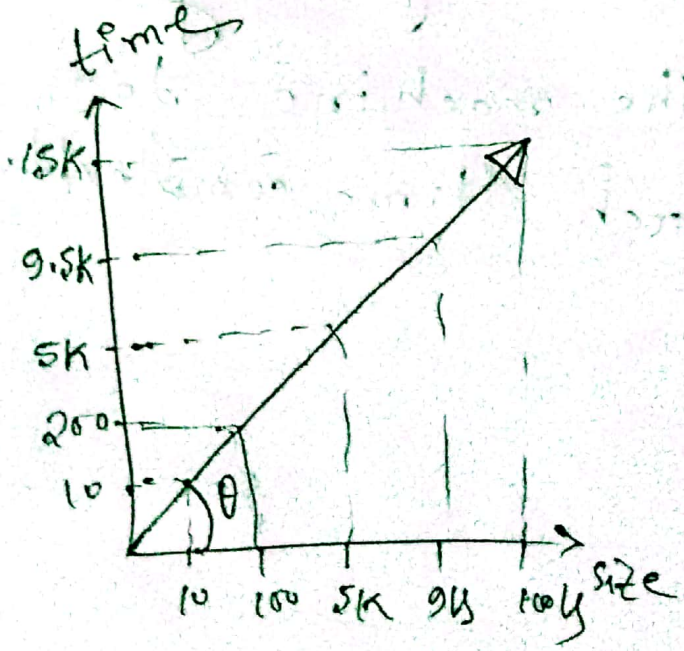
Time Complexity

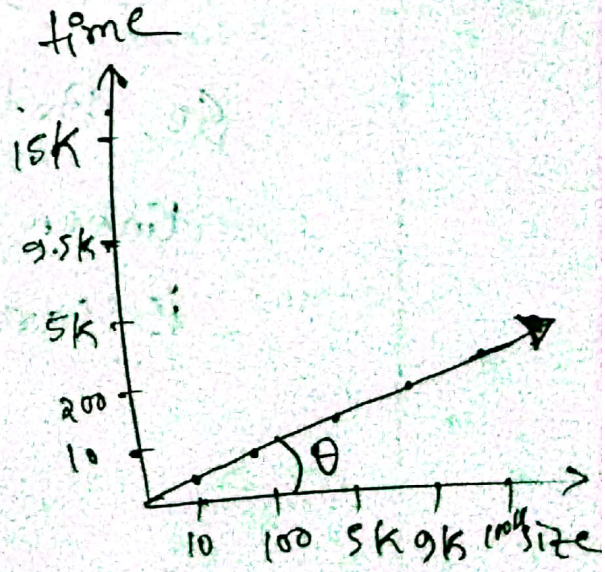
Time Complexity :

Time Complexity != Time taken

Example : An old machine takes 10 sec to perform linear search.

A new machine takes 1 sec to perform linear search.





- Time complexity is graph.

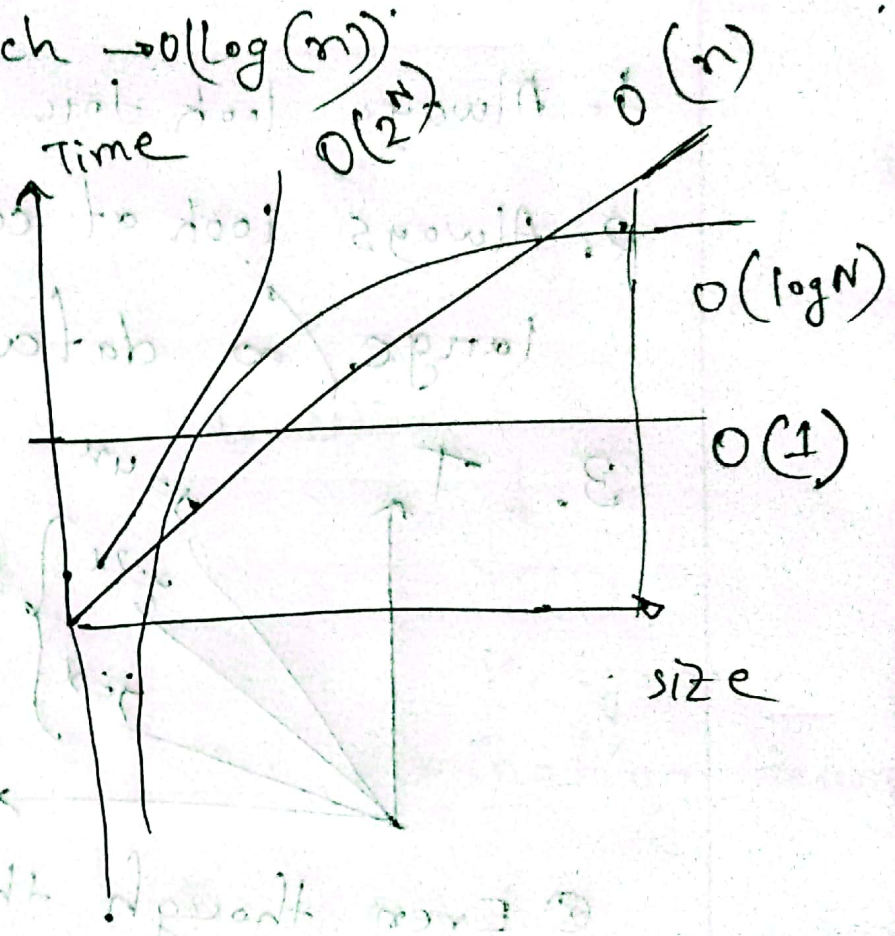
Time Complexity: Time complexity is function that tells us how the time is going to grow as inputs grows.

- In our example time is linearly growing as size growing. In Both the cases time is growing linearly. So, Both the machine dor linear search time complexity is same.

Linear Search → O(n)

Binary search → O(log(n))

Comparison:

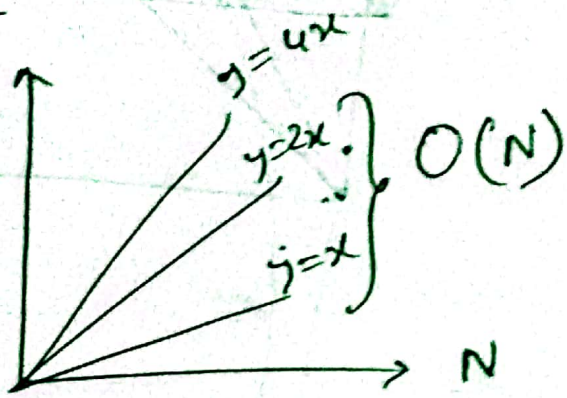


(*) For larger size linear search takes more time than binary search.

(*) 11.3 → we don't compare it for lowere number

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

Procedure for Analysing Complexity:

1. Always look for worst case complexity
2. Always look at complexity for large/∞ data
3. 

⊛ Even though the value of actual time is different ~~by~~ but they are all growing linearly.

④ We don't care about actual time.

⑦ This is why we ignore all constants.

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

Ex → N → 1 mil

$$\left((1\text{mil})^3 + \log(1\text{mil})\right)$$

$$= (1\text{mil})^3 + 6$$

↓ very small, so, ignore

* Always ignore less dominating terms.

Ex: $O(3N^3 + 4N^2 + 6N + 9)$

$= O(N^3)$

Big-oh Notation:

Ex: $O(N^3)$. It's means that complexity can't be largen than N^3. It's the upperbaund. It can be smallerr or constant for particularr case.

Mathmetical:

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

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

Ex: $O(\underbrace{6N^3 + 5N^2 + 6N + 7}_{f(N)}) = \underbrace{O(N^3)}_{g(N)}$

$$\lim_{N \to \infty} \frac{6N^3 + 5N^2 + 6N + 7}{N^3}$$

$$= \lim_{N \to \infty} \left(6 + \frac{5}{N} + \frac{6}{N^2} + \frac{7}{N^3}\right) = 6 < \infty$$

Big-omega Notation: Opposite of Big-oh

Ex: $\Omega(N^3)$, It's means that complexity can't be smaller than N^3. Complexity can be larger. It's the lowerc bound.

Mathmetics:

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

Theta - Notation: Combine both Big-oh and Big-omega

$\theta(N^2)$. If's means upperbound and lowerbound both is N^2

Mathmetics:

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

Little-o - Notation:

* This is also giving upperbound
* It's not strict upperbound

Big-oh	Little-oh
$f = O(g)$	$f = o(g)$
$f \leq g$	$f < g$, strictly (Stronger statement)

Maths:

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

$f(N) < g(N)$

Ex: $f(N) = N^2$, $g(N) = N^3$

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

little Omega :

* This will b. give lowerbound
* It's not strictly lowerbound

$$f = \omega(g)$$

Maths:

Big Ω	little ω
$f = \Omega(g)$	$f = \omega(g)$
$f \geq g$	$f > g$

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

Ex: $f(N) = N^3$, $g(N) = N^2$

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

Space complexity :

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Auxiliary space: Auxiliary space is the extra space or temporary space used by an algorithm.

- Linear Search, Binary search, Bubble sort, insertion sort, cyclic sort, selection sort all have space complexity O(1).

Linear Search:

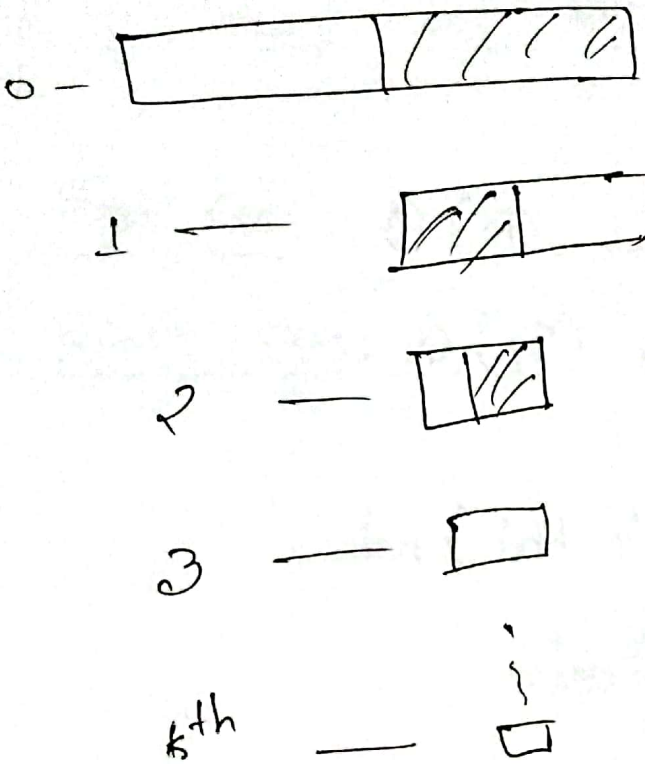
```
for ( i = 0 ; i < N ; i++ ) {
        //operation
}
```

so, loops runing N times,

∴ Worst case: Time complexity O(N)

Best case : Time complexity O(1)

Binary Search:




```
while (s <= e){
    if (mid = target) {
        return mid;}
    else if (target > mid)
        {s = mid+1}
    else{
        e = mid+1
    }
```

$$\therefore \frac{N}{2^k} = 1 \Rightarrow N = 2^k$$

$$\Rightarrow \log(N) = \log(2^k) = k(\log 2)$$

$$\Rightarrow k = \frac{\log N}{\log 2} = \log_2 N$$

* Total Numbers of comparison is K.

$$\therefore \boxed{\text{Time complexity} = O(\log N)}$$

Bubble Sort:

- ⊛ space complexity $O(1)$
  - ⊛ no extra space required
  - ⊛ also known as inplace sorting algorithms

Best Case: $O(N)$ → already sorted

Worst Case: $O(N^2)$ → sorted in reverse

```
for(int i=0; i<aN; i++) {
    for(int j=1; j<N-i; j++)
    {
        //swapping
    }
}
```

- outer loops runs : N − 1 times

- In the best case comparison happen only N − 1 times.

∴ Time Complexity $= O(N-1)$

$= O(N)$

Worst case:

Outer loops running : N-1 times

Internal loops running : N-i times

Total comparison:

$$(N-1) + (N-2) + (N-3) + (N-4) + \cdots + (N-(N-1))$$

$$= (N-1)(N) - (1+2+3+4 \cdots + N-1)$$

$$= N^2 - N - \frac{(N-1)(N)}{2}$$

$$= \frac{2N^2 - 2N - N^2 + N}{2}$$

$$= \frac{N^2 - N}{2}$$

$$\therefore O\left(\frac{N^2 - N}{2}\right) = O(N^2)$$

Selection Sort :

```
for( i=0; i< N; i++) {
    smallest = i
    for(j = i+1; j <N; j++) {
        // operation
    }
    // operation
}
```

Comparison Taking: $N-1, N-2, N-3, \ldots 1, 0$

Total Comparison: $0+1+2+3+ \ldots N-1$

$$= \frac{(N-1)(N)}{2}$$

$$= \frac{N^2 - N}{2}$$

$$\therefore O\left(\frac{N^2-N}{2}\right) = O(N^2)$$

Insertion Sort:

Worst Case: $O(N^2)$

Best Case: $O(N)$

```
for (i = 1 ; i < N-1 ; i++) {
    for (j = i+1 ; j > 0 ; j--) {
        // operation
        if (arr[j] < arr[j-1]) {
            //swap
        }
    }
}
```

Comparison taking: $N-1, N-2, N-3, \ldots 1$

Total Comparison: $1 + 2 + 3 + \ldots N-1$

$$= \frac{(N-1)(N)}{2} = \frac{N^2-N}{2}$$

$$O\left(\frac{N^2-N}{2}\right) = O(N^2)$$

Best case: 1, 2, 3, 4, 5 ...

Total comparison: N-1

$$O(N-1) = O(N)$$

Cyclic Sort: (1 to N)

```
i = 0
while (i < N) {
    correct = arr[i] - 1
    if (arr[i] != arr[correct])
        {   // swap
        }
    }
    else {
        i++
    }
}
```

worst case:  3, 5, 2, 1, 4

swaps = 4 and 3, 5, 2, 1, 4

last 5 swaps check 2, 5, 3, 1, 4

5, 2, 3, 1, 4

4, 2, 3, 1, 5

1, 2, 3, 4, 5

Total = 4 + 5

= (N-1) + N

= 2N - 1

$\therefore O(2N-1) = O(N)$

Example:

```
for(i=1; i≤N; i+=k){
    for(j=1; j≤k; j++){
        // works
    }
}
```

Inner loop running: $O(k)$ time

Ans: $O(k * $ how many times the outer loop is running$)$

$i = 1, 1+k, 1+2k, 1+3k, \ldots, 1+xk$

and outer loop is running x times

$$1+xk \leq N$$
$$\Rightarrow x \leq \frac{N-1}{k}$$

$$\therefore TC = O\left(k * \frac{N-1}{k}\right) = O(N)$$

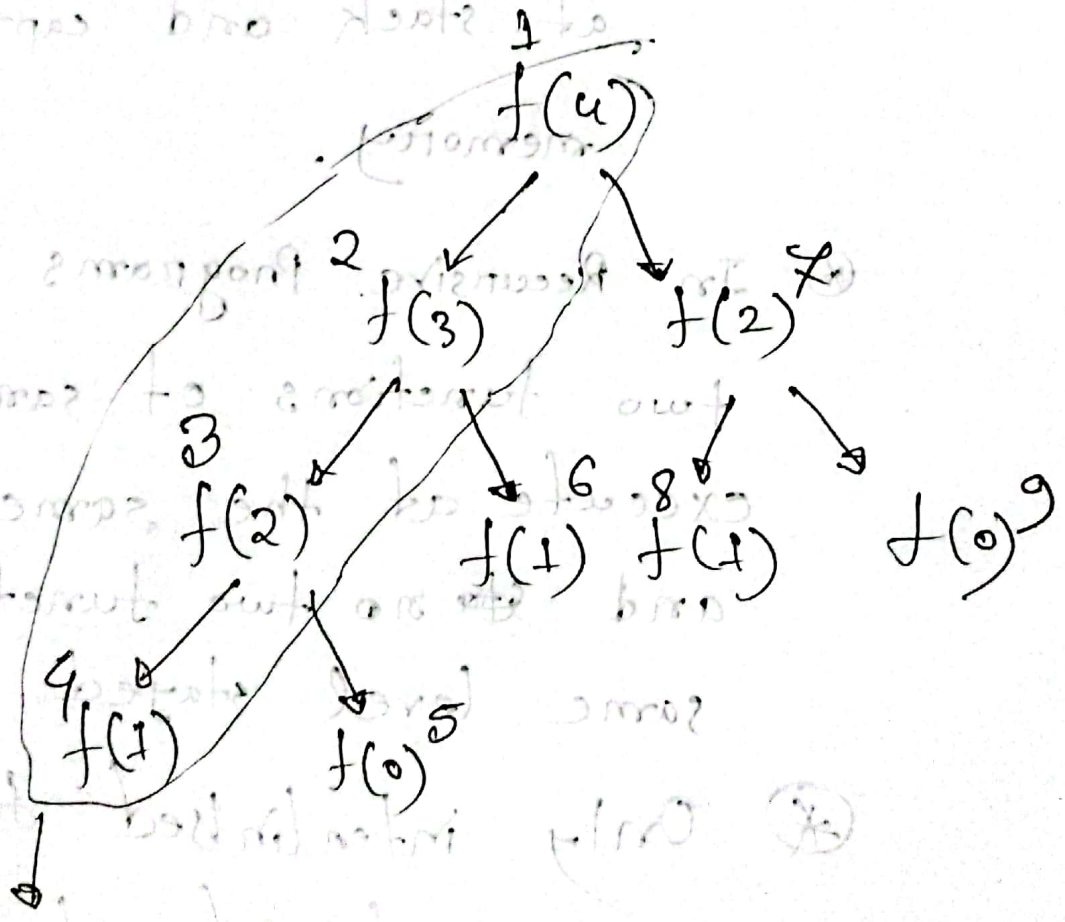
Recursive Programs:

- Space complexity is not going to O(1). Cause in Recursive Programs functions are stayed at stack and capture some memory.
- In Recursive Programs no two functions of same level execute at the same time and no two functions of same level stayed at stacks.
- Only interlinked functions are stayed at stack.

In Recursive Programs:

Space complexity would be the height of the tree.

Ex: Fibonacci Numbers



height of the tree.

∴ Space Complexity here = O(N)

Two types of recursion:

1. Linear
   ↓
   Fibonacci
   $F(N) = F(N-1) + F(N-2)$

2. Divide and Conquer
   ↓
   Binary Search
   ↓
   $F(N) = F(\frac{N}{2}) + O(1)$

(1) Divide and Conquer Recurrences:

Form: $T(x) = a_1 T(b_1 x + \epsilon_1(n)) + a_2 T(b_2 x + \epsilon_2(n)) + \cdots + a_k T(b_k x + \epsilon_k(n)) + g(x)$ for $x \geq x_0$

(x_0 ↓ constant)

Binary search:

$$T(N) = T\left(\frac{N}{2}\right) + C$$

comparing, $a_1 = 1$, $g(N) = C$

$b_1 = \frac{1}{2}$

$\epsilon_1(n) = 0$

another example:

$$T(N) = 9T\left(\frac{N}{3}\right) + \frac{4}{3} T\left(\frac{5N}{6}\right) + 4N^3$$

a_1 b_1 a_2 b_2 $g(N)$

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

a_1 b_1 $g(N)$

b (when · get ans from this + what we are doing with ans) — takes how much times → meaning $g(N)$

How to solve:

1. Plug and chug
2. Master's theorem.
3. Akra - Bazzi

Akra – Bazzi:

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

$p \rightarrow$ $\boxed{\sum_{i=1}^{k} a_i b_i^p = 1}$

Binary search:

$$T(N) = T\left(\frac{N}{2}\right) + C$$

$a_1 = 1 \qquad g(n) = C$

$b_1 = \frac{1}{2}$

we know, $\sum_{i=1}^{k} a_i b_i^p = 1$

$$a_1 b_1^p = 1$$

$$\Rightarrow 1 \times \left(\frac{1}{2}\right)^p = 1$$

$$\therefore p = 0$$

$$T(x) = \theta\left(x^0 + x^0 \int_1^x \frac{c}{v^{0+1}} \cdot dv\right)$$

$$= \theta\left(1 + \int_1^x \frac{c}{v}\, dv\right)$$

$$= \theta\left(1 + c \cdot \log x\right)$$

$$= \theta\left(\log x\right)$$

For N size we can say, it would be

$$\theta(\log N)$$

Ex: $T(N) = 2T\left(\frac{N}{2}\right) + (N-1)$ → Merge sort

$a_1 = 2 \qquad g(N) = N-1$

$b_1 = \frac{1}{2}$

we know $\sum_{i=1}^{k} a_i b_i^p = 1$

$\therefore \quad a_1 b_1^p = 1$

$\Rightarrow 2 \times \left(\frac{1}{2}\right)^p = 1$

$\Rightarrow \quad p = 1$

$$T(x) = \theta\left(x^1 + x^1 \int_1^x \frac{u-1}{u^{1+1}}\, du\right)$$

$$= \theta\left(x + x \int_1^x \left(\frac{u}{u^2} - \frac{1}{u^2}\right) du\right)$$

$$= \theta\left(x + x\left[\int_1^x \frac{1}{u}\, du - \int_1^x \frac{1}{u^2}\, du\right]\right)$$

$$= \theta\left(x + x\left[[\log u]_1^x + [u^{-1}]_1^x\right]\right)$$

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

$= \theta(x + x\log x + 1 - x)$

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

$= \theta(x\log x)$ → Time complexity

For size of N, Time complexity of merge sort is $O(N\log N)$

Example:

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

$a_1 = 2 \quad a_2 = \frac{8}{9} \quad g(N) = N^2$

$b_1 = \frac{1}{2} \quad b_2 = \frac{3}{4}$

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

$$\therefore p = 2$$

$$T(x) = \theta\left(x^2 + x^2 \int_1^x \frac{u^2}{u^3}\, du\right)$$

$$= \theta\left(x^2 + x^2 [\log u]_1^x\right)$$

$$= \theta\left(x^2 + x^2 \cdot \log x\right)$$

$$= \theta\left(x^2 \log x\right)$$

$$\therefore \theta\left(N^2 \log N\right)$$

If we can't find value of p

Ex) $T(x) = 3T(x/3) + 4T(x/4) + x^2$

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

$\boxed{1 < p < 2}$ $p = 1.5$

NOTE: when p < power of $(g(x))$

then answer will be $= g(x)$

∴ Answer here $= O(g(x))$

$= O(x^2)$

Solving Linear Recurrences:

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

Form:

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

$$\therefore f(x) = \sum_{i=1}^{n} a_i f(x-i)$$

a_i and

$n \rightarrow$ fixe

$n \rightarrow$ order of recurrence

Solution for fibonacci:

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

Steps:

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

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

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

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

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

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

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

(2) $f(n) = c_1 \alpha_1^n + c_2 \alpha_2^n$ → it's a sol^n

$= c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$ → take constants (2) no. of roots and multiply with roots powers n.

(3) Facts: no. of roots = no. of ans we already know

$f(0) = 0$

$f(1) = 1$

$\therefore f(0) = 0$

$\Rightarrow c_1 + c_2 = 0$

$\Rightarrow c_1 = -c_2$

$f(1) = 1$

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

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

$\Rightarrow c_1 = \frac{1}{\sqrt{5}}$

$c_2 = -\frac{1}{\sqrt{5}}$

∴ puting c_1, c_2 in (2)

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

↓

formula for n^{th} fibonacci numbers

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

as $n \to \infty$ $\left(\frac{1-\sqrt{5}}{2}\right)^n \to 0$, so

this less dominating term.

Hence ignore.

← Time complexity for n^{th} fibonacci numbers is: $O\left(\frac{1+\sqrt{5}}{2}\right)^n$ → Golden ratio

$$T(N) = O(1.6180)^n$$

Non-homogenous linear recurrence

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + a_3 f(n-3) + \cdots + a_d f(n-d) + g(n)$$

↓ when this extnam function is there, it is non-homogenous

(1) Replace $g(n)$ by 0 and follow usual step

(2) Take $g(n)$ on one side and find particular solutions

(3) Add both solutions together

Ex:

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

2. $f(n) = 4f(n-1)$

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

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

$$\Rightarrow \alpha - 4 = 0$$

$$\Rightarrow \alpha = 4$$

$$f(n) = c_1 \alpha^n$$

$$f(1) = c_1 (4)^n$$

Then,

$$f(n) - 4f(n-1) = 3^n$$

Guess something that is similar to $g(n)$

Guess: $f(n) = c3^n$

$$c3^n - 4c3^{n-1} = 3^n$$

$$\Rightarrow c = -3$$

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

~~(3) Add both Sol~~

$$f(n) = c_1 4^n - 3^{n+1}$$

$f(1) = 1$,

$$1 = c_1 4 - 3^2$$

$$\Rightarrow c_1 = \frac{5}{2}$$

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

Time complexity: $O\left(\frac{5}{2} 4^n - 3^{n+1}\right)$

How do we guess particular soln?

① If $g(n)$ is exponential, guess of same type:

Ex: $g(n) = 2^n + 3^n$,

Guess: $f(n) = a2^n + b3^n$.

② If it is polynomial, guess of same degree.

Ex: $g(n) = n^2 - 1 \Rightarrow$ guess of same degree

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

③ combination; $g(n) = 2^n + n$

$f(n) = a2^n + (bn + c)$

Lets say we guess $f(n) = a2^n$ and if it fails, then try $f(n) = (an+b)2^n$, if it's also fail, then increase the degree

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

Ex: $f(n) = 2f(n-1) + 2^n \quad , f(0) = 1$

① $2^n = 0$

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

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

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

$$\therefore \alpha = 2$$

② Guess particular sol^n.

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

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

$\Rightarrow$ $a = 2a + 1$ (wrong) ✗

then, Guess again,

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

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

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

$$\Rightarrow a = 1$$ — Discarded - b

$$f(n) = n\cdot 2^n$$

General:

$$f(n) = n\cdot 2^n + c_1 2^n$$

$$f(0) = 1$$

$$\Rightarrow c_1 = 1$$

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