Design & Analysis of Algorithm:- How to Design various Algo. (DAC, Greedy, Dynamic Branch and bound, Backtracking etc.) & How to Analysis Algo (Time & Space Complexity)

Why Study this Subject:- to minimize time & System resources (memory)

- Efficient Algorithm lead to efficient Programs.
- Efficient Program sell better.
- Efficient Program make better use of hardware.

NOTE:- Data structure is the Concept of
of Algo. used to structure the

Algorithm:- An algorithm is a finite step-step Procedure for solving Computational Problem

Ex:- Algo: swap(a, b)

```
{  temp = a
   a = b
   b = temp
}
```

Properties of Algo:-

- Input :- it should take zero or more input
- Output :- it Should Produce atleast one output
- Finiteness :- it Should terminate within finite time
- Definitness :- Every step in the Algo. Should be deterministic (Unambiguous)
- Effectiveness :- Every step in the Algo. Should Perform Something (don't write Unnecessary st
- It Should be Programming independent.

How to Solve Problem by Computer:-

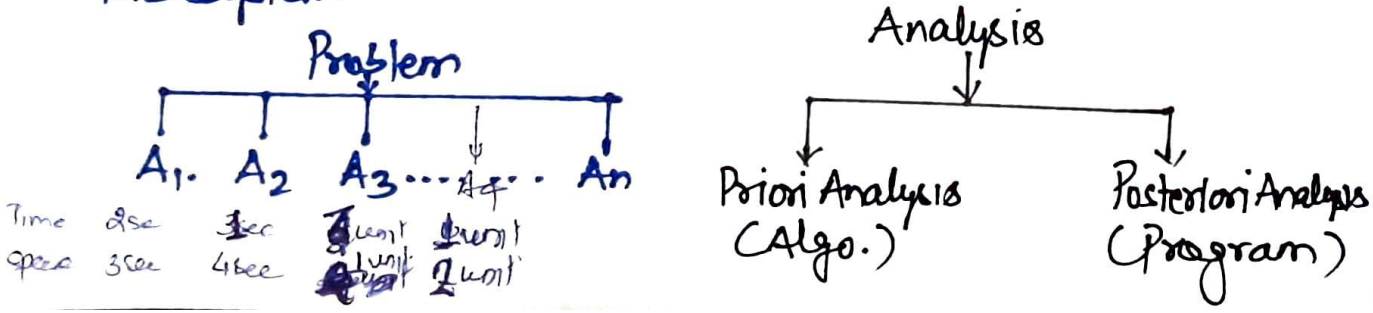
1. Problem definition
2. Design Algo. (May be DAC, Greedy, Dynamic etc.)
3. Construct flowchart
4. Validation
5. Implementation (coding)
6. Analysis (Testing) of Algo:-

Analysis is done on the basis of time & space, Time is more important than space, if time of Two Algo. are equal then only go for check space.

Time $\propto \frac{1}{\text{space}}$.

Purpose of Analysis: To check Best solution for a Problem.

- To check how much time algo. is taking to complete its execution
- To check how much space algo. is taking in The System



Difference b/w Algo. & Program:- (2)

Algorithm	Program
Design time	Implementation time
Domain Knowledge	Programmer
Any language	Programming language
Independent of S/w & H/w	dependent of S/w & H/w
Analysis	Testing

Priori Analysis	Posteriori Testing
for Algorithm	for Program
Independent of Language	Depends on Language
S/w & H/w Independent	S/w & H/w Dependent
Time & space function	Watch Time
Answer will be same in all System Ex:. $f(n) = 2n+3$ $f(n) = 5n^2+n+10$ $\rightarrow O(n)$	Answer change from System to System

Algorithm classification:-

- Divide & Conquer (DAC):- Quick Sort, Mergesort, Binary Search, Finding maxima/minima, selection Procedure Strassen's matrix multiplication etc.
- Greedy :- Knapsack Problem, Activity selection, Task scheduling, Huffman coding, optimal merge Pattern, Single Source Shortest path (DijKastra Algo.), Minimum spanning tree (Prim Algo, Kruskals Algo) etc.
- Dynamic Programming :- Matrix chain multiplication, LCS, 0/1 Knapsack, Sum of Subset, Travelling Salesman, Single Source Shortest path (Bellman ford Algo.), All pair shortest path (Floyd Washall, Transitive closure)
- Branch & Bound :- Graph Coloring, DFS etc.
- Backtracking Algo:- TSP, Knapsack Problem etc.
- Randomized Algo:- Randomized Quick Sort etc.
- Recursive Algo:- Binary Search, Fibonacci series etc.

Analysis of Algo:- depends on two factors

1- Time:- How much time it will take during execution. This Calculation will be Independent of Implementation details and Programming Language.

2- Space:- How much space it will take during execution (RAM space)

Ex:- Algo: Swap(a,b)

```
{ temp := a;
  a := b;
  b := temp;
}
```

Time		space	
	1	a → 1	
	1	b → 1	
	1	temp → 1	
f(n) = 3		S(n) = 3 words	
f(n) = O(1),		S(n) = O(1)	

Ex:- Algo: Sum(A,n)

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

Time		space	
	1	A[] → n	
	n+1	n → 1	
	n	S → 1	
	1	i → 1	
f(n) = 2n+3		S(n) = n+3	
= O(n)		= O(n)	

Asymptotic notation:- Asymptotic notations are mathematical tools to represent time & space complexity of algorithm for asymptotic analysis.

O-Bigoh :- Asymptotic upper Bound (it may be closed or not closed to actual value)

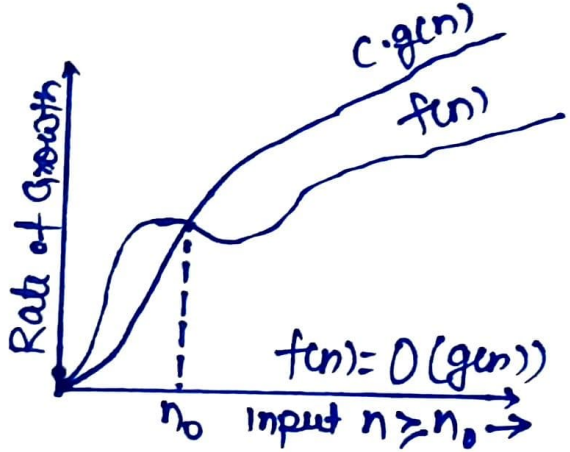
Ω-Bigomega :- Asymptotic Lower Bound (it may be closed or not closed to actual value)

θ-Theta :- Average bound (tightly closed). It is Practical useful notation.

o-Smalloh :- only upper Bound

ω-Smallomega :- only lower Bound

O-Bigoh: $f(n) = O(g(n))$ if there exist Positive constant c and n_0 such that $f(n) \leq c \cdot g(n)$, $\forall n \geq n_0$



Ex:- $f(n) = 3n+2$ OR $3n+2 \leq 3n^2+2n^2$

$= 3n+2 \leq 3n+2n$ $\quad 3n+2 \leq 5n^2, n \geq 1$

$= 3n+2 \leq 5n, n \geq 1$ $\quad f(n) \leq c \cdot g(n)$

$f(n) \leq c \cdot g(n), n \geq 1$ $\quad f(n) = O(n^2)$

$f(n) = O(n)$

$f(n) = 3n+2$

$= O(n)$ ✓

$= O(n^2)$ ✓ } TRUE

$= O(n^3)$ ✓

$\neq O(1)$

practically useful because closed value

Ex:- $f(n) = 3n^2+2n+5$, $f(n) = O(n^2)$ } practically useful. $\neq O(n)$

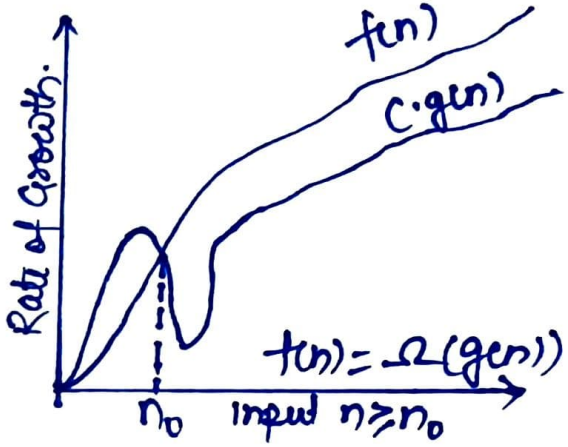
$= O(n^3)$ } TRUE $\neq O(1)$

$= O(n^4)$ TRUE

Ω-Bigomega:- $f(n) = \Omega(g(n))$ if There exist Positive Constant C and n_0 such that $f(n) \geq C \cdot g(n)$, $\forall n \geq n_0$

Ex:- $f(n) = 3n+2 \geq 3n$ → Practically useful b/c closest to Actual value.
$= \Omega(n)$ } TRUE
$= \Omega(1)$ }
$\neq \Omega(n^2)$

Ex:- $f(n) = 2n^2+3n+5$
$= \Omega(n^2)$ → Practically useful b/c closed to Actual value
$= \Omega(n)$ } TRUE
$= \Omega(1)$ }
$\neq \Omega(n3)$



Θ-Theta:- $f(n) = \Theta(g(n))$ if There exist Positive Constant C_1, C_2 and n_0 such that $C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$, $\forall n \geq n_0$

Ex:- $f(n) = 3n+2$, $3n \leq 3n+2 \leq 5n$
$= \Theta(n)$ ✓ TRUE Tightly closed value.
$\neq \Theta(n^2)$
$\neq \Theta(1)$

, $f(n) = 2n^2+3n+5$
$= \Theta(n^2)$ ✓
$\neq \Theta(n)$
$\neq \Theta(n^3)$

O-Smalloh:- $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Ex = $3n+2$, $f(n) = o(n^2)$, $\neq o(n)$

ω-Smallomega:- $f(n) = \omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Ex: $f(n) = 3n+2$, $f(n) = \omega(1)$, $\neq \omega(n)$

NOTE:- When WC = BC ⇒ Then use Θ-notation
Can not relate asymptotic notation with BC, WC and Ae of an Algo.

NOTE:-
$a \leq b \Rightarrow a = O(b)$
$a \geq b \Rightarrow a = \Omega(b)$
$a = b \Rightarrow a = \Theta(b)$
$a < b \Rightarrow a = o(b)$
$a > b \Rightarrow a = \omega(b)$

NOTE:-
In general:-
BC ≤ Avg ≤ WC
$B(n) \leq A(n) \leq W(n)$
$B(n) = O(A(n))$
$A(n) = O(W(n))$
$B(n) = O(W(n))$
$A(n) = \Omega(B(n))$
$W(n) = \Omega(A(n))$

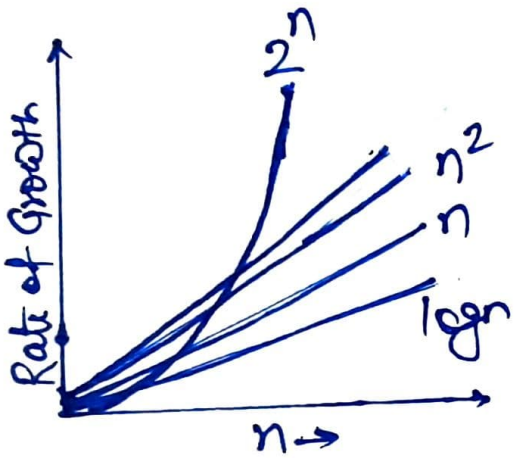
Conclusion of Asymptotic Notation

Types of time function:-

function	Rate of Growth
Constant	$O(1)$
Logarithm	$O(\log n)$
Root	$O(\sqrt{n})$
Linear	$O(n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Polynomial	$O(n^k), k>0$
Exponential	$O(c^n), c>1$

Compare class of function:

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



$\log n$	n	n^2	2^n
0	1	1	2
1	2	4	4
2	4	16	16
3	8	64	256
3.1	9	81	512

2^n will be moving faster speed at after Certain Point (Growth rate)

How to Compare function:-

1) $n^2 < n^3$

2) $2n \overset{v}{=} 3n$

3) $2^n > n^2$

4) $f(n) = 2^n$, $g(n) = 2^{2n}$

After log $n\log 2 < 2n \log 2$

$n < 2n$

don't say equal b/c This Condition generate after log

5) $n > (\log n)^{100}$

Sol^n Take log Both side

$\log n \quad 100 \log\log n$

Put n = large value & check

let $n = 2^{2^{10}}$

$\log 2^{2^{2^{10}}} \quad 100 \log\log 2^{2^{2^{10}}}$

$2^{2^{10}} > 100 \times 2^{10}$

proof $n > (\log n)^{100}$

6) $n^{\log n} > n \log n$

Proof:

7) $f(n) = n^2 \log n$, $g(n) = n(\log n)^{10}$

Sol^n: Take log Both side.

$\log(n^2 \log n)$ $\quad$ $\log(n(\log n)^{10})$

$2\log n + \log\log n$ $\quad$ $\log n + 10 \log\log n$

like 2 Rupees + 1 paisa > 1 Rupees + 10 paisa

$f(n) > g(n)$

$f(n) = \Omega(g(n))$
$g(n) = O(f(n))$ } TRUE

$f(n) = O(g(n))$
$g(n) = \Omega(f(n))$
$f(n) = \Theta(g(n))$
$g(n) = \Theta(f(n))$ } False. b/c $f(n)$ is upperbound of $g(n)$ function

8) $n^{\sqrt{n}}$ $\quad$ $n^{\log n}$

9) $f(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^2 & n \geq 10000 \end{cases}$

$g(n) = \begin{cases} n & 0 < n < 100 \\ n^3 & n > 100 \end{cases}$

TRUE:

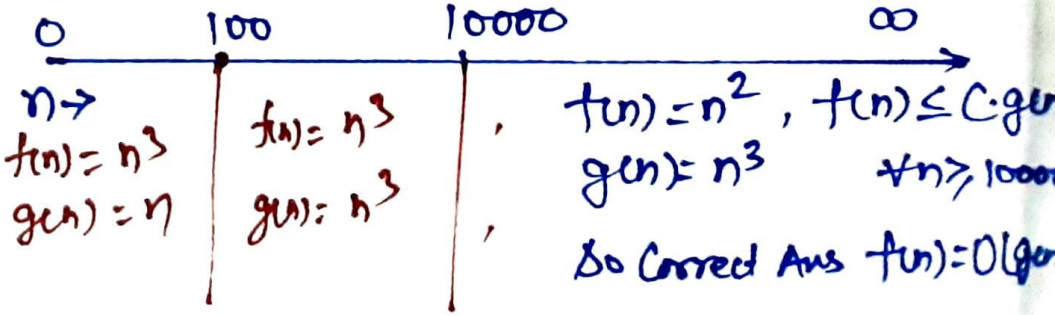
a) $f(n) = O(g(n))$

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

c) $f(n) = \Theta(g(n))$

d) None.

Sol^n:-



TRUE:

1) $2^{2n} = \theta(2^n)$ False

2) $\sqrt{\log n} = O(\log\log n)$ False

3) $n^2 \cdot 2^{3\log_2 n} = \theta(n^5)$ True

4) $2^{\log_2 n} = O(n^2)$

5) $2^{\log_2 n} = n^{\log 2} = \theta(n) = O(n^2)$
$\neq \theta(n^2)$

6) $f(n) = n^{\sqrt{n}}$, $g(n) = 3.2^{\sqrt{n}\log_2 n}$, $h(n) = n!$

~~$g(n) = 3.\sqrt{n}^{\sqrt{n}\log_2 2}$~~ $g(n) = 3.n^{\sqrt{n}}$

So $f(n) \cong g(n) \Rightarrow f(n) = \theta(g(n))$

But $h(n)$ is largest function

$f(n) \cong g(n) < O(h(n))$

factorial function: not tightly closed

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

$f(n) = n! = n \times (n-1) \times (n-2) \times \cdots 2 \times 1$

$1 \times 1 \times 1 \times \cdots \leq 1 \times 2 \times 3 \times 4 \cdots \times n \leq n \times n \times n \times \cdots n$

$1 \leq n! \leq n^n$

$\Omega(1) \leq n! \leq O(n^n)$

for small value of n nearer to 1 for
Larger value of n nearer to n^n
We can not fix the particular place for n!
That is why it is not tightly closed

Ex: $f(n) = \log n!$

$1 \leq \log n! \leq \log n^n$

$\Omega(1) \leq \log n! < O(n\log n)$

Properties of Asymptotic Notation

1) Reflexive: O, Ω, Θ notation satisfied Reflexive.
   - f(n) = O(f(n)) } Satisfied
   - f(n) = Ω(f(n)) } Satisfied
   - f(n) = Θ(f(n)) } Satisfied
   - f(n) ≠ o(f(n)) } not satisfied
   - f(n) ≠ ω(f(n)) } not satisfied Reflexive

2) Symmetric: only Θ notation satisfied
   - If f(n) = O(g(n)) ⇒ g(n) ≠ O(f(n))
   - If f(n) = Ω(g(n)) ⇒ g(n) ≠ Ω(f(n))
   - If f(n) = Θ(g(n)) ⇒ g(n) = Θ(f(n)) Satisfied
   - If f(n) = o(g(n)) ⇒ g(n) ≠ o(f(n))
   - If f(n) = ω(g(n)) ⇒ g(n) ≠ ω(f(n))

3) Transitive: All notation satisfied Transitive Property
   f(n) = O(g(n)) && g(n) = O(h(n)) ⇒ f(n) = O(h(n))

4) Transpose:
   - f(n) = O(g(n)) iff g(n) = Ω(f(n))
   - f(n) = Ω(g(n)) iff g(n) = O(f(n))
   - f(n) = Θ(g(n)) iff g(n) = Θ(f(n))

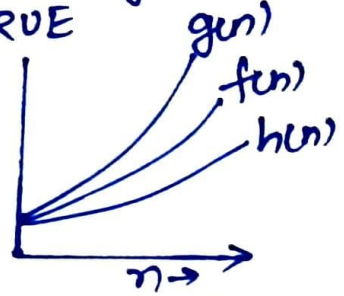
   Similarly o and ω satisfied Transpose Property

General Properties:-

1) If f(n) = Θ(g(n)) Then a * f(n) = Θ(g(n))
   Similarly for O, Ω satisfied
   Ex: f(n) = 2n² + 5, 7 * f(n) = 14n² + 35 ⇒ O(n²)
2) If f(n) = Θ(g(n)) and f(n) = Ω(g(n)) Then f(n) = Θ(g(n))
3) If f(n) = O(g(n)), d(n) = O(e(n))
   - f(n) + d(n) = O(max(g(n), e(n)))
   - f(n) * d(n) = O(g(n) * e(n))

Ex: Consider the following f(n) = 2^n, g(n) = n!, h(n) = n^log n
Which of the following statements about the Asymptotic behaviour of f(n), g(n) and h(n) is TRUE

Sol^n:



a) f(n) = O(g(n)), g(n) = O(h(n))
b) f(n) = Ω(g(n)), g(n) = O(h(n))
c) g(n) = O(f(n)), h(n) = O(f(n))
d) h(n) = O(f(n)), g(n) = Ω(f(n))

Ex: Let f(n), g(n) and h(n) be function defined for Positive integer such that

f(n) = O(g(n))
g(n) ≠ O(f(n))
g(n) = O(h(n))
h(n) = O(g(n))

a) f(n) + g(n) = O(h(n) + h(n))
b) f(n) = O(h(n))
c) h(n) ≠ O(f(n))
d) f(n) · h(n) ≠ O(g(n) · h(n))

Which of the following is FALSE

Recursion:- A function call itself.

Ex:-
```
main()
{ int x,f;
f= fact(x);
  Printf("factorial= %d",f);
}
fact(n)
{ int n;
  if (n<=1)
    return 1;
  else
    factorial = n * fact(n-1);
    return factorial;
}
```

Properties of Recursion:-

1. It should have termination condition.
2. It should have recursive state
3. Parameter value should be change from one function call to another function.

Draw back of Recursion:

- Take more space (STACK space)

Advantage of Recursion:

- Easy to understand (b/c reduce line of code)

Recurrence Relation: it is a relation in which nth term of the sequence will be represented in terms of its previous terms

Ex:- $$F(n) = \begin{cases} 1 & , \text{if } n \le 1 \\ n * F(n-1) & , \text{if } n > 1 \end{cases}$$

Ex:- find recurrence relation for multiplying two no. a & b

$$Mul(a,b) = \begin{cases} 0 & \text{if } b=0 \\ a + Mul(a,b-1), & \text{if } b \ge 1 \end{cases}$$

Ex:- Find recurrence relation for finding power of a. (i.e a^b)

$$pow(a,b) = \begin{cases} 1 & , \text{if } b=0 \\ a * pow(a,b-1), & \text{if } b>0 \end{cases}$$

Ex:- find recurrence relation for GCD(m,n)

Soln:

$$GCD(m,n) = \begin{cases} 0 & \text{, if } m=0 \text{ \& } n=0 \\ m & \text{, if } n=0 \\ n & \text{, if } m=0 \\ GCD(m\%n, n), & \text{otherwise} \end{cases}$$

if $m>n$

$\#TC = O(\log m)$ if $m>n$
$= O(\log n)$ if $m<n$
} Worst case (if m & n are prime)
$= O(1)$ Best case.

Ex:- Find recurrence relation for fibonacci series.

Fibonacci series:

n:	0	1	2	3	4	5	6	7	8	9	...
fib(n):	0	1	1	2	3	5	8	13	21	34	...

Recurrence Relation:

$$fib(n) = \begin{cases} n & \text{if } n=0 \text{ or } n=1 \\ fib(n-1)+fib(n-2), & \text{if } n>1 \end{cases}$$

Ex:- GCD(24,23)

↳ GCD(6,23)
↳ GCD(5,6)
↳ GCD(1,5)
↳ GCD(0,1)
=1

Algo:-
```
GCD(m,n)
{ if (m==0 && n==0)
    return(-1);
  else if (m==0) return n;
  else if (n==0) return m;
  else return (GCD(m%n,n));
}
```

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

Recurrence relation:- When an Algo. Contains a recursive call to itself it running time can be describe by a recurrence. A recurrence is an equation that describes a function in terms of its value on smaller input

Ex:. for decreasing function $T(n) = T(n-1)+1$
for Divide function $T(n) = T(n/2)+1$
for Root function $T(n) = T(\sqrt{n})+1$

How to Construct Recurrence Relation:-

Decreasing function:-

1) Algo A(n) — T(n)
{ if(n>0) — 1
{ pf(n); — 1
A(n-1); — T(n-1)
}
}
T(n-1)+2

Recurrence relation:
$$T(n) = \begin{cases} 1, & \text{if } n=0 \\ T(n-1)+1, & \text{if } n>0 \end{cases}$$

2) Algo: A(n) — T(n)
{ if(n>0) — T(1)
{ for(i=0; i<n; i++) — n+1
pf(n); — n
A(n-1); — T(n-1)
}
}
T(n-1)+n

Recurrence Relation
$$T(n) = \begin{cases} 1, & \text{if } n=0 \\ T(n-1)+n, & \text{if } n>0 \end{cases}$$

3) Algo. A(n) — T(n)
{ if(n>0) — 1
{ for(i=1; i<n; i=i+2) — logn
pf(i); — logn
A(n-1); — T(n-1)
}
}
T(n-1)+logn

Recurrence Relation:
$$T(n) = \begin{cases} 1, & \text{if } n=0 \\ T(n-1)+\log n, & \text{if } n>0 \end{cases}$$

4) Algo: A(n) — T(n)
{ if(n>0) — 1
{ pf(n); — 1
A(n-1); — T(n-1)
A(n-1); — T(n-1)
}
}
2T(n-1)+1

Recurrence Relation:
$$T(n) = \begin{cases} 1, & \text{if } n=0 \\ 2T(n-1)+1, & \text{if } n>0 \end{cases}$$

Division function:-

5) Algo: A(n)
{ if(n>1)
{ pf(n)
A(n/2)
}
}

Recurrence Relation:
$$T(n) = \begin{cases} 1, & \text{if } n=1 \\ T(n/2)+1, & \text{if } n>1 \end{cases}$$

6) Algo: A(n)
{ if(n>1)
{ for(i=0; i<n; i++)
stmt;
A(n/2);
A(n/2);
}
}

Recurrence Relation:
$$T(n) = \begin{cases} 1, & \text{if } n=1 \\ 2T(n/2)+n, & \text{if } n>1 \end{cases}$$

7) Algo: A(n)
{ if(n≤1)
return 1
else
return (2*A(n/2)+6*A(n/2)+n²)
}
T(n/2) T(n/2)

Recurrence Relation:
$$T(n) = \begin{cases} 1, & \text{if } n=1 \\ 2T(n/2)+1, & \text{if } n>1 \end{cases}$$

Solution of Recurrence Relation:

1) Iteration Method
2) Recursion Tree Method
3) Master Method

Iteration Method :- It means to expand the recurrence and express it as a summation of terms of n and initial Condition.

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

Sol^n: $T(n) = 2T(n-1)$

$= 2[2T(n-2)] = 2^2 T(n-2)$

$= 4[2T(n-3)] = 2^3 T(n-3)$

$= 8[2T(n-4)] = 2^4 T(n-4) \cdots (i)$

Repeat the Procedure for i times

$T(n) = 2^i T(n-i)$

put $n-i=1$ or $i=n-1$ in equation (i)

$T(n) = 2^{n-1} T(1)$

$= 2^{n-1} \times 1$

$= 2^{n-1}$

$= O(2^n)$

Ex:- $T(n) = T(n-1)+1$ and $T(1) = \Theta(1)$

Sol^n: $T(n) = T(n-1)+1$

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

$= T(n-3)+3$

$\vdots$

$T(n-k)+k$

put $n-k=1 \Rightarrow k=n-1$

$= T(1)+n-1$

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

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

Sol^n: $T(n) = 2[2T(n-2)+1]+1$

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

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

$\vdots$

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

$T(1)$ $\quad n-k=1 \Rightarrow k=n-1$

$= 2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + \cdots + 2^2+2^1+2^0$

$= \frac{2^n-1}{2-1} \Rightarrow 2^n-1 \Rightarrow O(2^n)$ Ans

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

Sol^n:- $T(n) = T(n-1)+\log n$

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

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

$\vdots$

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

$\Downarrow$

$T(0) \Rightarrow n-(k+1)=0 \Rightarrow k=n-1$

put k value in equation (i)

$= T(0)+\log(1)+\log 2+\log 3+\cdots+\log n$

$= 1+\log 1+\log 2+\log 3+\cdots+\log n$

$= 1+\log(1\cdot 2\cdot 3\cdot 4\cdot 5\cdots n)$

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

$= O(1)+O(n\log n)$

$= O(n\log n)$ Ans

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

$T(n) = T(n/4)+1+1$

$= T(n/8)+1+1+1$

$= T(n/2^3)+3$

$\vdots$

$= T(n/2^{k+1})+k+1$ ----(i)

$\Downarrow$

$T(1) \Rightarrow n/2^{k+1}=1 \Rightarrow k=\log_2 n-1$

Substitute k value in equation (i)

$= T(1)+\log_2 n-1 \Rightarrow 1+\log_2 n-1 = O(\log_2 n)$ Ans

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

$= T(n/2)+n$

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

$= T(n/8)+n/4+n/2+n$

$\vdots$

$= T(n/2^{k+1})+\frac{n}{2^k}+\frac{n}{2^{k-1}}+\cdots+\frac{n}{4}+n/2+n$

$\Downarrow$

$= T(1) \Rightarrow n/2^{k+1}=1 \Rightarrow k=\log_2 n-1$

$= T(1)+n\left[1+1/2+1/2^2+\cdots+\frac{1}{2^{k-1}}\right] \Rightarrow 1+n = O(n)$ Ans

Ex:- $T(n) = \begin{cases} 1 & , \text{if } n=1 \\ 2T(n/2)+n & , \text{if } n>1 \end{cases} \Rightarrow O(n\log n)$ Ans

Recursion Tree Method :-

1- Recursion Tree Method is a pictorial representation of an iteration Method which is in the form of a tree where at each level nodes are expanded.

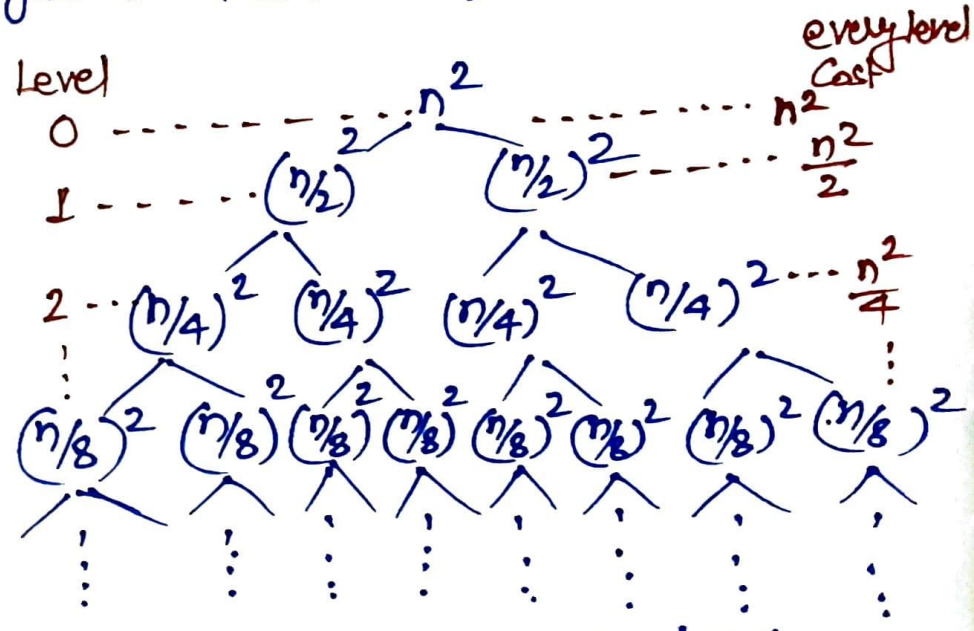
2- In general, we consider the second term in recurrence as root.

3- It is useful when the Divide & Conquer algorithm is used.

4- It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

5- We sum the costs within each of the levels of the Tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

6- A Recursion tree is best used to generate a good guess, which can be verified by the Substitution Method.



Total Cost = Sum of all level cost.

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

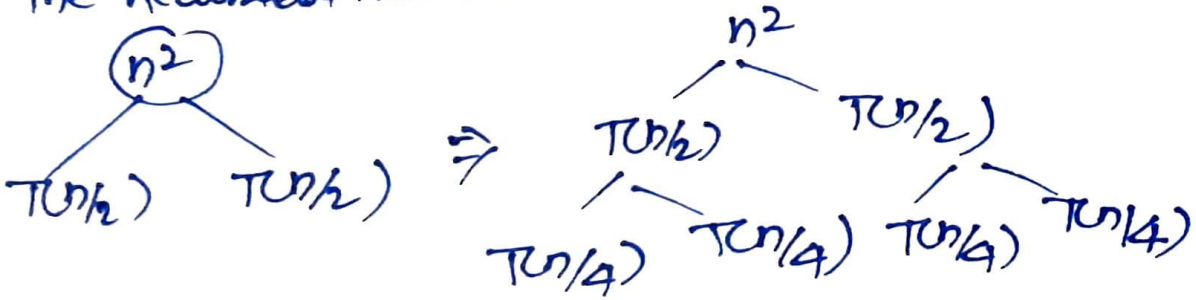
$= n^2 \left[1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right] + n$

$= C \cdot n^2 + n \quad \therefore = O(n^2)$

~~Leading term = n^2~~ Ans

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

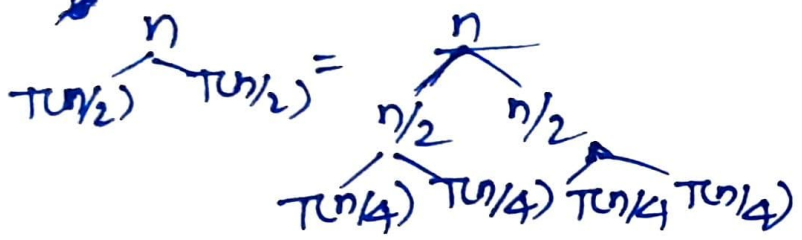
Sol^n:- The Recursion Tree for the above recurrence is

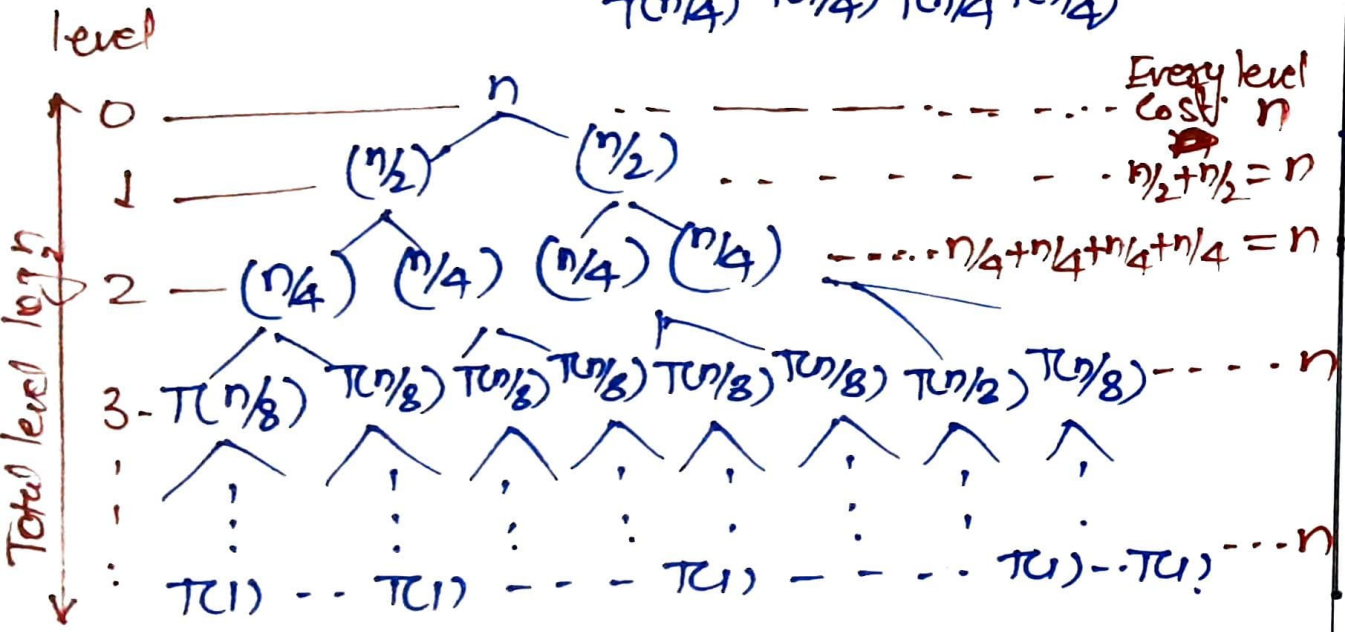


Ex:- Solve Recurrence Relation by Recursion tree Method

$$T(n) = \begin{cases} 1, & \text{if } n=1 \\ 2T(n/2)+n, & \text{if } n>1 \end{cases}$$

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





Total Cost = Sum of all level cost

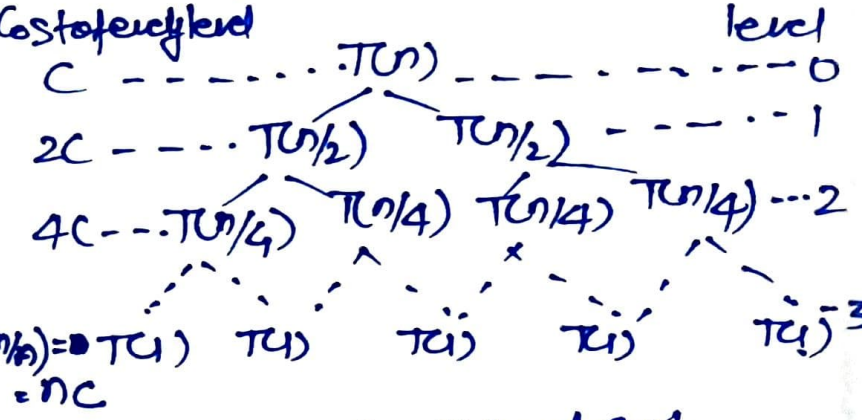
$= n+n+n+\cdots+\log n \text{ times}$

$= n \times \log n$

$= O(n \log n)$ Ans

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

Cost of each level



Total Cost = Sum of all level cost

$= c+2c+4c+\cdots+nc$

$= c[1+2+4+\cdots+2^k]$

let $n = 2^k$

$= c \times \left[\frac{2^{k+1}-1}{2-1}\right]$

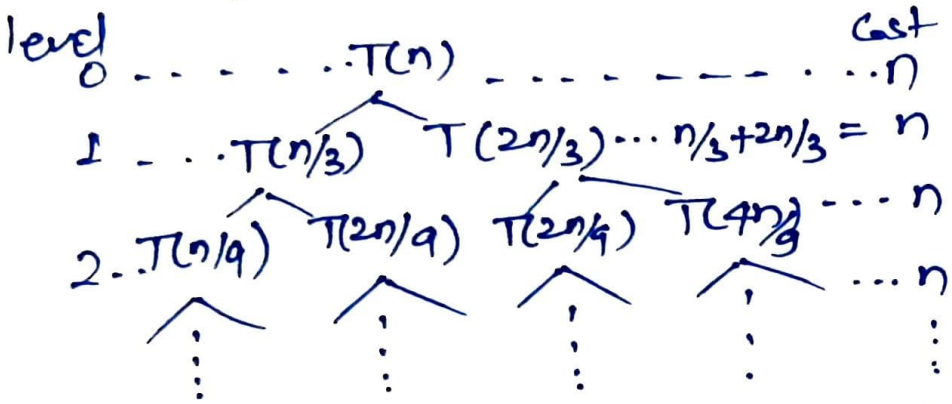
$= c \times (2 \cdot 2^k - 1)$

$= c[2n-1]$

$= 2cn - c$

$= O(n)$ Ans.

Ex:- $T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$



When we add the value across the levels of the recursion tree, we get a value of n for every level. The longest path from the root to leaf is

$n \to 2n/3 \to 4n/9 \to 1$

$T\left(\frac{n}{(3/2)^i}\right) = T(1) \Rightarrow \frac{n}{(3/2)^i} = 1 \Rightarrow i = \log_{3/2} n$

Thus the height of the tree is $\log_{3/2} n$

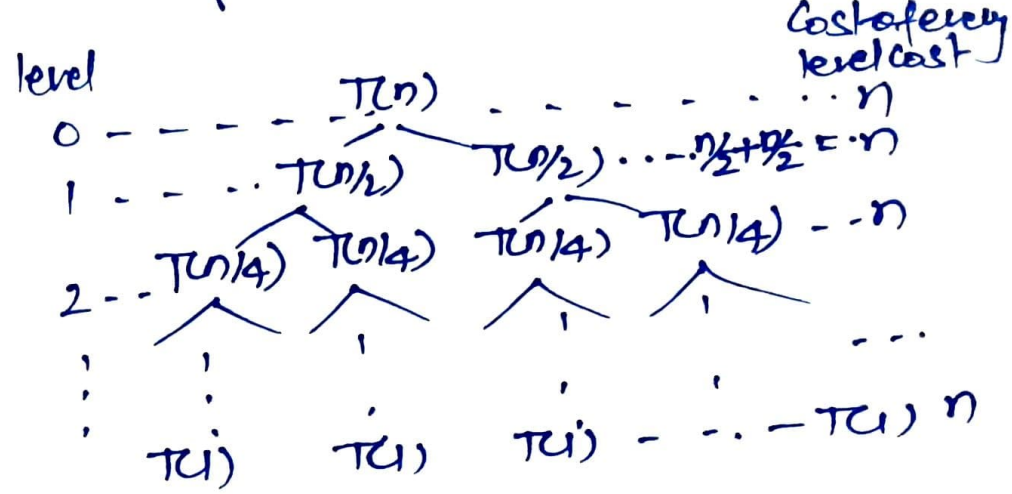
Total cost = Sum of all level cost

$= \underbrace{n+n+n+\cdots+n}_{\log_{3/2} n \text{ times}}$

$= n \log_{3/2} n = O(n \log n)$ Ans

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

(15)



Total cost = Sum of all level cost

$= \underbrace{n+n+n+\cdots+n}_{\log_2 n \text{ times}}$

$= n \times \log_2 n$

$= O(n \log_2 n)$

Ans

Master method for Decreasing function:-

$T(n) = aT(n-b) + f(n)$ where $a>0, b>0$ and $f(n) = O(n^k)$, $k \geq 0$

Case 1: if $a<1$, $O(f(n))$

Case 2: if $a=1$, $O(n * f(n))$

Case 3: if $a>1$, $O(f(n) \cdot a^{n/b})$

Ex: i) $T(n) = T(n-1) + 1$

$a=1, b=1, f(n)=1$

Case 2: TRUE

So Ans $O(n * f(n)) = O(n \times 1) = O(n)$

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

$a=1, b=1, f(n)=n$

Ans $= O(n * f(n)) = O(n * n) = O(n^2)$

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

4) $T(n) = T(n-1) + n^2 \Rightarrow O(n^3)$

5) $T(n) = T(n-2) + 1 \Rightarrow O(n/2) = O(n)$

6) $T(n) = T(n-100) + n \Rightarrow O(n^2/100) = O(n^2)$

7) $T(n) = 2T(n-1) + 1$, $a=2, b=1, f(n)=1$

Case 3: TRUE, Ans: $O(f(n) \cdot a^{n/b})$

$= O(1 \times 2^{n/1}) = O(2^n)$ Ans:

8) $T(n) = 3T(n-1) + 1 = O(3^n)$ Ans

9) $T(n) = 2T(n-1) + n = O(n \cdot 2^n)$ Ans

10) $T(n) = 3T(n-1) + n \Rightarrow O(n * 3^n)$ Ans.

Master method for Division :- $T(n) = aT(n/b) + \Theta(n^k \cdot \log^p n)$

where $a \geq 1, b>1, k \geq 0$ and p is real no.

Case 1: if $a > b^k$, Then $T(n) = \Theta(n^{\log_b a})$

Case 2: if $a = b^k$

i) If $p > -1$ Then $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$

ii) If $p = -1$ Then $T(n) = \Theta(n^{\log_b a} \cdot \log \log n)$

iii) If $p < -1$ Then $T(n) = \Theta(n^{\log_b a})$

Case 3: If $a < b^k$

i) If $p \geq 0$ Then $T(n) = \Theta(n^k \cdot \log^p n)$

ii) If $p < 0$ Then $T(n) = O(n^k)$

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

$\Rightarrow a=2, b=2, k=1, p=1$

Case 2(i) $T(n) = \Theta(n^{\log_2 2} \cdot \log^2 n) \Rightarrow \Theta(n \log^2 n)$

Ex:- $T(n) = 3T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Case 3(ii) $T(n) = \Theta(n^k \cdot \log^p n)$; $a=3, b=2$, $k=2, p=0$ $\hookrightarrow \Theta(n^k \cdot \log^p n)$)

Ex: $T(n) = 2^n T(n/2) + n^n$, not master required form

So master method can not be applied.

Ex:- $T(n) = 2T(n/2) + n/\log n = 2T(n/2) + n \log^{-1} n$ $a=2, b=2, k=1, p=-1$

$\hookrightarrow T(n) = \Theta(n^{\log_b a} \cdot \log \log n)$

Case 2(ii) $T(n) = \Theta(n \log \log n)$

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

$a = 0.5$, $\because a < 1$, not Apply master

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

↑ Master not applied

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

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

Case 1: $\theta(\sqrt{n})$ Ans $T(n) = \theta(n^{\log_b^a})$

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

$a = 2, b = 2, K = 1/2, P = 0$ Case 3: $\theta(n^{\log_b^a}) = \theta(n)$

Ex:- $T(n) = 3T(n/4) + n\log n$, $a = 3, b = 4, K = 1, P = 1$

Case 3(i) $\Rightarrow \theta(n\log n)$ $T(n) = \theta(n^K \cdot \log^P n)$

For Root function:-

$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$

put $n = 2^m$

$m = \log_2^n$

$T(2^m) = 2T(2^{m/2}) + m$

$S(m) = 2S(m/2) + m$

Case 2(i) $= \theta(m^{\log_2^2} \cdot \log^{0+1} m)$

$= \theta(m \log m)$

put m value $\Rightarrow \theta(\log n \cdot \log\log n)$

Ex:- $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + 1$, $T(1) = 1$

put $n = 2^m \Rightarrow m = \log_2^n$

$T(2^m) = 2T(2^{m/2}) + 1$

$S(m) = 2S(m/2) + 1$

where $a = 2, b = 2, K = 0, P = 0$

Case 1: Then $T(n) = \theta(m^{\log_b^a}) = \theta(m^{\log_2^2}) = \theta(m)$

$= \theta(\log_2^n)$ Ans

Summation & Formula:

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

$1 + x + x^2 + x^3 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$

$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

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

GP:- Sum of n-Terms $= \frac{a(r^n - 1)}{r - 1}$ if $r > 1$

$= \frac{a(1 - r^n)}{1 - r}$, if $r < 1$

$\log^K n = (\log n)^K$

$\log_b^a = \frac{1}{\log_a^b} = \frac{\log a}{\log b}$

$\log ab = \log a + \log b$

$\log a/b = \log a - \log b$

$a^{\log_b^n} = n^{\log_b^a}$

Calculate Time Complexity :-

1) frequency Count Method (For iterative Algo.)
2) Recurrence Relation Method (For Recursive Algo.)

frequency Count Method :-

```
for (i=0; i<n; i++) → O(n)
for (i=0; i<n; i=i+2) → O(n/2) = O(n)
for (i=n; i>1; i--) → O(n)
for (i=1; i<n; i=i*2) → O(log2 n)
for (i=1; i<n; i=i*3) → O(log3 n)
for (i=n; i>1; i=i/2) → O(log2 n)
for (i=0; i*i<n; i++) = O(√n)
for (i=n; i>=0; i=i-30) = O(n/30) = O(n)
```

Ex:- Algo: Add (a, b, c, m, n)

```
{ for(i=1 to m) do ⟶ m+1
    for(J=1 to n) do ⟶ m(n+1)
      C[i,J] = a[i,J] + b[i,J] → mn
}
```

Total Execution:

Total = 2mn + 2m + 1

≈ O(mn)

Ans:

Ex:- A( )

```
{ i=1, S=1;
  while(S<n)
  { i++;
    S=S+i;
    Pf("GATE");
  }
}
```

Soln:

S: 1 3 6 10 15 .... S ≥ n

i: 1 2 3 4 5 ..... K

$\frac{K(K+1)}{2} \geq n$ (K(K+1)/2 : ↑ 2)

$K^2 \geq n \Rightarrow K = O(\sqrt{n})$

Ex:- Algo: A( )

```
{ int i, J, K, n;
  for(i=1; i<n; i++)
  { for(J=1; J<=i; J++)
    { for(K=1; K<=100; K++)
        Pf("GATE");
    }
  }
}
```

Soln:

i = 1	i = 2	i = 3	.... i = n
J = 1	J = 2	J = 3	J = n
K = 100 times	K = 2*100 times	K = 3*100 times	K = n*100

Total Time Sum of All = 100 (1 + 2 + 3 + ... + n)

$= 100 \times \frac{n(n+1)}{2} = O(n^2)$ Ans

Ex:- Algo: A()
```
{ int i,j,k,n;
  for(i=1;i<n;i++)
   {for(j=1;j<=i^2;j++)
      {for(k=1;k<=n/2;k++)
         Pf("GATE");
      }
   }
}
```

Sol^n:

$i=1$	$i=2$	$i=3$	..... $i=n$
$j=1$	$j=4$	$j=9$	$j=n^2$
$k=n/2+1$	$k=n/2*4$	$k=n/2*9$	$k=n/2*n^2$ times

Total time $= \frac{n}{2}\left[1^2+2^2+3^2+4^2+\cdots\cdots+n^2\right]$

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

Ex:-
```
main()
{ while(n>=1)
   { n=n/2;
     n=n/3;
   }
}
```
$n=n/6 \Rightarrow O(\log_6 n)$ Ans

Ex:-
```
main()
{ while(n>=1)
   { n=n-10;
     n=n-20;
   }
}
```
$\Rightarrow n/30$ times $= O(n)$

Recurrence Relation Method:-

Ex:-
```
A(n)
{if(n<=1)
   return n
 else
   return(A(n/2)+A(n/2)+n)
}
```
Sol^n: Recurrence Relation for given Algo:

$$T(n)=\begin{cases}1, & \text{if } n\le 1\\ 2T(n/2)+c, & \text{if } n>1\end{cases}$$

Apply master Method Ans $=\Theta(n)$

Ex:-
```
A(n)
{if(n<=1)
   return 1
 else
   return (2*A(n/2)+6+A(n/2))+n^2)
}
```
Sol^n: Recurrence Relation for given Algo:

$$T(n)=\begin{cases}1, & \text{if } n\le 1\\ 2T(n/2)+c, & \text{if } n>1\end{cases}$$

$T(n)=2T(n/2)+c$, Apply master

Ans $=\Theta(n)$

Ex:-
```
int recursive(int n) {
 if(n==1)
   return 1
 else return (recursive(n-1)+recursive(n-1));
}
```
($T(n)$; each $T(n-1)$)

Sol^n: Recurrence Relation

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

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

Apply master for Decreasing function

$= O(2^n)$ Ans

Ex:- int Dosomething(int n){
if (n<=2)
return 1
else return (Dosomething (floor(sqrt(n))) + n);
}

Soln: Recurrence Relation for given function

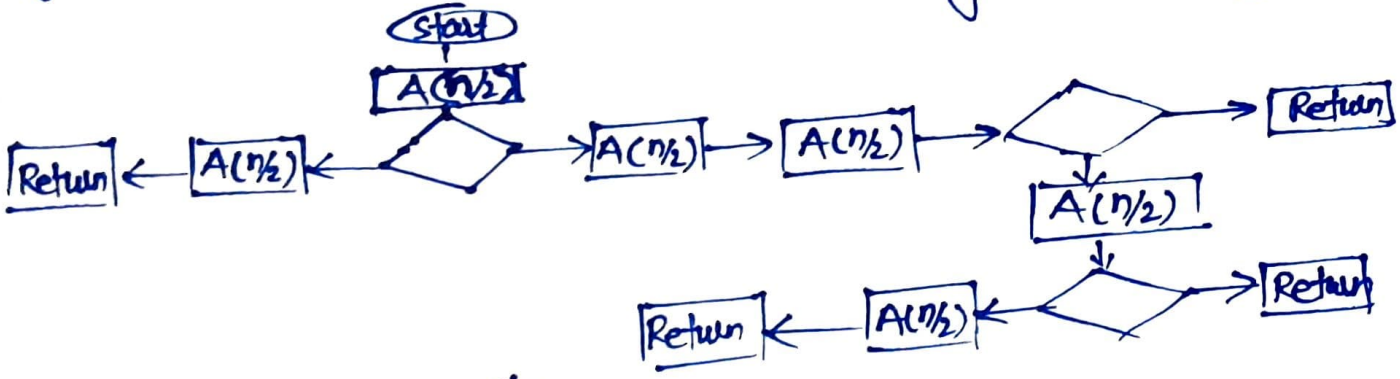
$$T(n) = \begin{cases} 1, & \text{if } n \le 2 \\ T(\sqrt{n}) + 1, & \text{if } n > 2 \end{cases}$$

Soln: of Recurrence Relation $T(n) = T(\sqrt{n}) + 1$, Put $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = T(2^{m/2}) + 1 \Rightarrow S(m) = S(m/2) + 1$, where $a=1, b=2, K=0, P=0$

Apply Master Case 2(i) TRUE $\Rightarrow \Theta(m^{\log_b a} \cdot \log^{P+1} m) = \Theta(m^{\log_2 1} \cdot \log^{0+1} m) \Rightarrow \Theta(\log m) = \Theta(\log \log_2 n)$ Ans

Ex:-



Worst Case Time cost = $O(n^{\alpha})$

What is α ?

Soln: Recurrence Relation for flow chart Worst Case $A(n) = 5A(n/2) + 1$

$A(n) = 5A(n/2) + 1$

Apply master Method Ans = $O(n^{\log_2 5}) = O(n^{2.32})$

$\alpha = 2.32$ Ans

Space Complexity:- for one variable one Unit space

Iterative Algo:

Ex:- Algo: Add (A,B,n)

```
{ Create C[n,n];
  for (i=0; i<n; i++)
    {for(J=0;J<n;J++)
       { C[i,J] =a[i,J] +b[i,J];
       }
    }
}
```

Space Complexity:

A[] → n^2] Not Considered Because of Modular Programming (it is already Considered in Calling function)
B[] → n^2]
C[] → n^2
n → 1
i → 1
J → 1

$S(n) = n^2+5 = O(n^2)$

Ex:- Algo: Mult(A,B,n)

```
{ Create C[n,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];
            }
        }
    }
}
```

Space

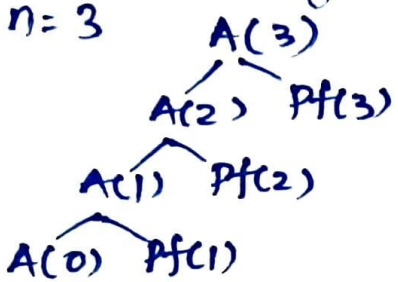
A[] → n^2] not Considered because memory already Allocated in Calling function
B[] → n^2]
C[] → n^2
n → 1
i → 1
J → 1
K → 1

Total = $n^2+6 \Rightarrow O(n^2)$ Ans

Recursive Algo: Space Complexity for Recursive Algo. is max. No. of function in the stack at particular time

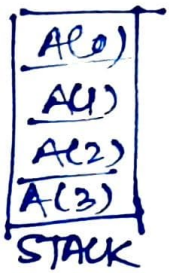
Ex:- Algo: A(n)
```
{ if(n>1)
  { A(n-1)
    Pf(n)
  }
}
```

Let n = 3



Print 1 2 3

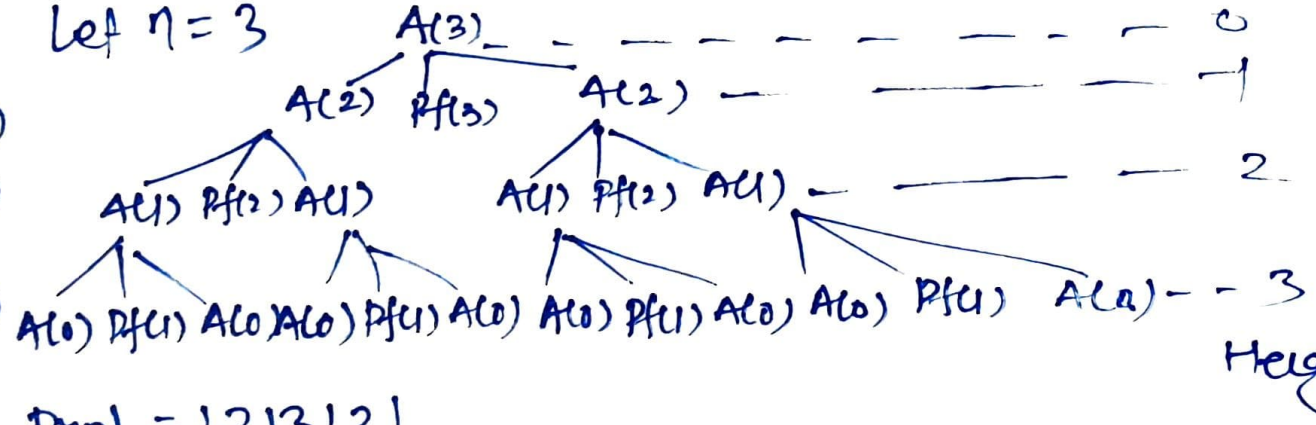
How many Recursive call = 4 ⇒ n+1



Max. No. of function in STACK = 4 = n+1
So Space Complexity = 4 = n+1 = O(n)

Ex:- Algo: A(n)
```
{ if(n>1)
  { A(n-1)
    Pf(n)
    A(n-1)
  }
}
```

Let n = 3

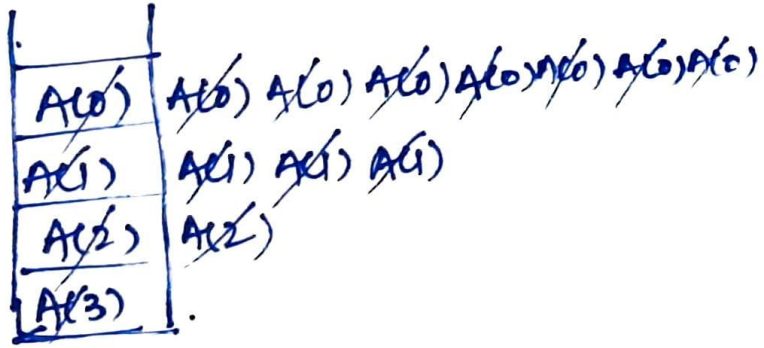


Height of Tree = 3

Print = 121312

STACK space:

Maximum No. of function in STACK = 4



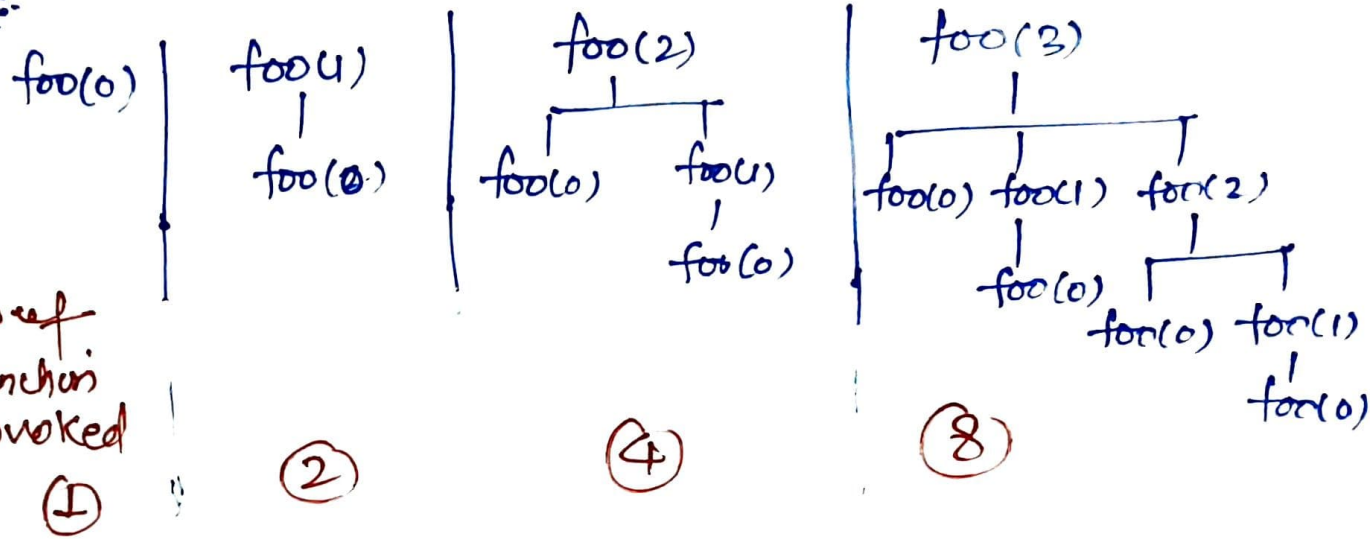
Space Complexity = Max. function in STACK
= 4 = n+1 = 3+1 = O(n)
↑ height

OR

Space cost = height + 1
= O(height) = O(n) Ans

Ex:-
```
double foo(int n){
  int i;
  double Sum;
  If (n==0) return 1.0;
  else {
      Sum = 0.0;
      for (i=1; i<n; i++)
        Sum += foo(i);
      return Sum
  }
}
```

Sol^n:-



No of function invoked: ① ② ④ ⑧

Similarly

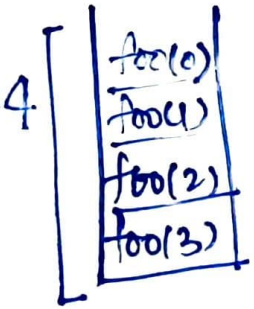
No of function invoked for $foo(n) = 2^n$

So Time Complexity = No of function invoked = $2^n = O(2^n)$

But space complexity = Max. No of function in STACK

If $n=3$

Max function in STACK = 4 = $n+1 = O(n)$



So space cost = $O(n)$, Time cost = $O(2^n)$

Q find space complexity

a) $O(1)$

b) $O(n)$

c) $O(n!)$

d) $O(2^n)$