UNIT - 2

Introduction to Algorithms :-

1. Algorithm :

"It is a step-by-step procedure for solving a computational Problem."

"An Algorithm is a finite set of instructions that, if followed accomplishes a particular task."

common terms that are used in Algorithm :

1) Variables : Variables refers to a specific location in computer memory used to store one and only one value

2) data-type : Set of variables takes its values.

3) Statement : "lines of code"

Characteristics of Algorithm :-

1. Input : Zero (or) more quantities are Externally supplied.

Algorithm uses values from a specific set {...}. Whatever the information, we are giving to algorithm that is called as Input.

2. Output :- Atleast one quantity is produced. For each input, algorithm produces values from a specific task.

→ Algorithm must generate result. Like a function. Even though function may not take any argument then also it returns result.

3. Definiteness :- Each instruction is clear and unambiguous. Every statement should have single and exact meaning.

4. Finiteness :- If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. Effectiveness :- Every instruction must be very basic. You should not write unnecessary steps in Algorithm.

→ Every step of algorithm should be feasible. In other words, every step of algorithm should be such that it can be carried out by pen and pencil.

The implementation of such algorithms can be done by programming languages.

→ These are the two terms are used for Algorithm & Programs.

Prior Analysis	Posterior Testing
– done on Algorithms	– done on programs
– Analyze the algorithm	– Run the program & check it.
– Independent of language	– Language dependent
– Hardware Independent	– Hardware dependent
– Time & space function	– Watch time & Bytes.

⇒ Algorithm can be expressed in different notations.

- Natural language
- It can be expressed in the form of pseudo code
- flow chart
- programming language.

2. Algorithm Specifications : Pseudocode conventions :

Algorithm is basically sequence of instructions written in simple English language. Based on algorithm there are two more representations used by programmer and those are flow chart and pseudo-code.

Flowchart is a graphical representation of an algorithm.

Similarly, Pseudo code is a representation of algorithm in which instruction sequence can be given with the help of programming constructs.

"Pseudo code is a compact & informal high-level description of a program."

Pseudo code useful only for description of a program. It is also called as "program Design Language" (PDL).

We present most of our algorithms using a Pseudo code that resembles C.

1. Algorithm is a procedure consisting of heading and body. The heading consists of name of the procedure and parameter list.

   The syntax is,

   Algorithm name_of_procedure (parameter1, - - , parametern)

2. The beginning and end of block should be indicated by { and } respectively. (or) BEGIN and END.

3. The delimiters ; are used at end of each statement.

4. comments begin with // and continue untill the end of line.

5. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context.

The pointer type is also used to point memory location. The compound data type such as structure or record can also be used.

6. Assignment of values to variables is done using the assignment statement.

<variable> := <Expression> ;

7. There are two boolean values "true" and "false". In order to produce these values, the logical operators and, or, and not and the relational operators $<, \leq, =, \neq, \geq$ and $>$ are provided.

8. Elements of multidimensional arrays are accessed using [and]. For Example, if A is a two dimensional array, the (i,j)th element of the array is denoted as A[i,j]. Array indices starts at zero.

9. The following looping statements are employed :

for, while, and repeat - until.

The while loop takes the following form :

```
while <condition> do
{
    <statement 1>
      :
    <statement n>
}
```

As long as <condition> is true, the statements get Executed. When <condition> becomes false, the loop is Exited. The value of <condition> is Evaluated at the top of the loop.

The General form of a for loop is,

```
for variable := value1 to value2 Step step do
{
    <statement 1>
      :
    <statement n>
}
```

Here value 1, value 2 and step are arithmetic expressions. A variable of type integer (or) real (or) a numerical constant is a simple form of an arithmetic expression.

The clause "Step step" is optional and taken as +1 if does not occur. step could either be positive (or) negative.

A "repeat-until" statement is constructed as follows:

```
repeat
    <statement 1>
        :
    <statement n>
until <condition>
```

The statements are executed as long as <condition> is false. The value of <condition> is computed after executing the statements.

→ The instruction "break;" can be used within any of the above looping instructions to force exit.

→ A "return" statement within any of the above also will result in exiting the loops. A return statement results in the exit of the function itself.

10. A conditional statement has the following forms:

if <condition> then <statement>

if <condition> then <statement 1> else <statement 2>

Here, <condition> is a boolean expression and <statement>, <statement 1>, and <statement 2> are arbitrary statements.

<u>case</u> <u>statement</u>:

```
case
{
    : <condition 1> : <statement 1>
                  :
                  :
    : <condition n> : <statement n>
    : else : <statement n+1>
}
```

Here, <statement 1>, <statement 2>, etc., could be either simple statements (or) compound statements.

A case statement is interpreted as follows. if <condition 1> is true, <statement 1> gets executed and case statement is exited. If <statement 1> is false, <condition 2> is Evaluated.

→ If <condition 2> is true, <statement 2> get executed and the case statement exited. and so on. If none of the conditions <condition 1> .... <condition n> are true, <statement n+1> is executed and the case stmnt is exited. The else clause is optional.

11. Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.

Examples :-

1. Write an Algorithm to count the sum of n numbers.

```
Algorithm Sum ( i, n )
{
    result := 0;
    for i=1 to n do i := i+1
    result := result + i;
}
```

2. Write an algorithm to check whether given number is Even or odd.

```
Algorithm  Eventest ( val )
{
   if ( val %.2 == 0) then
       write ("Given number is Even");
   else
     write ("Given number is odd");
}
```

3. Write an algorithm for sorting the elements.

```
Algorithm sort (a, n)
{
 // sorting elements in ascending order
 for i:=1 to n do
    for j:=i+1 to n-1 do
    {
        if (a[i] > a[j]) then
        {
            temp := a[i];
            a[i]:= a[j];
            a[j]:= temp;
        }
    }
    write (" List is sorted");
}
```

4. Write an algorithm to finds and returns the maximum of n given numbers :

```
Algorithm  Max(A, n)
// A is an array of size n
{
    Result := A[1];
    for i := 2 to n do
        if A[i] > Result then Result := A[i];
    return Result;
}
```

5. Write an Algorithm to find factorial of n number.

```
Algorithm  fact(n)
{  if n := 1 then
       return 1;
   Else
       return  n * fact(n-1);
}
```

6. Write an algorithm to perform multiplication of two matrices.

```
Algorithm  MUL(A, B, n)
{
```

```
for i:=1 to n do
    for j:=1 to n do
        c[i,j]:=0
    for K:=1 to n do
        c[i,j] := c[i,j] + A[i,K] * B[K,j];
}
```

3. Performance Analysis — Space & Time complexity.

The Efficiency of an algorithm can be decided by measuring the performance of an algorithm. performance analysis can be measured interms of space & time complexity.

An algorithm is said to be efficient & fast, if it takes less time to execute and it consumes less memory space. then we can say "algorithm is fit and performance wise it is good".

We can measure the performance of an algorithm by computing amount of time and storage requirement.

3.1. Space complexity :-

The space complexity can be defined as amount of memory required by an algorithm to run.

Space complexity can be calculated in two factors : (1) constant program 2) linear program based on program.

constant space complexity :

```
int square(int a)
{
    return a * a ;
}
```

Here, algorithm requires fixed amount of space for all input values. So, the space complexity is constant.

Linear space complexity :

The space needed for algorithm, can be calculated by standard instructions,

- Size of variable 'n' = 1 word
- array of a values = n word
- loop variable i = 1 word
- Sum of variable is = 1 word.

The amount of space required by an algorithm increases with an increasing input values. we call it as "linear Space".

For Each variable, input value increases means amount of space increases.

Ex:-
```
int sum(int A[], int n)
{
    int sum=0, i;
    for(i=0; i<n; i++)
    sum = sum + A[i];
    return sum;
}
```

1 + n + 1 + 1 = n + 3 words.

- For Each variable, we allocating some amount of space i,e., Space complexity.

Time complexity :-

- The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

- There are two types of computing time - compile time and run time. The time complexity is generally computed using run time (or) Execution time.

The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

It is difficult to compute the time complexity in terms of physically clocked time. For instance in multiuser system, Executing time depends on many factors such as,

- System load
- No. of other programs running
- Instruction set used
- Speed of underlying hardware.

∴ The time complexity is given in terms of "frequency count".

"Frequency count" is a count denoting no. of times of execution of statement.

- for Example,

If we write a code for calculating sum of 'n' numbers in an array then we can find its time complexity using

"frequency count": This frequency count denotes how many times the particular statement is executed.

Ex: Algorithm Sum(A, n)

```
{
    s = 0;
    for (i=0; i<n; i++)
    {
        s = s + A[i];
    }
    return s;
}
```

The statements inside the loop will execute 'n' times and loop itself n+1 times.

Thus, we get frequency count to be $f(n) = 2n+3$. The time complexity is normally denoted in terms of oh-notation (O). Hence, if we neglect the constants then we get the time complexity to be O(n).

Space complexity for this Example,

Variables : A — n words

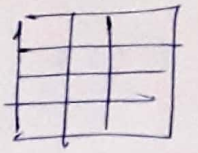
n — 1 word

s — 1 word

i — 1 word

$S(n) = n+3$ words.

$n \times n$ matrix



Ex: 2 Algorithm Add (A, B, n)

```
{
    for (i=0 ; i<n ; i++)                    — n+1
    {
        for (j=0 ; j<n ; j++)                — n x n+1
                                               = n^2+n
        {
            c[i,j] = A[i,j] + B[i,j]         — n x n = n^2
        }
    }
}
```

$\Rightarrow n+1+n^2+n+n^2$

Time complexity : $\boxed{f(n) = 2n^2 + 2n + 1}$

$\boxed{f(n) = O(n^2)}$

Space complexity : Variables :-

2-dimensional matrices $n \times n$

- A — n^2
- B — n^2
- C — n^2

Simple variables

- n — 1
- i — 1
- j — 1

$\boxed{S(n) = 3n^2 + 3}$

$\boxed{S(n) = O(n^2)}$

Ex:3 Algorithm Multiply (A, B, n)

```
{
    for (i=0 ; i<n ; i++)
    {
        for (j=0 ; j<n ; j++)
        {
            c[i,j] = 0 ;
            for (k=0 ; k<n ; k++)
            {
                c[i,j] = c[i,j] + A[i,k] * B[k,j] ;
            }
        }
    }
}
```

Time complexity : $f(n) = 2n^3 + 3n^2 + 2n + 1$

If we neglect constants $f(n) = O(n^3)$

Space complexity :

Variables : A — n^2
B — n^2
C — n^2
n — 1
i — 1
j — 1
k — 1

$S(n) = 3n^2 + 4$

$S(n) = O(n^2)$

Asymptotic Notation :-

The main idea of asymptotic analysis is to have a measure of efficiency of algorithm that does not depend on -

- Machine constants
- doesn't require algorithm to be implemented
- time taken by program to be compare.

It is the way to describe the behaviour of the function in the limit (or) without bounds.

Asymptotic Growth :-

The rate at which the function grows.

"Growth Rate" Meaning describes the complexity of a function (or) & the amount of resources it takes up to compute.

classification of growth :-

- growing with same rate
- growing with a slower rate
- growing with a faster rate.

The three different asymptotic notations are mostly used to represent the complexity of an algorithm.

1. Theta θ - notation

2. Big oh O - notation

3. Omega Ω - notation.

Theta notation :-

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Asymptotic - "Equality" (Same growth rate)

Big-oh Notation :-

$$f(n) \leq c\, g(n)$$

Asymptotic - "less than" (Slower growth rate)

Omega Notation :-

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

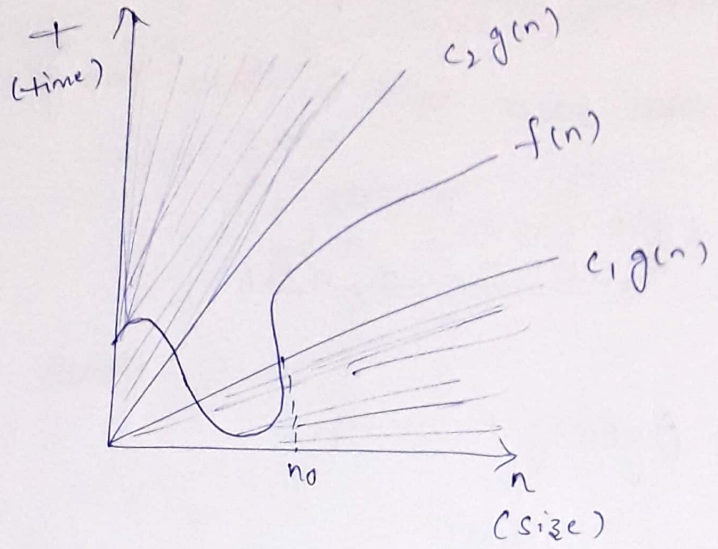
Asymptotic - "greater than" (Faster growth rate).

Theta Notation :-

We choose $f(n)$ & $g(n)$ as two differentiable function and say that they have same growth rate.

if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \quad \therefore\ 0 < c \leq \infty$

formally stated as, $f(n) = \Theta(g(n))$



$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Here, $c_1, c_2 > 0$

$n \geq n_0$

$n_0 \geq 1$

Here, $f(n)$ exist between these two functions.

Ex:- $f(n) = 3n+2$ $\quad g(n) = n$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\Downarrow$$

$f(n) \leq c_2 g(n)$ $\quad$ $f(n) \geq c_1 g(n)$ $\quad \because c_1 = 1$

$3n+2 \leq 4n$ $\quad$ $3n+2 \geq n$ $\quad$ {lower bound}

$\because c_2 = 4$ {upper bound}

$n_0 \geq 1$

Big - oh Notation :-

The two differentiable functions $f(n)$ & $g(n)$.

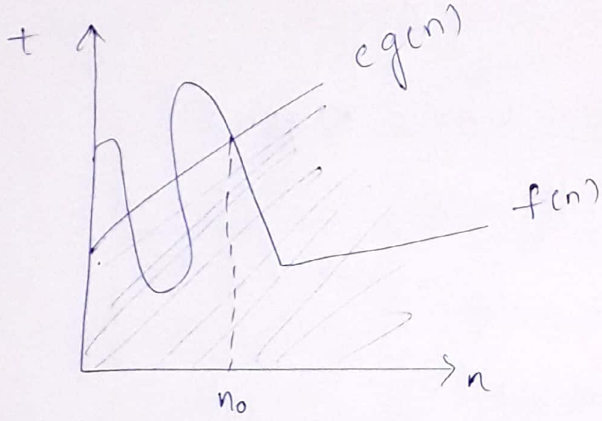
" $f(n)$ grows with same rate & slower than $g(n)$

$$f(n) \leq c\,g(n), \quad n \geq n_0$$

$$c > 0,\ n_0 \geq 1$$

denoted as,

$$F(n) = O\,g(n)$$



" $g(n)$ is an asymptotic upper bound for $f(n)$".

Ex:- $f(n) = 3n+2$; $g(n) = n$

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

$$f(n) \leq c\,g(n)$$

$$3n+2 \leq cn \quad \because c = 4 \text{ (assume)}$$

$$3n+2 \leq 4n \quad \because n \geq 2$$

Big - Omega Notation :- (Ω)

The two differentiable function $f(n)$ & $g(n)$.

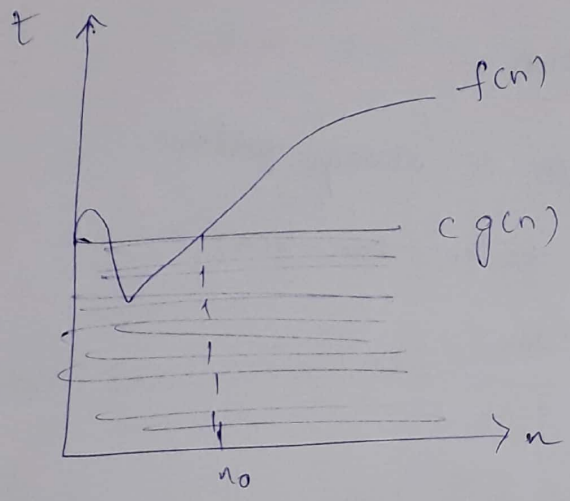
"$f(n)$ grows with same rate (or) faster than $g(n)$"

$$f(n) \geq cg(n) \quad \because n \geq n_0$$

$$c > 0, \; n_0 \geq 1$$

denoted as,

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



"$g(n)$ is an asymptotic lower bound for $f(n)$.

Ex :- $f(n) = 3n+2$ $\quad g(n) = n$

$$f(n) \geq cg(n)$$

$$3n+2 \geq cn$$

$$3n+2 \geq n \qquad n_0 \geq 1, \; c = 1$$

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

Little-oh Notation (o) :-

The growth rate is always slower

"f(n) grows slower than g(n)"

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

Formally stated as,

$$f(n) = o(g(n))$$

Little omega Notation :-

The growth rate is always faster

f(n) grows faster than g(n)

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

formally stated as,

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

Examples :-

1.
```
for (i=0; i<n; i++)
{
    stmnt;
}
```

stmnt — n times

$O(n)$

2. for (i=n ; i>0; i--)
{
stmnt ; — n times
}
O(n)

3. for (i=1 ; i<n ; i = i+2)
{
stmnt ; — This statement is not executing for n times. n/2 times
}
$f(n) = O(n)$

4. for (i=0; i<n ; i++) — n+1
{
for (j=0; j<n ; j++) — n × (n+1)
{
stmnt; — n × n
}
}
$\Rightarrow n+1+n^2+n+n$
$\Rightarrow 2n^2+2n+1$
$\Rightarrow O(n^2)$

5. for (i=0; i<n; i++)
{
for (j=0; j<i; j++)
{
stmnt ;
}
}

i	j	stmnt
0	0 ✗	0 times
1	0 ✓	1 time
	1 ✗	
2	0 ✓	2 times
	1 ✓	
	2 ✗	
3	0 ✓	3 times
	1 ✓	
	3 ✗	
⋮		
n		n times

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$f(n) = \frac{n^2+n}{2}$$

$$\boxed{f(n) = O(n^2)}$$

6. `for (i=1 ; i<n ; i = i*2)`

```
{
  stmt;
}
```

Assume i becomes $>= n$

i
$1 \times 2 = 2$
$2 \times 2^1 = 2^2$
$2^2 \times 2 = 2^3$
⋮
2^K times

$i >= n \Rightarrow 2^K >= n \quad \because i = 2^K$

Equate, $K = \log_2 n$

The statement will Execute $O(\log_2 n)$ times.

If we take $\log n$. $\log n$ may give decimal values (or) float value. We need to know whether ceil (or) floor operation has to perform on this $\log n$:

$n = 8$

i
1
2
4
8

] 3

$n = 10$

i
1
2
4
8
16

] 4

$\log_2^8 = 3$

$\log_2^{10} = 3.2 \ldots$

This log should take ceil operation $[\log_2^n]$

Ex:7

```
for (i=n ; i>=1 ; i = i/2)
{
    stmnt;
}
```

i
n
n/2
n/2^2
n/2^3
!
n/2^K

Assume, when i becomes < 1

$i < 1$

$\therefore \frac{n}{2^K} < 1$

if we Equate

$\frac{n}{2^K} = 1$

$n = 2^K$

$K = \log_2^n$

$\therefore$ The time complexity for this piece of code is

$T(n) = O(\log_2^n)$

Ex:8

```
for (i=0; i<n; i++)      - n+1
{
   stmnt;                - n
}
for (j=0; j<n; j++)      - n+1
{
   stmnt;                - n
}
```

$f(n) = 4n+2$

$f(n) = O(n)$

Ex: 9

```
for (i=0; i<n; i++)      - n+1
{
   for (j=1; j<n; j=j*2) - n x log n
   {
      stmnt;             - n x log n
   }
}
```

$\Rightarrow n+1+n\log n+n\log n$

$\Rightarrow 2n\log n+n+1$

$f(n) = O(n\log n)$

Ex: 10

```
for (i=0; i*i<n; i++)
{
   stmnt;
}
```

$i*i >= n$

$i^2 = n$

$i = \sqrt{n}$

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

```
for (i=0; i<n ; i++)          — O(n)
for (i=0; i<n ; i=i+2)        — n/2 times O(n)
for (i=n; i>1 ; i--)          — O(n)
for (i=1; i<n ; i=i*2)        — O(log2 n)
for (i=n; i>1 ; i=i/2)        — O(log2 n)
```

Ex:- 11

```
i=0                           — 1
While (i<n)                   — n+1
{
  stmnt ;                     — n
  i++ ;                       — n
}
```

if we take n = 10 then 11 times it check the condition and 10 times the statements will execute.

$f(n) = 3n + 2$

$\boxed{f(n) = O(n)}$

Same can be written in for loop,

```
for (i=0;  i<n ;  i++)        => 2n+2
     1     n+1    n
{
  stmnt ;     — n             -> n
}                             ------
                              3n+2
```

$f(n) = O(n)$

Ex: 12

```
a = 1;
while (a < b)
{
  Stmnt;
  a = a * 2;
}
```

$a = 1$

$a = 1 * 2 = 2$

$a = 2 \times 2 = 2^2$

$a = 2^2 \times 2 = 2^3$

!

! K times

2^K times

The loop terminates, $a \geq b$

$$2^K \geq b$$

If we Equate

$$2^K = b$$

$$K = \log_2^b \text{ times}$$

$$f(n) = O(\log_2^n) \text{ times}$$

⇓

for loop: -

```
for (i = 0; i < n; i = i * 2)
{
  Stmnt;
}
```

$$f(n) = O(\log_2^n)$$

13.
```
i = n ;
while (i > 1)
{
   stmnt ;
   i = i/2 ;
}
```

$\Rightarrow$

```
for (i = n ; i > 1 ; i = i/2)
{
   stmnt ;
}
```

$f(n) = O(\log_2 n)$

14.
```
i = 1 ;
K = 1 ;
while (K < n)
{
   stmnt ;
   K = K + i ;
   i++ ;
}
```

i	K
1	1
2	2 (1+1)
3	2+2
4	2+2+3
5	2+2+3+4
⋮	⋮
m times	Sum of 'm' natural no's $\frac{m(m+1)}{2}$

loop terminates when,

$K \geq n$

$\frac{m(m+1)}{2} \geq n$ (m^2)

~~$f(n) = O(n^2)$~~

Equates

$m^2 = n$

$m = \sqrt{n}$

$\Rightarrow$

```
for (K = 1 ; i = 1 ; K < n ; i++)
{
   stmnt ;
   K = K + i ;
}
```

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

15. while (m != n)

```
{
    if (m > n)
    m = m - n;
    else
      n = n - m;
}
```

This will repeat until m == n

→ m = 6 n = 3 initially

m = 3 n = 3

→ m = 16 n = 2

m = 14 n = 2

m = 12 n = 2

m = 10 n = 2

m = 8 n = 2

m = 6 n = 2

m = 4 n = 2

m = 2 n = 2

4 times

almost $\frac{16}{2}$ times

So, it can be taken as $n/2$ times

$f(n) = O(n)$.

Minimum time taken by this alg is $O(1)$.

Maximum time taken by this alg is $O(n)$.

Types of Time functions :-

O(1) – constant

$f(n) = 2$
$f(n) = 100$ → O(1)
$f(n) = 50000$

O(logn) – logarithmic

O(n) – linear

$f(n) = 2n + 3$
$f(n) = 500n + 700$ → O(n)
$f(n) = \frac{n}{5000} + 6$

$O(n^2)$ – Quadratic

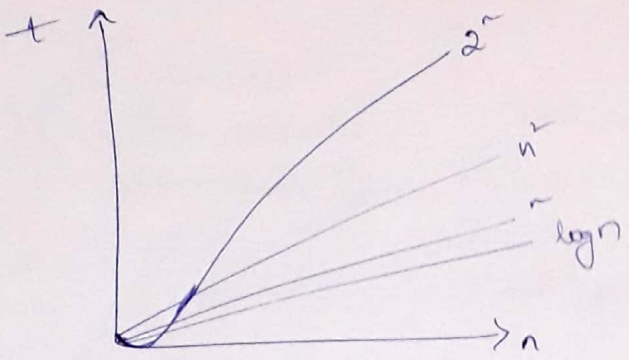
$O(n^3)$ – cubic

$O(2^n)$
$O(3^n)$ → Exponential
$O(n^n)$

Increasing order of classes of functions :-

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < n^n$.

logn	n	n^2	2^n
0	1	1	2
$\log_2^2 = 1$	2	4	16
2	4	16	16
3	8	64	256
3..	9	81	512



Ex: 16

```
A()
{
    i = 1;
    s = 1;
    while (s <= n)
    {
        i++;
        s = s + i;
        Pf("Algorithm");
    }
```

s =	1	3	6	10	15	21	- - -
i =	1	2	3	4	5	6	- - - K times

The value of 's' is sum of natural numbers of 'i'

$$s = \frac{K(K+1)}{2}$$

$s > n$

$$\frac{K(K+1)}{2} > n$$

if we equate

$K^2 = n$

$K = \sqrt{n}$

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

Ex: 17

```
A()
{
  int i, j, k, n
  for(i=1; i<n; i++)
  {
    for(j=1; j<=i; j++)
    {
      for(k=1; k<=100; k++)
      {
        Printf("Algorithm");
      }
    }
  }
```

i = 1	i = 2	i = 3
j = 1 time	j = 2 times	j = 3 times
K = 100 times	K = 2 * 100 times	K = 3 * 100 times

i = n

j = n

K = n * 100 times.

Total no. of times this printf statement will Execute.

$100 + 2*100 + 3*100 + 4*100 + \cdots n*100$

$100(1+2+3+4+\cdots+n)$

$100\left(\frac{n(n+1)}{2}\right) \Rightarrow O(n^2)$

Ex:- 18

```
A()
{
    int i, j, k, n
    for (i=1; i<=n ; i++)
    {
        for (j=1; j<=i ; j++)
        {
            for (k=1; k<=n/2 ; k++)
            {
                printf ("Algorithm");
            }
        }
    }
```

$i=1$	$i=2$	$i=3$
$j=1$ time	$j=4$ times	$j=9$ times
$k=\frac{n}{2}*1$	$k=\frac{n}{2}*4$	$k=\frac{n}{2}*9$

$i=n$

$j=n^2$ times

$k=\frac{n}{2}*n^2$

∴ The total no. of times the stmt is going to execute.

$$\frac{n}{2}*1+\frac{n}{2}*4+\frac{n}{2}*9+\cdots\frac{n}{2}*n^2$$

$$\frac{n}{2}\left[1+4+9+\cdots+n^2\right]$$

$$\frac{n}{2}\left[\frac{n(n+1)(2n+1)}{6}\right]=O(n^4)$$

Ex: 19

```
A()
{
    int i, j, k;
    for (i = n/2; i <= n; i++)        - n/2 times
        for (j = 1; j <= n/2; j++)    - n/2 times
            for (k = 1; k <= n; k * 2)  - log n
                printf("Algorithm")
}
```

$$f(n) = \frac{n}{2} * \frac{n}{2} * \log n \Rightarrow O(n^2 \log n)$$

Small oh - Notation :- [Little oh]

The function $f(n) = o(g(n))$ [read as "f of n is little oh of g of n") iff

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

Here, $g(n)$ is vaiably bigger than $f(n)$

Ex:-1 $f(n) = n$, $g(n) = n^2$

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

Whenever $f(n) = o(g(n))$ then it says $f(n)$ is asymptotically smaller than $g(n)$. $\therefore n = o(n^2)$.

Ex:-2 $f(n) = \log n$, $g(n) = n$

$$\lim_{n\to\infty} \frac{\log \infty}{\infty} = \frac{\infty}{\infty}$$ [It is called indefinite form]

Apply L-hospital Rule.

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

$$\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \frac{1/n}{1} \Rightarrow \lim_{n\to\infty} \frac{1}{n} = \frac{1}{\infty} = 0$$

"$\log n$ is asymptotically smaller than n"

$$\boxed{\log n = o(n)}$$

Small omega Notation :- [little omega]

$f(n) = w(g(n))$ iff

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

Ex: $f(n) = 3^n$ $\quad g(n) = 2^n$

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n$$

$$= \lim_{n\to\infty} (1.5)^n$$

$$= (1.5)^{\infty} = \infty$$

$\therefore 3^n = w(2^n)$

"$f(n)$ is asymptotically bigger than $g(n)$"

Ex: $f(n) = n^2$, $g(n) = \log n$

$$\lim_{n\to\infty} \frac{n^2}{\log n} = \frac{\infty^2}{\log\infty} = \frac{\infty}{\infty}$$

Apply L-Hospital Rule,

$$\lim_{n\to\infty} \frac{2n}{1/n} \Rightarrow 2n^2 \Rightarrow 2\infty^2$$

$$\Rightarrow \infty$$

$n^2 = w(\log n)$

Examples:-

1) prove that $5n+3 = O(n)$

consider. $f(n) = 5n+3$

$f(n) \leq c \cdot g(n)$

$5n+3 \leq c \cdot n$ $\quad \therefore c = 6$

$5n+3 \leq 6n$

$\boxed{f(n) = O(n)}$

for $n=1$, $8 \leq 6$ False

$n=2$, $13 \leq 12$ false

$n=3$, $18 \leq 18$ True

$\boxed{\therefore f(n) = O(n), \forall n \geq 3, c = 6}$

2) prove that $6n^2+2n+3 = O(n^2)$

consider $f(n) = 6n^2+2n+3$

$f(n) \leq c \cdot g(n)$

$6n^2+2n+3 \leq c \cdot n^2$ $\quad \therefore c = 7$

$6n^2+2n+3 \leq 7n^2$

for $n=1$, $11 \leq 7$ False

$n=2$, $32 \leq 28$ False

$n = 3$, $63 \leq 63$ True

$f(n) = O(n)$, $\forall\ n \geq 3$, $c = 7$

3) Prove that $5n+3 = \Omega(n)$

consider $f(n) = 5n+3$

$f(n) \geq c \cdot g(n)$

$5n+3 \geq c \cdot n$ $\quad \therefore c = 4$

$5n+3 \geq 4n$

for $n=1$, $8 \geq 4$ True

$f(n) = \Omega(n)$, $\forall\ n \geq 1$, $c = 4$

4) Prove that $6n^2+2n+3 = \Omega(n^2)$

consider $f(n) = 6n^2+2n+3$

$f(n) \geq c \cdot g(n)$

$6n^2+2n+3 \geq c \cdot n^2$ $\quad \therefore c = 5$

$6n^2+2n+3 \geq 5n^2$

for $n=1$, $11 \geq 5$ True

$f(n) = \Omega(n^2)$ $\forall\ n \geq 1$, $c = 5$

5) Prove that $5n+3 = \theta(n)$

consider $f(n) = 5n+3$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$c_1 \cdot n \leq 5n+3 \leq c_2 \cdot n \quad \therefore c_1 = 4, \; c_2 = 6$$

$$4n \leq 5n+3 \leq 6n$$

$n=1$, $4 \leq 8 \leq 6$ False

$n=2$, $8 \leq 13 \leq 12$ False

$n=3$, $12 \leq 18 \leq 18$ True

$f(n) = \theta(n) \quad \forall \; n \geq 3, \; c_1 = 4, \; c_2 = 6$

Recurrences :-

The word Recurrence is recursion (or) repeatition.

→ Recursion is a technique which call the same function repeatedly. Algorithm which use recursion is called recursive algorithm. The time complexity of recursive algorithm can be analyzed by a formula (or) equation called "Recurrence Relation".

There are three Methods to solve the recurrence.

1) Substitution Method 2) Recursion Tree Method

By Master Method.

Ex:-1
```
void Test (int n)          — T(n)
{
  if (n>0)
  {
    printf ("%d", n);      — 1
    Test (n-1);            — T(n-1)
  }
}
```

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

The Recurrance Relation,

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1)+1 & n > 0 \end{cases}$$

Ex:-2
```
void Test (int n)          — T(n)
{
  if (n>0)
  {
    for (i=0; i<n; i++)
    {
      printf ("%d", n);    — n
    }
    Test (n-1);            — T(n-1)
  }
}
```

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

Recurrence Relation,

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Ex:-3

```
Void Test (int n)          ——— T(n)
{
  if (n>0)
  {
    for (i=1; i<n; i = i*2)
    {
      Printf ("%d", i);      — logn
    }
    Test (n-1);              — T(n-1)
  }
}
```

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

The Recurrence Relation,

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + \log n & n > 0 \end{cases}$$

Ex: 4 Algorithm Test (int n) — T(n)

```
{
  if (n>0)
  {
    printf("%d", n);      — 1
    Test(n-1);            — T(n-1)
    Test(n-1);            — T(n-1)
  }
}
```

$$\boxed{T(n) = 2T(n-1) + 1}$$

The Recurrence Relation,

$$T(n) = \begin{cases} 1 & n = 0 \\ 2T(n-1)+1 & n > 0 \end{cases}$$

1. Substitution Method :-

Substitution Method follows two steps :

1) Assume that solution is correct

2) Prove this assumption by substitution Method.

Ex:- 1 $T(n) = \begin{cases} 1 & n = 0 \\ T(n-1)+1 & n > 0 \end{cases}$ find time complexity?

Sol:- We solve this recurrence relation by back substitution Method.

$T(n) = T(n-1) + 1$ — (1)

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

$= T(n-2) + 1$

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

$= T(n-2) + 2$ — (2)

$\therefore T(n-2) = T(n-2-1) + 1$

$= T(n-3) + 1$

$T(n) = T(n-3) + 1 + 2$

$= T(n-3) + 3$ — (3)

continue for K - steps

$T(n) = T(n-K) + K$ — (4)

let $n - K = 0$

$\boxed{n = K}$

Substitute 'n' in the place of 'K' in (4)

$T(n) = T(n-n) + n$

$= T(0) + n$

$= 1 + n$

$\boxed{T(n) = O(n)}$

Ex:-2 $T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$

By back substitution Method,

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

$$\therefore T(n-1) = T(n-1-1) + n-1$$
$$= T(n-2) + n-1$$

$$T(n) = T(n-2) + (n-1) + n \quad -(2)$$

$$\therefore T(n-2) = T(n-2-1) + n-2$$
$$= T(n-3) + n-2$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad -(3)$$

After K - steps

$$T(n) = T(n-K) + (n-(K-1)) + (n-(K-2)) + \cdots + n \quad -(4)$$

Let $n - K = 0$

$n = K$

Substitute 'n' in the place of 'K'.

$$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \cdots + n$$

$$= T(0) + \underbrace{1 + 2 + \cdots (n-1) + n}$$

$$= T(0) + \frac{n(n+1)}{2} \Rightarrow 1 + \frac{n(n+1)}{2}$$

$$\boxed{T(n) = O(n^2)}$$

Ex:-3 $$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + \log n & n > 0 \end{cases}$$

By back Substitution Method,

$$T(n) = T(n-1) + \log n \quad - (1)$$

$$\therefore T(n-1) = T(n-1-1) + \log(n-1)$$
$$= T(n-2) + \log(n-1)$$

$$T(n) = T(n-2) + \log(n-1) + \log n \quad - (2)$$

$$\therefore T(n-2) = T(n-2-1) + \log(n-2)$$
$$= T(n-3) + \log(n-2)$$

$$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n \quad - (3)$$

:

After K- steps.

:

$$T(n) = T(n-K) + \log(n-(K-1)) + \log(n-(K-2)) + \log n$$

Let $n - K = 0$

$n = K$

$$T(n) = T(n-n) + \log 1 + \log 2 + \cdots + \log(n-1) + \log n$$

$$T(n) = T(0) + \log n!$$

$$T(n) = 1 + \log n! \Rightarrow O(n \log n)$$

"Recursion Tree" gives a graphical view of recurrence.

Ex:-4 $$T(n) = \begin{cases} 1 & n = 0 \\ 2T(n-1)+1 & n > 0 \end{cases}$$

By back substitution Method,

$T(n) = 2T(n-1)+1$ —(1) $\because T(n-1) = 2T(n-1-1)+1$

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

$T(n) = 2T(n-2)+2$ —(2)

$\because T(n-2) = 2T(n-2-1)+1$

$= 2T(n-3)+1$

$T(n) = 2T(n-3)+1+2$

$= 2T(n-3)+3$ —(3)

After K steps.

$T(n) = 2T(n-K)+K$ —(4)

Let $n-K = 0$

$n = K$

Substitute 'n' in the place of 'K'.

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

$= 2T(0)+n$

$T(n) = 2+n = O(n)$

Ex:- 5 $T(n) = \begin{cases} 2T(n/2) & n > 0 \\ 0 & n = 0 \end{cases}$

Prove that $T(n) = O(n\log n)$ by Substitution Method.

Sol:- Assume $T(n) = O(n\log n)$

$$T(n) \leq c \cdot n\log n$$

$$T(n/2) \leq c \cdot \frac{n}{2}\log\frac{n}{2}$$

Given that,

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

$$T(n) \leq 2 \cdot c \cdot \frac{n}{2}\log\frac{n}{2}$$

$$T(n) \leq cn\log\frac{n}{2}$$

$$T(n) \leq cn[\log n - \log 2]$$

$$T(n) \leq cn(\log n - 1)$$

$$T(n) \leq cn\log n - cn$$

Here, we neglect 'cn' smallest value.

$$T(n) \leq cn\log n.$$

$$\therefore T(n) = O(n\log n)$$

Hence proved

Ex:6 :- $T(n) = 2T(n-1) + 1$

Prove that $T(n) = O(2^n)$ by Substitution Method.

Sol:- Assume that, $T(n) = O(2^n)$

$$T(n) \leq c \cdot 2^n$$

$$T(n-1) \leq c \cdot 2^{n-1} \quad -(1)$$

Given that,

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

$$T(n) \leq 2 \cdot c \cdot 2^{n-1} + 1$$

$$T(n) \leq \cancel{2} \cdot c \cdot \frac{2^n}{\cancel{2}} + 1$$

$$T(n) \leq c \cdot 2^n + 1 \quad \therefore \text{Neglect '1' Small Value}$$

$$T(n) \leq c \cdot 2^n$$

$$\therefore T(n) = O(2^n)$$

Hence proved

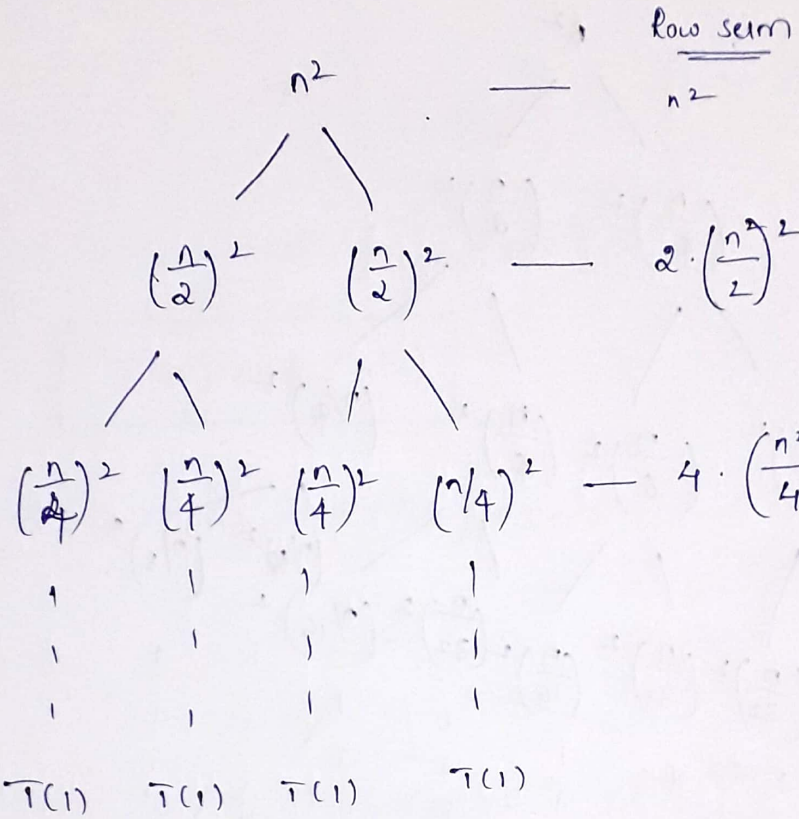
2. Recursion Tree Method :-

In this Method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic (or) geometric Series.

"Recursion Tree" gives a graphical view of recurrence.

Ex:-1 $T(n) = 2T(n/2) + n^2$

The second term in Recurrence relation becomes our root node.



$$T(n) = n^2 + 2\left[\left(\frac{n}{2}\right)^2\right] + 4\left[\left(\frac{n}{4}\right)^2\right] + 8\left[\left(\frac{n}{8}\right)^2\right] + \cdots$$

$$= n^2 + 2 \cdot \frac{n^2}{4} + 4 \cdot \frac{n^2}{16} + 8 \cdot \frac{n^2}{64} + \cdots$$

$$= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots$$

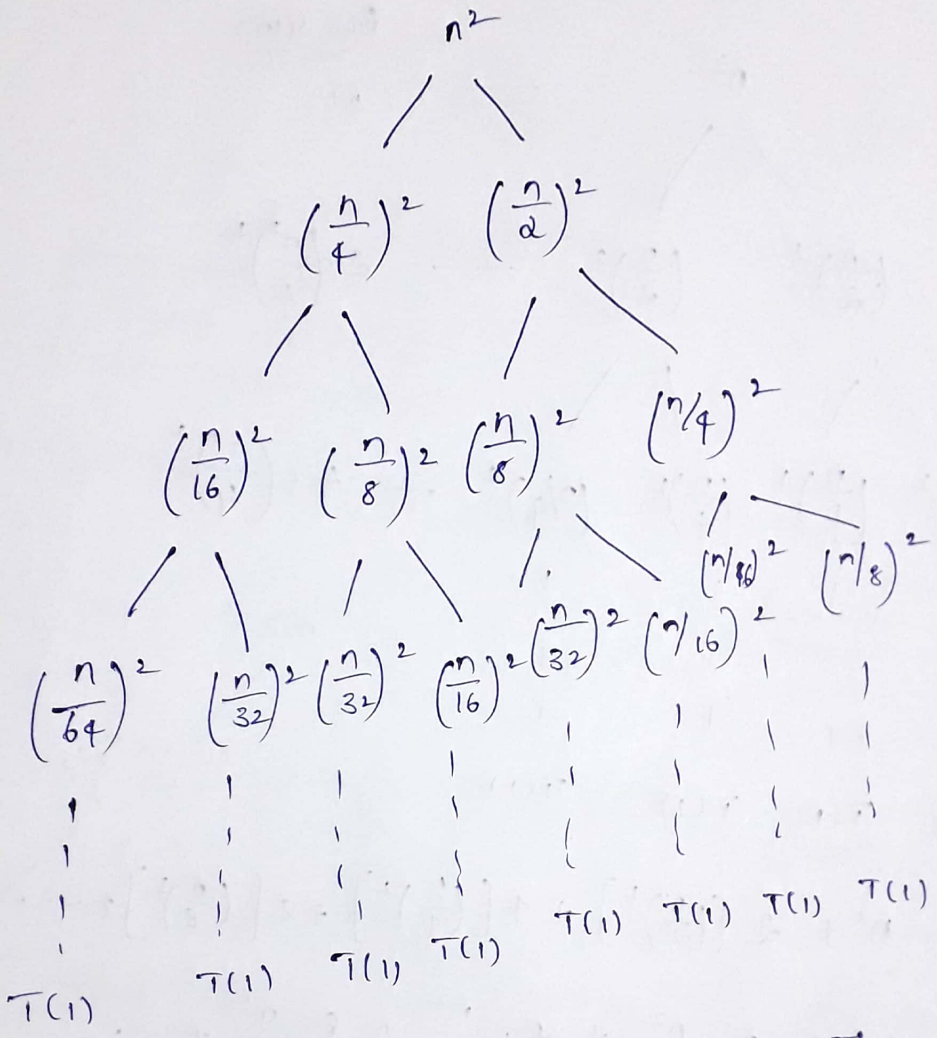
$$= n^2 \left[1 + \frac{1}{2} + \frac{1}{4} + \cdots\right]$$

$$= n^2 \left[\frac{1}{1 - 1/2}\right] \qquad \left[\because \frac{a}{a-r} \text{ in GP}\right]$$

$= 2n^2$

$$\boxed{T(n) = O(n^2)}$$

Ex:-2 $T(n) = T(n/4) + T(n/2) + n^2$



cost of level = addition of nodes present in the level

Total cost = addition of all levels.

Total time = $n^2 + \frac{5n^2}{16} + \frac{25}{256}n^2 + \cdots$

$= n^2 + \left(\frac{5}{16}\right)$

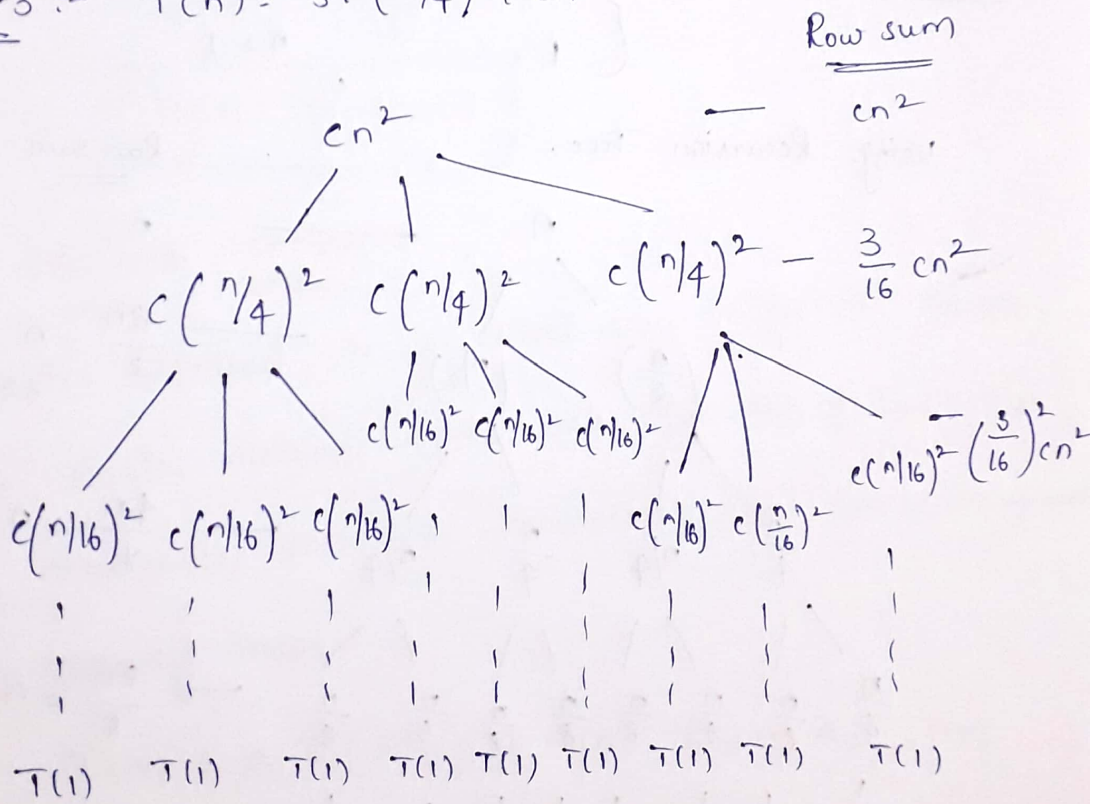
$= n^2\left[1 + \left(\frac{5}{16}\right)^1 + \left(\frac{5}{16}\right)^2 + \cdots\right]$

$= n^2\left[\frac{1}{1 - 5/16}\right]$ $\left[\because \frac{a}{a-r} \text{ in GP}\right]$

$= n^2$ [constant]

$\boxed{T(n) = O(n^2)}$

Ex:-3 :- $T(n) = 3T(n/4) + cn^2$



$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots$$

$$= cn^2\left[1 + \left(\frac{3}{16}\right)^1 + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \cdots\right]$$

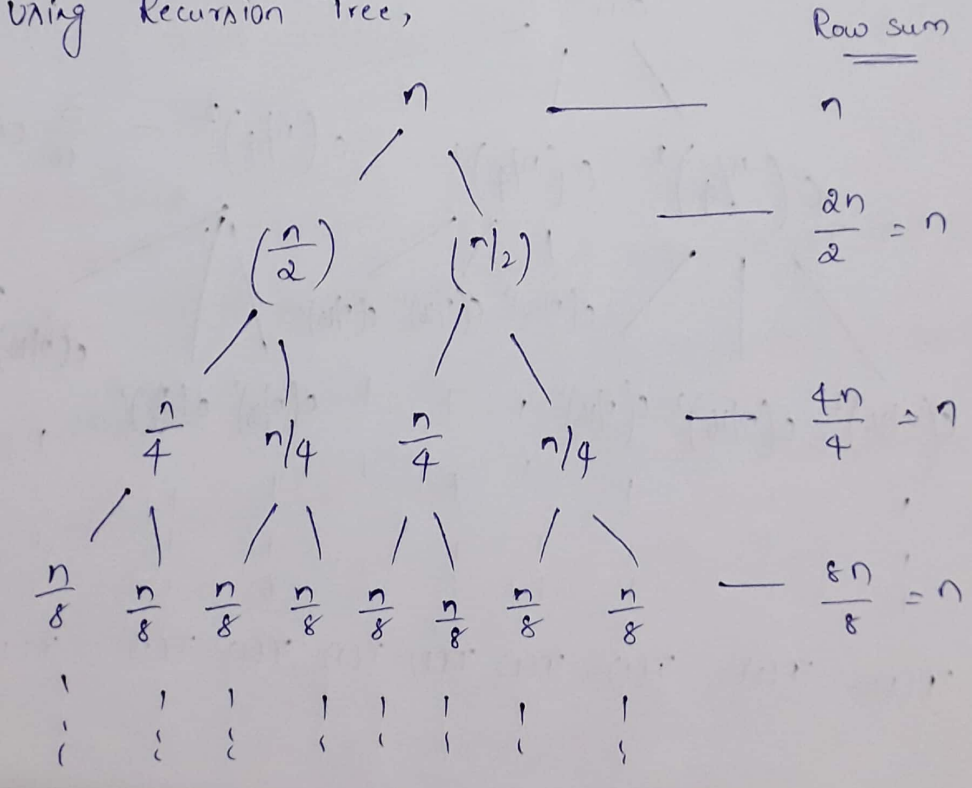
$$= cn^2\left[\frac{1}{1 - 3/16}\right] \Rightarrow cn^2\left[\frac{1}{\frac{16-3}{16}}\right]$$

$$= cn^2\left[\text{constant}\right] \Rightarrow cn^2\left[\frac{16}{13}\right]$$

$$\boxed{T(n) = O(n^2)}$$

Ex:- 4:- $T(n) = \begin{cases} 2T(n/2) + n, & n > 1 \\ 1, & n = 1 \end{cases}$

Using Recursion Tree,



Now, Each and Every level overall work done is 'n'.

Total work done is n x Total no. of levels.

No. of levels => $\frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \cdots \frac{n}{2^k}$

=> K+1 levels

Let $n = 2^K$

$K = \log n$

Total cost = $n \times (K+1)$

$= n(\log n + 1)$

$= n\log n + n$

$T(n) = O(n \log n)$

3. Master Method :-

Master Method is a direct way to get the Solution.

The Master method works only for the following type of recurrences (or) for recurrences that can be transformed to following type.

$T(n) = a\,T(n/b) + f(n)$ where $a \geq 1, b > 1$

$K \geq 0$ and P is real number.

$f(n) = \Theta(n^K \log^P n)$

case 1 : if $a > b^K$ then $T(n) = \theta(n^{\log_b^a})$

case 2 : if $a = b^K$

a) if $p > -1$ then $T(n) = \theta(n^{\log_b^a} \log^{p+1} n)$

b) if $p = -1$ then $T(n) = \theta(n^{\log_b^a} \log\log n)$

c) if $p < -1$ then $T(n) = \theta(n^{\log_b^a})$

case 3 : if $a < b^K$

a) if $p \geq 0$ then $T(n) = \theta(n^K \log^p n)$

b) if $p < 0$ then $T(n) = \theta(n^K)$

Ex:-1 $T(n) = 3T(n/2) + n^2$

Here, $a = 3$, $b = 2$, $K = 2$, $p = 0$

we compare a and b^K

3 and 2^2 i.e., $a < b^K$

$T(n) = \theta(n^K \log^p n)$

$= \theta(n^2 \log^0 n)$

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

Ex:-2 $T(n) = 4T(n/2) + n^2$

Here, $a = 4,\ b = 2,\ K = 2,\ P = 0$

compare a and b^K

4 and 2^2 i.e., $a = b^K$

Now check P value.

$\therefore P > -1$ then $T(n) = \theta(n^{\log_b^a} \log^{P+1} n)$

$T(n) = \theta(n^{\log_2^4} \log^1 n)$

$T(n) = \theta(n^2 \log n)$

Ex:-3 $T(n) = T(n/2) + n^2$

Here $a = 1,\ b = 2,\ K = 2,\ P = 0$

check a and b^K,

1 and 2^2 i.e., $1 < b^1$

$(\because a < b^K)$

$T(n) = \theta(n^K \log^P n)$

$= \theta(n^2 \log^0 n)$

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

Ex: 4 $T(n) = 16T(n/4) + n$

Here, $a = 16$, $b = 4$, $k = 1$, $p = 0$

check a and b^k, 16 and 4^1

$\therefore 16 > 4 \quad (a > b^k)$

$T(n) = \theta(n^{\log_b^a})$

$= \theta(n^{\log_4^{16}})$

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

Ex:- 5 $T(n) = 2T(n/2) + n \log n$

Here, $a = 2$, $b = 2$, $k = 1$, $p = 1$

check a and b^k, 2 and 2^1 $[\because a = b^k]$

check the value of p, $[\because p > -1.]$

$T(n) = \theta(n^{\log_b^a} \log^{p+1} n)$

$T(n) = \theta(n^{\log_2^2} \log^2 n)$

$\boxed{T(n) = \theta(n \log^2 n)}$

Ex:- 6 $T(n) = 2T(n/2) + \frac{n}{\log n}$

This can be written as, $T(n) = 2T(n/2) + n \log^{-1} n$

Here, $a = 2$, $b = 2$, $k = 1$, $p = -1$

Compare a and b^K values,

2 and 2^1 $\Rightarrow$ $2 = 2$ $[\because a = b^K]$

check the value of P, $P = -1$ $(\because P = -1)$

$$T(n) = \theta(n^{\log_b^a} \log\log n)$$

$$= \theta(n^{\log_2^2} \log\log n)$$

$$\boxed{T(n) = \theta(n \log\log n)}$$

Ex:- 7 $T(n) = 2T(n/4) + n^{0.51}$

Here $a = 2$, $b = 4$, $K = 0.51$, $P = 0$

compare a and b^K values,

2 and $4^{0.5}$ $\Rightarrow$ $2 < 4^{0.5}$ $(\because a < b^K)$

$$T(n) = \theta(n^K \log^P n)$$

$$= \theta(n^{0.51} \log^0 n)$$

$$\boxed{T(n) = \theta(n^{0.51})}$$

Ex:- 8 $T(n) = 6T(n/3) + n^2 \log n$

Here, $a = 6$, $b = 3$, $K = 2$, $P = 1$

compare a and b^K values, 6 and 3^2 $\Rightarrow 6 < 9$

$[\because a < b^K]$

Check the value of 'P'. i.e; P = 1 [$\because P >= 0$]

$$T(n) = \theta(n^K \log^P n)$$

$$= \theta(n^2 \log^1 n)$$

$$\boxed{T(n) = \theta(n^2 \log n)}$$

Ex :- 9 $T(n) = 7T(n/3) + n^2$

Here $a = 7$, $b = 3$, $K = 2$, $P = 0$

compare a and b^K,

7 and 3^2 $\Rightarrow$ $7 < 3^2$ [$\because a < b^K$]

Now, p value $\Rightarrow P = 0$ [$\because P >= 0$]

$$T(n) = \theta(n^K \log^P n)$$

$$= \theta(n^2 \log^0 n)$$

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

Ex :- 10 :- $T(n) = 4T(n/2) + \log n$

Here $a = 4$, $b = 2$, $K = 0$, $P = 1$

check a and b^K values, 4 and 2^0

$4 > 2^0$ [$\because a > b^K$]

$$T(n) = \theta(n^{\log_b^a})$$

$$= \theta(n^{\log_2^4})$$

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

Ex:- 11 $T(n) = \sqrt{2}\, T(n/2) + \log n$.

Here, $a = \sqrt{2}$, $b = 2$, $k = 0$, $P = 1$

$$[\because a > b^k]$$

$$T(n) = \theta(n^{\log_b^a})$$

$$= \theta(n^{\log_2^{\sqrt{2}}})$$

$$\boxed{T(n) = \theta(\sqrt{n})}$$

Ex:- 12 $T(n) = 2\, T(n/2) + \sqrt{n}$

Here, $a = 2$, $b = 2$, $k = 1/2$, $P = 0$

compare a and b^k values, 2 and $2^{1/2}$

$$\boxed{\because a > b^k}$$

$$T(n) = \theta(n^{\log_2^2})$$

$$\boxed{T(n) = \theta(n)}$$

Ex:- 13 $T(n) = 3T(n/2) + n$

Here, $a = 3$, $b = 2$, $K = 1$, $P = 0$

compare a and b^K values,

3 and 2^1 $\Rightarrow$ $3 > 2$ $[\because a > b^K]$

$T(n) = \theta(n^{\log_b^a})$

$\boxed{T(n) = \theta(n^{\log_2^3})}$

Ex:- 14 $T(n) = 3T(n/3) + \sqrt{n}$

Here, $a = 3$, $b = 3$, $K = 1/2$, $P = 0$

compare a and b^K values,

3 and $3^{1/2}$ $\Rightarrow$ $3 > 3^{1/2}$ $[\because a > b^K]$

$T(n) = \theta(n^{\log_b^a})$

$= \theta(n^{\log_3^3})$.

$\boxed{T(n) = \theta(n)}$

Ex:- 15 $T(n) = 4T(n/2) + cn$

Here, $a = 4$, $b = 2$, $K = 1$, $P = 0$

compare a and b^K values,

4 and 2^1 $\Rightarrow$ $4 > 2$ $[\because a > b^K]$

$T(n) = \theta(n^{\log_b^a})$

$T(n) = \theta(n^{\log_2^4}) \Rightarrow \theta(n^2)$ //

Ex:-16 $T(n) = 3T(n/4) + n \log n$

Here, $a = 3$, $b = 4$, $K = 1$, $p = 1$

compare a and b^K values,

3 and 4 $\Rightarrow 3 < 4$ $[\because a < b^K, p \geq 0]$

$T(n) = \theta(n^K \log^p n)$

$T(n) = \theta(n^1 \log^1 n)$

$\boxed{T(n) = \theta(n \log n)}$

Ex:-17 $T(n) = 64\, T(n/8) - n^2 \log n$

This is not applicable for masters theorm due to '-'.

Ex:-18 $T(n) = 0.5\, T(n/2) + 1/n$

Here $a = 0.5$

According to theorem $a \geq 1$ so, it is not applicable for masters theorem.

Ex:-19 $T(n) = 2^n\, T(n/2) + n^n$

According to theorem, a value must be some constant. Here 2^n so, not applicable for master theorem.

Example :- $T(n) = \begin{cases} 3T(n-1), & n>0 \\ 1, & n=0 \end{cases}$ find Time complexity ?

Sol :- $T(n) = 3T(n-1) \quad -(1)$ $\therefore T(n-1) = 3T(n-1-1)$

$= 3T(n-2)$

$T(n) = 3 \cdot 3 \cdot T(n-2)$

$= 3^2 T(n-2) \quad -(2)$ $\therefore T(n-2) = 3T(n-2-1)$

$= 3T(n-3)$

$T(n) = 3^2 \cdot 3T(n-3)$

$= 3^3 T(n-3) \quad -(3)$

After K-steps

$T(n) = 3^K T(n-K) \quad -(4)$

Let $n-K = 0$

$n = K$

Substitute 'n' in the place of 'K'

$T(n) = 3^n T(n-n)$

$= 3^n T(0)$

$= 3^n$

$\boxed{T(n) = O(3^n)}$