theorem

UNIT - I

INTRODUCTION.

Notation of an algorithm - Fundamentals of Algorithmic problem solving - importance problem Type fundamentals of algorithmic Efficiency - Analysis framework - Asymptotic Notations and their properties - Empirical Analysis - Mathematical analysis of Recurive and non-Recursive algorithms - visualization.

Algorithm:

* Algorithm is nothing but it refers to a step by step process performing some action
* A step by step method for solving a problem (or) doing a task.
* Set of instructions / steps to solve problem / task
* This algorithm is a most fundamental for all the programming lang.
* algorithm is mainly used to find the optimal solution for the problems.

many algorithm methods / types:

1. Searching (Linear Binary)
2. Sorting (Bubble, quick, etc)
3. Brute force (checking all the possibilities)
4. Greedy method
5. Dynamic programming
6. Divide and conquer
7. Recursive Programming.

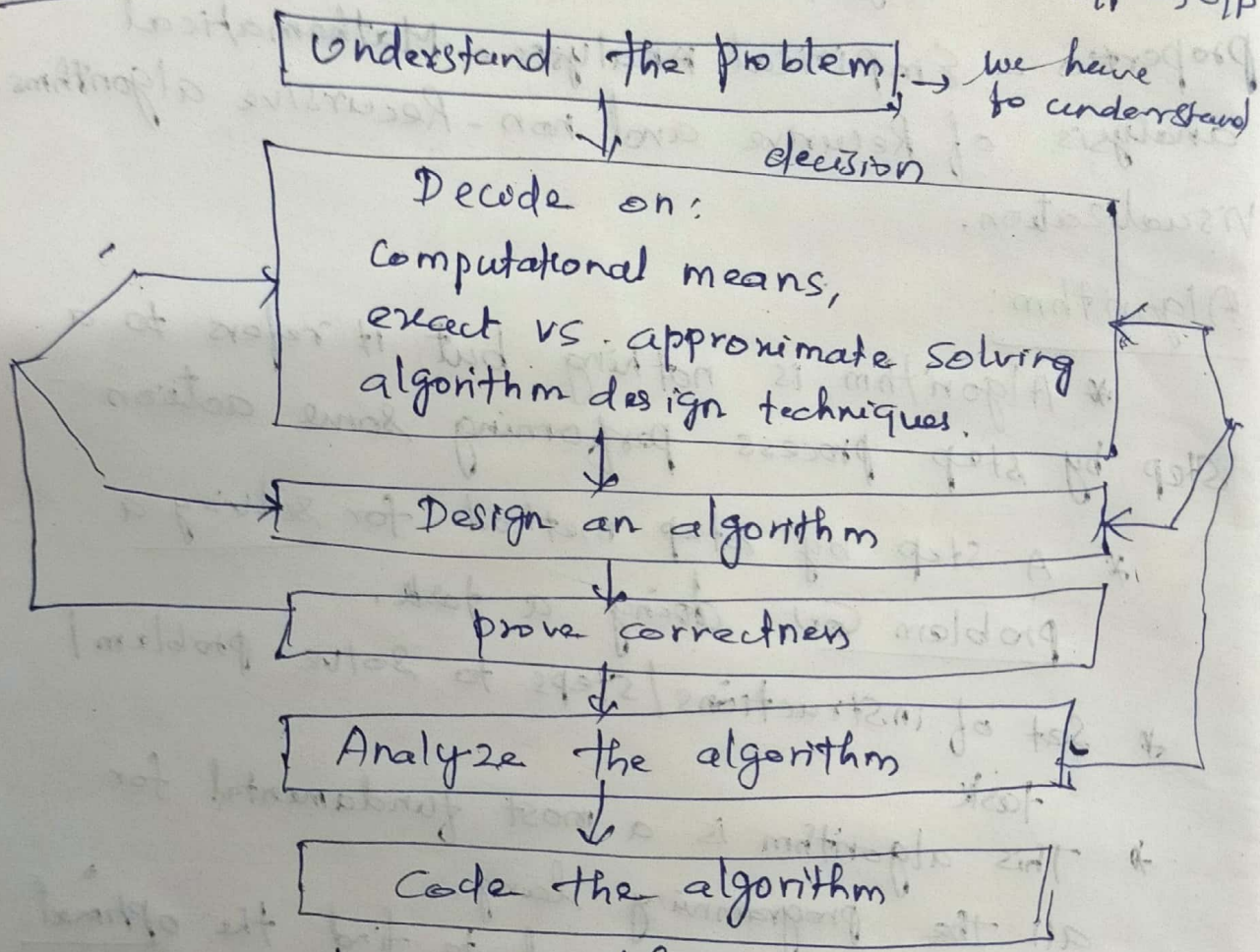
* We have to learn all the methods to find the solutions.

Problem
↓
algorithm
↓

whenever we have given with any sort of problem we have to design an algorithm and we have to feed the computer with proper i/p → o/p.

Input → Computer → output

Steps:



understand the plm: (Given complete plm stmt)

* First step in designing of algorithm.
* Read and understand the plm statements completly.
* Clarifying the doubts about the plm.
* Identify plm type and use existing algorithm to find solutions.

Decision making:

* Decide what type of algorithm we are going to use either (sequential (or) parallel)

Algorithm design techniques:

Alg + DS = programs.

Here we are going to specify what types of DS going to used (i.e) stack (or) queue etc.

Eg: Brute force, Divide and conquer etc.

methods of specifying an algorithm

There are three ways to specify an algorithm They are:

a) Natural language
b) Pseudocode
c) Flowchart.

(a) It is very simple and easy to specify an algorithm using natural languages. But many times specification of algorithm by using natural language is not clear and thereby we get brief specifications.

Ex: Addition of two no's

1. Read the first number, a
2. Read the second number, b
3. Add the above two no and store the results in c.
4. Display the result from c.

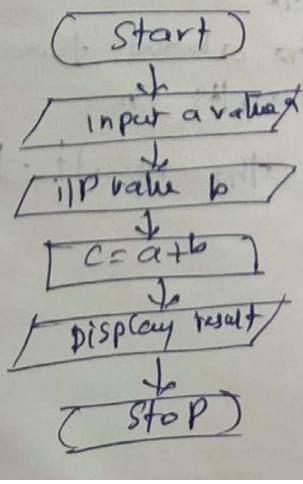
⑥ PSEUDOCODE:

* mixture of a natural language and programming language constructus.
* more precise than natural language.
  - for assignment operation left arrow, "←"
  - for comment "//"
  - if condition, for, while loops are used.

```
Alg Sum (a,b)
// Plm Descnption: add of two no
// input : two integers a and b
// output : Addition of two integers.
        c ← a+b
        return c
```

Flow chart:

* Graphical representation of an algorithm method of expressing an alg by a collection of connected geometric shapes containing description of the algorithm's steps.



proving an alg Correctness:

* once an algorithm has been specified then its correctness must be proved.
* Ex: The Correctness of Euclid's algorithm for computing Greatest common divisor stems from the correctness of the equality

$$gcd(m,n) = gcd(n, m \bmod n).$$

Analyzing an algorithm:

* For an algorithm the most important Efficiency in fact, there are two kinds of algorithm efficency

1. Time Efficeny : indicating how fast the algorithm runs
2. Space Efficency: how much extra memory it uses.

The Efficiency of an algorithm is determined by measuring both time and space efficency.

factors of analysing an algorithm are,

1. Time efficeny of an algorithm
2. Space efficeny of an algorithm.
3. Simplicity of an algorithm
4. Generality of an algorithm:

Coding an algorithm:

* The coding / implementation of an algorithm is done by a suitable programming lang like C, C++, Java.

Problem Types:

Sorting: Arranging the given elements in ascending / descending order.
(Insertion, Bubble, quick etc)

Searching: Search and find the given Search key element in the array.
Sequential, Binary search, etc.

String processing: Processing / dealing with the string.
Brute force String matching,

Graph problems:

Graph is a collection of points called vertices and connected by lines is called Edges.
DFS, BFS, Topological sorting etc

Combinatorial plm:

plm dealing with combinatorial objects such as permutation, combination (or) subset.

TSP, Graph coloring etc.

Geometric plm:

Used in computer Graphics, robotics etc.

closest pair plm, Convex Hull plm.

Numerical plm:

Solving equations, Evaluating fun, etc.

Fundametals of algorithm Efficeney:

* Algorithm analysis refers to the task of determining the computing time and storage space requirement of an algorithm.

* It is also known as performance analysis (or) efficeney of an algorithm which enables us to select an efficent algorithm.

* when we have a problem to solve, there may be many algorithm available. we would obviously like to choose the best. The selection of best algorithm is possible by analysing the algorithm.

* The Efficeny of an algorithm can be in terms of time and space. The algorithm efficeny can be analysed by the following ways.

a) Analysis framework.
b) Asymptotic Notation and its properties.
c) mathematical analysis for Recursive algorithms.
d) mathematical analysis for non-recursive algorithms.

Analysis framework:

① Time ② Space.

↓

also called time complexity, indicates how much time takes

The algorithm analysis framework consists of the following

1. Measuring an input size.
2. Units of measuring Running Time.
3. Orders of Growth.
4. Worst, Best, Average case Efficiency.

Measuring i/p size:

The i/p size denoted by 'n' and we use 'n' in most of the algorithm.

a) Searching and sorting 'n' indicates the no of array elements.

b) matrix multiplication n indicates the matrix order.

i/p size 'n' is much important in analysing the algorithm.

(2) Units of measuring Running Time:

* Cannot measure the running time by seconds, milliseconds, and so on bcz such a measurment depends on the typ of computer, compiler and the pgm.

* methods of computing the time Efficiency of algorithm.

1. Operation Counts
2. Steps Counts

Operations Count:

The time is measured by counting the no of basic operations or key operations.

Ex:

It will execute until the condition is fals.

A = a*b → executes one unit of time

for (i=0; i<n; i++) → The code takes 'n' units of time bcz, it execute for n times
  a = a+i; if, n=5

for (i=0; i<n; i++)
  for (j=0; j<n; j++) → This code take n^2 unit of times
    print ("hellow");

Step count:

Here, we count no of times one instruction is executing.

A = a*b → Step count 1

for (i=0; j<n; i++)
  a = a+i; → Step count n

for (i=0; j<n; i++)
  for (j=0; j<n; j++) → Step count n^2
    print ("hellow")

Order of Growth: (when the i/p size of the algorithm is increases the computation size of an algorithm also increase).

→ If the alg is fast, slow, avg. This order of growth is used to identify the algorithm is fast, slow, or Avg.

→ Order of growth: | AB upper bound.

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) \ldots < O(2^n) < O(n!)$

lower B

↑ Relation of this order of growth.

+ By using this we can identify the efficeny of algorithm.

Example table:

log n	n	n log n	n^2	n^3	2^n	n!
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	8	24	64	512	256	10080
4	16	64	256	4096	65536	4789 57u
5	32	160	1024	32768	4234967 296	2631 35

1. log complexity - O(logn)

$2^n = 2^5 = 32$ steps.

* plm will be divided into small parts
* After each division no. steps will be reduced to half

Ex: Binary search. search down = 10

10 20 30 40 50

Efficiency is decreasing the amount time is decreasing.

order of growth

	Description
1 - constant	
logn - logarithmic	
n - linear	
nlogn - linear log.	
n^2 - quadratic	
n^3 - cubic	
2^n - Exponential.	

no. of steps →

If the i/p n=8

$\log 8 = \log_2 2^3 = 4$

(3) steps is executed

i/p is executed

16 → $\log 16 = \log_2 4$

4 - is not executed steps

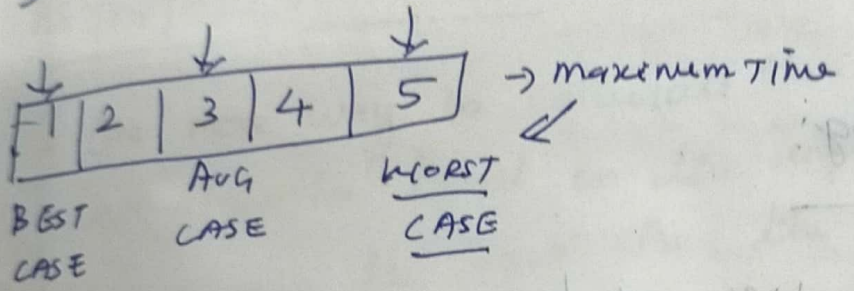
n=8, n=10

4log4 = 8

Time complexity:

* How much operation used.

1. BEST CASE : BIG OMEGA (Ω)
2. AVERAGE CASE : BIG THETA (Θ)
3. WORST CASE : BIG O(O)



①
a=5 → O(1)
b=10 → (1)
Sum = a+b → (1)

O(3)

O(1) → constant.

O(5) (or) O(n).

② n=5
```
for (i=1; i<=n); i++)
    print(i)
```

③
```
for (i=1; i<=n; i++) }  O(n)
    print(i)
for (j=1; j<=n; j++) }  O(n)
    print(j)
```

O(2n)
O(n)

Nested:

```
n=5
for(i=1; i<=n; i++)
  {
    for(j=1; j<=n; j++)
    {
      Print(i,j)
    }}
```

i = 1 = 5
i = 2 = 5
i = 3 = 5
i = 4 = 5
i = 5 = 5

Print Stmt 25 time execute.

$O(n^2)$

What is log:

$\log N$ mathematis $\rightarrow \log_{10}$

$\log N$ in CSE $\rightarrow \log_2$

$\log_2 16 =$

How many times do you multiply 2 to get 16?

$2 * 2 * 2 * 2 = 16 = 4$

$2^4 = 16$

$\log_2 16 = 4$

$\log_2 8 = 3$

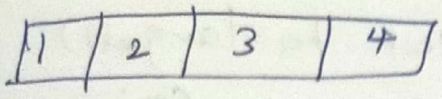
$2^3 = 2 \times 2 \times 2 = 8$

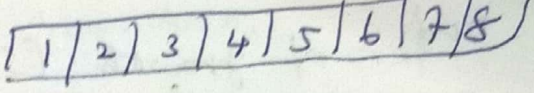
$\log_2 1024 = 10$

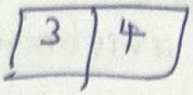
2^{10}

$\log_2 12 = 3.5$

$\log_3 27 = 3$







Array length = 4 >> 2

Array length = 8 >> 3.

$O(\log(n))$

If we have to do for every elements.

$n * \log(n)$

Time complexity $= n(\log n)$

ASYMPTOTIC NOTATIONS

* One more way to calculate the performance of an algorithm based on the input size
* Depends upon the input size the algorithm Execution time will be changed.

There are three categories:

1. Big O notation (O) → represent upper bound → max Time
2. Omega notation (Ω) → Lower bound → min Time
3. Theta notation (Θ) → Average bound → inbetween lower & upper bound.

Big - O(n) - worst case → max Time

Omega → $\Omega(n)$ - Best case → min Time

Theta → $\Theta(n)$ - Avg case → inbetween ob/lb.

n → linear time complexity.

Ex:

Searching:

10, 20, 30, 40, 50

Search key = 50.

order of growth (smallest to largest)

f(n)	Name of function	Ex:
Best O(1)	Constant	arithmetic operation
O(log n)	logarithmic	Binary search (sorted)
O(n)	linear	Linear search (unsorted)
O(n log n)	Linearithmic	merge sort with divide and conquer
$O(n^2)$	Quadratic	bubble sort
$O(n^3)$	Cubic	matrix multiplication
Worst $O(2^n)$	Exponential	Towes of hanoi.

How to calculate Time complexity

Steps to remember (or) Rules

1. Identify no of blocks
   - → Declaration } Time complexity will be constant O(1)
   - → Initialization }
   - → Iterations → Nested → multiplication (for (or) while)
   - → Conditional stmts → Max(if, else)
2. Avoid Constants
   If tc is $2n$, $n/2$, $n+1$
3. Add Time complexity of all blocks.
4. Time complexity will be highest order.
   in Equation
   Iteration — dependent
   Iteration — Independent

until i greater than n.

Ex:

① for (i=0; i<=n; i++) → n+1 times

{ ... }

Time complexity = O(n)

② for (i=1; i<=n; i=i*2) ③

{ ... }

$i = 1$	$0 < = n$ (T)
$2 = i = 2$	$2 < = n$ (T)
$2^2 = i = 4$	$4 < = n$ (T)
$2^3 = i = 8$	$8 < = n$ (T)
$2^4 = i = 16$	$16 < = n$ (T)

2^k (F) $= n$

$2^k = n$

$Tc = k = \log_2 n$

$TC = O(\log_2 n)$

③ for (i=0; i<=n; i++) → n times

{ for (j=0; j<=n; j++) → n times

{ Time complexity = $O(n*n)$

$= O(n^2)$

} for (k=0; i<=n; k++ → n times

} { }

Time complexity = $O(n*n*n)$

$= O(n^3)$

$TC = O(m*n)$

④ if (condition)

{ for (i=0; i<=n; i++) → n times

{ }

```
}
else
{
  for(j=0; j<=n; j=j*2)  → log n times
  {
  }
```

TC = O(n)

④
```
void main()
{
  int i;   → B1 → 1 time
  i=0;     → B2 → 1 time
  for(i=0; i<=n; i++)
  {
    printf("welcome");
  }
}
```
B3 → n+1 times

if i will be(n+1) loop will be terminate

$\Rightarrow 1+1+n+1 = n+3$

TC = O(n)

⑤

Mathematical analysis of Recursive and Non Recursive algorithms

Recursive:

* The algorithm it's calling by itself.

Eg:
```
void add()
{ int a,b;
  add();
}
```

General Condition:

1.

to compute factorial:

if $n=0$ return 1;

else return $F(n-1) * n$;

constant

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

GT

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

Replace n by $(n-1)$

$F(n-1) = [F(n-1-1)+1]+1$

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

$= F(n-2)+2$

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

$= F(n-3)+3$

$F(n-3) = F(n-1-3)+1+3$

$= F(n-4)+4$

$\vdots$

$F(n) = F(n-n)+n$

$= F(0)+n$

$F(n) = 1+n$

$F(n) \in O(n)$

$0! = 1$

$1! = 1$

$2! = 2 \times 1 = 2$

$3! = 3 \times 2 \times 1 = 6$

$4! = 4 \times 3 \times 2 \times 1 = 24$

$\vdots$

$n! = (n-1) * n.$

$F(n) = F(n-1) * n$

$n = 1$

$F(1) = F(0) * 1$

$= 1 * 1 = 1$

$F(2) = F(1) * 2$

$= 2$

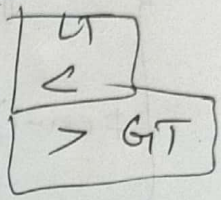
Big oh:

$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < 3^n < n^n$

The function $f(n) = O(g(n))$ iff $\exists$ +ve constant c and n_0.

Such that $f(n) \leq c * g(n)$ $\forall$ $n \geq n_0$ (GT) (GT) $c \geq 0$

or:



① $f(n) = 2n + 3$

This one: $2n + 3 \leq 10n$ ← Greater than GT $n \geq 1$

$f(n)$ c $g(n)$

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

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

$2n + 3 \leq 10n$ $c : g(n)$ $n = 1$

$2 + 3 \leq 10$

$7 \leq 20$ $n = 2$

$f(n) = O(n)$

② $2n + 3 \leq 3n^2 + 3n^2$

$2n + 3 \leq 5n^2$ $n \geq 1$

$\boxed{f(n) = O(n^2)}$ → upper bound.

~~Theta~~:

omega:

$2n + 3 \leq 10n$ $n = 1$

$n = 2$

$2 + 3 = 5$

$7 = 20$

$f(n) \leq c \cdot g(n)$ Big oh.

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

$f(n) = 2n + 3$

$2n + 3 \overset{GT}{\geq} c \cdot g(n)$ $n = 1$

$2n + 3 \geq 1 \cdot n$ $n = 2$

$5 \geq 1$

$7 \geq 2$

$f(n) \Omega(n)$

Theta: $f(n) = \Theta(g(n))$ for all possible c_0, c_1, n_0.

(LT) $c \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$

$f(n) = 2n+1$

$1 \cdot n \leq 2n+1 \leq 10n$

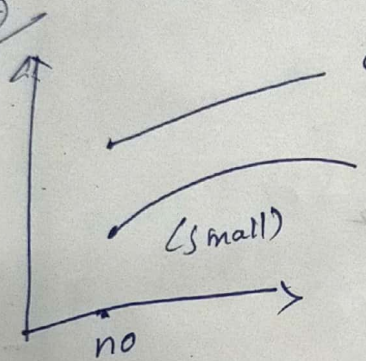
lower bound — upper bound.

Big oh

$\Omega(n)$ $\Theta(n)$ $O(n)$

represent average bound

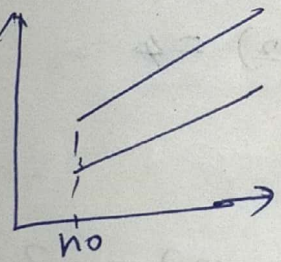
① Θ



→ upper bound
→ worst case.
→ complex p/m.
→ Big oh.

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

② Ω



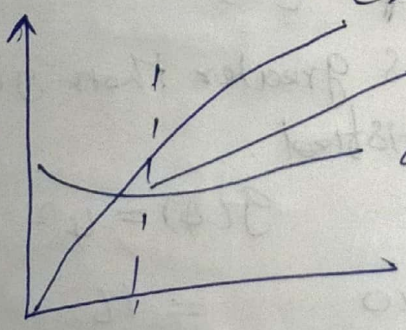
→ lower bound.
→ best case.
→ Running time min.

(GT) $F(n) \geq c * g(n)$.

③ Θ

$c_1 g(n) \leq f(n) \leq c_2 (g(n) \quad n \geq n_0$.

$F(n) \in \Theta(g(n))$



→ Big oh ⇒ $f(n) \leq c * g(n)$

→ omega ⇒ $f(n) \geq c * g(n)$

→ Thota ⇒ $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

$1 < \log n < n \log n < n < 2n < n^2 < n^3 .. 2^n$.

$1 < 2 < 3 < 4$.

Ex:

$f(n) = 2n + 2 \quad g(n) = n^2$

$n = 0, 1, 2$

$f(0) = 2(0) + 2 \quad g(0) = 0^2$

$f(0) = 2$

$f(0) > g(0) \quad (x)$

$2 > 0$

$f(1) = 2(1) + 2 \quad g(1) = 1^2$

$= 2 + 2 = 4 \quad g(1) = 1$

$f(4) > g(1) \cdot x$

$f(2) = 2(2) + 2 \quad g(2) = 2^2$

$= 4 + 2 = 6 \quad g(2) = 4$

$= f(6) > g(2)$

$f(3) = 2(3) + 2 \quad g(3) = 3^2$

$= 6 + 2 = 8 \quad = 9$

$f(8) < g(9) \quad (L)$

When $f(n)$ is greater than $g(n)$ is condition satisfied.

$\therefore$

$f(4) = 2(4) + 2 \quad g(4) = 4^2$

$= 8 + 2 = 10 \quad = 16$

$f(n) \in O(g(n)) \quad$ when $n \geq 3$

omega:

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

(GT)

$f(n) = 2n + 2 \quad g(n) = n.$

$f(0) = 2(0) + 2 \quad g(0) = 0$

$= 2$

$f(0) > g(0)$ - ✓

$f(1) = 2(1) + 2 \quad g(1) = 1$

$= 2 + 2 = 4, \quad 1$

$f(4) > g(1)$ ✓

$f(n) \in \Omega(g(n))$ when $n \geq 0$.

$f(n) > g(n)$

$g(n) < f(n)$

Theta:

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

$n \leq 2n+2 \leq n^2$ $\quad n \geq 3.$

$n > 0 \qquad n \geq 3$

Recurrence Relation

```
Void Test (int n)  → T(n)
{
 if (n>1)
 {
  for (i=0; i<n; i++)  → n
  {
   stmt;
   {
    Test (n/2);  → T(n/2)
    Test (n/2)   → T(n/2)
   }
  }
```

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

(2)

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

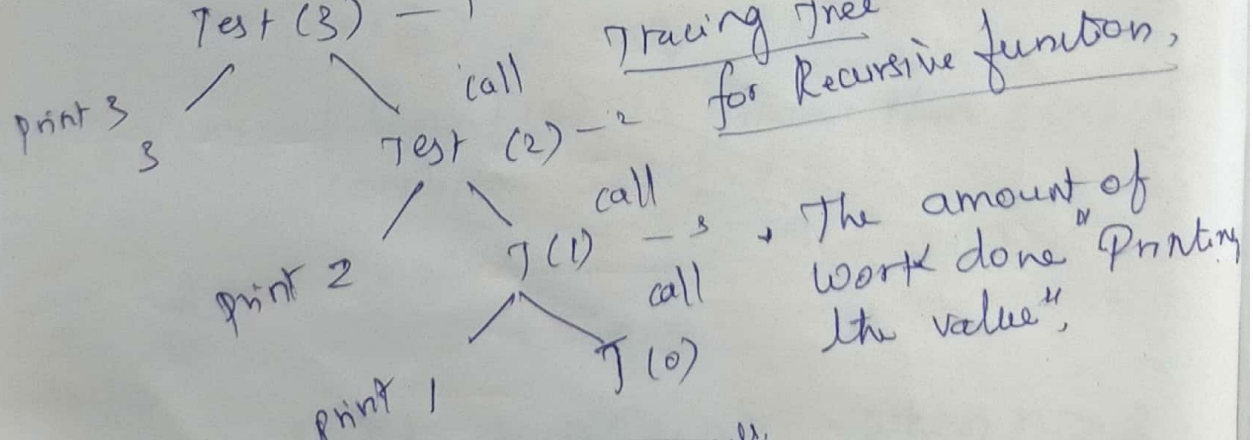
T(n) is equal to time taken to complete the work.

Recurrence

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

If pass 3 what will happen.

Tracing Tree for Recursive function,



- The amount of work done "Printing the value".

Time 3+1 call. fun = n+1 call

O(n)

Recurrence Relation (or) function:

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

Backsubstitution method:

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

Substitute $T(n-1)$

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

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

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

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

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

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

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

⋮ continue for k times

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

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

Assume $n - k = 0$

$\therefore n = k$

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

$T(n) = T(0) + n$

$T(n) = 1 + n$

$= O(n)$

③ void Test (int n) — T(n)

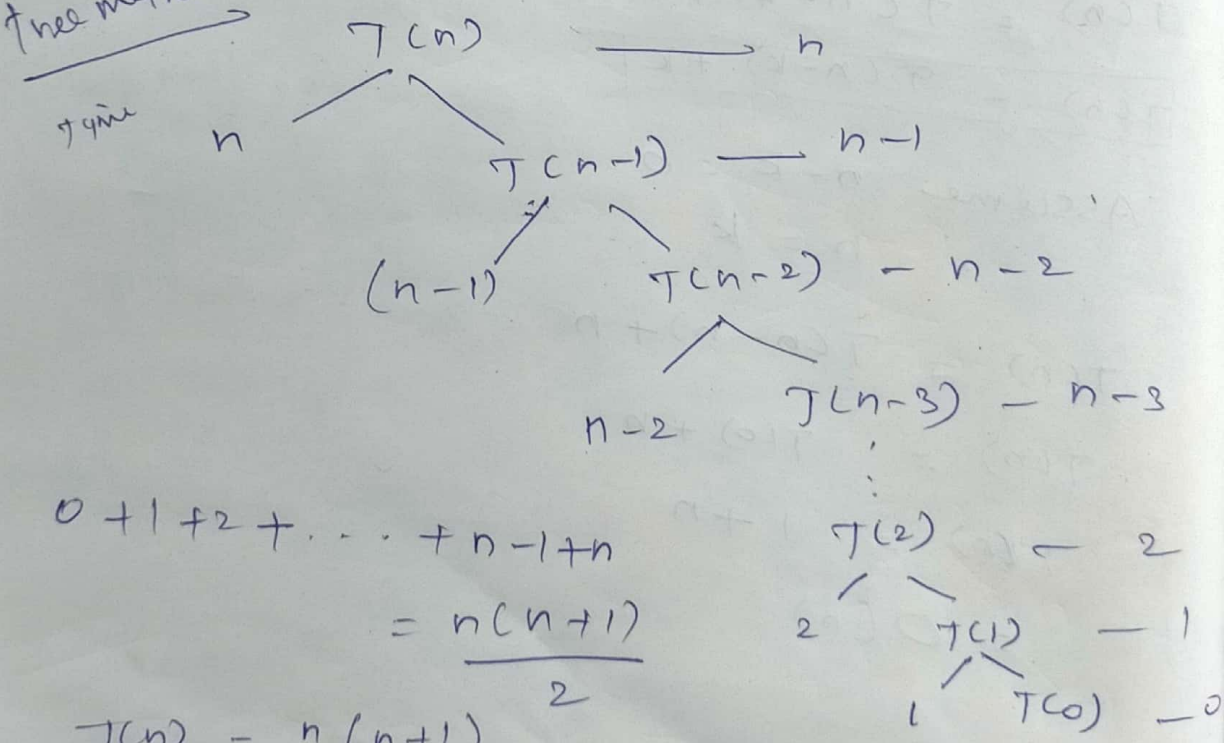
```
{
1   if (n>0)
    {
n+1   for (i=0; i<n; i++)
      {
n       printf("%d", n);
      }
T(n-1)  Test (n-1);
    }
}
```

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

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

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

Tree method:



$0 + 1 + 2 + \cdots + n-1 + n$

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

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

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

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

Substitution method.

$$T(n) = T(n-1) + n \quad \text{---(1)}$$

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

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

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

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

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

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

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

$$\vdots$$

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \cdots + (n-1) + n \quad \text{---(4)}$$

Assume $n - k = 0$

$\therefore n = k$

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

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

Sum of n natural nos.

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

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

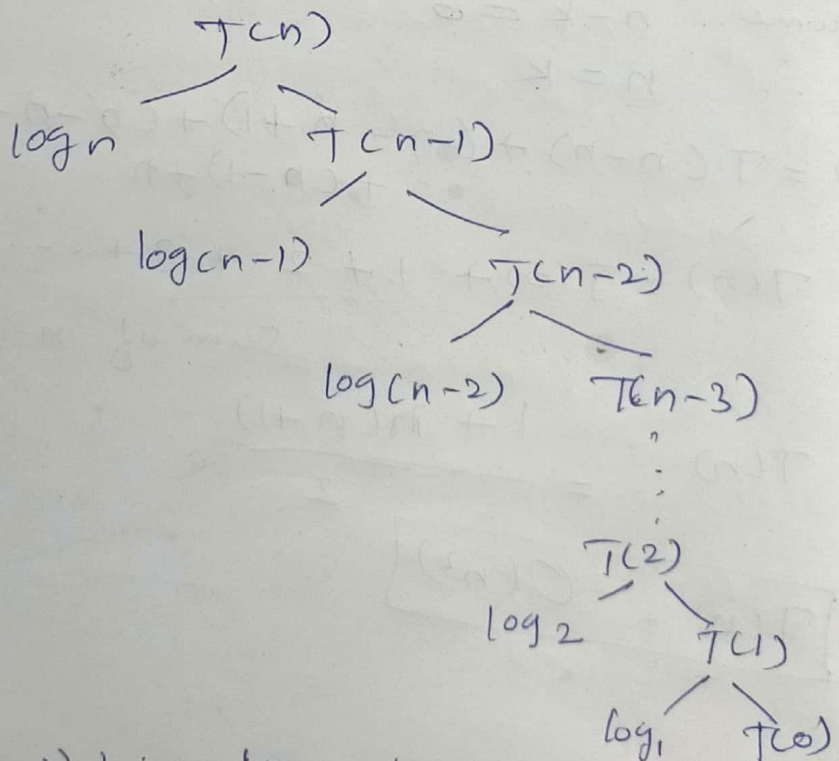
(3) void Test (int n) —— T(n)

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

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

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

Tree method:



Sum of:

$\log n + \log(n-1) + \cdots \log_2 + \log_1$

$\log[n \times (n-1) \times \cdots \times 2 \times 1]$

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

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

$T(n) = T(n-1) + \log n$ ——— (1)

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

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

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

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

⋮ Assume k times.

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

$= n - k = 0$

$n = k$

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

$= 1 + \log n!$

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

$T(n) = T(n-1) + 1$ — $O(n)$

$T(n) = T(n-01) + n$ — $O(n^2)$

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

$T(n) = T(n-1) + n^2$ — $O(n^3)$

$T(n) = T(n-2) + 1$ — $n/2 \rightarrow O(n)$

$T(n) = T(n-100) + n$ — $O(n^2)$

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