Analysis of Algorithms

- Analysis of Algorithms (Background)
- Example:- Sum of n natural number

I/P:- 3
O/P:- 6

⇒ Code:- 01>

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

Time token:- C_1

⇒ Code:- 02>

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

Time token:- $C_2n + C_3$

⇒ Code:- 03>

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

Time token:- $C_4n^2 + C_5n + C_6$

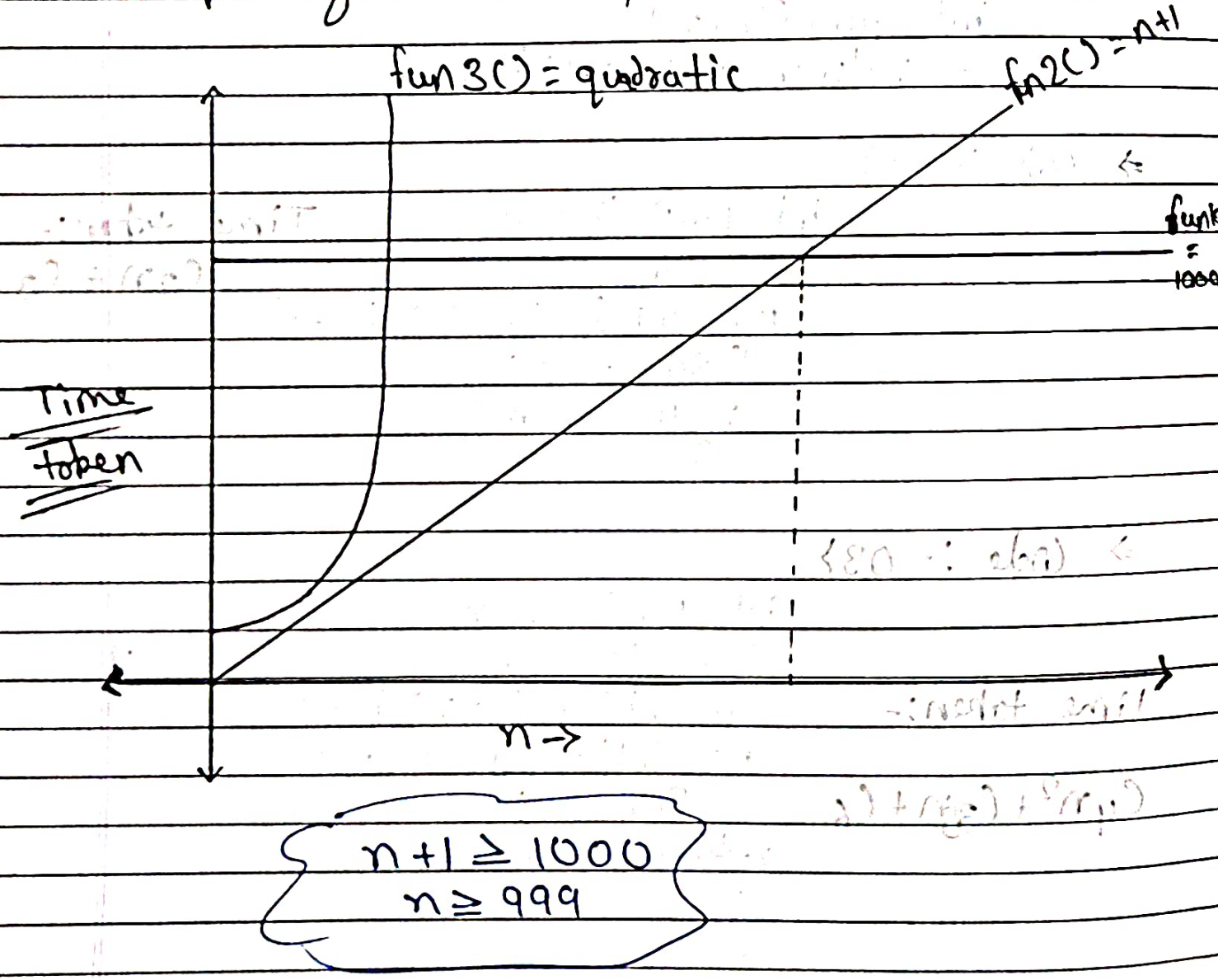
** * . Asymtotic Analysis:-

⇒ The idea is to measure order of growth.

⇒ Does not depend upon machine, programming language etc.

⇒ No need to implement, we can analyse algorithms.

* Graph of fun1(), fun2() & fun3().



$n+1 \geq 1000$

$n \geq 999$

- **Order of Growth**

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

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

$f(n)$ & $g(n)$ represent Time taken.

or

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

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

$f(n) = n+1$
$g(n) = 1000$

for $n = 0$ to 999 $f(n)$ is is taking less time but after 999 onwards $g(n)$ is better this is how asymtotic Analysis.

⇒ **Example:-**

A function $f(n)$ is said to be growing faste than $g(n)$ if.

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

$F(n) = n^2+n+6$
$g(n) = 2n+5$

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

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

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

***
**
*

Direct way to find & compare growths.

① Ignore lower order Terms

② Ignore leading Term constant.

Examples :- $f(n) = 2n^2 + n + 6$, order of growth: n^2 (quadratic)

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

* How do we compare terms?

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

$< n^2 < n^3 < n^4 < 2^n < n^n$.

* Best, Average & Worst Cases :-

- Example :- 01) Simple function with some 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 Token :- $C_1 n + C_2$

Order of Growth :- n (or linear)

- Example :- 02) Multiple orders of growths.

Best Case :- Constant

Average Case :- Linear
(under the assumption that even and odd cases are equally likely)

Worst Case :- Linear.

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

- **Asymptotic Notation:-**

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

Imp

⇒ Code:-

```
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}

1) Big O Notation (O):

- Describes the upper bound of an algorithm's running time. It provides the worst-case scenario of how an algorithm performs as the input size grows for example, $O(n^2)$ means the running time increases at most quadratically as the input size increases.

⇒ Direct way :-

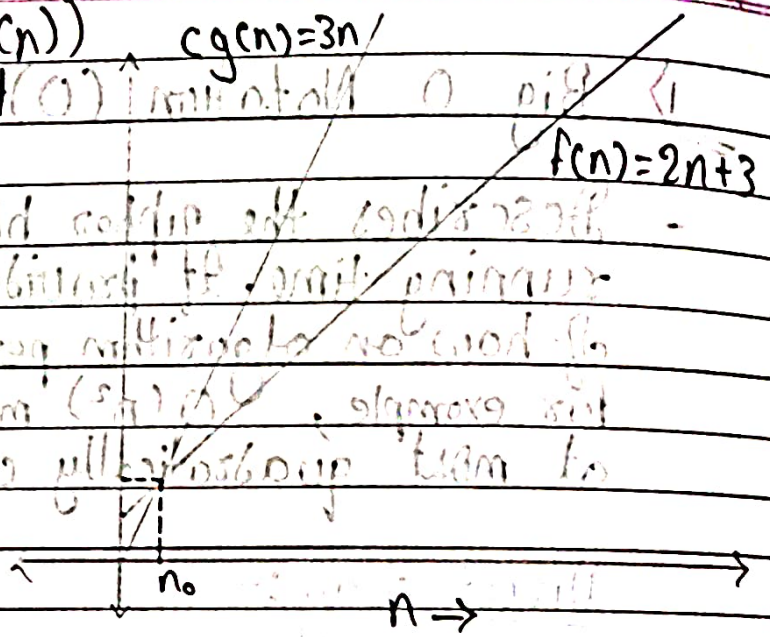
- Ignore lower order terms.
- Ignore leading term constant.

- $3n^2 + 5n + 6$ → $O(n^2)$
  - $3n^2$ → constant
  - $5n$ → leading terms & constant
  - 6 → constant

- $3n + \log\log n + 3$ → $O(n \log n)$
  - $3n$ → lower value term
  - $\log\log n$ → leading terms & constant
  - 3 → constant

- $10n^3 + 40n + 10$ → $O(n^3)$
  - $10n^3$ → constant
  - $40n$ → leading terms & constant
  - 10 → constant

- We say $f(n) = O(g(n))$ if there exist constant c & n_0 such that $f(n) \leq (g(n)$ 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 $n_0 = 3$

$2n+3 \leq n$ for all $n_0 \leq n$

-

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

All the constant terms are referred to be said as $O(1)$

- $U\left\{\frac{n}{4}, 2n+3, \frac{n}{100} + \log n, n+100, \log n+10, \ldots\right\} \in O(n)$

- $U\{n^2+n, 2n^2, n^2+100n, n^2+n\log n+n, \frac{n^2}{1000} \ldots\} \in O(n^2)$

- Big O notation works for multiple variable also:-

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

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

* Application:-

- Used when we have an upper bound. **

```
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;
    }
```

$O(\sqrt{n})$

2) Omega Notation :- (Ω)

Describes the lower bound of an algorithm's running time. If it provides the best case scenario. For instance, $\Omega(n)$ means the algorithm will take at least linear time in the best case as the input size grow.

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

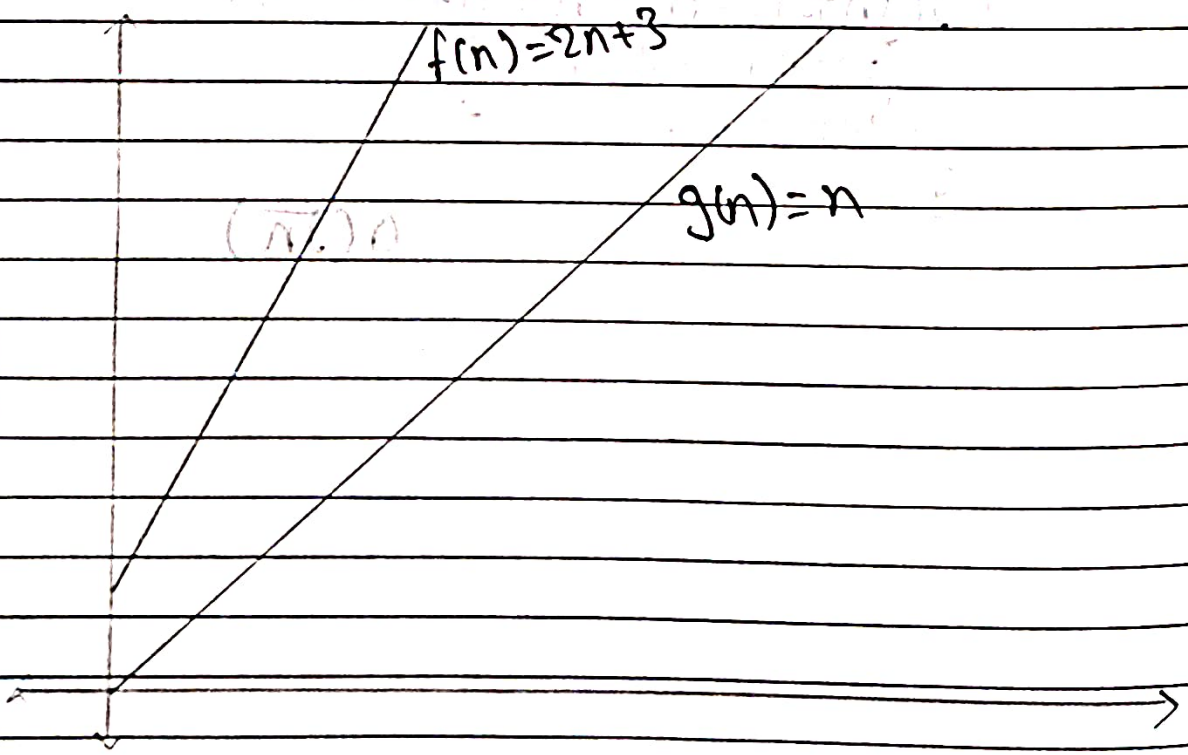
$$f(n) = 2n+3 = \Omega(n)$$

$c = 1$

$n \leq 2n+3$

$-3 \leq n$

$n_0 = 0$



⊛ $\{n^2, 2n^2+5, 100n^2, 2n^3+n, \ldots\} \in \Omega(n^2)$

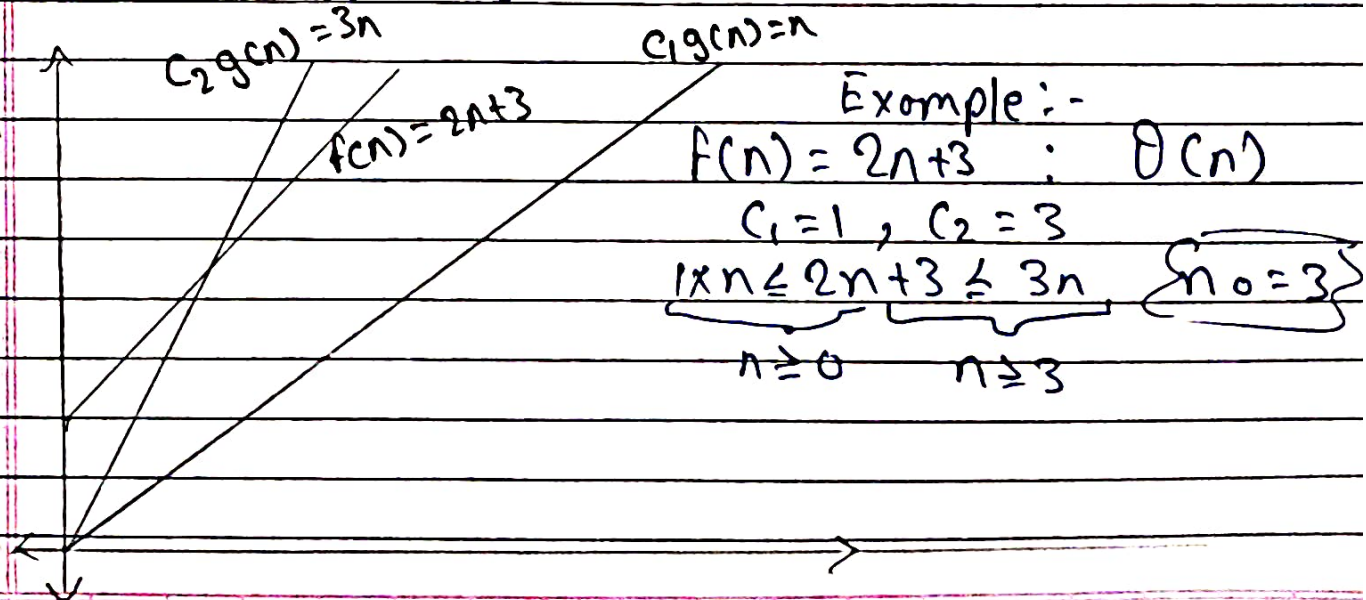
$\cup \{2n+3; 100n+\log n, \ldots\} \in \Omega(n)$

$\cup \{5000, (10)^5, \log 200, \ldots\} \in \Omega(1)$

⊛ If $f(n) = O(g(n))$
$g(n) = \Omega(f(n))$

3) Theta (Θ) :- Describes the exact bound of an algorithm's running time. It tightly bounds a function from above & below, giving a precise estimate. for example:- $\Theta(n \log n)$ means the running time grows exactly in proportion to $n \log n$ as the input size increases.

$f(n) = \Theta(g(n))$ iff there exist constant 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 :-
$f(n) = 2n+3 \quad : \quad \Theta(n)$
$c_1 = 1, \; c_2 = 3$
$1 \times n \leq 2n+3 \leq 3n \quad n_0 = 3$
$n \geq 0 \quad n \geq 3$

* Direct Method:-

- $1000n^2 + 100n\log n + 2n : O(n^2)$
- $200n^3 + 3n + 5 :- O(n^3)$
- $200n + 2\log n :- O(n)$

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

⊛ Represents Exact Bound:-

⊛ $\{100, 10^5, \log 2000, \dots\} \in O(1)$

$\{100n, 2n+\log n, 5n+3, \dots\} \in O(n)$

$\{2n^2, \frac{n^2}{4} + 5n\log n, \dots\} \in O(n^2)$

- **Analysis of common loops**

- Example:- 01)

n: User Input
c: constant

```
for (int i=0; i<n; i=i+c)
  { // some O(1) work }
```

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

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

Loop Runs $\left\lceil \frac{n}{c} \right\rceil$ times

Time complexity:- $O(n)$

- Example:-

```
for (int 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

$\left\lceil \frac{n}{c} \right\rceil$ Time complexity :- $\theta(n)$

- Example 3:- (Multiplication)

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

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

$\lceil \log_c n \rceil$

$c^0, c, c, \dots c^{k-1}$

$c^{k-1} < n$

$k < \log_c n + 1$

⊛ Time complexity :- $\theta(\log_c n)$

- Example :- 04) (Division)

```
for(int i=n; i>1; i=i/c)
  { // Some θ(1) work }
```

n = 33, c = 2

i = 33
i = 16
i = 8
i = 4
i = 2

$\lceil \log_c n \rceil$

$n/c^0, n/c, n/c^2 \cdots \frac{n}{c^{k-1}}$

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

$c^{k-1} < n$

$k-1 < \log_c n$

$k < \log_c n + 1$

$\boxed{\theta(\log_c n)}$

- Example :- 05

```
for(int i=2; i<n; i=pow(i,c)){
  { // Some O(1) work }
}
```

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

c = 514	i = 2	$2^3 = 8$
c = 3	i = 8	$8^3 = 512$
	i = 512	

$2, 2^c, (2^c)^c \dots (2^c)^c \dots$

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

$2^{c^{k-1}} < n$

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

$k-1 < \log_c \log_2 n$

$k < \log_c \log_2 n + 1$

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

Analysis of Multiple Loops:-

- Example 01:-> Subsequent loops

```
Void fun (int n)
 { for (int i=0; i<n; i++) } O(n)
   { // Some O(1) work } } +
 for (int i=1; i<n; i=i*2) } O(log n)
   { // Some O(1) work } } +
 for (int i=1; i<100; i++) } O(1)
  { // Some O(1) work } }
  }
```

We are going to add all time complexities & ignore the lower one & constant.

So, over all time complexities of the fun(int n) is

$$= O(n)$$

- Example :- 02> Nested loop

```
Void fun (int n)
 { for (int i=0; i<n; i++) -> O(n)
  { for (int j=1; j<n; j=j*2) -> O(log n)
   { // Some O(1) work }
  }
 }
```

So, over all time complexities is :- $O(n \log n)$

- Example:- 03) Mixed loops

```
Void fun (int n)
{
  for (int i = 0; i < n; i++)
  { for (int j = 1; j < n; j = j x 2)
    { Some θ(1) work }
  }}
```

} $\theta(n \log n)$

```
  for (int i=0; i<n; i++)
  { for (int j=0; j<n; j++)
    { // Same θ(1) work }
  }
```

} $\theta(n^2)$

So, over all time complexities is $\theta(n^2)$

- Example 04 :-) Different Input

```
Void fun (int n, int m)
  { for (int i=0; i<n; i++)
    { for (int j=1; j<n; j=j*2)
      { // Some θ(1) work }
    }
        for (int i=0; i<m; i++)
        for (int j=1; j<m; j++)
        {//Some θ(1) work}

        }
```

} $\theta(n \log n)$

} $\theta(m^2)$

So, over all time complexities is :-

$$\boxed{\theta(n \log n + m^2)}$$

Analysis of Recursion :- (Introduction)

- Example 01 :-)

```
Void fun(int n)
{ if(n<=0)
    return;
print("%f");
fun(n/2);
fun(n/2);
}
```

$n > 0$

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

$$\boxed{\begin{aligned} T(n) &= 2T(n/2) + \theta(1) \\ T(0) &= \theta(1) \end{aligned}}$$

This is the recurrsion relation of this function.

Base case

- Example 02 :-

```
Void fun(int n)
{ if(n<=0)
    return;
for(int i=0; i<n; i++)
θ(1) Print("%f");
T(n/2) fun(n/2);
T(n/3) fun(n/3);
}
```

$n > 0$

$$\boxed{\begin{aligned} T(n) &= T(n/2) + T(n/3) + \theta(1) \\ T(0) &= \theta(1) \end{aligned}}$$

Steps to draw

1. Draw recursive tree for given recurrence relation.
2. Calculate the cost at each level & count the total no of leaves in the recursion tree.
3. Count the total number of nodes in the last level & calculate the cost of last level.
4. Sum up the cost of all the levels in the recurvice tree.

- Example 3:-

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

```
Void fun(int n)
{ if (n<=1)
    return;
  Print("GfG")
  fun(n-1);
}
```

Recursion Tree Method for Solving Recurrences.

⇒ We consider the recursion tree & compute total work done.

⇒ We write non-recursive part as root of the tree & write the recursive part as children.

⇒ We keep expanding until we see a pattern.

Consider,

recursive part

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

Step 3:-

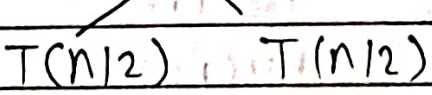
No. of nodes at level $0 = 2^0 = 1$
No. of nodes at level $1 = 2^1 = 2$
. . . . . .
No. of nodes at the $(k+1)$th level (or last level) $= 2^k = 2^{\log_2(n)} = n^{\log_2(2)} = n$

Cost of sub problem at level last $= n \times \Theta(1) = \Theta(n)$

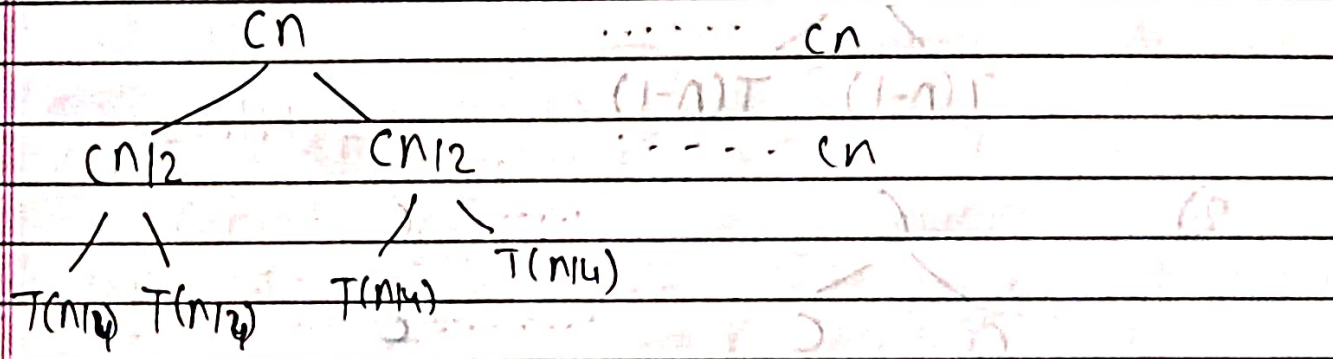
Using GP Sum formula we get

Sum of the series as $n = c(n) + \theta(n)$

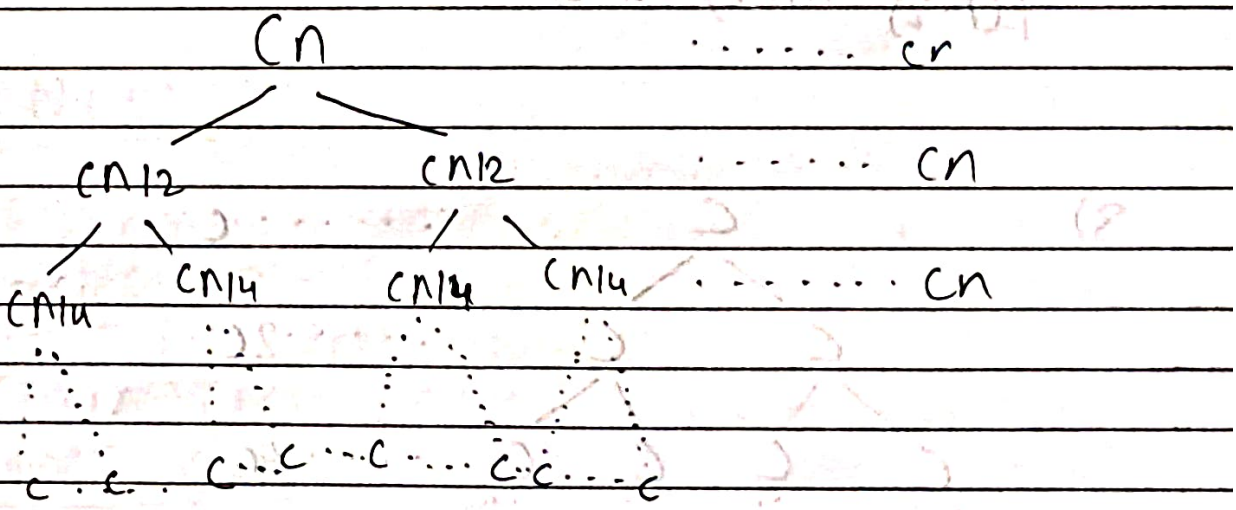
1) $T(n) = 2T(n/2) + cn$



2)



3)



- Asymtotic value of this recursion tree is $\theta(n \log n)$

Choose the longest path from root node to leaf node

$n/2^0 \rightarrow n/2^1 \rightarrow n/2^2 \rightarrow \cdots\cdots \rightarrow n/2^K$

Size of problem at last level = $n/2^K$

At last size of problem become = 1

$n/2^K = 1$

$2^K = n$

$K = \log_2(n)$

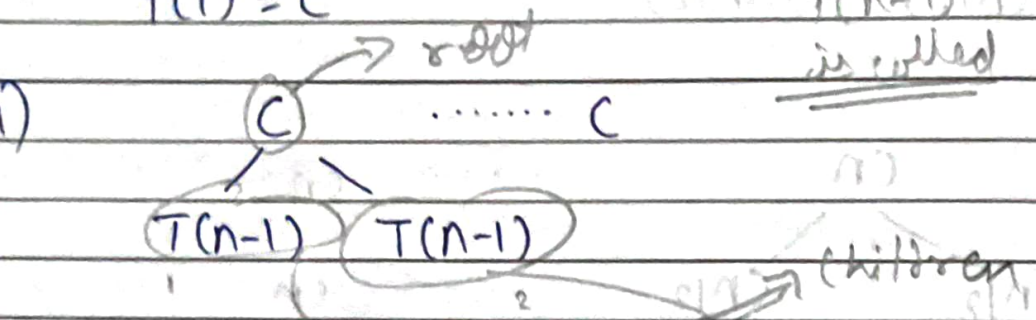
Total no of leaf in tree = $K+1 = \log_2(n) + 1$

⇒ Example:- (01)

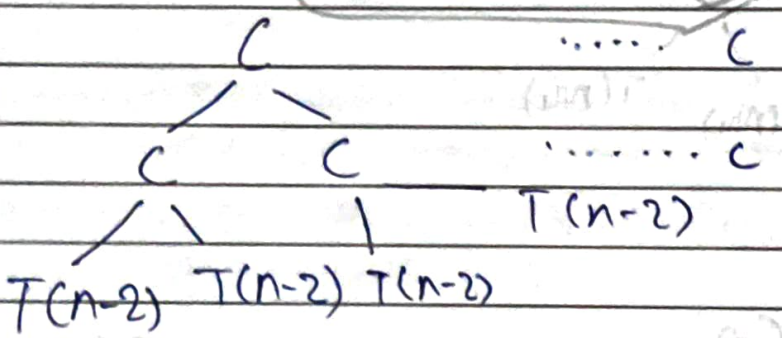
$T(n) = 2T(n-1) + C$ — 2 time $T(n-1)$ is called

$T(1) = C$

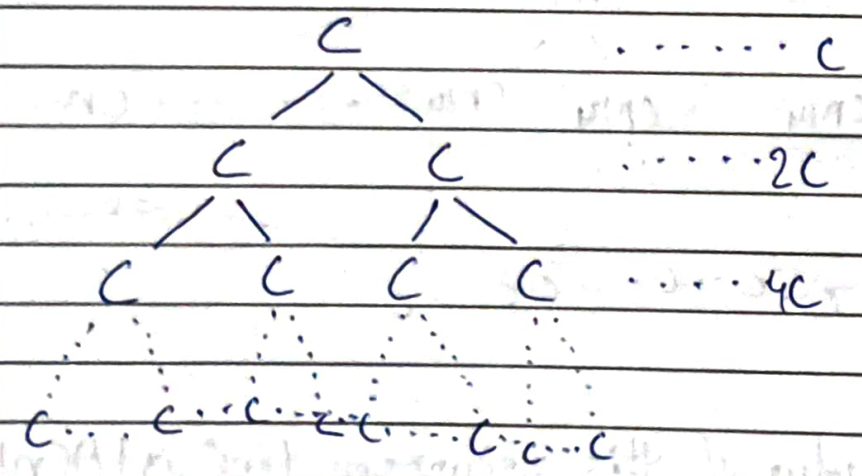
1)



2)



3)



- Step 01:- Non-recursive part as root then recursive part as childeren
- Step 02:- Then we do the some part
- Step 03:- We keep the values of at all the level of work done at first C, 2C & 4C

The value is at every level is $c, 2c, 4c, \cdots$

$\underbrace{c + 2c + 4c \cdots \cdots}_{n}$ It is a (g.p) → geometric Progression.

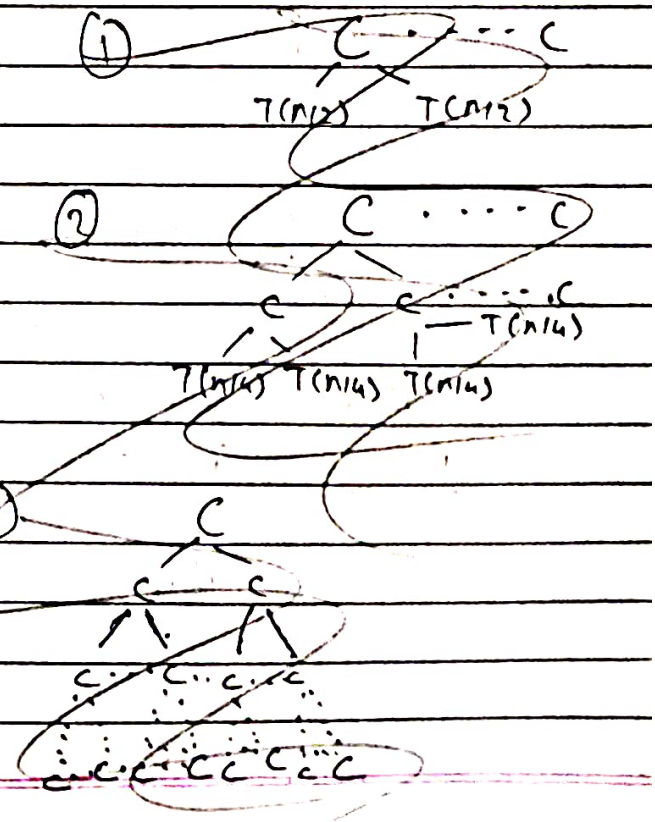
$c(1 + 2 + 4 + \cdots)$

$\left[\frac{c \; 1 \times (2^n - 1)}{2-1}\right]$ $\left[\frac{c \; 1 \times 2^{\log_2 n} - 1}{2-1}\right]$

$\boxed{\theta(2^n)}$

⇒ Example :- 02

$T(n) = T(n/2) + c$
$T(1) = c$



⇒ ① C .... C C+C+C+ C

T(n/2) $(\log_2 n)+1$

② C C

C C

T(n/2) $\theta(\log n)$

③ C C

C C

⋮

C C

⇒ Example :- (03)

$T(n) = 2T(n/2) + C$

$T(1) = C$

① C C

T(n/2) T(n/2)

② C C

C .... C — T(n/2)

T(n/2) T(n/2) T(n/2)

③ C C

C C ... 2C

C C C C ... 4C

C ... C C ... C C ... C

$\theta(n)$

Upper Bounds Using Recursion Tree Method:-

- $T(n) = T(n/4) + T(n/2) + cn$
  $T(1) = c$

① cn — cn

$T(n/4)$ $T(n/2)$

② cn ..... cn

$cn/4$ $cn/2$ ..... $3cn/4$

$T(n/16)$ $T(n/8)$ $T(n/8)$ $T(n/4)$

③ cn .... cn

$cn/4$ $cn/2$ ..... $\frac{3}{4}cn$

$cn/16$ $cn/8$ $cn/8$ $cn/4$ ..... $9cn/16$

$cn/64$ $cn/32$ $cn/32$ $cn/16$ $cn/32$ $cn/16$ $cn/16$ $cn/8$ .... $(27/64)cn$

Geometric progression is :- $cn + 3cn/4 + 9cn/16 \ldots$

$\theta(\log n)$

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

- Example:-

$T(n) = T(n-1) + T(n-2) + C$

$T(1) = C$

$T(0) = C$

If one term is changing with different speed & another with different speed. then we can get upper bound only.

① 

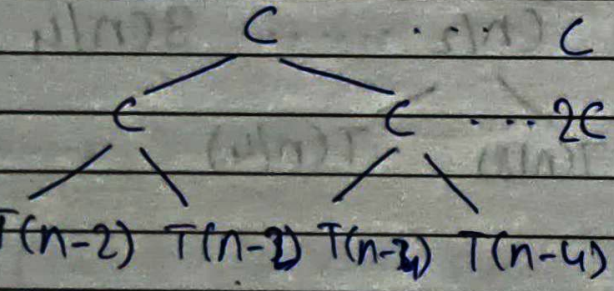
$O(C + 2C + 4C \cdots\cdots)$

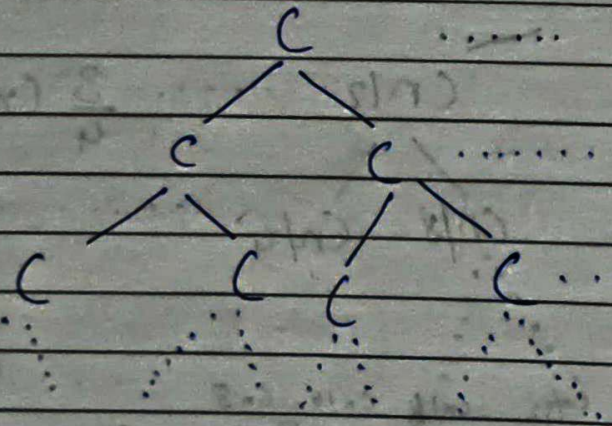
↓ n terms

$O(2^n)$

↓

We cannot find the actual answer therefore we are using Big O.

② 

③ 

SPACE COMPLEXITY

- Order of growth of memory (or RAM) space in terms of input size.
- We use some Asymptotic Notation for Space Complexity & Time complexity.

- Example:-

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

O(1) or Θ(1)

- Example:-

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

Θ(1) or O(1)

- Example:-

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

$O(n)$

* 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;
    }
```

- Auxiliary Space :- $O(1)$
- Space Complexities :- $O(n)$

- Example:-

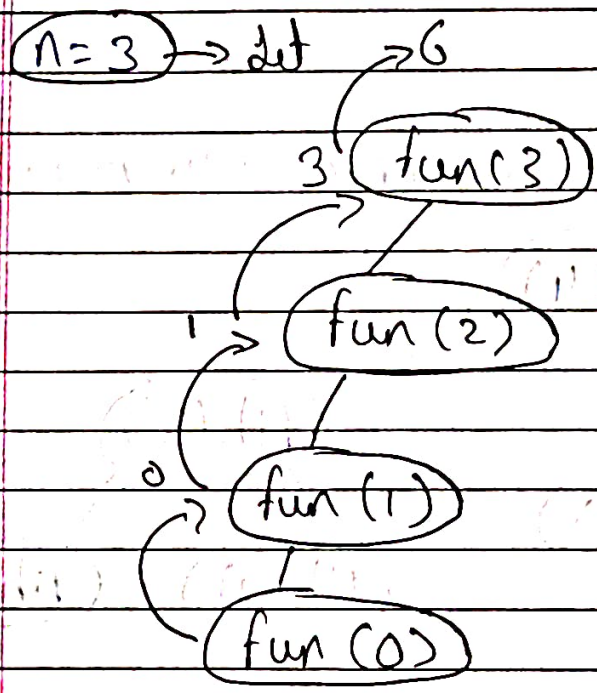
```
int fun (int n)
{ if (n<=0)
    return 0;
  else return n+fun(n-1);
}
```

$\Theta(n+1)$

$\Theta(n)$

or

$O(n)$



- Exercise considering different sol^n of fibbonacci number:.
Lets find Auxiliary space & space complexities for all this solution

On next page

⇒ (Q1)

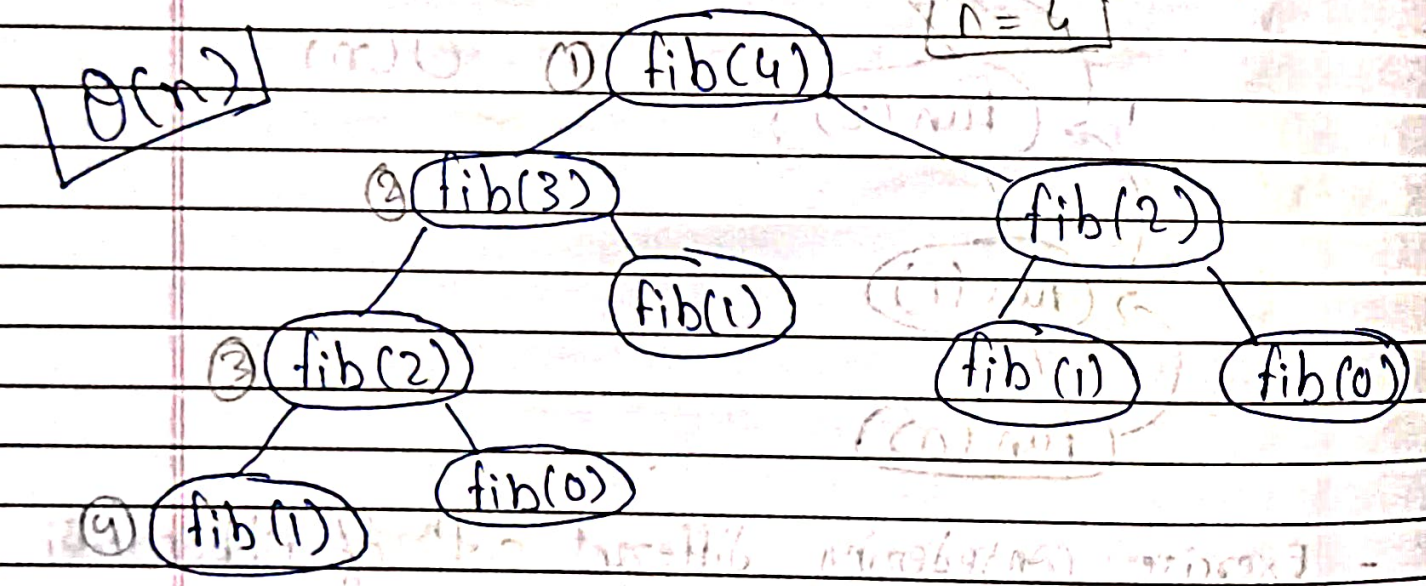
fibonnaci number:-

0, 1, 1, 2, 3, 5, ...

n=0 n=1 n=2 n=3 n=4 n=5

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

- recursion tree for fibonnaci number.



⊛ Shortcuts :- To find the auxiliary space just count the maximum height of the tree from root till leaf.

for this tree the auxiliary space is "4"

⇒ (2)

```
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];
}
```

n = 4

0	1	1	2	3
0	1	2	3	4

Auxiliary Space = $\theta(n)$ & Space complexities = $\theta(n)$

⇒ (03)

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

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

Auxiliary Space :- $\theta(1)$ & Space complexities :⇒ $\theta(1)$