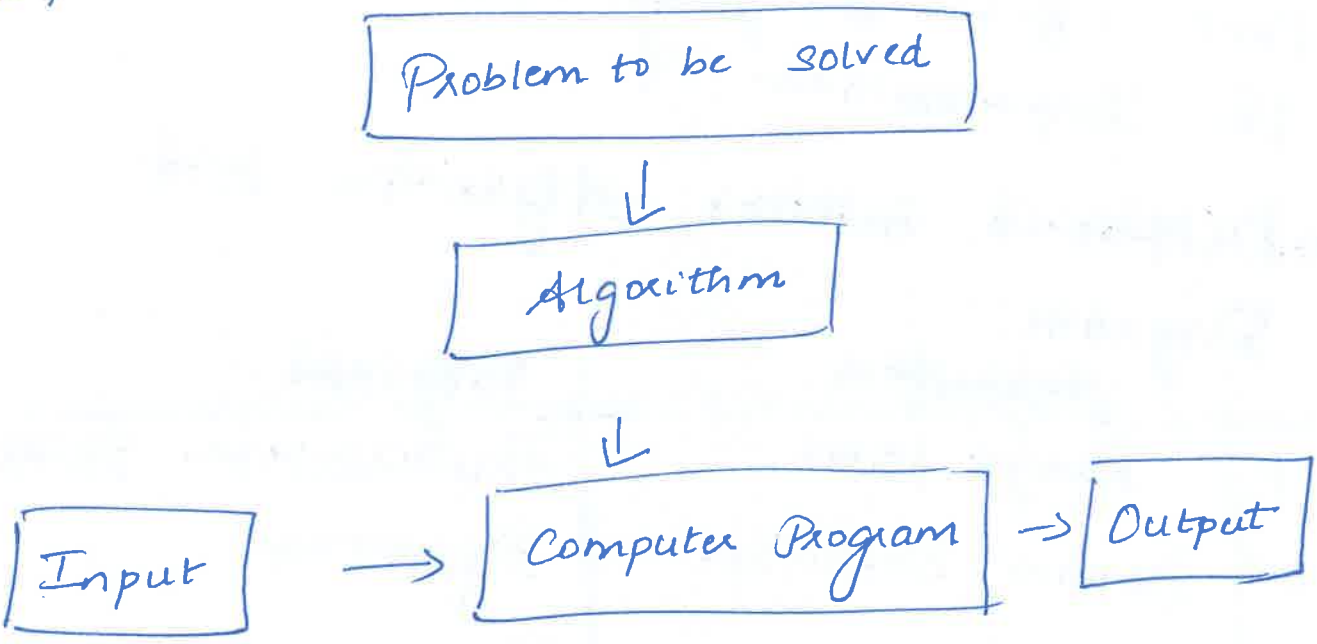
UNIT - I ALGORITHM DESIGN

1.1 INTRODUCTION

1.1.1 ALGORITHM

An Algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Algorithm is a step-by-step procedure for solving computational problems. Program is also a step by step procedure for solving a problem. Then what is the difference between Algorithm and Program? So by

Comparing algorithm and program, we can understand the importance of it.

In Software Development Life Cycle (SDLC), the phases of developing a Software Project, there are two important phases namely, Design Phase and Implementation Phase. First step is designing and then comes the Implementation

1.1.2 Difference between Algorithm and Program

	Algorithm	Program
1.	Design phase	Implementation Phase
2.	Domain Knowledge	Programme
3.	Independent on any Language	Programming Language
4.	Independent on H/W & OS	Dependent on H/W & OS
5.	Analyze	Testing

1.1.3 Priori Analysis and Posteriori Testing

S.NO	Priori Analysis	Posteriori Testing
1.	Algorithm	Program
2.	Independent on any Lang	Lang Dependent
3.	Hardware Independent	Hardware Dependent
4.	Time and Space function	Watch Time & Bytes

1.1.4 Characteristics of Algorithm

* Input
  An algorithm should have 0 or more well defined inputs.

* Output
  An algorithm should have 1 or more well defined outputs.

* Definiteness
  An algorithm should be precise.

* Finiteness
  Algorithms must terminate after a finite no. of steps.

* Feasibility

Should be feasible with the available Resources

* Independent

An algorithm should have step-by-step directions which should be independent of any programming code.

* Unambiguous

Algorithm should be clear and unambiguous.

1.1.5 How to Write an Algorithm?

Algorithm swap(a,b)
{
temp=a;
a=b;
b=temp;
}

Example 1:

Algorithm swap(a,b)
Begin
temp:=a;
a:=b;
b:=temp;
End

Example 2:

Algorithm swap(a,b)
Begin
temp←a;
a←b;
b←temp;
End

Example 3

1.1.6 How to Analyze an Algorithm?

→ Time Function
→ Space Memory
→ Network (How much data going to be transfered)
→ Power Consumption
→ Usage of CPU Registers.

Algorithms are represented / documented in two ways.

* Pseudocode
* Flowchart

Pseudocode

* Capitalize key commands (IF number > 10 THEN)
* Write one statement per line
* Use Indentation
* Be specific
* Keep it simple

Example 1

```
READ a,b
COMPUTE sum = a+b
PRINT SUM
```

Example 2

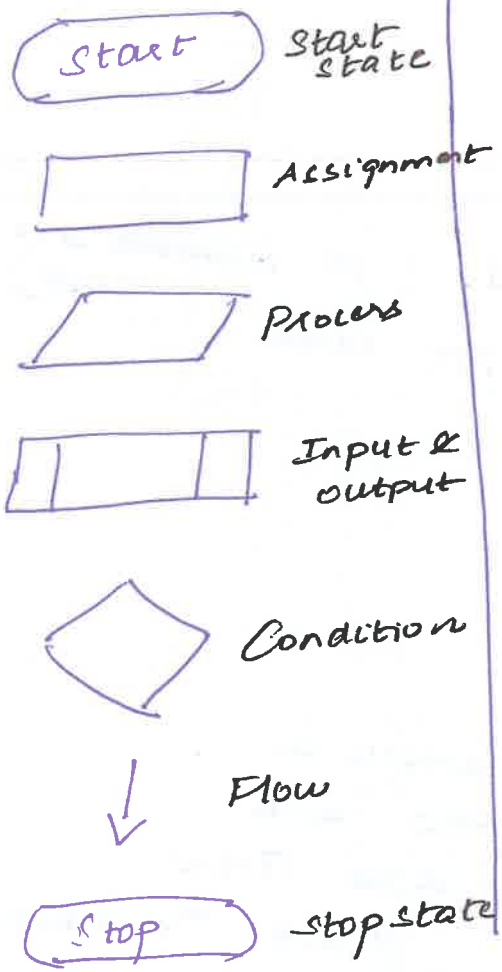
```
READ a,b
IF a>b THEN
    PRINT "a is big"
ELSE
    PRINT "b is big"
END IF
```

Flowchart

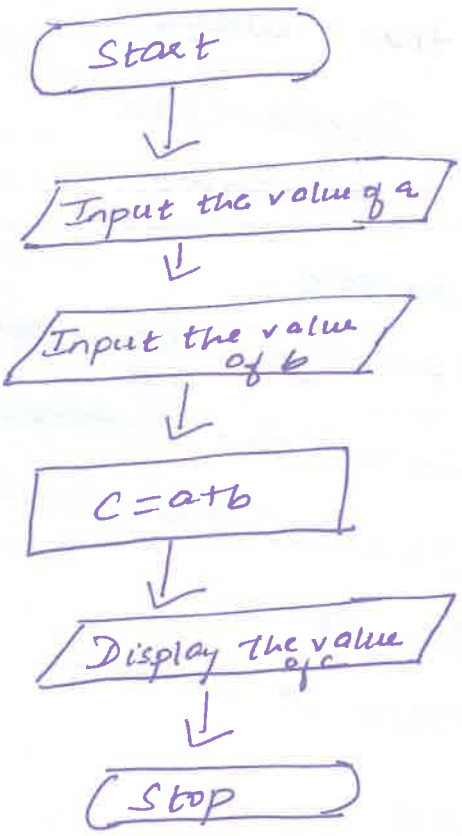
Flowchart is a graphical representation of an Algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing discriptions of Algorithm's steps.

Example

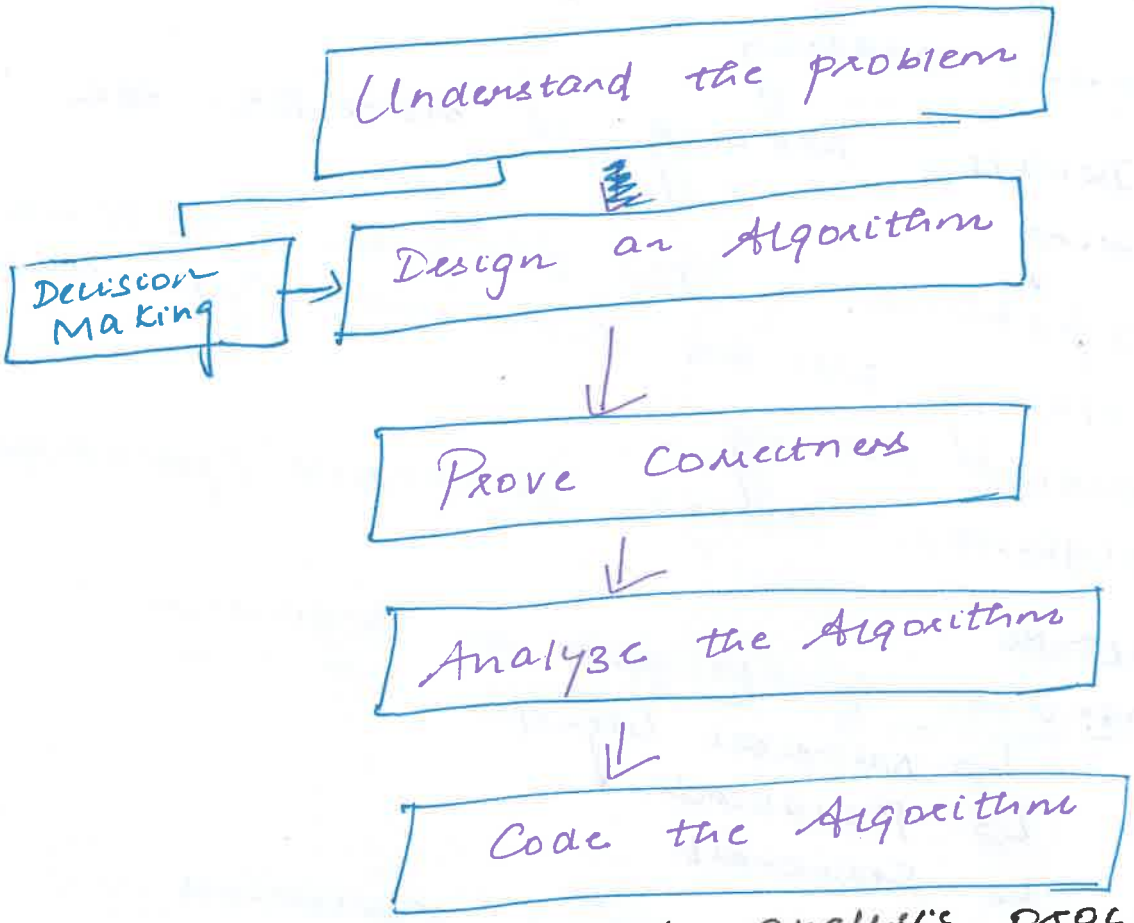
Symbols



Example: Addition of a & b



1.2. Fundamentals Of Algorithm.



Algorithm design and analysis process.

(1) Understand the Problem

* This is the first step in designing of Algorithm
* Read the problem's description carefully to understand the problem statement completely
* Ask questions for clarifying the doubts about the problem
* Identify the problem types and use existing algorithm to find solution.

* Input (instance) to the problem and range of the input get fixed.

(ii) Decision Making

The Decision Making is done on the following.

* Capabilities of the Computational Device
* choosing between exact and approximate problem solving.
* Algorithm design Techniques / paradigms

(iii) Methods of Specifying an algorithm

* Methods
  - Natural Language
  - Pseudocode
  - Flowchart

(iv) Proving an Algorithm's Correctness

→ Once an algorithm has been specified then its correctness must be proved.

→ An algorithm must yield a required result for every legitmate input in a finite amount of time.

(v) Analyzing an Algorithm

For an algorithm the most important is efficiency. There are two kinds of

algorithm efficiency.

* Time efficiency: indicating how fast the algorithm runs, and
* Space efficiency: indicating how much extra memory it uses.

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

(VI) Coding an Algorithm

→ The coding / implementation of an Algorithm is done by a suitable programming Language like, C, C++, Java

→ Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by inefficient implementation.

→ It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

1.3. Correctness of Algorithm

→ A proof of correctness requires that the solution be stated in two forms.

* One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus.

* The second form is called a specification, and this may also be expressed in the predicate calculus.

→ A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output.

A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct.

→ A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work

correctly for all possible inputs.

Methods of proving correctness.

* Proof by
  - → Counter example (indirect proof)
  - → Induction (direct proof)
  - → Loop Invariant

* Other approaches
  - → proof by cases/enumeration
  - → proof by contradiction
  - → proof by contrapositive.

Proof by counter example.

A counter-example is an example that shows that a statement is not always true. It is sufficient to just give one example.

Show by counter-example that the following statements are not always true:

1) $4n+4$ is always a multiple of 8 for all positive integer values of n.

when $n=2$, $4n+4=12$

not a multiple of 8

2) If $x>y$ then $\frac{x}{y}>1$

If $x=6, y=-3$, ie $6>-3$

then $\frac{x}{y} = \frac{6}{-3} = -2$

3) $p-q \leq p^2-q^2$ for all values of p and q

when $p=4, q=-5$

$p-q = 4--5$

$= 9$

$p^2-q^2 = (4)^2-(-5)^2$

$= -9$

So $p-q$ is not less than p^2-q^2 for all values of p and q.

4) $2n^2-16n+31$ is always positive for all values of n.

when $n=4$,

$2n^2-16n+31 = 2(4)^2-16(4)+31$

$= 32-64+31$

$= -1$

Proof by mathematical Induction (Direct Method)

Direct Method of Proof

Suppose the hypothesis P is true, Then the implication P -> Q can be proved if we can prove that Q is true by using the rules of inference and some other theorems.

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

Basis: $n=1$

$1=1$ ✓

Induction: Assume True $n=K$.

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

Show true $n=K+1$

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

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

$$K(K+1) + 2(K+1) = (K+1)(K+2)$$

$$K^2+K+2K+2 = K^2+K+K+2$$

Proof by mathematical induction involves four steps.

Basis: This is the starting point for an induction. Here it is proved that the result is true for $n = 1$

Induction Hypothesis: Assume the result is true for $n = k$.

Induction step: Prove that the result is true for some $n = k+1$.

Use Induction to prove $2+4+6+\cdots+2n = n(n+1)$?

Domino effect

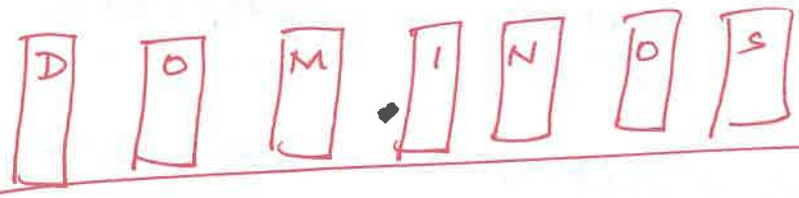
Formula True for the 1st term

↓

Formula True for the 2nd term

↓

Domino effect:

→ 

Condn 1: When the first domino falls, it will hit the second one

Cond 2: Each domino will hit the next one and each domino that is hit will fall.

Goal → All the dominos fall.

Base case:

Condition 1: P(1) is True for n=1

n=1 → First term

P → Propositional statement

Eg: statement x is Cold

P(x) → True or false (depends on x)

x is ice → P(x) is True

x is fire → P(x) is false.

Inductive case:

Condition 2: If P(n) true for n=k, then P(n) true for n=k+1

{P(k) true → P(k+1) true}

Inductive Hypothesis:

$2+4+6+\cdots+2n = n(n+1)$

$P(n)$: $2+4+6+\cdots+2n = n(n+1)$

Condition 1: $P(1)$ is True for $n=1$

$n=1 \rightarrow$ First term

$P(1)_{LHS} = 2$

$P(1)_{RHS} = 1(1+1) = 2$

Hence Proved.

$P(K)$: $2+4+6+\cdots+2K = K(K+1)$

Condition 2: If $P(K)$ is true

$\downarrow$

$P(K+1)$ is also true

$P(K+1)_{LHS} = 2+4+6+\cdots+2K+2(K+1)$

$= K(K+1)+2(K+1)$

$= (K+1)(K+2)$

$P(K+1)_{RHS} = K(K+1)$

$= (K+1)([K+1]+1)$

$= (K+1)(K+2)$

Hence Proved.

Proof by cases:

In a proof by cases, we must cover all possible cases that arise in a theorem.

Example: Prove if n is an integer $n \leq n^2$

case 1: $n \leq -1$

case 2: $n = 0$

case 3: $n \geq 1$

} must cover all possible cases.

Prove "If n is an integer, $n \leq n^2$"

case 1: $n \leq -1$ $\quad n^2 \geq 0$

Put $n = -1$

$-1 \leq (-1)^2 = -1 \leq 1$ true

case 2: $n = 0$ $\quad 0 \leq (0)^2 = 0 \leq 0$ true

case 3: $n \geq 1$ $\quad 1 \leq (1)^2 = 1 \leq 1$ true.

Since our inequality $n \leq n^2$ is true for all possible cases, we can conclude $n \leq n^2$ for all integers.

H/W Prove "If n is an integer, $n^2 + 3n + 2$ is even"

Proof by Contradiction.

A way to prove something is true by showing that if it wasn't true, that would lead to a logical error.

Assume I did rob the bank ⟶ Then I'd be rich

So I didn't rob the bank ⟵ I'm not rich.

→ We want to prove that a statement P is true

* We assume that P is false
* then we arrive at an incorrect conclusion
* therefore, statement P must be true.

Example.

Theorem: $\sqrt{2}$ is not rational

Proof:

Assume by contradiction that it is rational

$\sqrt{2} = n/m$

n x m have no common factors

We will show that this is impossible

1. Prove that $\sqrt{2}$ is irrational by giving a proof using contradiction

Solution:

Assume the contradiction that $\sqrt{2}$ is rational

$\therefore \sqrt{2} = \frac{a}{b}$ where a and b have no common factors.

$$\frac{a}{b} = \sqrt{2}$$

$$\therefore \frac{a^2}{b^2} = 2 \Rightarrow a^2 = 2b^2 \rightarrow (1)$$

This means that a^2 is even $\Rightarrow$ a is even. Let

$$a = 2c$$

$$(1) \Rightarrow a^2 = 2b^2$$

$$\Rightarrow (2c)^2 = 2b^2$$

$$\Rightarrow 4c^2 = 2b^2$$

$$= 2c^2 = b^2$$

$$\Rightarrow b^2 = 2c^2$$

$\Rightarrow b^2$ is even $\Rightarrow$ b is even

Thus, a and b are even. Hence, they have a common factor 2, which is a contradiction to our assumption $\sqrt{2}$ is irrational.

Proof by Contrapositive

If P, then Q

If ~Q then ~P

If Johny ~~doesn't~~ likes apples, then Johnny eats fruit.

If Johnny doesn't eat fruit, then Johnny doesnt like apples

1) If ~~7x9~~ $7x+9$ is even, then x is odd.

Suppose x is not odd, Thus x is even

So $x = 2a$ for an integer.

Then $7x+9 = 7(2a)+9 = 14a+9 =$

$= 14a+8+1 = 2(7a+4)+1$

Thus $7x+9 = 2b+1$, where b is the integer $7a+4$.

Therefore $7x+9$ is odd.

HW Prove that if $3n+2$ is odd then n is odd

Proof by Loop Invariant (Direct Method)

Definition

A loop invariant is a logical predicate such that. If it is satisfied before entering any single iteration of the loop, then it is also satisfied after that iteration.

Example

```
Linear Search (A, n, x)
for i = 0 to n-1
if a[i] = x return (i)
return not found.
```

At the start of each iteration of step 1 if x is present in array a then it is present in sub array from a[i] through a[n-1].

3 steps

→ Initialization

→ Maintenance

→ Termination

Initialization:

i=0: subarray [0:n] whole array so holds true.

Maintenance:

If x is present in a[0:n]
if a[i] ≠ x then we do not have x
everytime it is incremented, it holds true.

Termination:

If a[i] = x
if (i > n-1) then contrapositive of invariant. if ¬A then B if ¬B then ¬A.

Mathematical Induction eg pblms

1) $\sum_{i=1}^{n} i = n(n+1)/2, n \geq 1$

2) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6, n \geq 1$

3) $\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x-1), x \neq 1, n \geq 0$

1.4. Time Complexity Analysis

Time Complexity is the total time needed to solve the problem & run the algorithm.

T(P) = Compilation time + run/execution time

= C + T(P) (Instant characteristics).

↓

I/p & O/p (size & Number)

TP(n) = c_1 (addition) + c_2 (Subtraction) + ...

↓

Current load, Nature of compiler.

Desktop, Supercomputer, Mainframe computer depends. ~~on the~~.

frequency -> How many times repeated.

Nature of statements.

1) Comment // or /*....*/ = 0

2) assignment = 1

3) Computation = 1

4) Loops Range ('1' to 'n') n times.

Some basics of Time Complexity

(1)
```
for (i=0; i<n; i++)      → n+1
{
    stmt;                → n
}
```
2n+1 → O(n)//

(2)
```
for (i=n; i>0; i--)      → n+1
{
    stmt;                → n
}
```
2n+1 → O(n)//

(3)
```
for (i=1; i<n; i=i+2)
{
    stmt;                → n/2
}
```
O(n)//

(4)
```
for (i=1; i<n; i=i+20)
{
    stmt;                → n/20
}
```
O(n)//

Nested for loops

(1)

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

$n+1+n^2+n+n^2$

$2n^2+2n+1$

$O(n^2)$ //

(2)

```
for (i=0; i<n; i++)
{
for (j=0; j<i; j++)
{
    stmt;
}
}
```

i	j	no of times
0	0	0 ✗
1	0 ✓ 1 ✗	1
2	0 1 2 ✗	2
3	0 1 2 3 ✗	3
⋮	⋮	⋮
n		n

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

$\frac{n^2+n}{2}$

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

$O(n^2)$ //.

(3)

```
P=0;
for (i=1; p<=n; i++)
{
    p=p+i;
}
```

$1+2+3+4+\cdots+K$

i	p	n
1	0+1=1	
2	1+2=2	
3	1+2+3=6	
4	1+2+3+4=	
⋮		
K		

assume

$p>n$ it stops

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

$\frac{K(K+1)}{2} > n \Rightarrow K^2 > n \Rightarrow K > \sqrt{n} \Rightarrow O(\sqrt{n})$ //

(4)

```
for (i=1; i<n; i=i*2)
{
    stmt;
}
```

Assume $i >= n$

$i = 2^k$

$2^k >= n$

$2^k = n$

$k = \log_2 n$ $O(\log n)$ //.

i
1
$1\times2=2$
$2\times2=2^2$
$2^2\times2=2^3$
⋮
2^K

(8)

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

→ n

```
for (j=0; j<n; j++)
{
    stmt;
}
```

→ n

$2n$

$O(n)$ //

9,

```
P=0
for (i=1; i<n; i = (i*2))
{
    p++;
}
```

→ $\log n$

```
for (j=1; j<P; j = j*2)
{
    stmt;
}
```

→ $\log P$

$O(\log(\log n))$

(5)
```
for (i=1; i<n; i=i*2)
{
    stmt;
}
```

n=8	n=10
i	i
1	1
2	2
4	4
	8
8 ×	16 ×

3

If i value is multiplied then $\log_n$ times.

$[\log n]$.

(6)
```
for (i=n; i>=1; i=i/2)
{
    stmt;
}
```

Assume $i<1$

i
n
$n/2$
$n/2^2$
$\frac{n}{2^3}$
⋮
$\frac{n}{2^k}$

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

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

$n = 2^k$

$k = \log_2 n$ $\quad O(\log_2 n)$ //.

(7)
```
for (i=0; i*i<n; i++)
{
    stmt;
}
```

$i*i<n$

$i \times 1 \geq n$

$i^2 \geq n$

$i^2 = n$

$i = \sqrt{n}$

```
for (i=0; i<n; i++)          — O(n)
for (i=0; i<n; i=i+2)        — 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)
```

Analysis of if & while

(1)
```
i=0;          -> 1
while (i<n)   -> n+1
{
  stmt;       -> n
  i++;        -> n
}
```

f(n) = 3n+2
= O(n)//

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

a
1
1×2=2
2×2=2^2
2^2×2=2^3
.
.
.
2^k

Terminate

a >= b

∴ a = 2^k

2^k >= b

k = log2 b

O(log n)//

(3)

```
i=1;
k=1;
while (k<n)
{
   stmt;
   k = k+i;
   i++;
}
```

i	k
1	1
2	1+1=2
3	2+2=4
4	...
	...
5	⋮
	m times.

$2+2+3+4+\cdots m$

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

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

$\Rightarrow m^2 = n$

$m = \sqrt{n}$

$O(\sqrt{n})$ //.

Time Complexity

↳ Two Methods.

* Counter Method
* Tabular Method

↳ other Methods.

* Step count Method
* Operation count Method.

Counter Method

→ We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways. In this we introduce a new variable, count.

```
Algorithm Sum(A, n)
{
    S := 0;
    for i := 1 to n do
        S := S + A[i];
    return s;
}
```

1 + n + n + 1 + 1

3 + 2n

T(sum(n)) = 2n + 3

T(sum(n)) ≥ 2n + 3

```
Algorithm Sum(A, n)
{
S := 0;                          → 1
count++;
for i := 1 to n do
{
  count++;  // for loop          → n times
  s := s + A[i];
}
count++;                         → n times
}
count++; // false for case       → 1
count++;                         → 1
return s;
}
```

Recursive

```
Algorithm RSum(A, n)
{
  if (n <= 0)
  return 0;
  else
  return RSum(A, n-1) + A[n];
}
```

```
Algorithm RSum(a, n)
{
  count++;          -> (1)
  if(n<=0)
  {
    count++;
    return 0;
  }
  else
  {
    count++;        -> (2)
    return RSum(A, n-1) + A[n];
  }
}
```

True

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

$TRSum(n-1)$

→ These Recursive formulas are referred to as recurrence relations.

→ One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} the right hand side until all such occurrences disappear.

$T(n) = 2$ if $n=0$

$= 2+T(n-1)$ if $n>0$

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

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

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

$\vdots$

$= 2 * n + T(n-n) = 2n + T(0)$

$T(n) = 2n+2$

Time Complexity Table Method

→ The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

→ First determining the number of steps per execution (s/e) of the statement and the total number of times (ie frequency) each statement is executed.

→ By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algo is obtained.

(1)

Statement	step/execution	freq	Total
Algorithm Sum(a,n)	—	0	0
{	—	0	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i];	1	n	n
return s;	1	1	1
}	—	0	0
			2n+3

(2)

Statement	S/e	freq n=0	freq n>0	Total n=0	Total n>0
Algo RSum(a,n)	—	0	0	0	0
{	—	0	0	—	—
if (n<=0) then	1	1	1	1	1
return 0	1	1	—	1	—
else	—	—	—	—	—
RSum(a,n-1)+a[n]	1+x	—	1	—	1+x
}	—	0	0	0	0
				2	2+x

x = tRSum(n-1)

(3)

Statement	s/e	freq	total
Algo Add (a,b,c, m,n)	0	—	0
{	0	—	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+ b[i,j];	1	mn	mn
}	0	—	0
			2mn+2m+1

Step Count Method

→ The step count is ~~used for~~ tells how the run time for an algorithm changes with changes in the instance characteristics.

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

	Cost	freq	Tot cost
int sum = 0;	C_1	1	C_1
for (int i=0; i<n; i++)	C_2	n+1	$C_2(n+1)$
sum += a[i];	C_3	n	$C_3 n$
return sum;	C_4	1	C_4

$$T_{sum}(n) = C_1 + C_2(n+1) + C_3 n + C_4$$

$$= C_2 n + C_3 n + C_1 + C_2 + C_4$$

$$T_{sum}(n) = (C_2 + C_3) n + C_1 + C_2 + C_4$$

Operation Count Method

```
int max (int a[], int n)
{
    max = a[0]
    for (i=1; i<n; i++)
    {
        if a[i] > max)
            max = a[i];
    }
    return max;
}
```

* Basic operation (time consuming)

Max execution time

1) Execution time of Basic operation (C_b)

2) Freq of basic op (n)

1) Comparison.

$b = 1$ till $n-1$

$C(n) = n-1$

$T(n) = b\,C(c(n))$

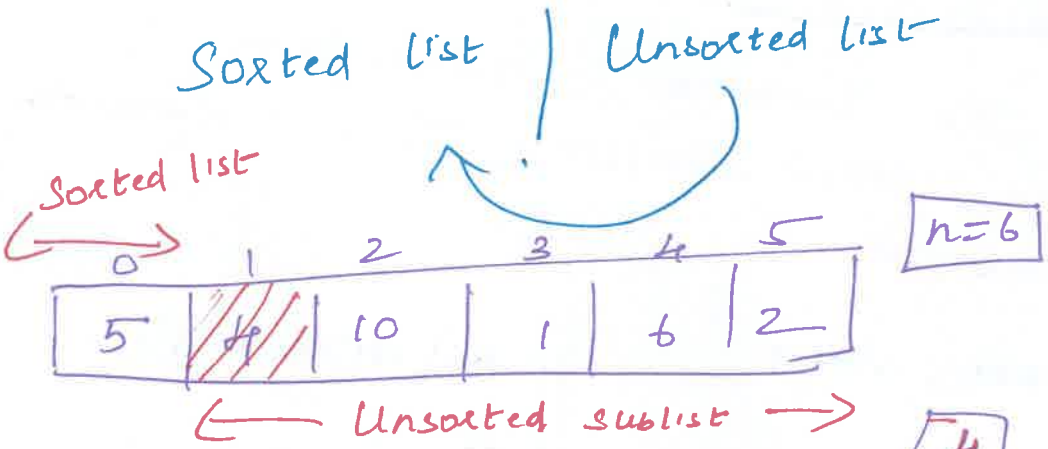
$= 1 \times n-1$

$= n-1$

$O(n)$ //

⇒ The efficiency

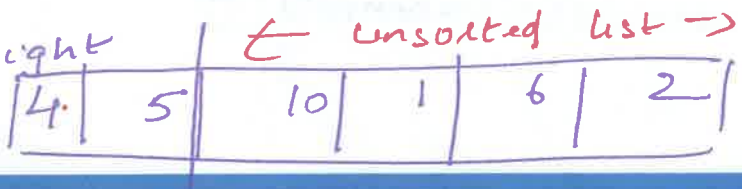
Lab 1: Insertion Sort - Line count, Operation Count

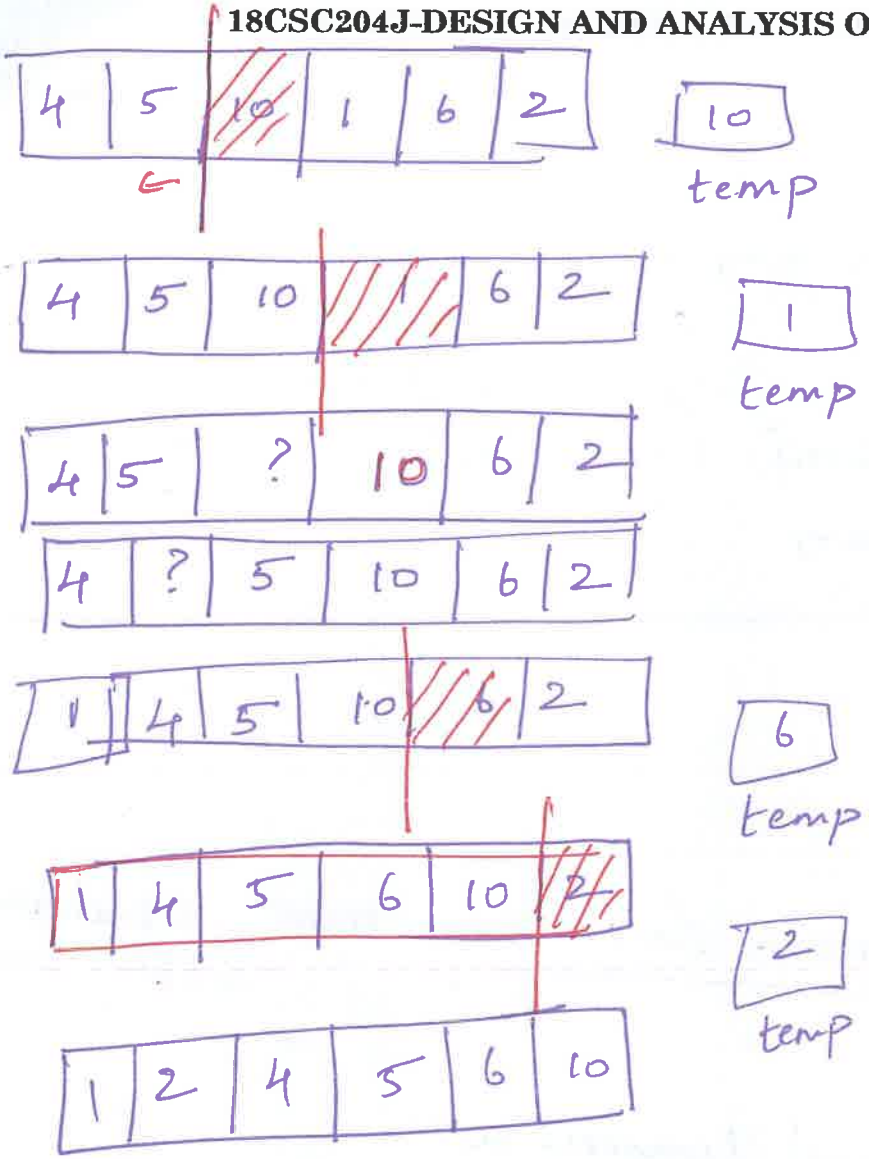


Step 1 → store it in temp

Step 2 Compare (sorted) 5 > temp

Step 3 Shift Right





One value from unsorted is taken and inserted in sorted sublist. This is known as Insertion sort.

```
for (i=1; i<n; i++)
{
   temp = a[i];
   j = i-1;
while (j>0 && a[j]>temp)
 {
   a[j+1] = a[j];
   j--;
}
a[j+1] = temp;
}
```

Insertion sort Time Complexity

	cost	freq	total cost
`for (i=1; i<n; i++)`	c_1	n	$c_1 n$
`{`			
`temp = a[i];`	c_2	$n-1$	$c_2(n-1)$
`j = i-1;`	c_3	$n-1$	$c_3(n-1)$
`while (j>0 && a[j]>temp)`	c_4	$\sum_{j=0}^{n} t_j+1$	$c_4 \sum_{j=0}^{n} t_j+1$
`{`			
`a[j+1] = a[j];`	c_5	$\sum_{j=0}^{n} t_j$	$c_5 \sum_{j=0}^{n} t_j$
`j--;`	c_6	$\sum_{j=0}^{n} t_j$	$c_6 \sum_{j=0}^{n} t_j$
`}`			
`a[j+1] = temp;`	c_7	$\sum_{j=0}^{n} t_j - 1$	$c_7 \sum_{j=0}^{n} (t_j - 1)$
`}`			

Worst case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left[\sum_{j=0}^{n} t_j + 1\right] + c_5 \sum_{j=0}^{n} t_j + c_6 \sum_{j=0}^{n} t_j + c_7(n-1)$$

$$= (c_4 + c_2 + c_3 + c_7)n + c_4 \sum_{j=0}^{n} t_j + 1 + (c_5 + c_6) \sum_{j=0}^{n} t_j - (c_2 + c_3 + c_7)$$

Worst case $= O(N^2)$

Average case $= (C_1+C_2+C_3+C_7)\, n/2 + C_4 \sum_{j=0}^{n/2} t_j + 1$

$+ (C_5+C_6) \sum_{j=0}^{n/2} t_j - (C_2+C_3+C_7)$

$= O(N^2)$

Best case $= (C_1+C_2+C_3+C_7)\, n - (C_2+C_3+C_7)$

$= O(N)$

1.5 Algorithm Design Paradigms.

Algorithm Techniques provides interest because

→ The provide templates suited to solve a broad range of diverse problems

→ They can be translated into common control and data structures provided by most-high-level languages

→ The time and space requirements of the algorithms which result can be precisely analyzed.

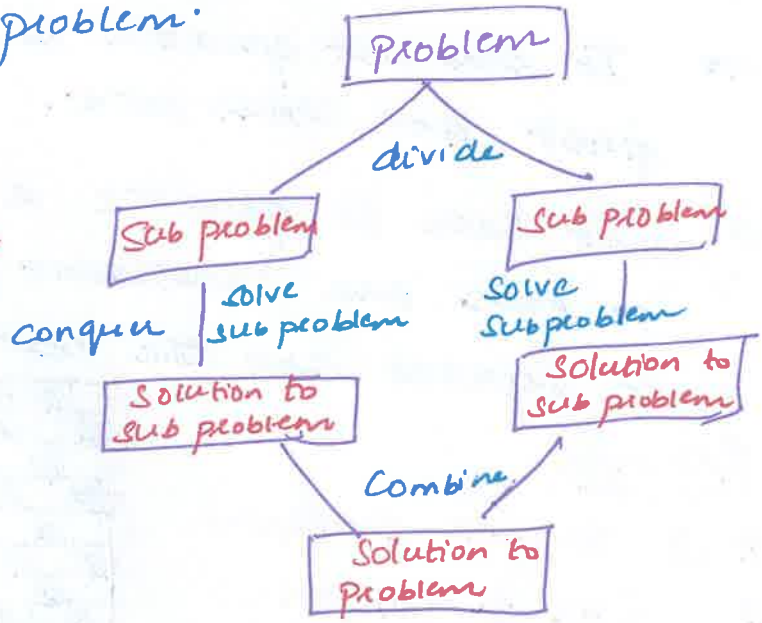
Various Algorithm paradigms are

* Divide and Conquer
* Dynamic Programming
* Backtracking
* Greedy Approach
* Branch & Bound

* Selection of the Paradigms depends upon the problem to be addressed.

Divide and Conquer

→ It divides the problem into smaller sub parts until these sub parts become simple enough to be solved, and the sub parts are solved recursively. Then the sub parts can be combined to give a solution to the original problem.



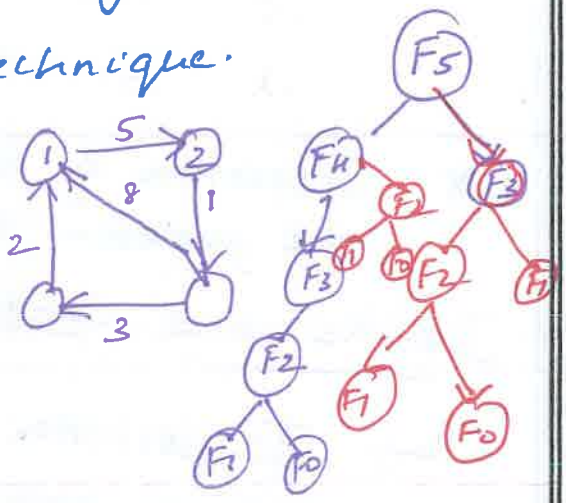
Examples:

* Binary Search
* Merge Sort
* Quick Sort

Dynamic Programming

→ It solves a given complex problem by breaking it into sub-problems and stores the results of sub-problems to avoid computing the same results again.

→ It is an optimization technique.



Examples

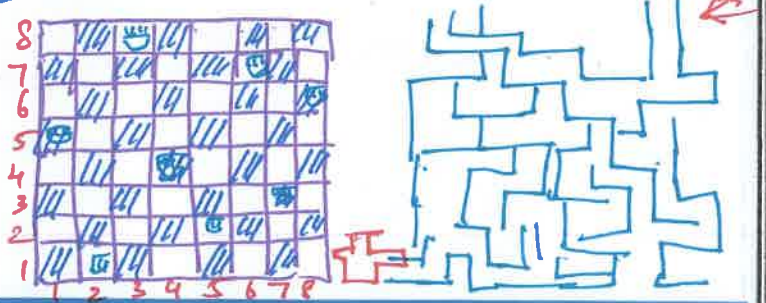
* All pairs of shortest path
* Fibonacci series.

Back tracking

→ It improves the time complexity of the exhaustive search technique if possible.

→ It does not generate all possible solutions first and checks later.

→ It tries to generate a solution & as soon as even one constraint fails, the solution is rejected and the next solution is tried.



Example

* 8 Queens problem
* Sum of subsets

Greedy Approach

→ Greedy Algorithm build a solution part by part, choosing the next part in such a way, that it gives an immediate benift.

→ This approach is mainly used to solve optimization problems

→ Finding the shortest path between two vertices using Dijkstra's algorithm.

36-20=16 (20)

16-10=6 (20) (10)

6-5=1 (20) (10) (5)

1-1=0 (20) (10) (5) (1)

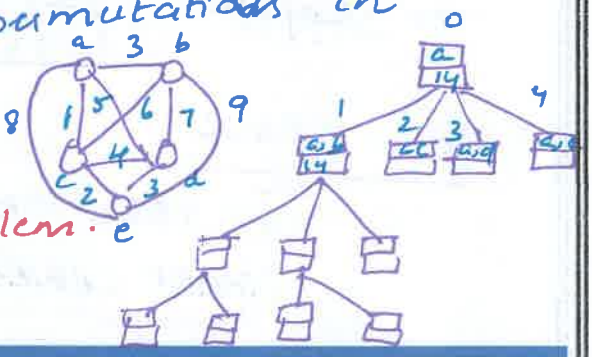
Examples

* Coin exchange problem
* Prim's
* Krustal's algorithm
* Traveling salesman problem
* Graph map coloring

Branch and Bound

→ It is generally used for solving combinatorial optimization problems.

→ These problems are typically exponential in terms of time complexity and may require exploring all possible pumutations in worst case.



Example:

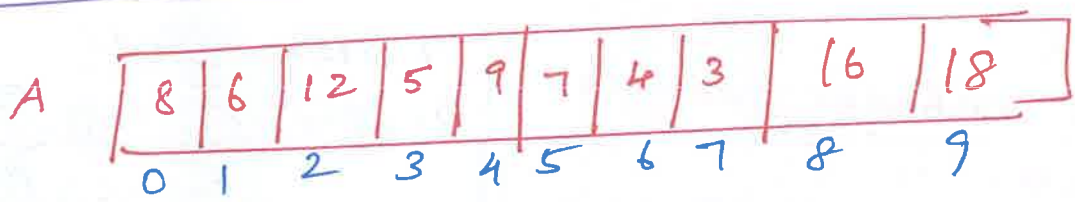
Travelling Salesman Problem.

1.6. Designing an Algorithm and its analysis, Best, Worst and Average case

Lets Learn with example

1. Linear Search
2. Binary search Tree.

(1) Linear search



→ List of Elements, I want to search

Key = 7 index 5

→ start checking one by one sequentially from left to right until the key element is found.

6 comparisons. to identify key element.

→ suppose Key = 20 Not in the list

Best Case

If the key element is present in the first index then it is Best case

Key = 8

Best case Time - 1 O(1)

Best B(n) = O(1)

Worst case.

→ Searching a key at the last index

Key = 18

→ Taking the maximum amount of time

Worst case Time = n

W(n) = O(n)

Average case:

→ $\frac{\text{all possible case time}}{\text{no: of cases}}$

→ may not be possible for all algorithms

→ similar to worst case

If the search of key element is present in first index, second and so on then

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

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

Applying Asymptotic Notations in this?

$B(n) = 1$ $\quad W(n) = n$

$B(n) = O(1)$ $\quad W(n) = O(n)$

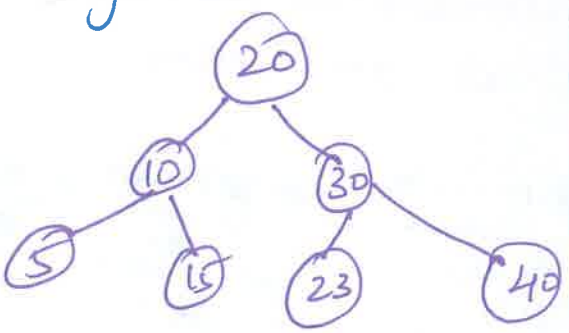
$B(n) = \Omega(1)$ $\quad W(n) = \Omega(n)$

$B(n) = \Theta(1)$ $\quad W(n) = \Theta(n)$

Note: There is no fixed notation to show that this is for best case, avg case or worst case

(2) Binary Search Tree

→ Searching the element in such a way that the elements are organized such that for any node all the elements smaller are in the left hand side and all the elements greater are in the right hand side.



Key = 15 → 3 comparison

log n (height)

Best case → Search root node

Best case Time – B(1) – B(n) = 1

Worst case – search leaf element

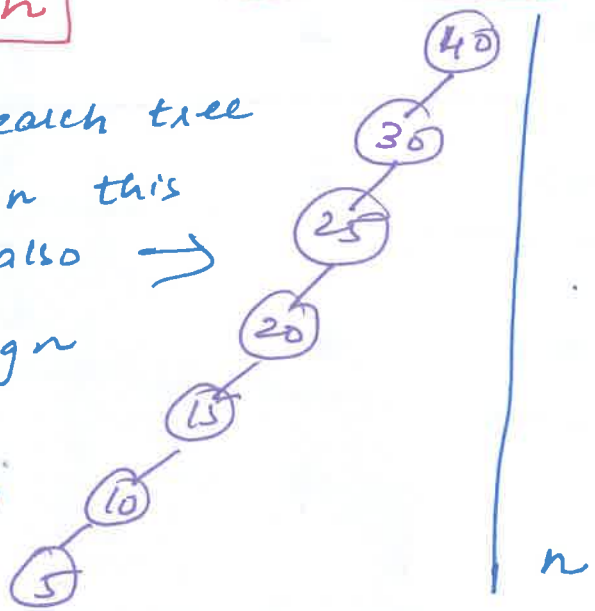
Worst case Time – W(n) = log n

W(n) = n

A Binary search tree can be in this format also →

min W(n) = log n

max W(n) = n



Note: Worst case depends on the height of the tree

1.7 Asymptotic Notations Based on Growth functions

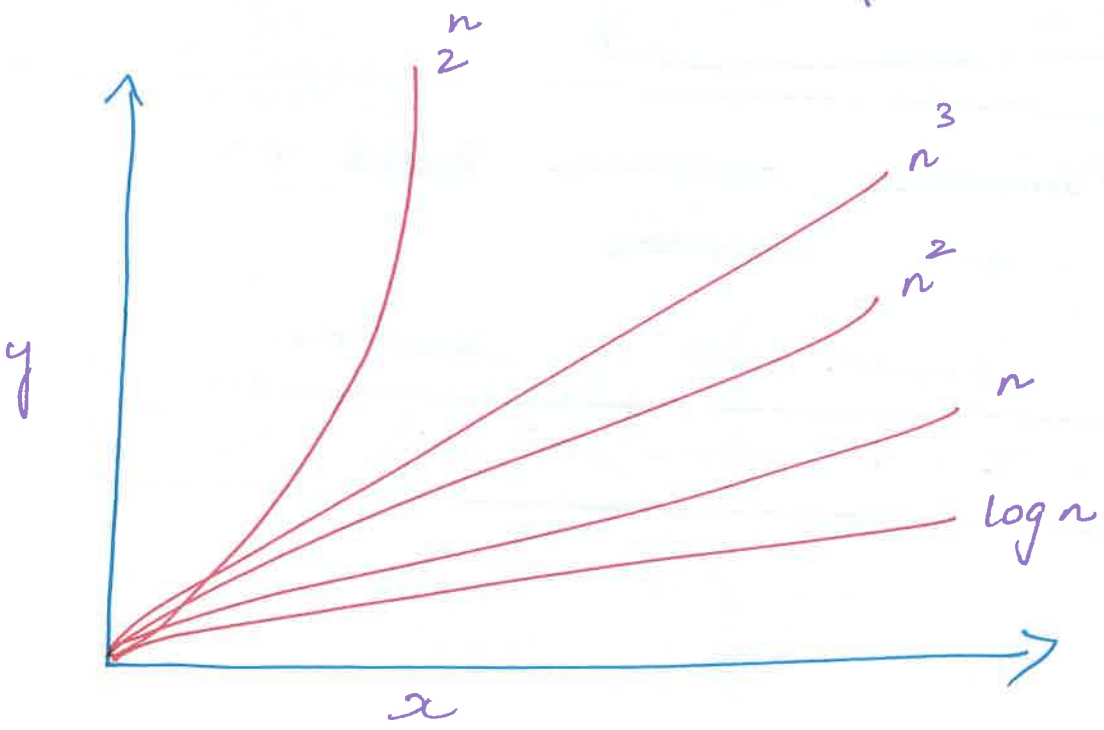
Compare class of functions.

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

$\log n$	n	n^2	2^n
0	1	1	2
$\log_2^2 = 1$	2	4	4
$\log_2^4 = 2$	4	16	16
$\log_2^8 = 3$	8	64	256
3.1	9	81	512

$n^{100} < 2^n$

$n^k < 2^n$



Asymptotic Notations Analysis

→ This topic is coming from Mathematics

→ The time required by an algorithm falls under three types.

Best case – Minimum time required for pgm execution

Average case – Average time required for pgm execution

Worst case – Maximum time required for pgm execution

→ Asymptotic analysis refers to defining the mathematical boundation / framing of its run-time performance

→ Derive the best case, avg case & worst case scenario of an algorithm

→ Asymptotic analysis is input bound
  ↳ Specify the behaviours of the algorithm when the input size increases.

→ Theory of approximation

→ Asymptote of a curve is a line that closely approximates a curve but does not touch the curve at any point of time.

Asymptotic Notations

→ Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

→ Asymptotic order is concerned with how the running time of an algorithm increases with the size of the input, if input increases from small value to large values.

1. Big-Oh notation (O)
2. Big-Omega notation (Ω)
3. Theta notation (Θ)
4. Little-oh notation (o)
5. Little-omega notation (ω)

Big-Oh Notation (O)

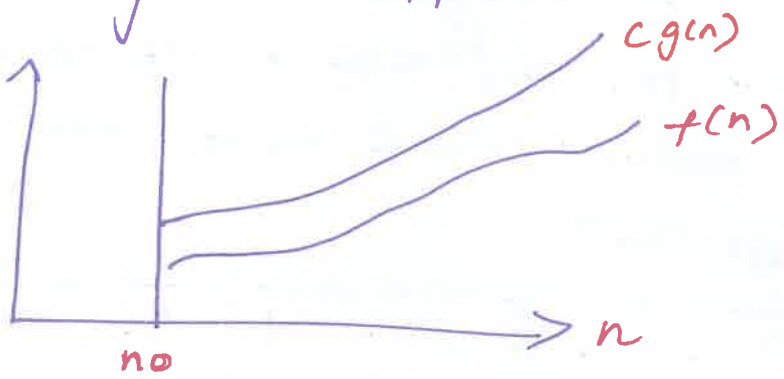
Definition: A function $f(n)$ is said to be in $O(g(n))$, denoted as $f(n) \in O(g(n))$, if $f(n)$ is bounded above by some constant multiple of $g(n)$ for all large n.

(i.e) if there exist some positive constant c and some non-negative integer n_0 such

that

$$f(n) \leq c^{*} g(n) \text{ for all } n \geq n_0$$

O big-oh upper bound



Example

$f(n) = 2n+3$

$2n+3 \leq 10\,n \quad n \geq 1$

put $n=1$

$5 \leq 10$ ✓

$2n+3 \leq 2n+3n$

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

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

$\leq 5n^2 \quad n \geq 1$

$f(n) = O(n^2)$ ✓

$f(n) = O(2^n)$ ✓

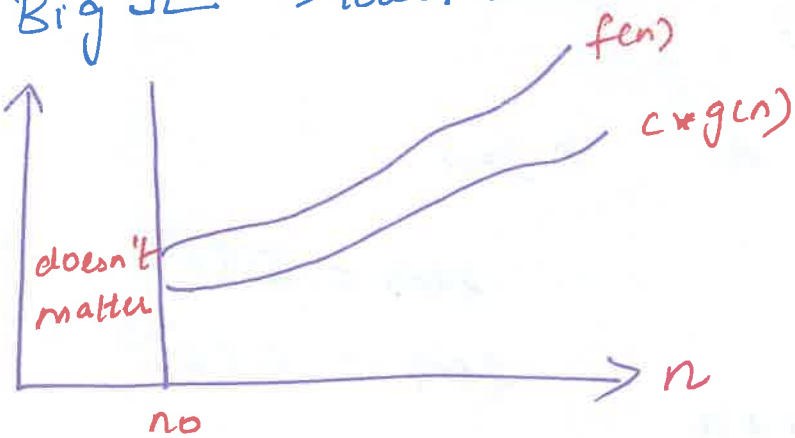
$f(n) = O(\log n)$ X

Big - Omega notation (Ω)

Definition: A function $f(n)$ is said to be in $\Omega g(n)$, denoted as $f(n) \in \Omega(g(n))$ if $f(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n. ie there exist some positive constant c and some non-negative integer no. such that

$$f(n) \geq c * g(n) \text{ for all } n \geq no$$

Big Ω → lower bound.



Example

$f(n) = 2n+3$

$2n+3 \geq 1 \times n \quad \forall n \geq 1$

$1 \quad 1$

$c \quad g(n)$

$\therefore f(n) = \Omega(n)$ ✓

$f(n) = \Omega(\log n)$ ✓

$f(n) = \Omega(n^2)$ ✗

$f(n)$

$2n+3 \geq 1 \times \log n$

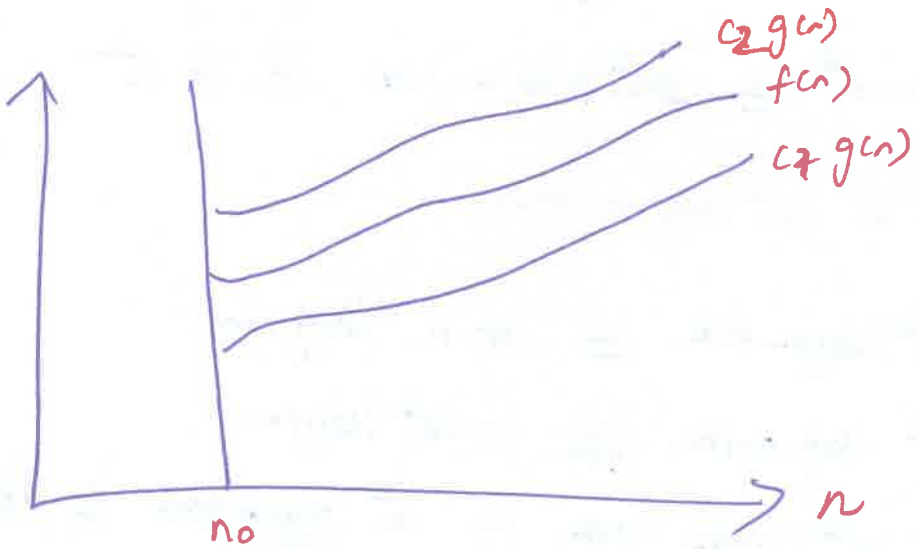
Theta Notation (Θ)

Definition: A function $f(n)$ is said to be in $\Theta(g(n))$, denoted $f(n) \in \Theta(g(n))$, if $f(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n. (ie), if there exist some positive constant c_1 and c_2 and some non-negative integer n_0 such that

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

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Theta Θ → Average bound.



Example

$f(n) = 2n+3$

$1 \times n \leq 2n+3 \leq 5 \times n$ $\quad \therefore f(n) = \theta(n)$

$c_1\ g(n) \quad f(n) \quad c_2\ g(n)$ $\quad f(n) = \theta(n^2)$ ✗

Note. $g(n)$ should be same.

Example Problems

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

1. $f(n) = 2n^2 + 3n + 4$

Big → $2n^2 + 3n + 4 \leq 9n^2 \quad n \geq 1$

$c\ g(n)$

Omega → $2n^2 + 3n + 4 \geq 1 \times n^2$

Theta → $1 \times n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$

2. $f(n) = n^2 \log n + n$

Big $\quad n^2 \log n + n \leq 10 n^2 \log n$

Omega $\quad n^2 \log n + n \geq 1 n^2 \log n$

Theta $\quad 1 n^2 \log n + n \leq n^2 \log n + n \leq 10 n^2 \log n + n$

Little -oh Notation (o)

Definition : A function $f(n)$ is said to be in $o(g(n))$, denoted $f(n) \in o(g(n))$, if there exist some positive constant c and some non-negative integer such that

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

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

Little -omega notation (ω)

The function $f(n) = \omega(g(n))$ iff

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

Properties of O, Ω and Θ

General Property:

→ If $f(n)$ is $O(g(n))$ then $a * f(n)$ is $O(g(n))$

Similar for Ω and Θ

Transitive Property

→ If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$; that is O is transitive.

Also Ω, Θ, o and ω are transitive.

Reflexive Property

→ If $f(n)$ is given then $f(n)$ is $O(f(n))$

Symmetric Property

→ If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$

Transpose Property

→ If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$

Orders of Growth

→ Measuring the performance of an algorithm in relation with the input size, n is called Order of growth.

→ Order of growth rates are:

* Constant
* Logarithmic
* Quadratic and
* Exponential

Order of growth of functions

Time complexity		Example
$O(1)$	constant	Adding to the front of a linked list

Time Complexity		Example
$O(\log N)$	log	Finding an entry in a sorted array
$O(N)$	Linear	Finding an entry in an unsorted array
$O(N \log N)$	n-log-n	Sorting n items by 'divide and conquer'.
$O(N^2)$	quadratic	Shortest path between two nodes in a graph
$O(N^3)$	Cubic	Simultaneous linear equations
$O(2^N)$	exponential	The Towers of Hanoi problem

Lab 2: Bubble Sort

Algorithm

```
Bubble sort (Arr[], n)
{
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {  if (arr[j] > arr[j+1])
       {  swap (arr[i], arr[j]);
    } } }
```

Example

	0	1	2	3	4
A	15	16	6	8	5

n=5

Two adjacent elements are compared and a[j] > [j+1] then swap.

pass 1

	15	16	6	8	5
	15	6	16	8	5
	15	6	8	16	5
	15	6	8	5	16

pass 2

	15	6	8	5	16
	6	15	8	5	16
	6	8	15	5	16
	6	8	5	15	16

pass 3

	6	8	5	15	16
	6	5	8	15	16
	6	5	8	15	16

pass 4

	6	5	8	15	16
	5	6	8	15	16
	5	6	8	15	16

Optimized Algorithm

	Cost	freq	Tot cost
for (~~int~~ i=0; i<n; i++)	C_1	n+1	$C_1(n+1)$
{ flag=0;			
for (j=0; j<n-1-i; j++)	C_2	n+1	$C_2(n+1)$
{			
if (arr[j] > arr[j+1])	C_3	n(n-1)	$C_3 n(n-1)$
{			
swap(arr[i], arr[j]);	C_4	$\sum_{j=0}^{n(n-1)} t_j$	$C_4 \sum_{j=0}^{n(n-1)} t_j$
}			
flag=1;			
}			
if (flag==1);			
break;			
}			

Worst case

$$T(n) = (n+1)C_1 + (n+1)C_2 + n(n-1)C_3 + \sum_{j=0}^{n(n-1)} t_j C_4$$

$$= (C_1 + C_2 - C_3)n + n^2 C_3 + C_4 \sum_{j=0}^{n(n-1)} t_j + (C_1 + C_2)$$

$$= n^2 C_3 + C_4 \sum_{j=0}^{n(n-1)} + (C_1 + C_2 - C_3)n + (C_1 + C_2)$$

$$= O(n^2) \text{ //.}$$

Average case

$$T(n) = (n/2+1)C_1 + (n/2+1)C_2 + n/2\,(n/2-1)C_3 + \sum_{j=0}^{n/2(n/2-1)} t_j C_4$$

Optimized Algorithm

	Cost	freq	Tot cost
`for (int i=0; i<n; i++)`	C_1	$n+1$	$C_1(n+1)$
`{ flag=0;`			
`for (j=0; j<n-1-i; j++)`	C_2	$n+1$	$C_2(n+1)$
`{`			
`if (arr[j] > arr[j+1])`	C_3	$n(n-1)$	$C_3 n(n-1)$
`{`			
`swap(arr[i], arr[j]);`	C_4	$\sum_{j=0}^{n(n-1)} t_j$	$C_4 \sum_{j=0}^{n(n-1)} t_j$
`}`			
`flag=1;`			
`}`			
`if (flag==1);`			
`break;`			
`}`			

Worst case

$$T(n) = (n+1)C_1 + (n+1)C_2 + n(n-1)C_3 + \sum_{j=0}^{n(n-1)} t_j C_4$$

$$= (C_1 + C_2 - C_3)n + n^2 C_3 + C_4 \sum_{j=0}^{n(n-1)} t_j + (C_1 + C_2)$$

$$= n^2 C_3 + C_4 \sum_{j=0}^{n(n-1)} + (C_1 + C_2 - C_3)n + (C_1 + C_2)$$

$$= O(n^2) \text{ //.}$$

Average case

$$T(n) = (n/2+1)C_1 + (n/2+1)C_2 + n/2\,(n/2-1)C_3 + \sum_{j=0}^{n/2(n/2-1)} t_j C_4$$

1.8 Recurrence Relation

Recurrence

→ Any problem can be solved either by writing recursive algorithm or by writing non-recursive algorithm

→ A recursive algorithm is one which makes a recursive call to itself with smaller inputs. We often use a recurrence relation to describe the running time of a recursive algorithm.

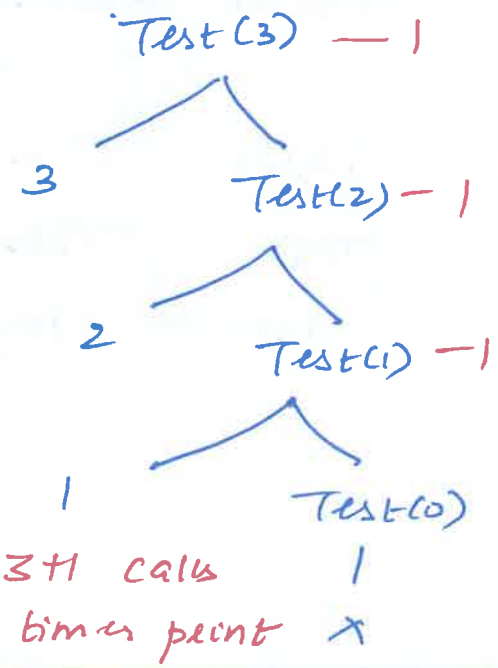
→ Recurrence relations often arise in calculating the time and space complexity of algorithms.

Example 1

n=3

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

Tree Method



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

Back Substitution Method

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

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

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

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

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

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

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

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

$\vdots$ k times

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

Assume $n - k = 0$

$n = k$

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

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

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

Example 2

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

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

Take Asymptotic Notation and remove constant

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

Back Substitution

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

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

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

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

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

Assume $n-k = 0$

$\therefore n = k$

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

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

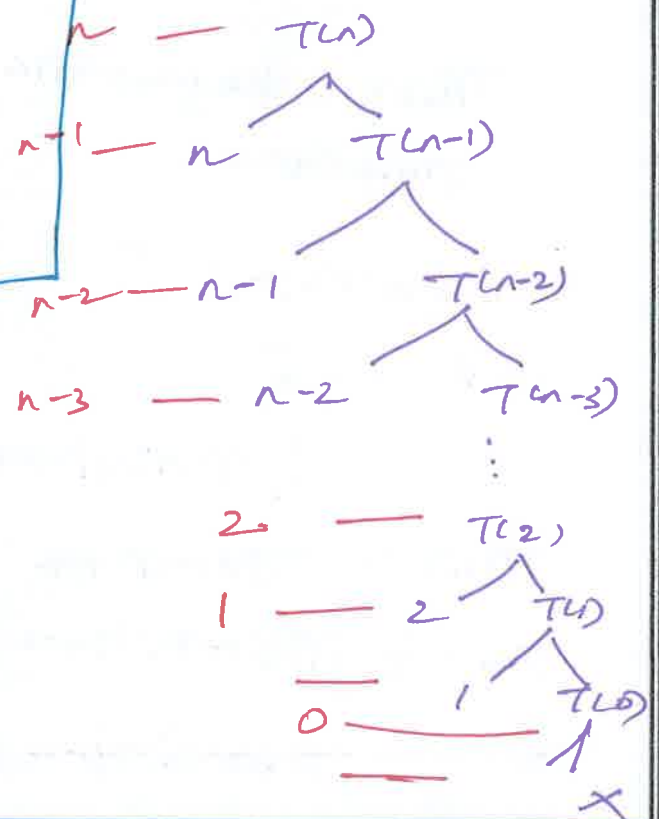
$$T(n) = T(0) + 1 + 2 + \cdots + (n-0) + n$$

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

$$= \frac{2 + n^2 + n}{2}$$

$O(n^2)$ //

Tree Method



$0 + 1 + 2 + 3 + \cdots n-1+n$

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

$O(n^2)$ //

Example 3

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

$\longrightarrow T(n)$

$\longrightarrow \log n$

$\longrightarrow T(n-1)$

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

```
for (i=1; i<n; i = i*2)
{
  Stmt;
}
```

i
1
$1 \times 2 = 2$
$2 \times 2 = 2^2$
$2^2 \times 2 = 2^3$
$\vdots$
2^k

$i = 2^k$

Assume $i \geq n$

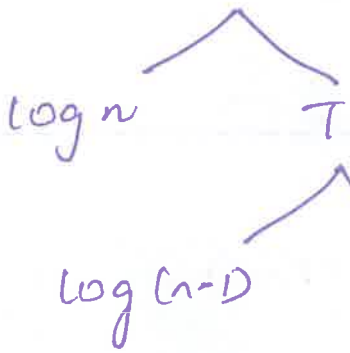
$2^k \geq n$

$2^k = n$

$k = \log_2 n$

Tree Method.

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



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

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

$\Rightarrow \log n!$

$\Rightarrow \log n^n$

$\Rightarrow \boxed{O(n \log n)}$

Back Substitution.

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

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

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

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

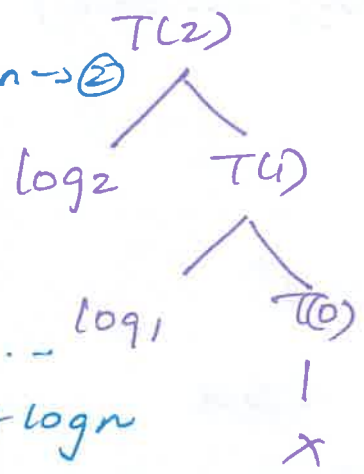
$\therefore n - k = 0$

$n = k$

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

$= 1 + \log n!$

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



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

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

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

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

Shortcut / Gate Questions

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

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

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

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

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

$T(n) = T(n-100) + n = O(n^2)$

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

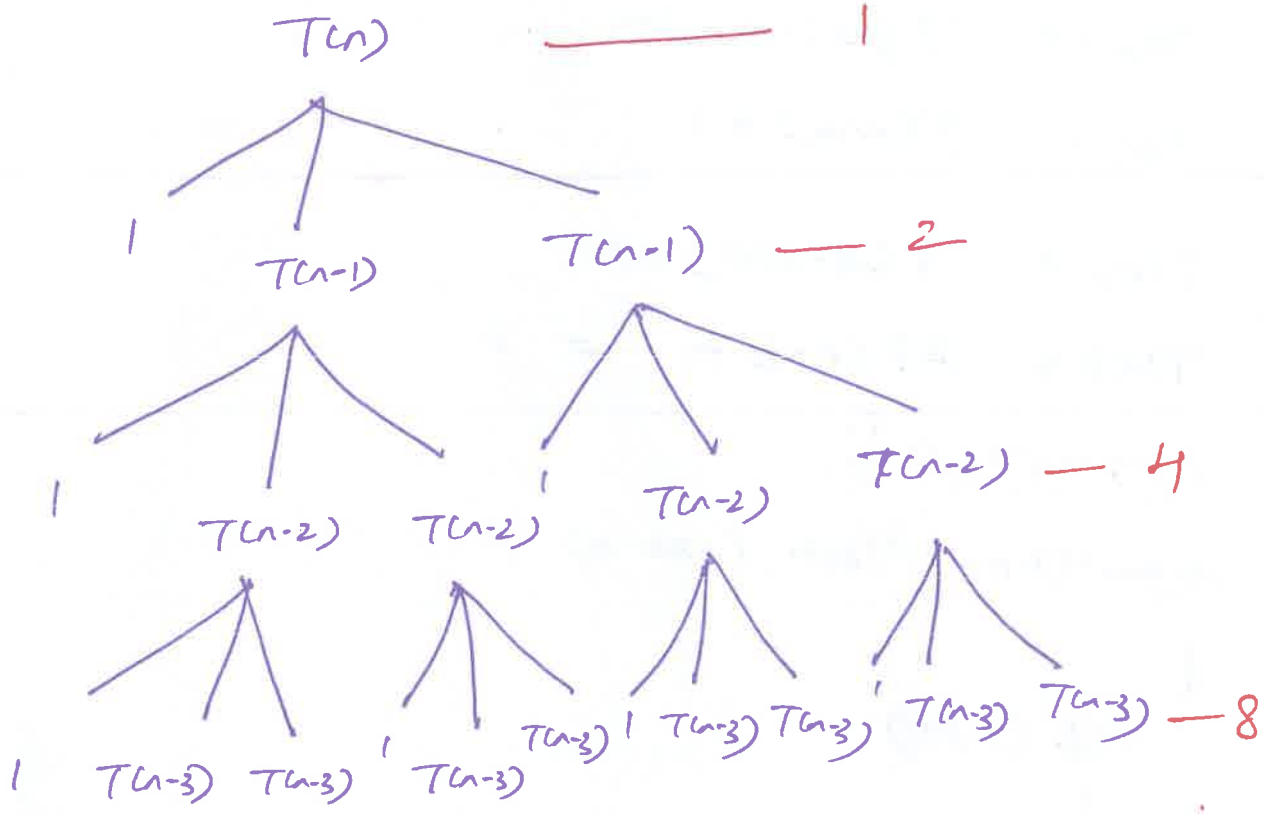
Example 4

```
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)
    }
}
```

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

Tree Method

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



$$1 + 2 + 2^2 + 2^3 + \cdots + 2^k = 2^{k+1} - 1$$

Sum of GP series

How?

$$a + ar + ar^2 + ar^3 + \cdots + ar^k = \frac{a(r^{k+1} - 1)}{r - 1}$$

$a = 1 \quad r = 2 \qquad \frac{2^{k+1} - 1}{2 - 1} = \boxed{2^{k+1} - 1}$

Assume $n-k=0$

$n=k$

$2^{n+1}-1 \quad O(2^n)$ //.

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

Back Substitution Method

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

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

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

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

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

$\vdots$

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

Assume $n=k$

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

$= 2^n \times 1 + 2^k - 1$ assume $n=k$

$= 2^n + 2^{n-1}$

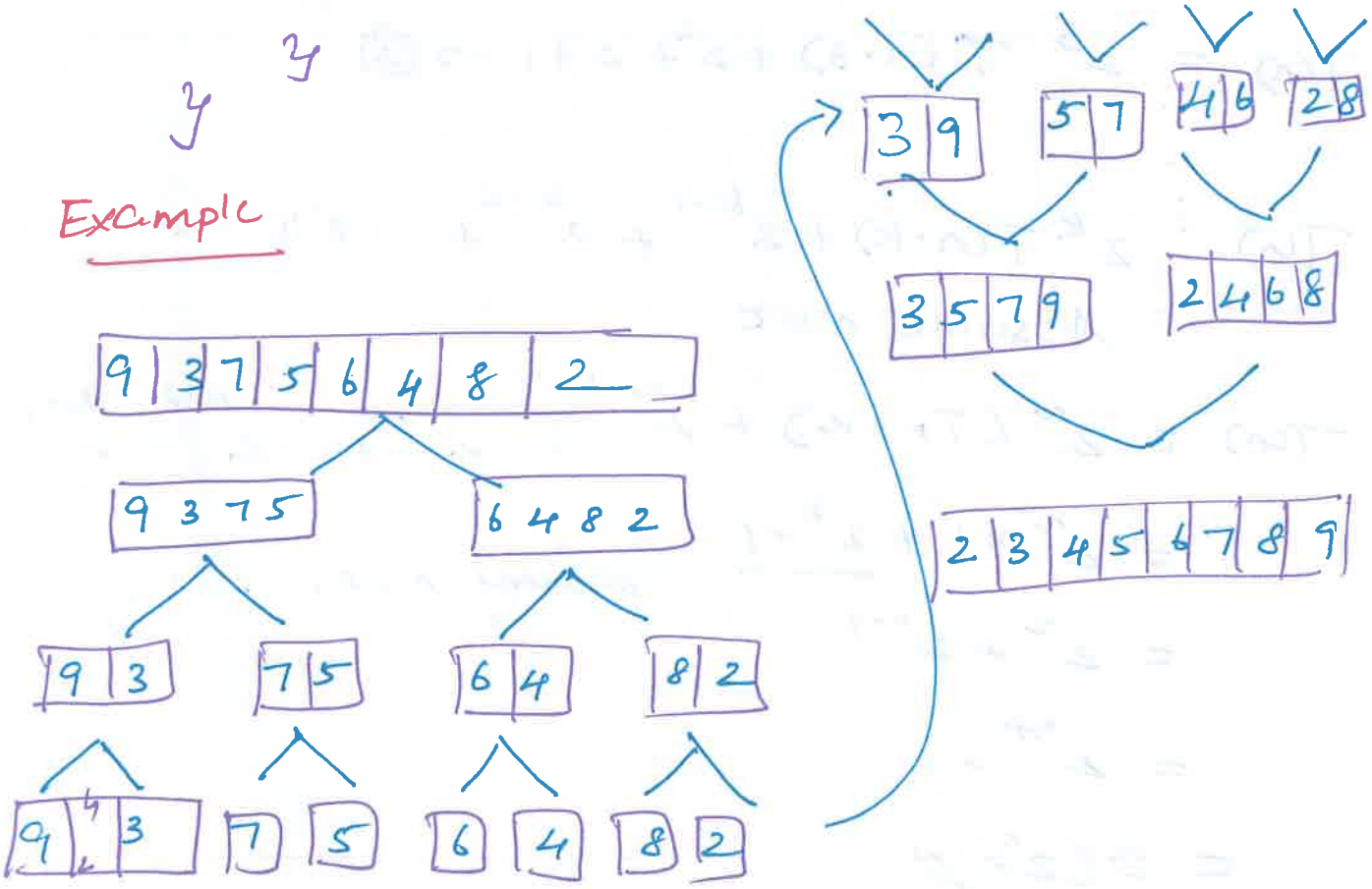
$= 2^{n+1} - 1$

$= O(2^n)$ //

Lab 3a: Merge Sort

```
Algorithm MergeSort ( arr, low, high)
{
    if (low < high)
    {
        mid = (low + high)/2 ;
        Mergesort ( arr , low, mid);
        MergeSort ( arr, midth, high);
        Merge ( arr , low, mid, high)
    }
}
```

Example



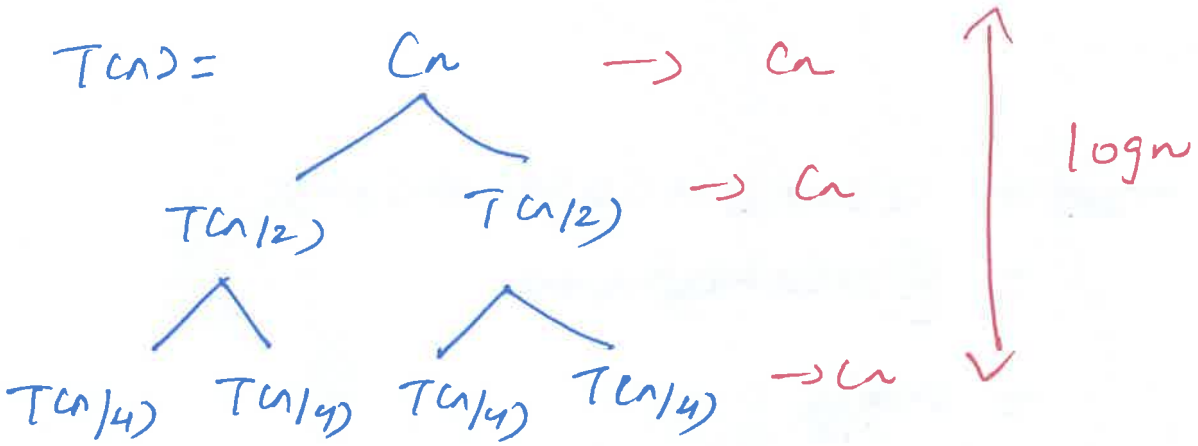
Time Complexity

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

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

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

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



Total cost = $n \log n + cn$

Worst case = $O(N \log N)$ //

3 b, Linear Search

```
Algorithm LinearSearch( a, i, k)
for (i=0; i<n; i++)          — c1      n+1     c1(n+1)
{
  if (key == a[i])           — c2      n       c2(n)
  {
    return i;                — c3      n       c3(n)
  }
}
return -1;
```

Worst case

$T(n) = c_1(n+1) + c_2(n) + c_3(n)$

$= (c_1+c_2+c_3)n + c_1$

$= O(n)$ //

Avg case: $T(n) = c_1(n/2+1) + c_2(n/2) + c_3(n/2)$

$= (c_1+c_2+c_3)n/2 + c_1$

$= O(n)$ //

Best case $T(n) = 1$

$O(1)$ //

UNIT-I INTRODUCTION - ALGORITHM

Operation Count

```
for (i=0; i<n; i++)
{
    if (key == a[i])      -> n
    return i;
}
```

worst case = O(n)//

Example

Key = 6