11/2/19

Unit-I :-

(1) • **Algorithm :-** An algorithm is defined as a collection of "unambiguous" instructions occuring in somespecific sequence and such an algorithm should produce output for given set of input in finite amount of time.*

(clear) @=1 x=2 x=string ambiguity

→ **Properties of Algorithm :-**

1) Algorithm should possess the following properties

* **Input :** The input of zero (or) more number of quantities should be given to the algorithm

2) **Output :** The Algorithm should produce atleast one quantity as an output.

3) **Definiteness :** Each instruction in algorithm should be specific and unambiguous (clear)

4) **Finiteness :** The algorithm should be finite i.e after finite no. of steps it should terminate

5) **Effectiveness :** Every step of algorithm should be feasible. (The prob should be solved)

→ Design of an Algorithm :-

Various steps are involved in the design of algorithm.

1) Understanding the problem.
2) Decision making on
   a) Capabilities of devices.
   b) choice for either exact or approximate Problem solving method.

c) Data structures.
d) Algorithmic strategies.

3) Specification of Algorithm.
4) Algorithmic verification
5) Analysis of Algorithm.
6) Implementation (or) coding of Algorithm.

1 → Understanding the problem :-

• This is the very, step (first) of designing of an algorithm

While understanding the problem statements read the problem description carefully and clarify the doubts regarding problem statement. After carefully understanding the problem statements find out what are the necessary inputs for solving the problem.

→ The input to the algorithm is called instance of the problem. It is very important to decide the range of inputs so that the boundary values of algorithm get fixed.

2 : Decision making :-

• Capabilities of devices :- It is necessary to know the computational capabilities of devices on which the algorithm is running.

• Choice for either exact (or) appropriate problem solving method :- If the problem needs to be solved correctly then exact algorithm is needed.

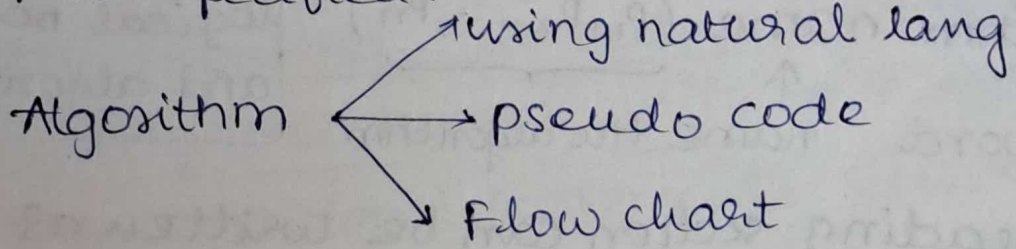
If the problem is so complex then the appropriate algorithm is needed to solve the problem.

- Data structures :- The choice of proper data structure is required before designing the actual algorithm.
- Algorithmic strategies :- They are also called as algorithmic techniques.

Ex:- 1) Divide and conquer. 2) Dynamic programing
3) Greedy Technique 4) Backtracking.

- Specifications of Algorithm :-

There are various ways by which algorithm can be specified.



- Algorithmic verification :-

After specifying an algorithm go for checking its correctness.

- Analysis of Algorithm :- While anaylzing an algorithm, the following factors are considered.

1) Time complexity 2) Space efficiency.
3) Simplicity 4) Generality.
5) Range of Input.

→ Pseudo code for expressing an algorithm:- (4)
also known as false code

SFt Algorithm is basically a sequence of instructions written in simple english language. The Algorithm is broadly devided into 2 sections:

1) Algorithm heading 2) Algorithm body

→ 1) Algorithm is a procedure consisting of heading and body. The heading consisting of the keyword "algorithm." and name of the algorithm and parameters list.

Syntax:-

Rules for writing Pseudo code:

Algorithm name ($P_1, P_2 \ldots, P_n$)

↑ Keyword ↑ name the algorithm

write parameters

Algorithm heading
It consists of name of algorithm, problem description Input and output.

Algorithm body
It consists of logical body and algorithm

2) The heading section can be written as follows

// problem description:

//Input:

//output:

3) The body of algorithm is written in various programming constructs like if, for, while (or some assignment statements.

4) The compound statements should be enclosed within { }.

5) Single line comments are written using "//" (Comment lines)

6) The identifies should begin by letter not by digit.

7) Using assignment operator "←" the assignment can be done.

variable ← Expression. x ← n [value of n is stored in x]

8) There are other types of operators such as (i) Boolean operators :- True or False.

(ii) logical operators :- And, or, not.

(iii) Relational operators :- >, <, >=, <=, ≠, =

9) The array indices are stored by using the "[ ]"

10) The inputing and outputing can be done by using read and write ✗

11) The conditional statements such as if then, if-then-else can be written as follows

If (condition) then statement

If (condition) then statement else statement

12) While statement can be written as

```
while (condition) do
{   statement 1
    :
    statement n
}
```

13) The general form of writing the for loop is for variable ← value, to value n do

{ stat 1 ... stat n }

value 1 → initialization.
value 2 - termination.

Ex:- for i ← 1 to n do
{ write(i)
}

14) The repeat-until statement can be written as

repeat
statement 1.
statement 2
⋮
statement n
until (condition)

15) The break statement is used to exit from inner loop.

Ex:- Write an algorithm to count the sum of n-numbers:-

Algorithm Sum (1, n)
// problem Description: This algorithm is for finding sum of given n numbers.
// Input: 1 to n numbers.
// Output: The sum of n numbers.

Algorithm heading

Algorithm body

result ← 0
for i ← 1 to n do
i ← i + 1
result ← result + i
return result.

write an algorithm for sorting the elements. (7)

```
Algorithm sort(a,n)
// Problem description: Sorting the elements
is ascending order.
//Input : An array in which the elements are
//sorted and
// n is the total number of elements in array.
//output : sorted array
  for i←1 to n do
  for j←i+1 to n-1 do
   { if (a[i] >a[j]) then
     {  temp←a[i]
        a[i]←a[j]
        a[j]← temp
     }
   }
   write ("List is sorted").
```

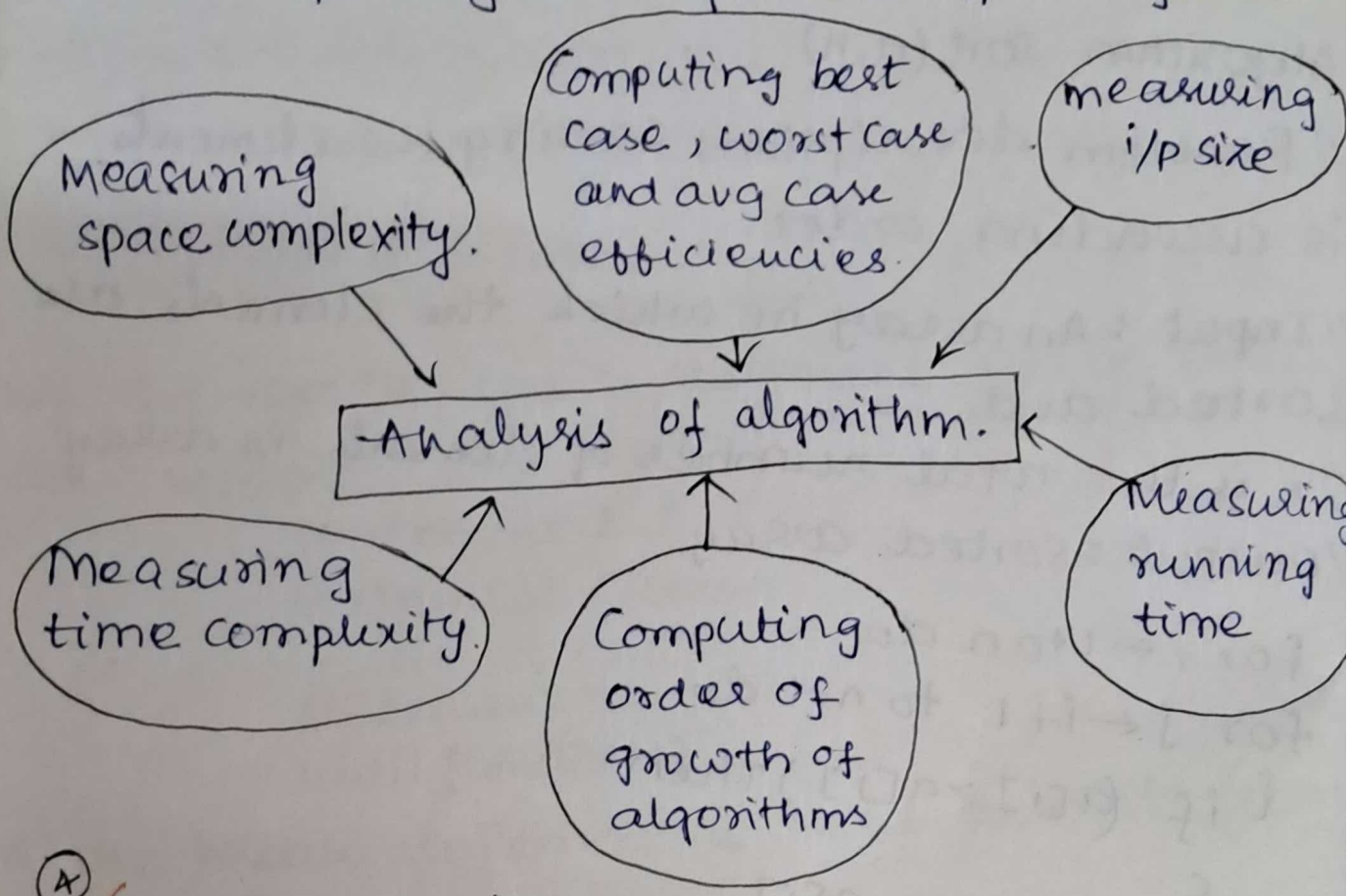
* Performance Analysis :-

The efficiency of an algorithm can be decided by measuring the performance of algorithm. The performance of an algorithm can be measured by computing 2 factors

1) Amount of time required by an algorithm to execute. (TC → Time Comp)

2) Amount of storage required by an algorithm. This is popularly known as

time complexity and space complexity. (8)



(4)
→ Space complexity :-
The space complexity can be defined as an amount of memory required by an algorithm to run. To compute the space complexity 2 factors are used : constant and instance characteristics. The space requirement S(P) can be given as : $S(P) = C + Sp.$
where C - is a constant, is fixed part and it denotes the space of inputs and outputs.
Sp - is the space dependant upon instance characteristics.

Ex:- Algorithm Add (a, b, c)

//problem Description : The algorithm computes the
//addition of 3 elements.
//Input : a, b and c of floating type
//output : Addition is returned.

return a+b+c

Space complexity : $S(P) = C + Sp$

$= C + 0$

$S(P) = 3. (a, b, c)$

Algorithm Add(x, n)
//problem description : The algorithm performs
//addition of elements in an array.
//Input : An array x and n is total number
//of elements in array.
//output : returns sum which is of data type
//float.

```
sum ← 0.0
for i ← 1 to n do
    sum ← sum + x[i]
return sum
```

n x 3.

Space complexity :-

$S(P) \geq (n+3)$

i.e, The n-space required for x[], one unit for n, one unit for i and one unit for sum.

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

- It is difficult to compute the time complexity in terms of physically clocked time because in multiuser system executing time depends on many factors such as 1) System load
2) Number of other programs running.
3) Instruction set used.
4) Speed of underlying hardware.

Therefore Time complexity is given interms of "frequency count".

Frequency count is a count denoting number of times of execution of statement.

- Measuring on Input size :-

If the input size is longer than usually algorithms runs for a longer time, Hence efficiency of an algorithm can be computed as a function to which input size is passed as a parameter.

- Measuring Running time :-

The time complexity is measured interms of a unit called frequency count. The time which is measured for analyzing an

algorithm is generally "running time".

→ From an algorithm :-

(a) First identity the important operation or logic of an algorithm. This operation is called basic operation.

The operation which is more time consuming is a basic operation in the algorithm. Such basic operation is in inner loop. It is not difficult to identify the basic operation of algorithm.

Problem statement	Input size	Basic operation
Searching a key element from least con n elements	list of n element	comparing key with every element of list.
Performing matrix multiplication	The 2 matrix with order n x n	Actual multiplication of elements into matrices
Computing GCD of 2 numbers	two number	Division.

* Then compute the total no of time taken by this basic operation.

* Using following formula the computing time can be obtained.

$$T(n) = C_{op}((n)$$

- $T(n)$: Running time of basic operation
- C_{op} : Time taken by basic operation to execute
- $((n)$: No of times the operation needs to be executed.

• Order Of Growth :-

Measuring the performance of an algorithm in relation with input size (n) called order of growth.

n	$\log n$	$n \log n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536

→ Step Count Method :-

(1)

Statement	s/e (Steps for execution)	frequency	total steps
1. Algorithm sum(a,n)	0	–	0
2. {	0	–	0
3. S:= 0.0; (initializing)	1	1	1
basic op 4. for i:=1 to n do	1	n+1	n+1
basic op 5. S:= S+a[i]	1	n	n
6. return s;	1	1	1
7. }	0	0	0
			2n+3

for i=1 i<=5

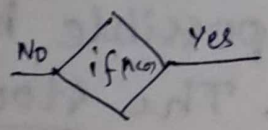
i=1 ✓ 1
2 ✓ 1
3 ✓ 1
4 ✓ 1
5 ✓ 1
6 (condition doesn't satisfy) ✓ 1

n+1

② statement

statement	s/e	frequency		total steps	
		n=0	n>0	n=0	n>0
Algorithm RSum(a,n)	0	-	-	0	0
{	0	-	-	0	0
if (n≤0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else return	-	-	-	0	0
RSum (a,n-1)+ a[n]	1+x	0	1	0	1+x
				0	0
}	0	-	-	2	2+x

1) if (n≤0)



③ statement

statement	s/e	frequency	total steps
1. Algorithm Add(a,b,c, m,n)	0	-	0
2. {	0	-	0
3. for i:=1 to m do	1	m+1	m+1
4. for j:=1 to n	1	m(n+1)	mn+m
5. c[i,j]:=a[i,j] +b[i,j]	1	mn	mn
6. }	0	-	0
		T.C:	2mn+2m+1

only logic is executed.

⑤ $\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}_{3\times3}$ $\begin{bmatrix}7&8&9\\4&5&6\\1&2&3\end{bmatrix}_{3\times3}$ → [1][1], [2][8], ... } 9 (m×n 3×3=9)

Asymptotic Notations:- (10M)

To choose the best algorithm it is req to check the efficiency of each algorithm. The efficiency of an algorithm can be measured by computing time complexity of each algorithm.

- Asymptotic Notation is the easiest way to represent the time complexity.
- By using Asymptotic Notations, the time complexity can be given as fastest possible is known as Best case of algorithm. The slowest possible is known as worst case of alg. The average possible is known as Average case. (time taken for the execution of algorithm)

Best	Fastest possible
worst	slowest possible
Average	Average time.

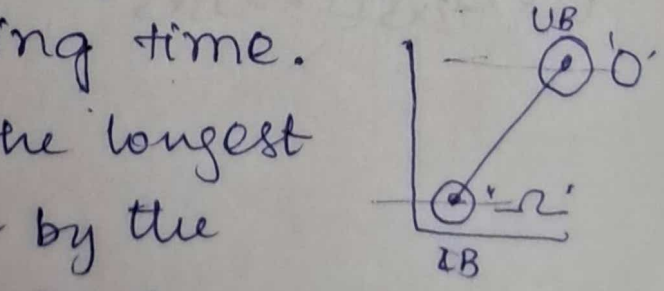
*** Asymptotic Notations.

→ The various notations are used as asymptotic Notations are "Ω" (omega), Θ (theta) and O (Big oh)
(Best, avg, worst)

BIG OH NOTATION (O) :- [Represents Worst case]

Remember:-

The Big oh notation is denoted by "O". It is a method of representing the upper bound of an algorithm running time. Using Big oh Notation the longest amount of time taken by the algorithm to complete will be given.



Definition:- Let $f(n)$ and $g(n)$ be two non-negative functions.

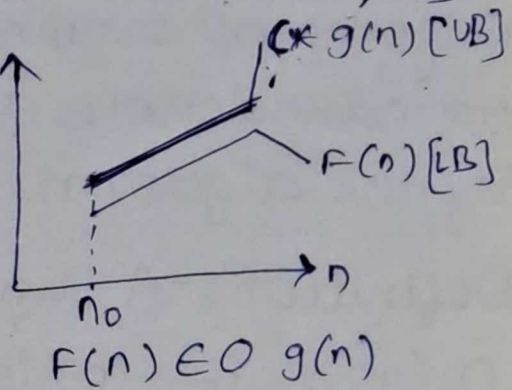
* Let n_0 and constant 'c' are 2 integers such that n_0 denotes some value of input and $n > n_0$, c is some constant such that $c > 0$.

$$F(n) \leq C * g(n)$$

Then $F(n)$ is big oh of $g(n)$.

It is also denoted as

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



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

Ex:- consider a function $F(n) = 2n+2$, Find the $g(n) = n^2$

constant c so that $F(n) \leq c * g(n)$.

For $n=1$ $F(n) = 2n+2$ $g(n) = n^2$ $F(n) \leq g(n)$

$= 2(1)+2 = 4$ $= 1$ ✗

for $n=2$ $F(n) = 2(2)+2$ $g(n) = 4$ ✗

$= 6$

For $n=3$ $F(n) = 2(3)+2$ $g(n) = 9$ ✓

$= 8$

For $n=4$ $F(n) = 2(4)+2$ $g(n) = 16$ ✓

$= 10$

→ For $n \geq 3$ the condition $f(n) \leq g(n)$ is satisfied.

So UB ⇒ $n \geq 3$. c can be any value.

∴ $F(n) \leq C * g(n)$ is satisfied.

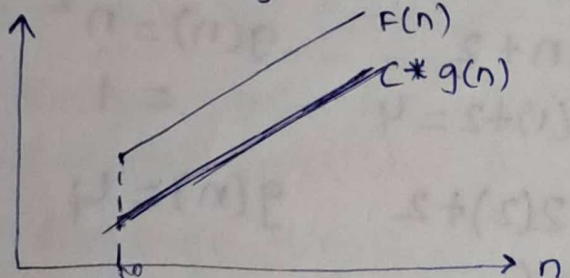
Omega Notation:- [Best case]

• Omega notation is denoted by "Ω". This is used to represent the lower bound of algorithms running time. Using 'Ω' notation we can denote shortest amount of time taken by the algorithm.

Definition:- A Function $F(n)$ is said to be in $\Omega(g(n))$ if $F(n)$ is bounded below by some positive constant multiple of $g(n)$ such that

$$F(n) \geq C * g(n) \quad \forall n \geq n_0$$

It is denoted by $F(n) \in \Omega(g(n))$



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

Ex:- $F(n) = 2n^2+5$ $g(n) = 7n$. Find lower bound.

$n+3$ n

$n=1$ $F(n) = 2(1)+5 = 7$ $g(n) = 7$.

$n=2$ $F(n) = 2(2)^2+5 = 2(4)+5 = 13.$ $g(n) = 7(2) = 14$

$n=3$ $F(n) = 2(3)^2+5 = 18+5 = 23$ $g(n) = 7(3) = 21$

$F(n) \geq g(n)$ $\forall n \geq 3$

lower bound $= 3$ $n \geq 3$

Ex:-
1) $2n+2$ n^2 Wo
2) $2n^2+5$ $7n$ Be
3) $2n+8$ $7n$ Av

$2(25)+5$ =

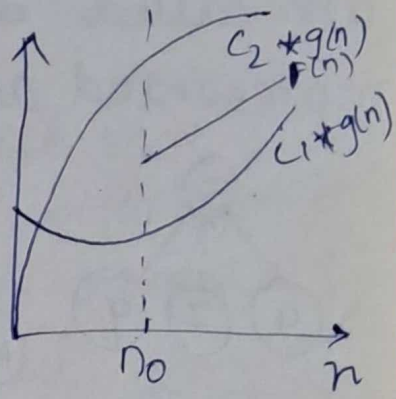
Theta Notation (Θ) :- [Avg case]

The theta notation is denoted by 'Θ'. By this method the running time is in between upper bound and lower bound.

Definition:- Let $F(n)$ and $g(n)$ be 2 non-negative functions. There are two positive constants namely c_1 and c_2 such that

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

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



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

Ex:- $F(n) = 2n+8$ $\quad g(n) = 7n$

$n=1$ $\quad F(n) = 2(1)+8 = 10$ $\quad g(n) = 7$. $\rightarrow F(n) > g(n)$

$n=2$ $\quad F(n) = 2(2)+8 = 12$ $\quad g(n) = 14$ $\quad F(n) < g(n)$

$n=3$ $\quad F(n) = 2(3)+8 = 14$ $\quad g(n) = 31$ $\quad F(n) < g(n)$

let $c_1 = 19$ $\quad c_2 = 5$ $\quad n=1$ $\quad c_1 * g(n) \Rightarrow 19 * 7(1) \leq 2(1)+8 \leq 5 * 7(1)$

$133 \leq 10 \leq 35$

→ little oh Notation :-

The little oh notation is denoted as 'o'.

Let $F(n)$ and $g(n)$ be the non-negative functions then $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{0}{0}$ → little 0

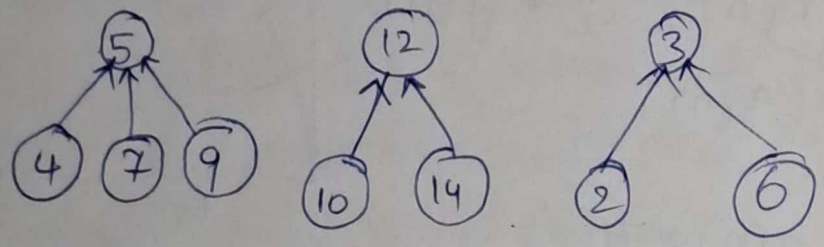
Such that, $f(n) = o(g(n))$, i.e $\frac{f \text{ of } n}{f(n)}$ is little oh of $g(n)$.

$f(n) = \frac{o}{0}(g(n))$ if and only if $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$

→ Disjoint sets:- (2M (or) 3M)

A disjoint set is a kind of Data structure that contains partitioned sets, these partition sets are seperate & non overlapping sets.

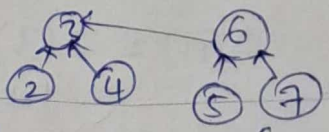
Ex:- $S_1 = \{5,4,7,9\}$, $S_2 = \{10,12,14\}$, $S_3 = \{2,3,6\}$ are called "disjoint sets". Each set can be represented as tree.



There are 2 operations that can be performed on the disjoint set DS these are union & find operations

1. DISJOINT SET UNION → If there exists 2 sets S_i and S_2 then $S_1 \cup S_2 = \{$all the elements from set S_i and $S_j\}$

Representation as trees



Ex:- $S_1 = \{2,3,4\}$, $S_2 = \{5,6,7\}$ $S_1 \cup S_2 = \{2,3,4,5,6,7\}$

2. Find (i):- For finding the element 'i' from given set, is done by this operation

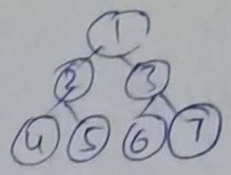
→ Algorithm for union and Find :-

```
Algorithm union (x1, x2)
{
    a[x1] = x2;
}
```

Time complexity is O(n)

```
Algorithm Find (x)
{
```

```
while (a[x] ≥ 0) do
    x = a[x];
    return x;
}
```

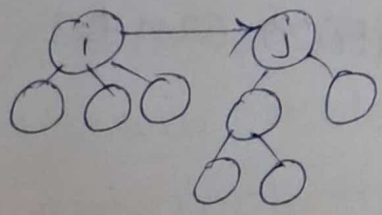


TO check one elem it has to check 2 times (n×n)

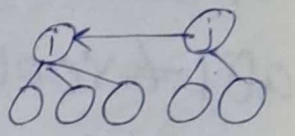
Time complexity $O(n^2)$

→ Weiguted union operation :-

Weighing rule :- If the tree with root i has the no of nodes less than a tree with root j then the tree with root j becomes parent of i otherwise root i is parent of root j



j is parent of i



i is parent of j

Algorithm wt-union (i, j)

```
{ /*problem Description: This algorithm generates a tree
using union operation
Input: Trees with root i & j
output:- performs union of all nodes. */
    a[i] ← count[i]
    a[j] ← count[j]
    temp ← a[i] + a[j] ;
    if (a[i] > a[j]) then
    {   a[i] = j;
        a[j] = temp;
```

i=2 j=3

2+3

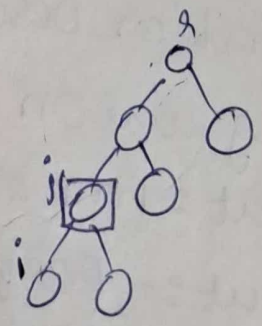
```
    }
    else
    { a[j] = i;
      a[i] = temp;
    }
  }
```

→ Collapsing Find operation :-

Collapsing rule :-

If j is a node on the path from i to its root and a[i] ≠ root[i] then set a[j] to root[i]. using collapsing rule, the find operation can be Performed.

```
Algorithm collapse_find(i)
{
//Problem Description: This algorithm collapse
//all nodes.
//from i to root node.
   root = i;
  while (i != root) do
  {
      temp = parent[i];
      parent[i] = root;
      i = temp;
  }
  return root;
}
```

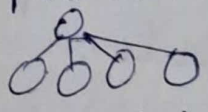


Divide and conquer Method :-

. Divide and conquer Method is a algorithmic strategy for solving the problems. In this strategy the big problem is broken into small subproblems and solution to these sub problems is obtained.

General Method :-

In D&C Method a given problem is divided into

1) smaller sub problems. 
2) These sub problems are solved independently.
3) combining all the solutions of sub problems into a solution of the whole problem. 
4) If the sub problem is large enough then re-apply the divide and conquer.
5) The generated sub problems are usually of same type as the original problem.

* The control abstraction for Divide and conquer

```
Algorithm DC(P)
{
    if P is too small then return solution
    of P.
    else
    {
        Divide (P) and obtain P1, P2 .... Pn
        where n≥1
        Apply DC to each sub problem
        return combine (DC(P1), DC(P2) .... DC(Pn));
    }
}
```

→ The computing time of the Divide and conquer is given by recurrence relation.

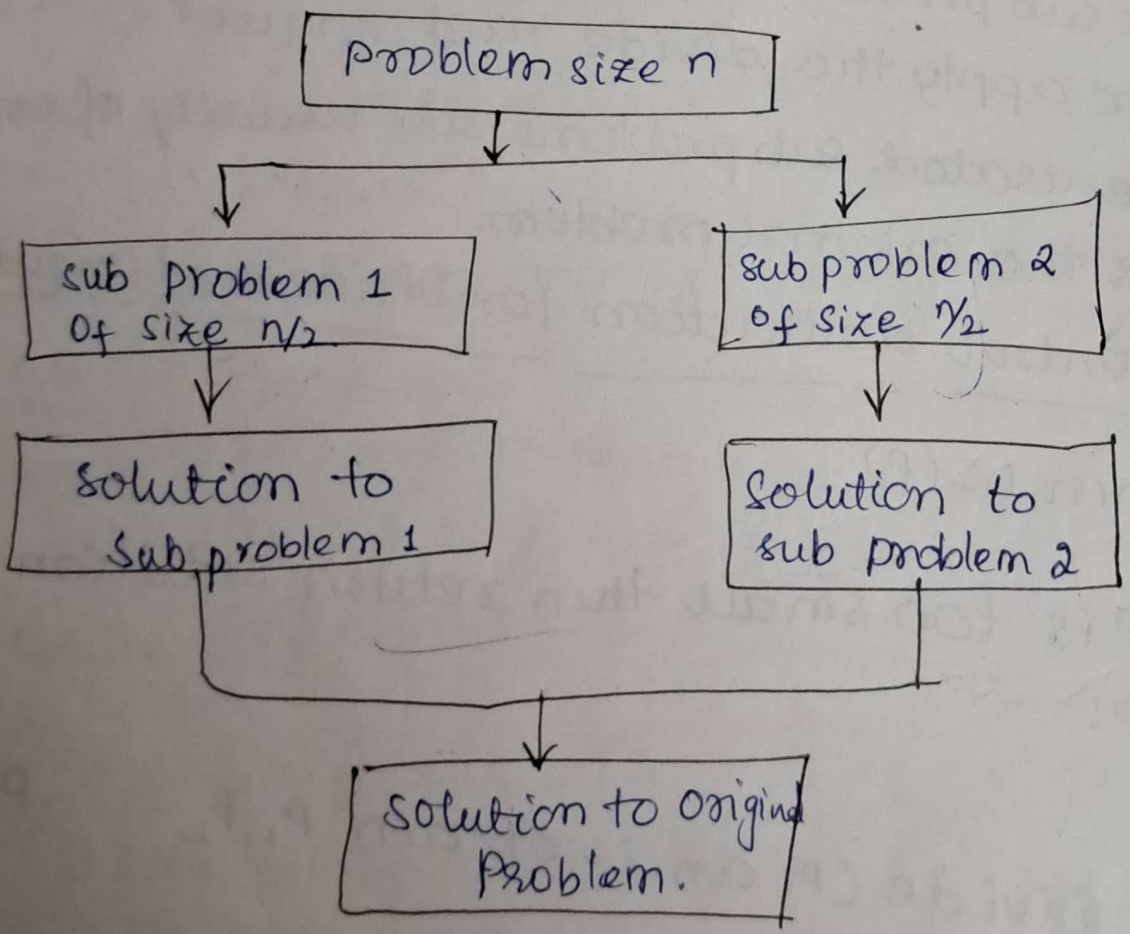
$$T(n) = \begin{cases} g(n) & \text{if } n \text{ - is small.} \\ T(n_1)+T(n_2)+\cdots+T(n_r)+F(n) & \text{when} \end{cases}$$

n is sufficiently large where T(n) is the time for divide and conquer of size n.

* The g(n) is the computing time required to solve small inputs.

* The F(n) is the time required in dividing Problem 'p' and combining the solutions of sub problems.

Divide and conquer technique :-



Applications of Divide and conquer :-

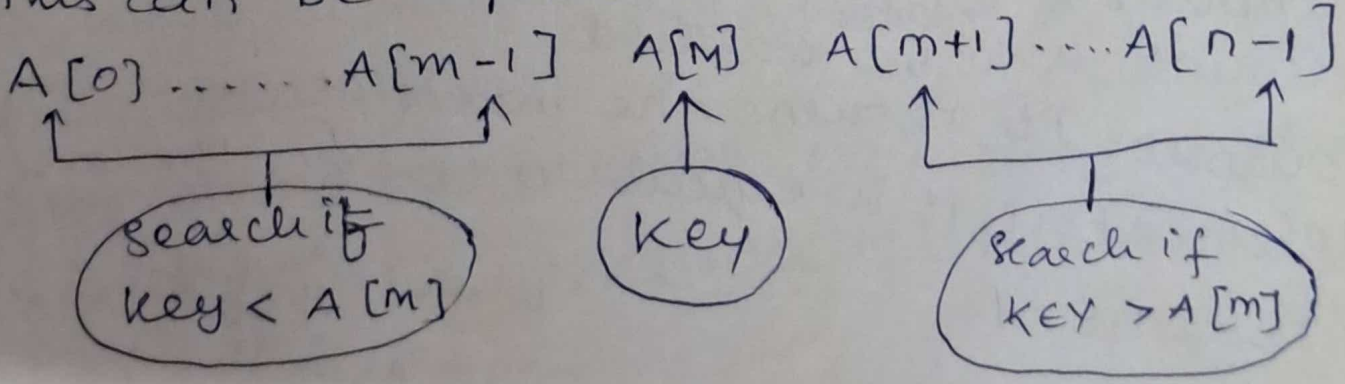
1. Binary Search 2. Quick Sort 3. Merge Sort
4. Strassen's Matrix Multiplication..

→ **Binary Search :-** Binary search is an efficient searching method. While searching the elements by using binary search method. The most essential thing is the elements in array should be sorted.

- An element which is to be searched from the list of elements sorted in an array is 0 to n-1 is called "key" [A[0.....n-1]] ← (sorted array)
- Let A[m] be the mid element of array A. Then there are 3 conditions that needs to be tested while searching the array using this method.

1. If KEY = A[m] then desired element is present in the Mid/list.
2. else, if key < A[m] then search the left sublist
3. else, if key > A[m] then search the right sublist

This can be represented as

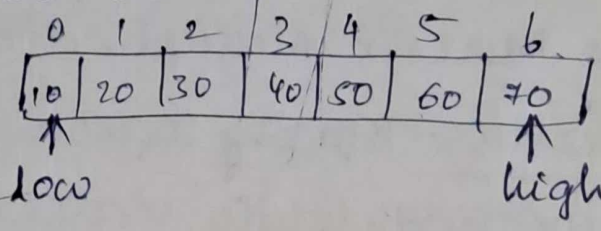


Ex:- A list of elements in array A is

0	1	2	3	4	5	6
10	20	30	40	50	60	70

A[0] = 10

A[6] = 70



KEY = 60

M = (low + high)/2

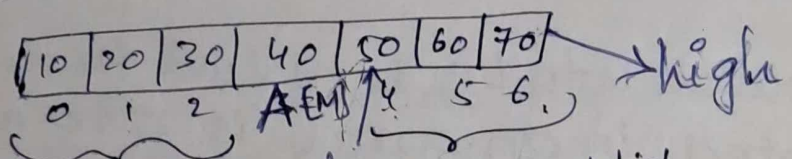
= (0+6)/2 = 3

M = 3

A[M] = 40

[A[m] = 40] < (KEY = 60)

∴ search in the right sub list.



M = (low + high)/2 = (4+6)/2 = 5

A[M] = 60 A[M] = 60 = KEY.

∴ The element is present in list.

→ Algorithm :-

Algorithm BinSearch (A[0....n-1], KEY)

//problem Description: This algorithm is for //searching the ele using binary search method.

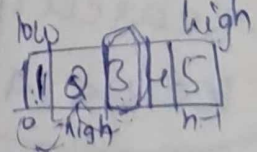
// Input: A → array A from which the "KEY" sorted

//element is to be searched

//output: It returns the index of an array

//element if it is equal to "KEY" otherwise

//returns -1



low ← 0

high ← n-1

while (low < high) do

{

(A, 3) (0 < n-1)

{

m = (low + high)/2

0+1/2 = 0.5

0+4/2 = 2

mid = 2

```
m ← (low + high)/2
if (KEY == A[m]) then
return m
else if (KEY < A(m)) then
high ← m-1
else
low ← m+1
}
return -1
```

*Time complexity is $O(\log_2 n)$

Advantages of Binary search:-

Binary Search is an optimal searching algorithm using which can search the desired element very efficiently.

Disadvantages:-

1) This algorithm requires the list to be sorted, then only this method can be applicable.

Applications:-

This is an efficient searching method and is used to search the desired record from database.

2) For solving non-linear equations with one unknown, this method is used.

Time complexity:-

BEST CASE	AVERAGE CASE	WORST CASE
$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

Quick Sort :- (v)

Quick sort is a sorting algorithm that uses the divide and conquer strategy.
In this method division is dynamically carried out.

- There are 3 steps in quick sort :-

1) **Divide** : split the array into 2 sub-arrays that each element in left sub array is less than (or) equal to the pivot element (or) middle ele. and each element in right sub array is greater than the pivot element. The splitting of array into 2 arrays is based on pivot ele.

2) **Conquer** :- Recursively sort the 2 sub arrays

3) **combine** :- combine all the sorted elements in a group to form a list of sorted elements.

A[0], ... A[m-1], A[m], A[m+1] .... A[n-1]

Left pivot — pivot — greater than Pivot

The quick sort algorithm is performed using two important functions.

1. Quick
2. partition.

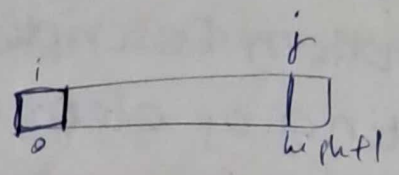
Algorithm Quick (A[0....n-1], low, high)
//problem Description: This algorithm performs
//sorting of elements given in Array[0....n-1]
//Input: An array A[0....n-1] in which unsorted
//elements are given. The low indicates left most
//element in the list and high indicates the right
most element in the list.
//output: creates a subarray which is sorted in
//ascending order.

```
if (low < high) then
m ← partition(A[low....high])
Quick(A[low.....m-1])
Quick(A[m+1,....high])
```

. The partition algorithm peforms arrangement of the elements in ascending order.
. The recursive quick is routine for dividing the list into 2 sub lists.

→ Algorithm Partition (A[low....high])
//problem Description: This algorithm partitions
//the subarray using the first element as
//pivot element.
//Input: A sub array A with low as left most index
//At the array and high as the right most
//index of the array.
//Output: The partitioning of array A is
//done and pivot occupies if proper position.

//And the right most index of the list is returned.




```
Pivot ← A[low]
i ← low
j ← high+1
while (i<=j) do
{
    while (A[i]<=pivot)
      i ← i+1
    while (A[j] >= Pivot)
      j ← j-1
    if (A[i] > pivot) && A[j] < pivot) then
      swap(A[i], A[j])
}
if (i >= j)
swap (A[low], A[j])
return j ;
```

Ex:- 50 | 30 | 10 | 90 | 80 | 20 | 40 | 70

low = 0, high = 7

pivot / i=0 ; j=8 [i+1=7+1=8]

STEP 1 :- i=0 < j=8

A[i] = 50 pivot = 50 (i = i+1)

50 30 10 90 80 20 40 70

Pivot i=1 j=8

STEP 2 :- i < j | A[i] = 30 | i = i+1

1 < 8 | pivot = 50

50 30 10 90 80 20 40 70

Pivot i=2

STEP 3 :- i < j | A[2] = 10 | i = i+1

2 < 8 | Pivot = 50

50 30 10 90 80 20 40 70
pivot :- i=3 j=8

STEP i<j A[i]=A[3]=90 | A[i] ≥ pivot.
3<8 pivot = 50

check A[j] ⇒ A[8] = 70
A[j] = 70 >= pivot(50) j=j-1

50 30 10 90 80 20 40 70
i=3 j=7.

STEP-5:- i<j A[i] = 90 > pivot
3<7 A[j] = 40 < 50
Swap(90, 40)

50 30 10 40 80 20 90 70
i=3 j=7

i<j
A[i] < pivot | i=i+1
40 50

50 30 10 40 80 20 90 70
i=4 j=7

i<j
A[i] > pivot | A[j] = 90 > pivot.
80 > 50 j=j-1

50 30 10 40 80 20 90 70
i=4 j=6

i<j
4 7
A[i] = 80 > 50
A[j] = 20 < 50 swap.

50 30 10 40 20 80 90 70
i=4 j=6

i<j A[i]=20<50
4<6 i=i+1

50 30 10 40 20 80 90 70
j=6
i=6

A[i]=80>50
A[j]=80>50
j=j-1

50 30 10 40 20 80 90 70
low j i

∧ i>=j So swap low with A[j]

20 30 10 40 50 80 90 70
i pivot < j >

i<j 20<=20
i=i+1

Time complexity:-

Best case	Avg case	Worst case
$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$

→ Merge Sort:- the merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method, division is carried out dynamically. Merge sort on an input array with n elements consist of 3 steps

1) Divide:- Partition the array into 2 sub lists S_1 and S_2 with $n/2$ elements each.

2) conquer :- Then sort sublist S_1 and sublist S_2

3) combine :- Merge S_1 and S_2 into a unique sorted group.

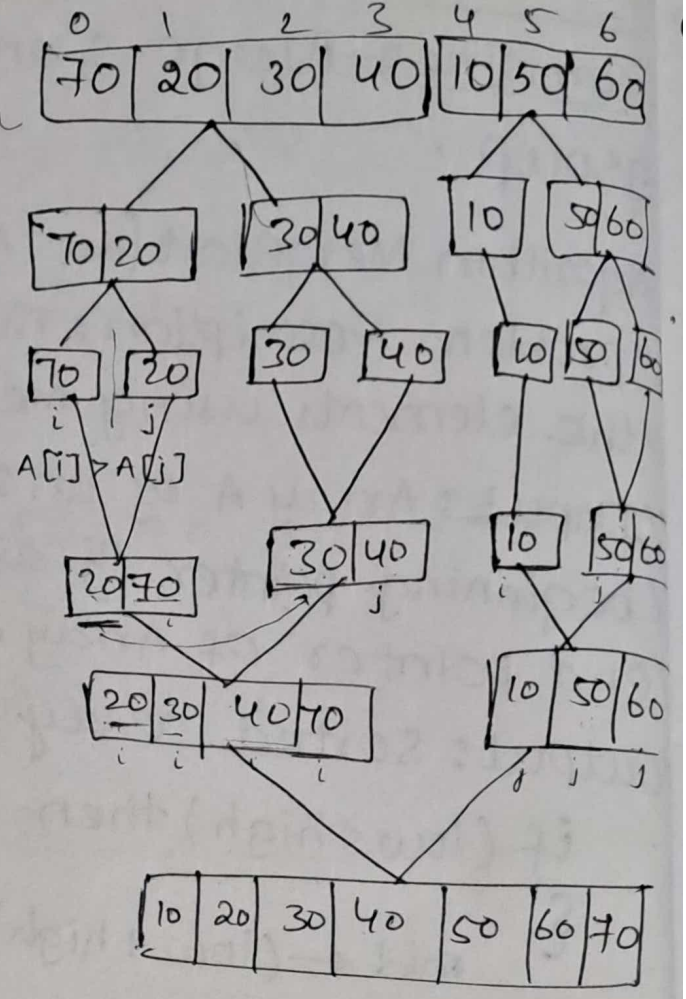
```
Algorithm Mergesort(int A[0....n-1], low, high)
//problem Description: This algorithm is for sorting
//the elements using Merge sort.
//Input: Array A of unsorted elements, low
//beginning pointer of array A and high as
//end pointer of Array A.
//Output: sorted array A[0.....n-1]
    if (low<high) then
    {
        mid ← (low + high)/2
        mergesort (A, low, mid)
        mergesort (A, mid+1, high)
        Combine (A, low, mid, high)

    }

Algorithm Combine (A[0. ..n-1], low, mid, high)
{
        k ← low;
        i ← low;
        j ← mid+1;
        while (i<=mid and j<=high) do
        {
            if (A[i] <= A[j]) then
            { temp[k] ← A[i]
            } i ← i+1
              k ← k+1 ;
```

```
    else
    {  temp[k] ← A[j]
       j ← j+1 ;
       k ← k+1 ;
    }
}
while (i<=mid) do
{   temp[k] ← A[i]
    i ← i+1 ;
    k ← k+1 ; // for array
}
while (j<=high) do
{   temp[k] ← A[j]
    j ← j+1 ;
    k ← k+1 ;
}
```

Ex:- n/2.



→ Time complexity :-

Best case	Average case	Worst case
$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

→ Strassen's Matrix Multiplication →

To accomplish 2x2 Matrix multiplication there are total 8 multiplications and 4 additions are required. The time complexity of Matrix Multiplication is

order of n^3 $[O(n^3)]$

Strassen shown that 2x2 Matrix Multiplication can be accomplished in 7 Multiplications and 18 additions con subtractions.

. With this the time complexity reduced to $O(n^{2.81})$.

→ to divide and conquer approach can be used for implementing strassen's matrix multiplication.

Divide :- Divide the matrices into sub matrices $A_0, A_1, A_2 \ldots\ldots$

Conquer :- Use a group of matrix multiply equation

Combine :- Recursively multiply submatrices and get the final result of multiplication after performing required additions & subtraction

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$S_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$

$S_2 = (A_{21} + A_{22}) \times (B_{11})$

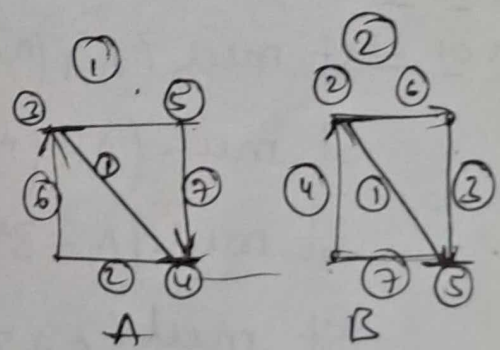
$S_3 = A_{11} \times (B_{12} - B_{22})$

$S_4 = A_{22} \times (B_{21} - B_{11})$

$S_5 = (A_{11} + A_{12}) \times B_{22}$

$S_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$

$S_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$



$C_{11} = S_1 + S_4 - S_5 + S_7$

$C_{12} = S_3 + S_5$

$C_{21} = S_2 + S_4$

$C_{22} = S_1 + S_3 - S_2 + S_6$

1			1
4	3	2	3
5	5	4	2
7			6

$C_{11} = S_1 + S_4 - S_5 + S_7 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) + A_{22} \times (B_{21} + B_{11}) - (A_{11} + A_{12}) \times (B_{22}) + (A_{12} - A_{21}) \times (B_{21} + B_{22})$

$C_{11} = A_{11} B_{11} + A_{12} \cdot B_{21}$

```
Algorithm st-mul (int*A, int*B, int*C, int n)
{ if(n==1) then
  { (*C)=(*C)+(*A)*(*B)
  }
  else
  {
     st-mul(A, B, C, n/4);
     st_mul(A, B+n/4, C+n/4, n/4);
     st_mul(A+2*(n/4), B, C+2*(n/4), n/4);
00 22 11 33 st_mul(A+2*(n/4), B+(n/4), C+3*(n/4), n/4);
01 01 23 23
01 23 01 23 st_mul(A+(n/4), B+2*(n/4), C, n/4);
        st_mul(A+(n/4), B+3*(n/4), C+(n/4), n/4);
        st_mul(A+3*(n/4), B+2*(n/4), C+2*(n/4), n/4);
        st_mul(A+3*(n/4), B+3*(n/4), C+3*(n/4), n/4);
  }
}
```

Time complexity :- $T(n) = n^{\log 7} = n^{2.81}$