Analysis of Algorithms

* for a single problem we can have multiple solution, but how we will decide which one is better, for that we will perform algorithm analysis

Ex: Sum of n natural numbers.

Input : n = 3.
output : 6. // 1+2+3

input : n = 5
output : 15 // 1+2+3+4+5

Solu:- 3- different type of solution for a problem :-

```
int fun1(int n){
    return n*(n+1)/2;
}
```

=> return => 6.

```
int fun2(int n){
    int sum = 0;
    for(int i=1; i<=n; i++){
        sum = sum + i;
    }
    return sum;
}
```

1+2+3 = 6.

```
int fun3(int n){
    int sum = 0;
    for(int i=1; i<=n; i++){
        for(int j=1; j<=i; j++){
            sum++;
        }
    }
    return sum;
}
```

sum++ => 1 + (1++) + (1+1+1)
=> 1+2+3
=> 6

=> for All the 3-Solution, How we can say which one is more efficient?

=> All codes depends on multiple factors like:-

- -> System configuration.
  - -> Hardware Specifications.
  - -> Software specifications.
- -> Input Size
- -> programming language (like)
  - -> C, C++ -> fast
  - -> Java, py -> slow.
- -> etc....

* by looking on the code we can't directly say which one is more effiient, because when same code runs on two different machine it can take different time. This is why we need some mathematical analysis method to compare code efficiency.

⇒ Asymptotic Analysis will help in this case.

⇒ We can compare the Algorithm with Asymptotic Analysis.

⇒ With Asymptotic Analysis we even don't have to implement all the solutions, by looking at the Algorithm, by looking all the solutions we can tell which one is more efficient theoritically.

Asymptotic Analysis

→ The Idea is to measure order of growth of time taken by a function program in terms of input size.

→ Does not depends upon machine, programming language, testcases, etc.

→ No Need to implement, we can analyze algorithms.

* The first thing we do is to get an expression of time taken by a code in terms of input size, we can see with the same example that we use eirlier for sum of n natural number.

Idea of Asymptotic Analysis

```
int fun1(int n){
    return n*(n+1)/2;
}
```

Time taken : C_1

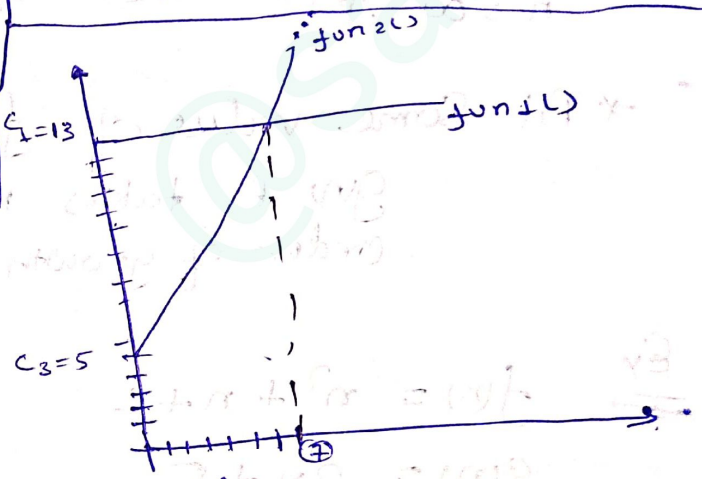
```
int fun2(int n){
    int solu = 0;
    for(int i=1; i<=n; i++){
        solu = solu + i;
    }
    return solu;
}
```

Time taken : $C_2n + C_3$

```
int fun3(int n){
    int solu = 0;
    for(int i=1; i<=n; i++){
        for(int j=1; j<=i; j++){
            solu++;
        }
    }
    return solu;
}
```

$\left(\frac{n^2}{2} + \frac{n}{2}\right)$ times.

Time Taken : $C_4n^2 + C_5n + C_6$



$C_1 = 13$, $C_2 = 5$

fun1() = 13.

fun 2() = 2n+5

after n=7 the fun2() is always higher than the fun1(); means the fun 2() will always takes more time than fun1(), so, we can say as the input size increase for fun 2() it takes more time.

* Time taken for order of growth, we simply taken the leading term from the expression so,

  fun1() → Constant
  fun2() → linear
  fun3() → Quardratic.

* We can always ignore the lower order of growth terms. & we can say the order of growth.

Order of growth

A function $f(n)$ is said to be growing faster than $g(n)$ iff

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

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

$f(n)$ and $g(n)$ represent time taken.

$n \geq 0$

$f(n), g(n) \geq 0$

* At some value of $f(n) = n+1$ it cross $g(n)$ & takes more time for their order of growth.

Ex

$f(n) = n^2 + n + 6$

$g(n) = 2n + 5$

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

$$= \lim_{n\to\infty} \frac{2/n + 5/n^2}{1 + 1/n + 6/n^2}$$

$$= \lim_{n\to\infty} \frac{0+0}{1+0+0} \quad \left(\frac{1}{\infty} = 0\right)$$

$$= 0$$

* As we wrote earlier

$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \Rightarrow f(n)$ is said to be growing faster than $g(n)$

* Now we see with example also, here, $f(n)$ term is quadratic & it is growing faster as we compare to $g(n)$

* Direct Way to find and compare Growths.

① Ignore lower order terms.

② Ignore leading constant terms.

Example: $f(n) = 2n^2 + n + 6$; order of growth $\Rightarrow n^2$ (Quardatic)

$g(n) = 100n + 3$, order of growth $= n$ (linear).

* How do we compare terms?

$$c < \log\log n < \log n < n^{1/3} < n^{1/2} < n < n^2 < n^2 < n^3 < n^4 < 2^n < n^n$$

* Best, Average & Worst Case

Example:- Simple function with same order of growth for every input

```
int getSum(int arr[], int n){
    int Sum = 0;
    for(int i=0; i<n; i++){
        Sum = Sum + arr[i];
    }
    return Sum;
}
```

Time Taken : $C_1 n + C_2$

Order of growth : n (linear).

Example 2: Multiple orders of Growths.

Best Case: Constant

Avg Case: Linear
(under the assumption that even & odd cases are equally likely.

Worst Case: Linear

```
int getSum(int arr[], int n){
    if(n%2==0){
        return;
    }
    int sum=0;
    for(int i=0; i<n; i++)
        sum=sum+arr[i];
    }
    return sum;
}
```

⇓

Suppose user provide every time Odd size, then the Worst Case will be 'n' i.e linear

Avg Case ⇒ $\frac{(C_1 n + C_2) + (C_3)}{2} = n$

(half time odd, half time even)

(If user provide equally odd size & even size input.

Note:- Worst Case, We always Want to know, for knowing the order of growth.

* Asymptotic Notations

→ Asymptotic Notation are the mathematical Notation is used to represent the order of growth of any mathematical function.

→ There are mainly Three Popular asymptotic Notation:-

- Big O : Exact or Upper
- Theta : Exact
- Omega : Exact or lower.

Example:-

```
int Search(int arr[], int n, int x) {
    for(int i=0; i<n; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

arr[] = {10, 5, 30, 40, 80}.

O(n).

* for this algorithm we can say; for worst case it takes O(n).

* worst case means :- exact or upper
So, we can say O(n), O(n²), O(n³) also, because, exact or upper.

eg for expression $C_1 n + C_2$
we can say -> O(n), O(n²) } we can write any thing higher than n not, less than n.

O(1), O(1) } constants.

* Big O Notation :- used to Represent exact or upper bound of order of growth.

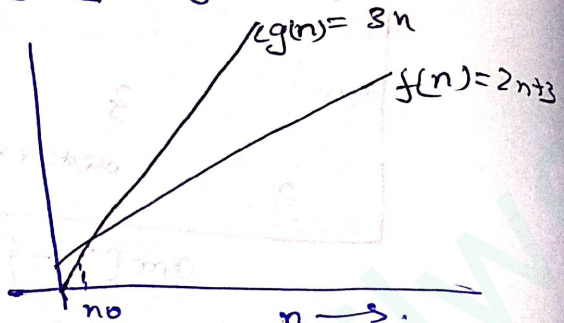
Direct Way
- Ignore lower order terms.
- Ignore leading term constant

$3n^2 + 5n + 6 \Rightarrow n^2 \Rightarrow O(n^2)$

$3n + 10\log n + 3 \Rightarrow n \log n \Rightarrow O(n \log n)$

$10n^3 + 40n + 10 = n^3 \Rightarrow O(n^3)$

$\Rightarrow$ We say $f(n) = O(g(n))$ iff there exist constant c & n_0 such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$.



Example

$f(n) = 2n+3$.

Can be written as $O(n)$ [$g(n) = n$]

let us take $c = 3$

$2n + 3 \leq 3n$

$3 \leq n$

We get $n_0 = 3$.

$2n + 3 \leq cn$

for all $n \geq n_0$.

We take $c=3$, because just take constant of higher term & add '1' to it $\rightarrow 2n \rightarrow (2+1) = 3$

Examples

$\{100, \log 2000, (10)^4 \ldots\} \in O(1)$

$\cup \{\frac{n}{4}, 2n+3, \frac{n}{100} + \log n, n + 10000, \log n + 10, \ldots\} \in O(n)$

$\cup \{n^2+n, 2n^2, n^2 + 1000n, n^2 + n\log n + n, \frac{n^2}{10000}, \ldots\} \in O(n^2)$

Big-O Notation works for multiple variable also.

$100n^2 + 1000m + n : O(n^2 + m)$.

$1000m^2 + 200mn + 30m + 20n : O(m^2 + mn)$

Note:- When you know exact order of growth, we would use Θ, theta Notation, Instead of Big-O, Notation.

Applications of Big-O Notations

-> used when we have an upper Bound.

Eg

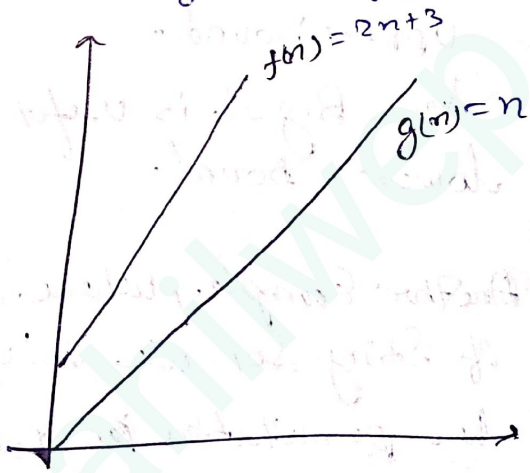
```
bool isPrime(int n) {
    if (n==1) return false;
    if(n==2 || n==3) return true;
    if(n%2==0 || n%3==0) return false
    for(int i=5; i*i<=n; i=i+6){
        if(n%i==0 || n%(i+2)==0){
            return false;
        }
    }
    return true
}
```

TC = $O(\sqrt{n})$

* Big-O Notation used in such case when we don't know the exact order of growth, & then we can use Big-O Notation.

Omega Notation : used to represent exact or lower bound of order of growth.

$f(n) = \Omega(g(n))$ iff there exist constant C (where $C > 0$) & n_0 (where $n_0 \geq 0$) such that $C\,g(n) \leq f(n)$ for all $n \geq n_0$.



Eg:- $f(n) = 2n+3 = \Omega(n)$

$C = 1$

$n \leq 2n+3$

$-3 \leq n$

$n_0 = 0$

* Example for application of Big-Ω. :-

→ Consider a situation where we have an algorithm to implement an online game & Say there are 'n' players at the game, & this game keeps on running forever until & unless a player quits the game, when player quit's the game, then the game is over.

→ Now we can't say about upper bound the time taken by the algorithm or program because its keeps on running, but we can have a lower bound,

→ Suppose our algorithm has initilizing scores of n player then we can say the time complexity of this algorithm or program is Big Omega of n, its going to take atleast Big Omega of 'n' time, because it might be running a loop which is initializing scores of n people, so, its going to take linear time but we don't know the upper bound.

→ So, Big Ω is useful when we have a lower bound.

→ Another Example, where we suppose the permutation of String. So, we have a String with say 3 characters then permutation are one 3!, in general factorial or permutation, whatever algorithm we write, its print factorial or permutation So, we can say the algorithm for printing permutation going to take big Ω (n!), because it takes atleast n! times to print.

Ex $3n^2 + 4n + 5$.

$\Omega(n^2)$ or $\Omega(n)$

(#) $\{n^2, 2n^2+5, 1000n^2, 2n^3+n, \ldots\} \in \Omega(n^2)$

$\cup \{2n+3, 100n + \log n, \ldots\} \in \Omega(n)$

$\cup \{5000, (10)^5, \lg 2000\} \in \Omega(1)$

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

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

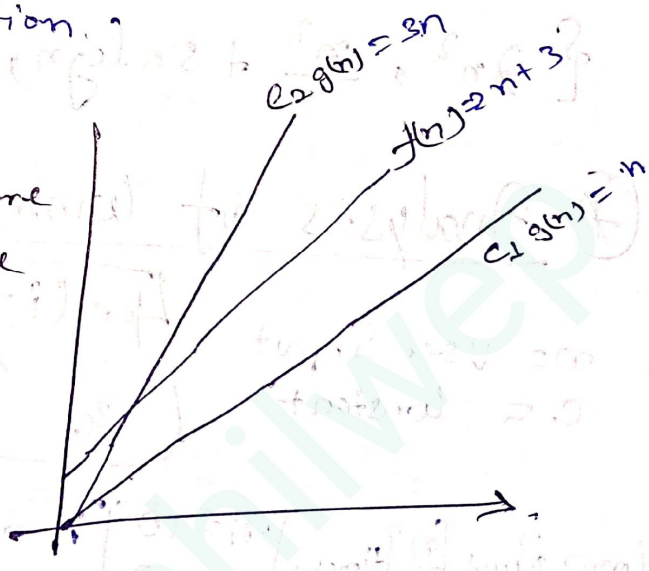
* Note: for exact order of growth, we can use any Big-O, Big-Ω, or Big-Θ.

Theta Notation: Used to Represent exact bound of a function.

$f(n) = \Theta g(n)$ iff there exist constants C_1 & C_2 (where $C_1 > 0$ & $C_2 > 0$) & n_0 (where $n_0 \geq 0$) such that

$C_1 g(n) \leq f(n) \leq C_2 g(n)$

for all $n \geq n_0$.



Example

$f(n) = 2n+3$. $\Theta(n)$

$C_1 = 1$, $C_2 = 3$ $\quad n_0 = 3$.

$1 \times n \leq 2n+3 \leq 3n$.

$n \geq 0$ $\quad n \geq 3$.

⊛ Direct Method.

$1000n^2 + 100n \log n + 2n$: $\Theta(n^2)$

$200n^3 + 30n + 5$: $\Theta(n^3)$

$2000n + 2 \log n$: $\Theta(n)$

⊛ If $f(n) = \Theta\, g(n)$

then $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

& $g(n) = O(f(n))$ & $g(n) = \Omega\, f(n)$

⊛ Represent Exact Bound.

⊛ $\{100, 10^5, \log 2000, \ldots\} \in \Theta(1)$.

$\{100n, 2n + \log n, 5n + 3, \ldots\} \in \Theta(n)$

$\{2n^2, \frac{n^2}{4} + 5n \log n, \ldots\}\ \Theta(n^2)$

⊛ Analysis of common loops

n = user Input
c = constant

```
for (i=0; i<n; i=i+c) {
    // Some Θ(1) work.
}
```

loop Runs $\lceil \frac{n}{c} \rceil$ times.

Time complexity : $\Theta(n)$.

n = 10, c = 2	i = 0
	i = 2
	i = 4
	i = 6
	i = 8

n = 20, c = 6	i = 0
	i = 6
	i = 12
	i = 18

Example 2

```
for (i=n ; i>0 ; i = i-c) {
    // some O(1) work
}
```

n = 10	i = 10
c = 2	i = 8
	i = 6
	i = 4
	i = 2

n = 20	i = 20
c = 6	i = 14
	i = 8
	i = 2

$\frac{n}{2}$

$TC = \theta(n)$

Example 3:

```
for (i=1 ; i<n ; i = i*c) {
    // some O(1) work
}
```

$(\log_c n)$

$c^0, c^1, c^2, \ldots, c^{K-1}$

$c^{K-1} < n$.

Taking log both side

$K < \log_c n + 1$

$TC = \theta(\log_c n)$

n = 33	i = 1
c = 2	i = 2
	i = 4
	i = 8
	i = 16
	i = 32

n = 81	i = 1
c = 3	i = 3
	i = 9
	i = 27

Example 4 :

```
for (i=n ; i>1 ; i = i/c) {
    // some O(1) work
}
```

$(\log_c n)$

$\frac{n}{c^0}, \frac{n}{c^1}, \frac{n}{c^2}, \ldots \frac{n}{c^{K-1}}$

$\frac{n}{c^{K-1}} > 1$

$c^{K-1} < n$.

Taking log both side.

$K-1 < \log_c n$.

$K < \log_c n + 1$

$\theta \log_c n$

n = 33	i = 33
c = 2	i = 16
	i = 8
	i = 4
	i = 2

n = 81	i = 81
c = 3	i = 27
	i = 9
	i = 3

Example `for(i=2; i<n; i=pow(i,c)){ // some O(1) work. }`

$2, 2^c, (2^c)^c \ldots ((2^c)^c)^{c \ldots}$ (k times)

$2^0, 2^{c^1}, 2^{c^2}, \ldots 2^{c^{k-1}}$

n = 33	i = 2
c = 2	i = 4
	i = 16

n = 514	i = 2
c = 3	i = 8
	i = 512

$2^3 = 8$
$8^3 = 512$

$2^{c^{k-1}} < n$ taking log both sid

$c^{k-1} < \log_2 n$.

taking log both sid

$k-1 < \log_c \log_2 n$.

$k < \log_c \log_2 n + 1$

$2^3 = 8 \to 2^c$
$8^3 = 512 \to (2^c)^c$
$512^3 \to$ break

$\boxed{\Theta(\log_2 \log_c n)}$

(*) Analysis of Multiple loop.

Ex: Subsequent loops.

```
void fun(int n){
    for(i=0; i<n; i++){          } Θ(n)
        // some Θ(1) work.       }
    }                                  +
    for(i=1; i<n; i=i*2){        } Θ(log n)
        // some Θ(1) work        }
    }                                  +
    for(i=1; i<100; i++){        } Θ(1)
        // some Θ(1) work.       }
    }
}
```

$T_c = \Theta(n)$.

Example 2 : Nested loop :-

$T_c = \Theta(n \log n)$

```
void fun(int n) {
    for (i=0 ; i<n ; i++) {          → Θ(n)
        for (j=1 ; j<n ; j=j*2) {
            // some Θ(1) work.       ↓ Θ(log n)
        }
    }
}
```

Example 3 : Mixed loop :-

$T_c = \Theta(n^2)$

```
void fun(int n){
    for(i=0 ; i<n ; i++){
        for (j=1 ; j<n ; j=j*2){        } Θ(n log n)
            //some Θ(1) work
        }
    }
    +
    for(i=0 ; i<n ; i++){
        for(j=1 ; j<n ; j++){           } Θ(n²)
            //some Θ(1) work
        }
    }
}
```

Example 4 :- Different Input

$T_c = \Theta(n \log n + m^2)$

```
void fun(int n, int m){
    for(i=0 ; i<n ; i++){
        for(j=1 ; j<n ; j=j*2){         } Θ(n log n)
            // some Θ(1) work
        }
    }
    for(i=0 ; i<m ; i++){
        for(j=1 ; j<m ; j++){           } Θ(m²)
            // some Θ(1) work.
        }
    }
}
```

⊛ Analysis of Recursion

Example

```
void fun(int n){
    if(n<=0){
        return;
    }
    print("Hello");
    fun(n/2);
    fun(n/2);
}
```

* → To solve we will write recurrence relation
→ assume function takes T(n) times. & make Two Recursive calls.

(n > 0)

$T(n) = T(n/2) + T(n/2) + \Theta(1)$

$\Rightarrow 2T(n/2) + \Theta(1)$

(n <= 0)

$T(n) = \Theta(1)$

(or)

$T(0) = \Theta(1)$

Example

```
void fun(int n){
    if(n<=0) return;
    for(i=0; i<n; i++){
        print("Hello");
    }
    fun(n/2);
    fun(n/3);
}
```

n > 0

$T(n) = T(n/2) + T(n/3) + \Theta(n)$

n ≤ 0

$T(n) = \Theta(1)$

(or)

$T(0) = \Theta(1)$

Example

```
void fun (int n){
    if (n<=1) return;
    print ("Hello");
    fun (n-1);
}
```

$T(n) = T(n-1) + \Theta(1)$

$T(1) = \Theta(1)$

(*) Recursion Tree method for solving Recurrences.

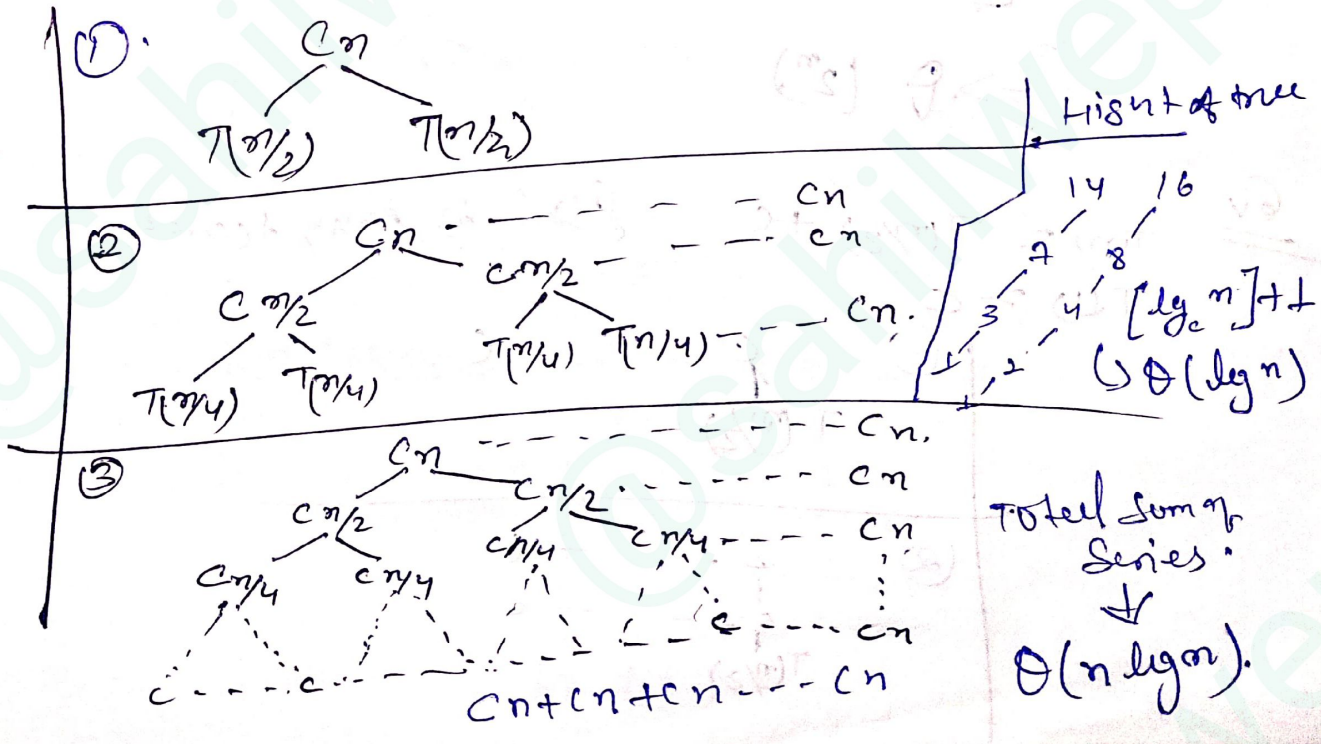
→ We consider the recursion tree & complete total work.

→ We write non-recursive part as Root of the tree & write the recursive part as children.

→ We keep expanding until we see the pattern.

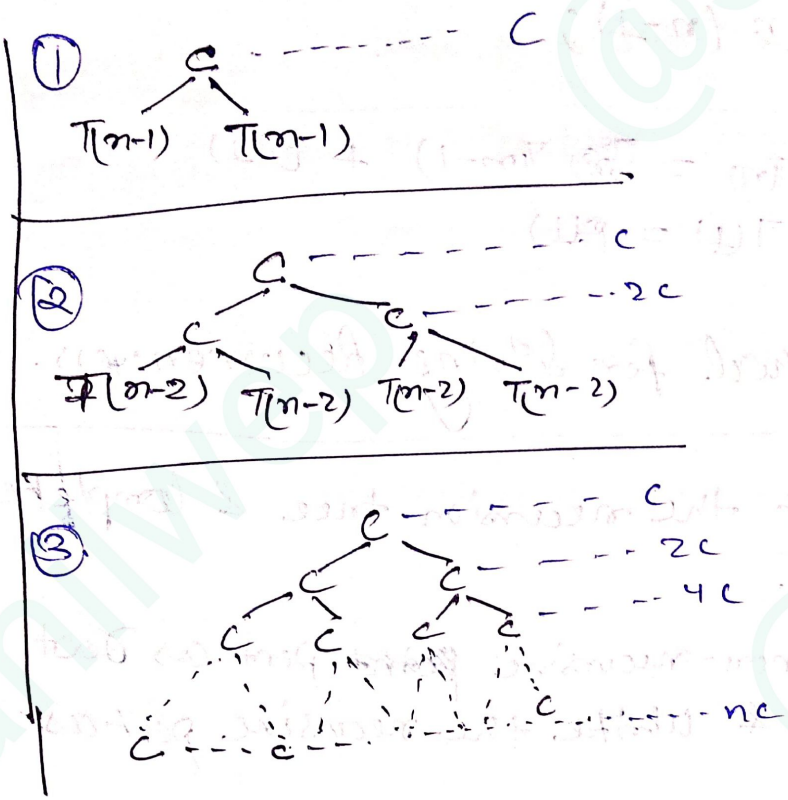
$T(n) = 2T(n/2) + \Theta(n)$ → Non Recursive Part

$T(1) = c$ → Recursive Part.



Ex $T(n) = 2T(n-1) + c$ } Seen in merge sort

$T(1) = c$



$c + 2c + 4c + \cdots$ (n terms)

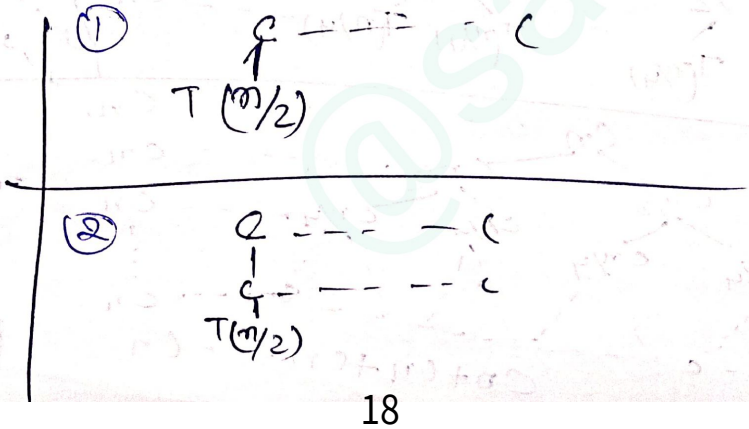
GP $(c[1+2+4\cdots)$

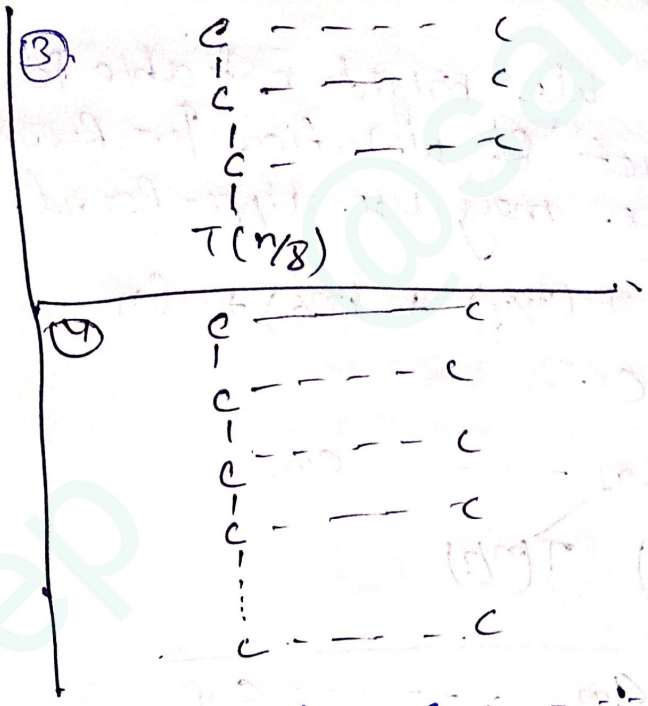
$\left[c \times \frac{1 \times 2^n - 1}{2-1}\right]$ $\left(a \times \frac{2^n - 1}{2-1}\right)$

$\Rightarrow \theta(2^n)$

Ex $T(n) = T(n/2) + c$ } Seen in Binary Search.

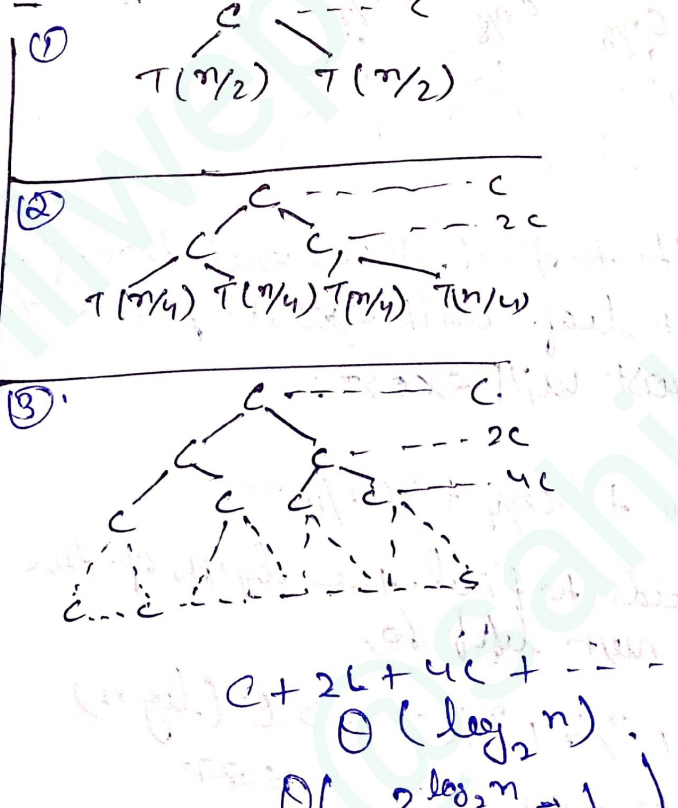
$T(1) = c$





Ex $T(n) = 2T(n/2) + c$

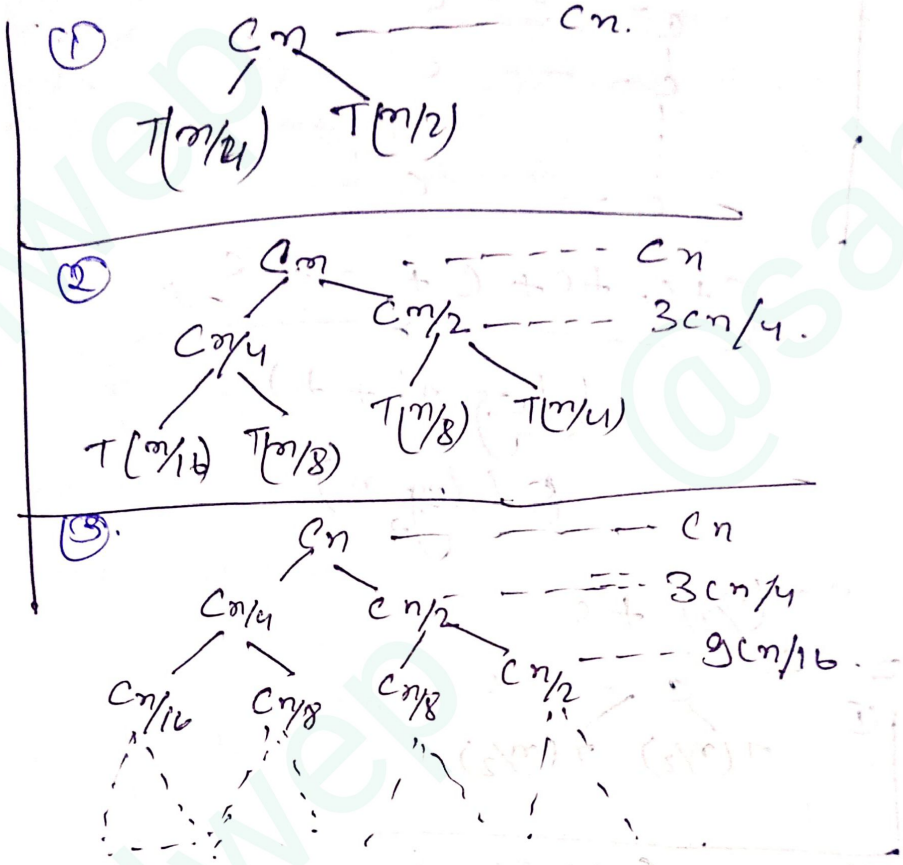
$T(1) = c$



*In some case we might not able to find the exact 'O' Notation for Recurrence Relation, we may use Upper Bound.

Eg $T(n) = T(n/4) + T(n/2) + cn$

$T(1) = c$.



(*) It's Difficult to find the exact bound, because the left most leaf will exist first & then the right most will exist.

So, G.P.:- $cn + 3cn/4 + 9cn/16 \cdots$

Geometric progression we need to find the depth of the right most left so,

$n, n/4, n/16 \cdots \rightarrow O(\log n)$

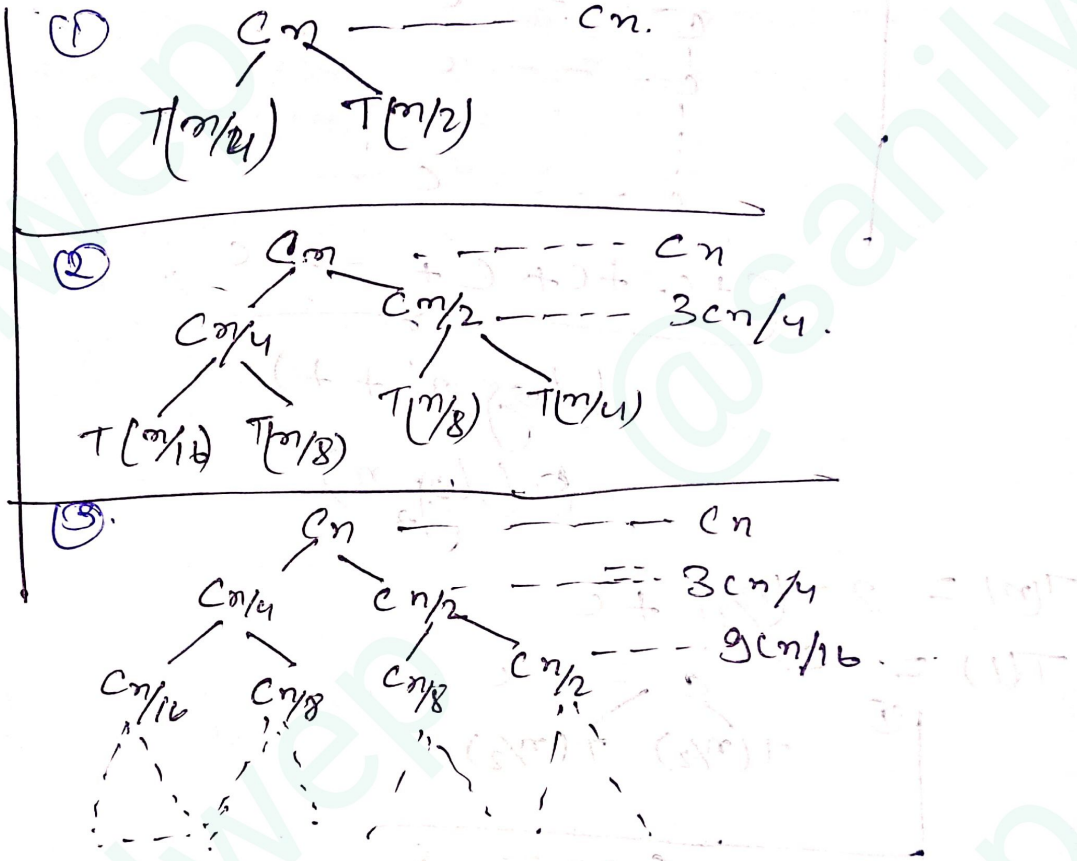
$O\left(\frac{cn}{1 - 3/4}\right)$

Upper bound $\rightarrow O(n)$.

*In some case we might not able to find the exact 'O' Notation for Recurrence Relation, we may use Upper Bound.

Eg $T(n) = T(n/4) + T(n/2) + cn$

$T(1) = c$.



(*) It's Difficult to find the exact bound, because the left most leaf will exost first & then the right most will exost.

So, G.P:- $cn + 3cn/4 + 9cn/16 \cdots$

Geometric progression we need to find the depth of the right most leaf so,

$n, n/4, n/16 \cdots \rightarrow O(\log n)$

$O\left(\frac{cn}{1 - 3/4}\right)$

Upper bound $\rightarrow O(n)$.

⊛ Space Complexity

→ order of growth of memory (or RAM) space in terms of input size.

```
int getSum(int n){
    return n*(n+1)/2;
}
```

Sc Θ(1) or O(1)

```
int getSum(int n) {
    int num=0;
    for(int i=1; i<=n; i++){
        num= num+i;
    }
    return num;
}
```

Θ(1) or O(1)

⊛ for Space Complexity we use same Asymptotic Notation, that we do for time complexity.

Ex

```
int arrSum(int arr[], int n){
    int sum=0;
    for(int i=0; i<n; i++){
        sum=sum+arr[i];
    return sum;
}
```

SC = Θ(n)

↳ Depends on Input size of array.

Auxiliary Space :- Order of growth of extra Space or temporary Space in terms of input Size.

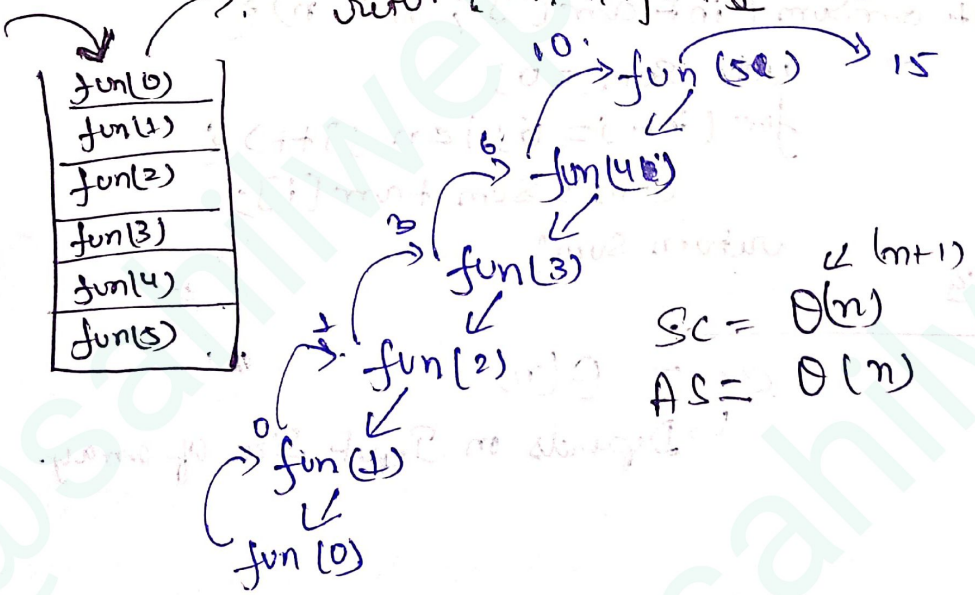
```
int arrSum (int arr[], int n) {
    int Sum = 0;
    for (int i=0; i<n; i++) {
        Sum = Sum + arr[i];
    }
    return Sum;
}
```

SC = $\Theta(n)$

Auxilary Space = $\Theta(1)$
(Extra Space)

Ex

```
int fun (int n) {
    if (n <= 0) return 0;
    return n + fun (n-1);
}
```

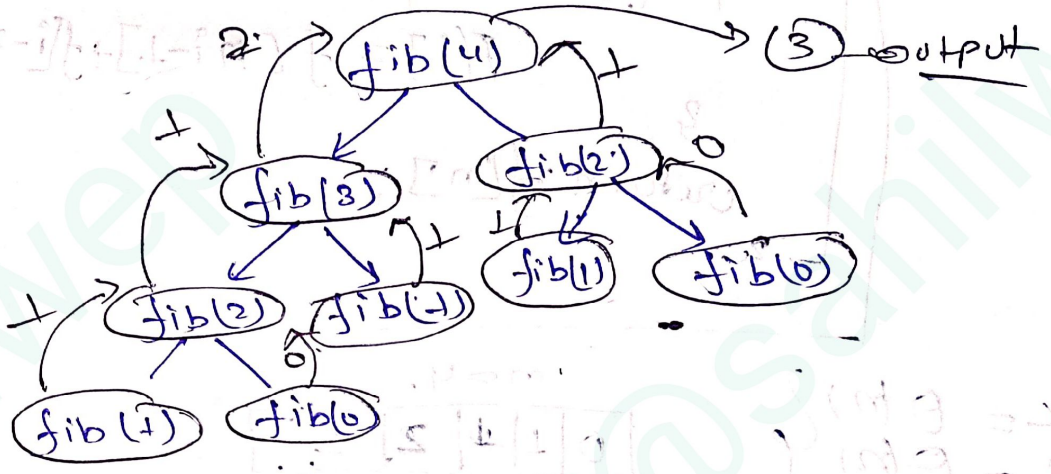


SC = $\Theta(n)$ ← (n+1)
AS = $\Theta(n)$

Ex

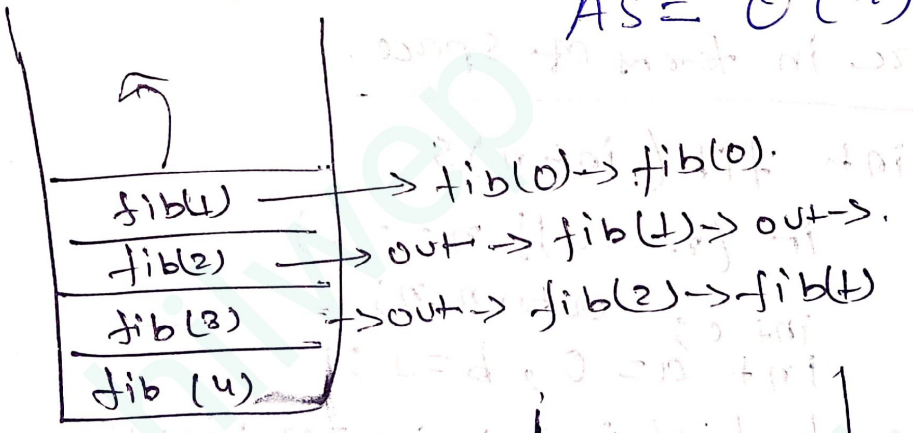
```
int fib (int n) {
    if (n==0 || n==1);
        return;
    return fib(n-1)+fib(n-2);
}
```

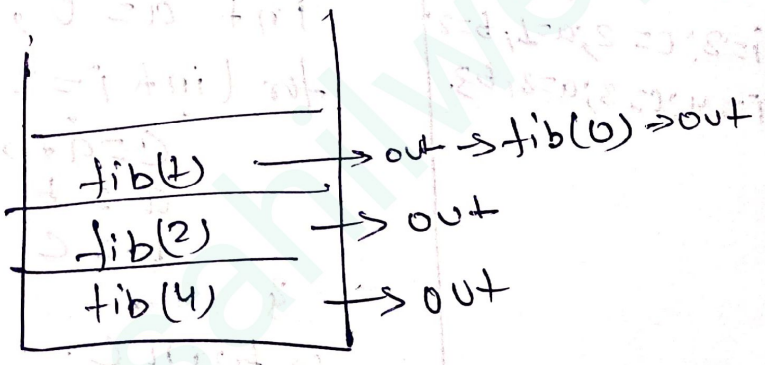
0, 1, 1, 2, 3, 5 - - - - -



* Auxiliary Space for any Recursive function is equal to
→ its height of the tree (n = 4,)

AS = O(n)

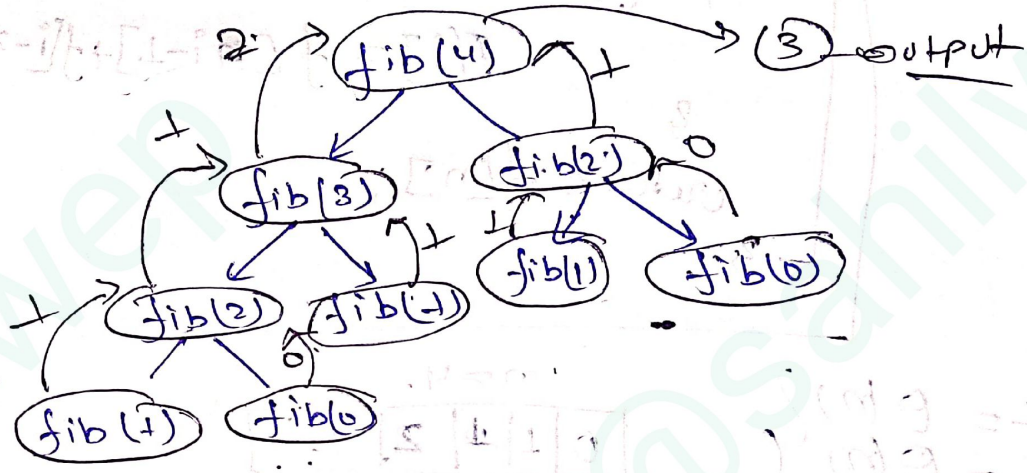




Ex

```
int fib (int n) {
    if (n==0 || n==1);
        return;
    return fib(n-1)+fib(n-2);
}
```

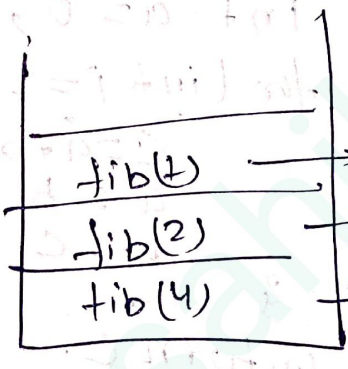
0, 1, 1, 2, 3, 5 - - - - -



* Auxiliary Space for any Recursive function is equal to
  → its height of the tree.

$AS = O(n)$ $(n = 4,)$

fib(4) → fib(0) → fib(0).
fib(2) → out → fib(1) → out →.
fib(3) → out → fib(2) → fib(1)



→ out → fib(0) → out
→ out
→ out

Ex

```
int fib (int n){
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for(int i=2; i<=n; i++){
        f[i] = f[i-1]+f[i-2];
    }
    return f[n];
}
```

$\{SC = O(n)$
$AS = O(n)\}$

n=4.

0	1	1	2	3
0	1	2	3	4

↳ array grows in terms of Input size.

Ex :- Optimize in terms of space.

n=4
a=0, b=1

i=2: c=1, a=1, b=1
i=3: c=2, a=1, b=2
i=4: c=3, a=2, b=3.

```
int fib(int n){
    if (n==0 || n==1)
        return n;
    int c;
    int a = 0, b = 1;
    for(int i=2; i<=n; i++){
        c = a+b;
        a = b
        b = c
    }
    return c.
}
```

SC = O(1)
AS = O(1)