exceed N^3.

3) For Eg : your algo that you've written may be solved in a constant time, it may be solved in $O(N)$ time, or $O(\log N)$ or $O(N^2)$ times etc. But it will never be solved or exceed the time complexity, relationship, the graph, the function value, it will never exceed more than N^3.
It will never be like $O(N^4)$ or $O(N^3 \log N)$ etc.

* Maths :

$$f(n) = O(g(n)) \quad \text{-- (1)}$$

$$\lim_{n \to \infty} = \frac{f(n)}{g(n)} < \infty \quad \text{-- (2)}$$

(2) It is actually some finite value.

As we said the "Always look for worst case complexity" & "Always look at complexity for large / ∞ data".

So, Here we're applying that.

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

Eg:

$$O(N^3) \sim O(6N^3 + 3N + 5)$$

$g(n)$ $f(n)$

$$= \lim_{n \to \infty} \frac{6N^3 + 3N + 5}{N^3}$$ - ∴ when the value of n reaches ∞ (closed to limit) ∴

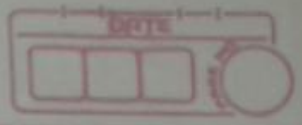
$$= \lim_{n \to \infty} 6 + \frac{3}{N^2} + \frac{5}{N^3}$$

$$\therefore 6 + \frac{3}{\infty} + \frac{5}{\infty}$$ - ∴ anything ÷ by ∞ = 0 ∴

$$= 6 + 0 + 0$$

$$= 6 < \infty$$

Hence, proved! It is a finite value because this is showing an upper bound.



Note: Our algorithm will never exceed the complexity of this. It can be better than this like it can also be solved in less complexity, but at any case it will never exceed.

* Little - Oh Notation : o(n)

- As we know that the Big - Oh notation was giving the upper bound this Little - oh notation also gonna give the upper bound only.

But, what is the distinct? This is not strict upper bound

It is a loose upper bound

Big - Oh	Little - oh.
1) if $f = O(g)$	$f = o(g)$
it means that the growth of f is no faster than g	it means that it is more like a smaller then g
$\therefore f \leq g$	$\therefore f < g$
	It is strictly slower than g.
	It is more stronger statement

Maths

If f is strictly slower than g, then you can say neumerater is slower than denominator, it should give us 0

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

$$f = N^2 \qquad g = N^3$$

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

$$= \lim_{n \to \infty} \infty \frac{1}{N} = 0.$$

** Big Omega Notation : def:
Opposite of Big-Oh notation

Suppose than an algo has complexity of

$\Omega(N^3)$

Q what does it means in simple term?

Ans: 1) This means that it will take atleast N^3 time complexity.
2) So, this means it the lower bound.
3) It will take atleast N^3, it can also take N^4, $N^3 \log n$ or $N^3 * 2^n$ etc
But, it will never be lesser than N^3
4) Minimum N^3 time complexity will be required.

** Matho:

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

Note: But we actually care about Big-Oh notation
Why?
Ans: We always look at the worst case

* Little omega :-

As we know big omega was giving the lower bound. This little omega will also going to give lower bound. But it will not be tight. So, it's gonna be like loosely lower bound.

Eg:- ~~$f = \omega(g)$~~

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

Big Ω

$f = \Omega(g)$

this means

$f \geq g$

lower bound.

\$ It is giving a lower bound. It can increase more than that \$

little ω

$f = \omega(g)$

this means

$f > g$ (strictly greater) difference.

Maths:

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

Q When do use this?

Ans

When you want something like strictly greater or strictly less than etc.

Note: In reality we use Big-Oh notations only.

* Space complexity or Auxilary Space :-

1) Auxilary space :- It is the extra space or temporary space taken by an algorithm.

∴ Space complexity = input space + Auxilary space.

Suppose :-
Take an ip of array ~~size~~ size N & do something with it.
So the sap: Space complexity will be the input you're taking from the size N + extra space the algo is using
This is known as Space complexity.

- So, in the binary search the space complexity was constant. So, that means auxilary space was constant it was not taking any extra space.
- Taking 3 variable i.e Start, End & Mid.
- If the array is of size 100 or more than that. Every single time it's only going to take 3 that variable i.e start, end & mid.
Hence, constant

Q

```
for( i = 1 ; i ≤ N ; ' ){
    for( j = 1 ; j ≤ k ; j++) {
        // some operations that
           takes time at t
    }
    i = i+k
}
```

Here,

1) inner loop is running k times & for every time it's running it's taking t amount of time.

∴ $O(kt)$ time.

2) If inner loop is running once, once, so it's taking t amount of time. So, here it's actully running k times. Hence kt.

3) Ans :- $O(kt *$ times outer loop is running)

conditions :- here i will starts from 1, & the loop will break when i is ≤ N & i is incrementing with k

4) So let's say, $i_{last} = 1, 1+k, 1+2k, 1+3k \ldots 1+xk$
So, if the x^{th} value is 'xk' that means is & satisfy the condions. Hence x should be ≤ N

'1+xk is the value & x is the no. of times its running

$\therefore 1 + xk \leq N$

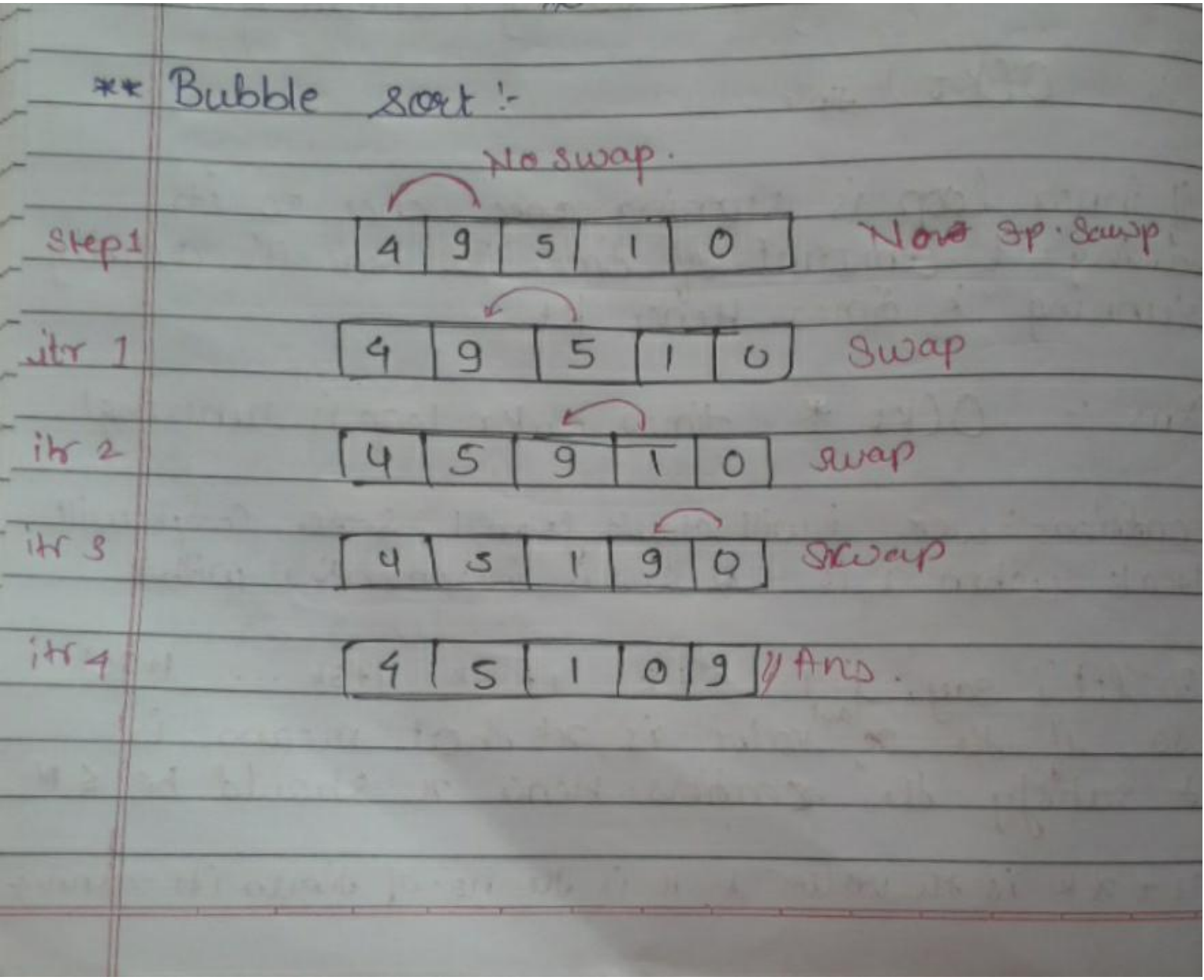
$xk \leq N-1$

$\therefore x = \frac{N-k}{k}$ — x = no of times the outer loop is running

Complexity : $O\left(kt * \frac{(N-1)}{k}\right)$ — const are removed

$\therefore = O(N * t)$

** Bubble sort :-


No swap.
step1
4 9 5 1 0
No sp. swap
itr 1
4 9 5 1 0
swap
itr 2
4 5 9 1 0
swap
itr 3
4 5 1 9 0
swap
itr 4
4 5 1 0 9
// Ans.

* Worst & Average case time complexity :-
O(n*n)
Worst case occurs when array is reverse sorted.

* Best case time complexity :-
O(n)
Best case occurs when array is already sorted

* Auxilary space
O(1)

* Boundary cases :-
Bubble sort takes minimum time (order of n) when elements are already sorted.

* Sorting in place :- Yes

* Stable :- Yes

** Selection Sort :

Worst complexity : n^2

Average complexity : n^2

Best complexity : n^2

Space complexity : 1

Method : Selection

Stable : No

Note: The good thing about selection sort is it never makes more than O(n) swaps & can be useful when memory write is a costly operation.

Eg:- (max) 4 , 5 , 1 , 2 , 3 — swap 5 & 3

4 , 3 , 1 , 2 , 5 — swap 4 & 2

2 , 3 , 1 , 4 , 5 — swap 3 & 1

2 , 1 , 3 , 4 , 5 — swap 2 & 1

1 , 2 , 3 , 4 , 5 ✓

** Insertion sort :-

Time complexity :- O(n * 2)

Auxiliary Space :- O(1)

Boundary Cases :- Insertion sort takes a maximum time to sort if element are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Sorting in Place :- Yes.

Eg :- 5 , 3 , 4 , 1 , 2

(Swap)

3 , 5 , 4 , 1 , 2

(sort & swap)

3 , 4 , 5 , 1 , 2

(sort)

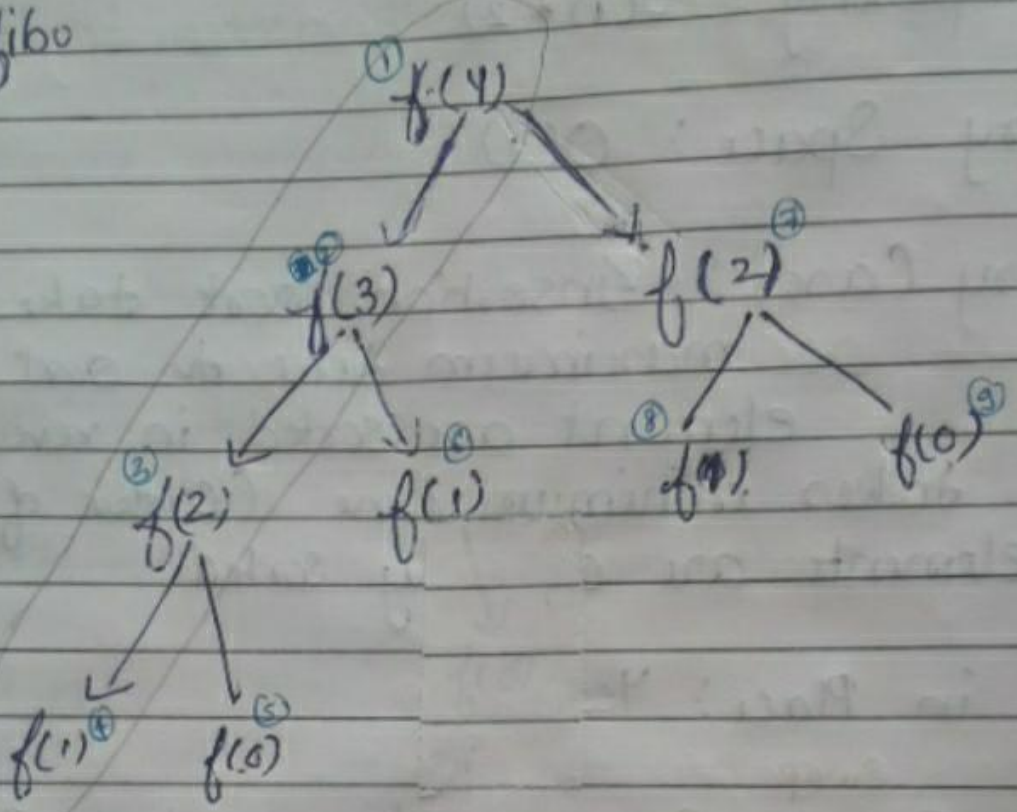
1 , 3 , 4 , 5 , 2

(sort)

1 , 2 , 3 , 4 , 5 #.

* Recursive & algorithms

• fibo



O(n)

Q what is the space complexity ?

Ans:-

1) Sop Space complexity is not going to be constant in Recursive program because

a) In Recursion we know that functions calls are stored in stack.

b) So, those function call actually takes some memory in stack

2) Therefore Recursive program do not have constant space in complexity

3) So, the space complexity is actually the height of the tree

"" At any particular point of time NO two funct^n call at perform at the same level of recursion will be in the stack at the same time.

Points:- Space complexity of recursive programs:-

1) When we talked about the flow of recursive program do you think that at any particular level

eg:- ③ ↙ ↘ ⑥ ⑧ ↙ ↘ ⑨
f(2) & f(1) f(1) f(0)

is there going to be a possibility for more than one function calls in this level to be in the stack at the same time?

2) Can we say that f(2) & f(0) are present in stack at the same time? NO.
Because this f(0) not even executed until f(2) finishes.

3) Please refer the fig:-
So, this 7, 8, 9 will for eg, will only execute when the ans of 3 is given.

Trick:-
Only calls that are interlinked with each other will be in the stack & at the same time.
because the previous one will be waiting for the next one to execute, next one will be wait for the next one ..... and these should be interlinked together at the same time.

from the fig:-

① for eg. when the function call no. 7 will be executing fun^{n} call no 2, 3, 4, 5, 6 would have already been finished.

② So, we suppose we take function call no 7. then 2, 3, 4, 5, 6 would have been already executed. Why?
Because those are not linked with function call no. 7

③ At one particular level there won't be any more than one calls that are in the stack at the same time.

④ So, what is the maximum space that it'll be taking?
Ans: We know that the longest st chain starts from the root till the leaf.
So, the longest chain will be the answer.

∴ Space complexity = height of the tree or path

∴ On

⑤ So, what is the auxiliary space complexity required when we're calculating Nth fibonacci number?
Ans O(n)
Because these all will be in the stack at the same time. so, the maximum amt of space req^{d} at the time.

* Types of Recursions :-

1) Linear ; Eg :- fibonacci no. $f(n) = f(n-1) + f(n-2)$

2) Divide & Conquer ; Eg :- binary search $f(n) = f(N/2) + a_1$

Note :-
You can represent recursion in the form of any question.

* Divide and Conquer Recurrences :-

This is actually same as $f(n) = f(N/2) + O(1)$

Form :-

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

for $x \geq x_0$ → same constant

If we relate this with $f(n) = f(N/2) + a_1$ we can say in binary search

Eg :- $T(N) = T(N/2) + C$ → constant.

$\therefore a_1 = 1$
$b_1 = 1/2$ $\quad g(N) = \Theta C$
$\varepsilon_1(x) = 0$

So, both are same & this is the form of divide & conquer recurrences

Other relationship can be something like this

$$T(N) = \underbrace{9}_{a_1}\, T\left(\underbrace{\frac{N}{3}}_{b_1}\right) + \underbrace{\frac{4}{3}}_{a_2}\, T\left(\underbrace{\frac{5}{6}N}_{b_2}\right) + \underbrace{4N^3}_{g(n)}$$

((a)) ((b)).

Q. What is the $g(n)$?

Ans: It means that

let say ((a)) ((b)) ↑

When you get the answer from ((a)) + what you're doing with that answer it takes how much time!

((b)) basically means the ((a)) recursion call is over then what amount of time complexity is required to do actually something with those recursion calls that are over right now. Extra time require at that step!

So, Extra time require at that step is equal to checking whether a number is greater greater than or equal to or less than middle that takes constant amount of time so, that's why we have constant over here.

Q How to actually solve to get complexity?

Ans Few Few 3 ways :-

1) Plug & Chug

OR

2) Master's theorem.

OR.

3) Akra Bazzai (1996) :- ~~Solve~~ ~~any~~

** Akra Bazzi :-

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

Words :- Time complexity = $T(x)$

$g(u)$:- this g funt" is basically a time complexity it-self. & we know that in time complexity like constants & more dominating terms are ignored.
we know that this going to be the simplest form of the function because all the less dominating terms and other things will be removed.

P :-

$$a_1 b_1^{\,P} + a_2 b_2^{\,P} + \cdots = 1$$

i.e $\sum_{i=1}^{k} a_i + b_i^{\,P} = 1$ //

$$T(N) = T\left(N/2\right) + C$$

Q In constants why do we write $O(1)$?

Ans Because we don't really care about constants for eg :- we can write $O(1)$ or $O(2k+1)$ all of these are constant, ∴ we can ignore the constants

∴ Basically $O((2k+1) \times 1)$

∴ Anything multiplied by anything, you can ~~ing~~ ignore the ~~constant~~ constants.

So, Hence anything in constants can be written as $O(1)$.

Eg: 1) $T(N) = 2T\left(\frac{N}{2}\right) + (N-1)$

$a_1 = 2$

$b_1 = \frac{1}{2}$ $\qquad g(x) = xN-1$

$$\therefore \quad 2 \times \left(\frac{1}{2}\right)^P = 1$$

$$\therefore \quad P = 1.$$

Once you've found the P then substitute it into the Akra-Bazzi formula :-

$$T(x) = \Theta\left(x^{p} + x^{p}\int_{1}^{x} \frac{u-1}{u^2}\,du\right)$$

$$= \Theta\left(x + x\int_{1}^{x} \frac{1}{u} - \frac{1}{u^2}\,du\right)$$

$$= \Theta\left(x + x\left[\int_{1}^{x} \frac{du}{u} - \int_{1}^{x} \frac{du}{u^2}\right]\right) \quad \int u^{-2} = \frac{u^{-1}}{-1} = -\frac{1}{u}$$

$$= \Theta\left(x + x\left[(\log u) + \left[\frac{1}{u}\right]\right]\right)$$

Put ↑.

$$= \Theta\left(x + x\left[\log x + \frac{1}{x} - 1\right]\right)$$

$$= \Theta\left(x + x\log x + \frac{1}{\log x} - x\right)$$

$$= \Theta(x\log x + 1)$$

$\therefore \Theta(x \log x)$. // Time complexity.

So, for array of size N :-
Merge sort complexity $= O(N \log N)$.

2) $T(N) = 2T\left(\frac{N}{2}\right) + \frac{8}{9} T\left(\frac{3}{4}N\right) + N^2$

$\therefore \quad 2 \times \left(\frac{1}{2}\right)^P + \frac{8^2}{9_3} \times \frac{3^1}{4_1}^P = 1$

$1 + \frac{2}{3}$ so, put $P = 1$

$2^1 \times \frac{1}{4_2} + \frac{8^1}{9} \times \frac{69}{16_2} = 1$

$\therefore \frac{1}{2} + \frac{1}{2} = 1$

$\therefore P = 2$ //.

Subsidute.

$$T(x) = \Theta\left(x^2 + x^2 \int_1^x \frac{xy^2}{3y^3}\, dy\right)$$

$= \Theta(x^2 + x^2 \log x)$ ✓ ignore the less dominating term ✓

$= \Theta(x^2 \log x)$

If you're unable do find value of P:

Q. $T(n) = 3T\left(\frac{n}{3}\right) + 4T\left(\frac{n}{4}\right) + n^2$

① $P = 1$ Case 1

$= 3 \times \left(\frac{1}{3}\right)^1 + 4 \times \left(\frac{1}{4}\right)^1 = 1$

$\therefore 1 + 1 = 1$

$2 \neq 1 \quad \because 2 > 1$ This means what? I need do increase the denominator

② $P = 2$ Case 2

$3 \times \frac{1}{9} + 4 \times \frac{1}{16} = 1$

$\frac{1}{3} + \frac{1}{4} = \frac{7}{12} < 1$

Hence, P is less than 2.

So, P actually lies in range 1 & 2

Note:

When P < power of $(g(n))$ than you your ans = $g(n)$.

Here, $g(n) = n^2$

$p < 2$ {i.e power of $g(n)$}

Hence, ans = $O(n^2)$.

$$T(x) = \Theta\left(x^p + x^p \int_1^x \frac{u^2}{u^{p+1}} \, du \right)$$

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

$$= \Theta\left(x^p + x^2 \right)$$

$\because p < 2$

$\therefore$ x^p will become less dominating term x^2 Hence, ignore.

$\therefore \Theta(x^2)$ //

Hence proved.

** Solving Linear Recurrence :- (Homogenous eqn)

Eg: $F(N) = F(N-1) + F(N-2)$

form :-

$$f(x) = a_1 f(x-1) + a_2 f(x-2) + a_3 f(x-3) + \dots + a_n f(x-n)$$

$$\therefore f(x) = \sum_{i=1}^{n} a_i f(x-i)$$ for a_i's, n is find.

n is the order of recurrence.

Solution :- for ~~filter~~ fibonacci no.

$$f(n) = f(n-1) + f(n-2) \quad -①$$

Steps :-

1) Put $f(n) = \alpha^n$ for some constant α.

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

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

$\div$ by α^{n-2}

$$\alpha^2 - \alpha - 1 = 0 \quad -(a)$$

$\left\{ \frac{1}{\alpha^{-1}} = \alpha \right.$

$\therefore \frac{\alpha^n}{\alpha^{n-2}}$

$\frac{\alpha^n \times \alpha^2}{\alpha^n}$

This eqn is also known as characteristics of recurrence

2) take roots of (a) by using quadratic eqn.

①

$$\alpha^2 - \alpha - 1 = 0$$

formula : $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

$$\therefore \alpha = \frac{1 \pm \sqrt{5}}{2}$$

$$\alpha_1 = \frac{1+\sqrt{5}}{2} \qquad \alpha_2 = \frac{1-\sqrt{5}}{2}$$

② If α_1 & α_2 have 2 roots, you can write the eqn in such a way

$f(n) = c_1 \alpha_1^n + c_2 \alpha_2^n$ is a sol^n for fibonacci.

it will equal to
$= f(n-1) + f(n-2)$

Note:

Not just for this any equation you have the number of roots, take that many no. of constants & x that many no. of roots with the power of n, add all that will equal to the Q.
So, for any constant C_1 & C_2

$$f(n) = C_1\left(\frac{1+\sqrt{5}}{2}\right)^n + C_2\left(\frac{1-\sqrt{5}}{2}\right)^n \quad \text{—②}$$

Note:

So, if you had three roots, you will write something like $C_1\alpha_1^n + C_2\alpha_2^n + C_3\alpha_3^n$.

③ Fact

No of roots that you have = no of ans you have already

So, here we have 2 roots α_1 & α_2.

Hence, we should have 2 ans already

We know that

$$f(0) = 0 \quad \& \quad f(1) = 1$$

Note:-

When you have n. no of roots or any number of roots you will have that many amount of answer.

$$f(0) = 0 = C_1\alpha_1^0 + C_2\alpha_2^0$$
$$= C_1 + C_2$$

$$\therefore C_1 = -C_2 \quad \text{—③}$$

for $f(1) = 1 = C_1\left(\frac{1+\sqrt{5}}{2}\right) + C_2\left(\frac{1-\sqrt{5}}{2}\right)$

from ③

$$1 = C_1\left(\frac{1+\sqrt{5}}{2}\right) - C_1\left(\frac{1-\sqrt{5}}{2}\right)$$

$$C_1 = \frac{1}{\sqrt{5}} \quad \& \quad C_2 = -\frac{1}{\sqrt{5}}$$

put in ②

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

- formula for n^{th} fibonacci no.

Q. How do we get the time complexity from this?

Ans:

① $f(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$ from the formula of fibonnaci number.

② As n increases or as $n \to \infty$ "$\left(\frac{1-\sqrt{5}}{2}\right)^n$" will be close to 0.

and as we know about time complexity that we should ignore the less dominating term
Hence, ~~igore~~ ignore the "$\left(\frac{1-\sqrt{5}}{2}\right)^n$".

③ Time complexity = $O\left(\frac{1+\sqrt{5}}{2}\right)^n$. for n^{th} fibonacci no.

Note:
Time complexity of fibo = $T(N) = O(1.6180)^n$

④ This is also known as golden ratio in mathematics.

⑤ So, this was the reason why our program was hanging for even small no. because exponential time complexity is very bad.

Code :-

```
public class FiboFormula {
    public static void main (String[] args){
        System.out.println ( formula (50));
    }

    static int formula (int n){
        return (int) ((Math.pow(((1+Math.sqrt(5))/2
                ,n)-Math.pow((1-Math.sqrt(5))/2
                n))/Math.sqrt(5));
    }

    static int fibo (int n){
        if (n<2){
            return n;
        }
        return fibo(n-1) + fibo(n-2)
    }
}
```

Q When you'll get equal no. of roots

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

I put $f(n) = \alpha^n$

$\therefore \quad \alpha^n = 2\alpha^{n-1} + \alpha^{n-2}$

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

both LHS & RHS

$= \alpha^2 - 2\alpha + 1 = 0$

$\therefore \alpha = 1$ ~~and~~ on both. (double root)

So, In such case what will happen is that if these root is repeated twice. so, let's say in the general case:-

* general case :-

If α is repeated 'r' times then, α^n, $n\alpha^n$, $n^2\alpha^n$ ... $n^{r-1}\alpha^n$ all are solutions to the recurrence.

Note:-

So, we know that if the number of roots are 2, so we know that there should be two roots but, we're getting only 1 root So, we can take extra roots from here because these are also a sol^n. we just multiply n, as we know that n divides n.

Hence, we can take two roots one is 1 and the other can be $n \cdot \alpha^n$.

as we know that $\alpha = 1$

$\therefore \quad 1, n$

putting in formula:

$$f(n) = c_1 (1)^n + c_2 n \alpha^n$$

$$= c_1 + c_2 n$$

let's say the ans given us is

$f(0) = 0$ & $f(1) = 1$

$\therefore f(0) = 0 = c_1$.

$\therefore c_1 = 0 \quad -(1)$

$\therefore f(1) = 1 = c_1 + c_2$

$\therefore c_2 = 1 \quad -(2)$

So, if we put eq (1) & (2) together,

$f(n) = n$.

$\therefore$ Time complexity of the above relation is $O(n)$.

* Homogenous linear recurrence :-

Q What does how homogenous means ?

Ans:- The form of a recurrance relationship where we do not have a particular other function like g(x).
So, there is no seperat function that is why it is known as homogenous.

Eg: Solving Linear Recurrance.

** Non - Homogenous Linear recurrences :-

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + a_3 f(n-3) + \ldots + a_d f(n-d) + \underbrace{g(x)}_{\neq 0}$$

So, when this extra function is present then it is known as Non homogenous linear recurrences.

Q How to solve ?

Steps:-

1) Replace g(n) by 0 & solve usually.

Eg:- $f(n) = 4f(n-1) + 3^n \quad , f(0) = 1$

$\therefore f(n) = 4f(n-1) + 0.$

$\alpha^n = 4\alpha^{n-1}$

$\therefore \alpha^4 - 4\alpha^{n-1} = 0$

$$\alpha - 4 = 0$$

$$\therefore \alpha = 4$$

∴ Homogeneous soln :-

$$f(n) = C_1 4 \alpha^n$$

$$f(n) = C_1 4^n \quad - (A)$$

$$\therefore f(n) = C_1 4^n + 3^n$$

2) Take g(n) on one side and find particular soln.

$$\therefore f(n) - 4f(n-1) = 3^n \quad - (1)$$

g(n) - as per the step

Q What is particula solution?

Ans: We ~~knee~~ need to do guess something that is similar to g(n), this is known as particular soln.

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

Kunal's guess :-

$$f(n) = C\, 3^n \quad - (2)$$

put in eqn (1)

$$\therefore C3^n - 4C3^{n-1} = 3^n$$

$\therefore C = -3$ — (a)

$\therefore$ the particular sol^n $\therefore$ put (a) in (2)

$$\therefore f(n) = -3 \times 3^n$$

$$\therefore f(n) = -3^{n+1} \quad - (B)$$

3) Add both the sol^n together

$$f(n) = C_1 4^n + (-3^{n+1})$$

$f(1) = 1$ — {we know}

$$C_1 4 - 3^2 = 1$$

$$\therefore C_1 = \frac{5}{2} \text{ //}$$

put the value of C_1 * in Orignal eq^n.

$$f(n) = \frac{5}{2} 4^n - 3^{n+1} \text{ //}$$

Short:

1) Replace $g(n)$ by 0 & solve usually.
2) Take $g(n)$ on one side & find particular sol^n
3) Add both the sol^n together

Abbreviation :-

1) First, put $g(n) = 0$ & take the normal sol^n then guess the particular sol^n and get the answer for that.

2) After that put $g(n)$ on one side, guess the particular solution. put that sol^n in the eq^n where you $g(n)$ one one side you'll get the value of 'c' and then you can put it in the orignal eq^n so, got your particular sol^n.

3) So, doing this we can get 'c' & then find answer of a particular sol^n.
Then just add the (1) eq^n with the particular eq^n and put it the ans ("which is already provided") use it then you'll get your orignal answer.

Q How do we guess a particular solution?

Ans:-

1) If $g(x)$ is exponential, guess of the same type.

Eg: $g(n) = 2^n + 3^n$

guess: $f(n) = a2^n + b3^n$

So, this is your particular solution.

2) But if it is polynomial, $g(n)$, in that case guess of same degree.

① Eg:- $g(n) = n^2 - 1$

then guess should be of same degree. Here it is 2 so, 2.

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

② Eg:- $g(n) = 2^n + n$

guess:- $f(n) = a2^n + (bn + c)$

Note:-

1) If it's exponential just multiply with the constant.
2) OR, if it's a polynomial take eq^n of that degree.
So, that's how you guess the particular sol^n.

3) let say you guessed, Eg:- $g(n) = a2^n$ & it fails, then try $\underline{(an + b)2^n}$, if this also fails increase the degree.

$\therefore (a^2n + bn + c)2^n$

Keep trying.

Eg: $f(n) = 2f(n-1) + 2^n$. $f(0) = 1$

① put $n\,2^n = 0$

$\therefore f(n) = 2f(n-1)$

put $f(n) = \alpha^n$

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

$\alpha = 2$ //

② guess P.S

$g(n) = 2^n$

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

put it in main eqn

$a2^n = 2a2^{n-1} + 2^n$

$a = a + 1$ ✗ wrong

Hence, guess another one from our rules because this is not working

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

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

$an + b = an - a + b + 1$

$\therefore a = 1$

discard b.

$f(n) = n 2^n$

our particular soln

3) General answer :-

$f(n) = c_1 2^n + n 2^n$ - ∫ some of both the eqns.

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

$c_1 = 1$

$\therefore f(n) = 2^n + n 2^n$ //

Complexity = $O(n 2^n)$