Subject Code: 21CS42

Subject : Design & Analysis of Algorithm

Module-02

Divide & conquer & Decrease & Conquer Approach...

Chapter-01

Divide & conquer

Topic-01 : General Method

- In Divide & conquer method, a given problem is
  - (i) Divide into smaller sub problems
  - (ii) These sub problems are solved Independently.

(iii) Combining all the solutions of sub problems into a solution of the whole.

* If the sub problems are large enough then divide & Conquer is Reapplied.

* The generated sub ~~pro~~ problems are Usually of same type as the original problem.

* A control abstraction for divide & Conquer is given Below - Using control abstraction a flow of control of a procedure is given.

```
Algorithm DC (P)
{
  if P is too small then
     return solution of P.
     else
    {
      Divide (P) and obtain $P_1, P_2 - - - P_n$
        where $n \geq 1$
Apply DC to each subproblem
 return combaine (DC(P1), DC(P2), -- DC(Pn))
}}
```

* The Computing Time of above procedure of divide & Conquer is given by the Recurrence Relation.

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

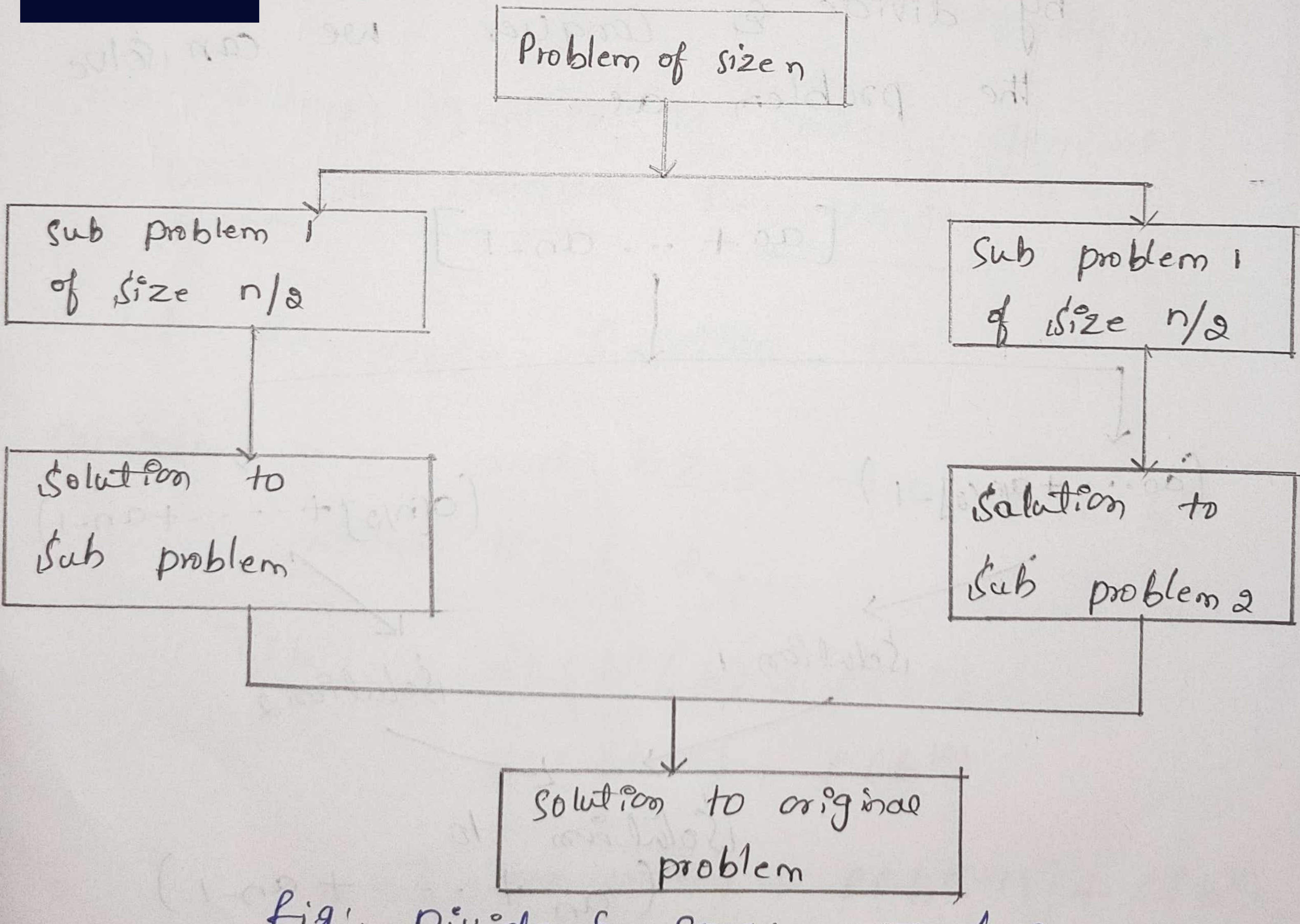

Problem of size n
Sub problem 1 of size n/2
Sub problem 1 of size n/2
Solution to sub problem
Solution to sub problem 2
Solution to original problem

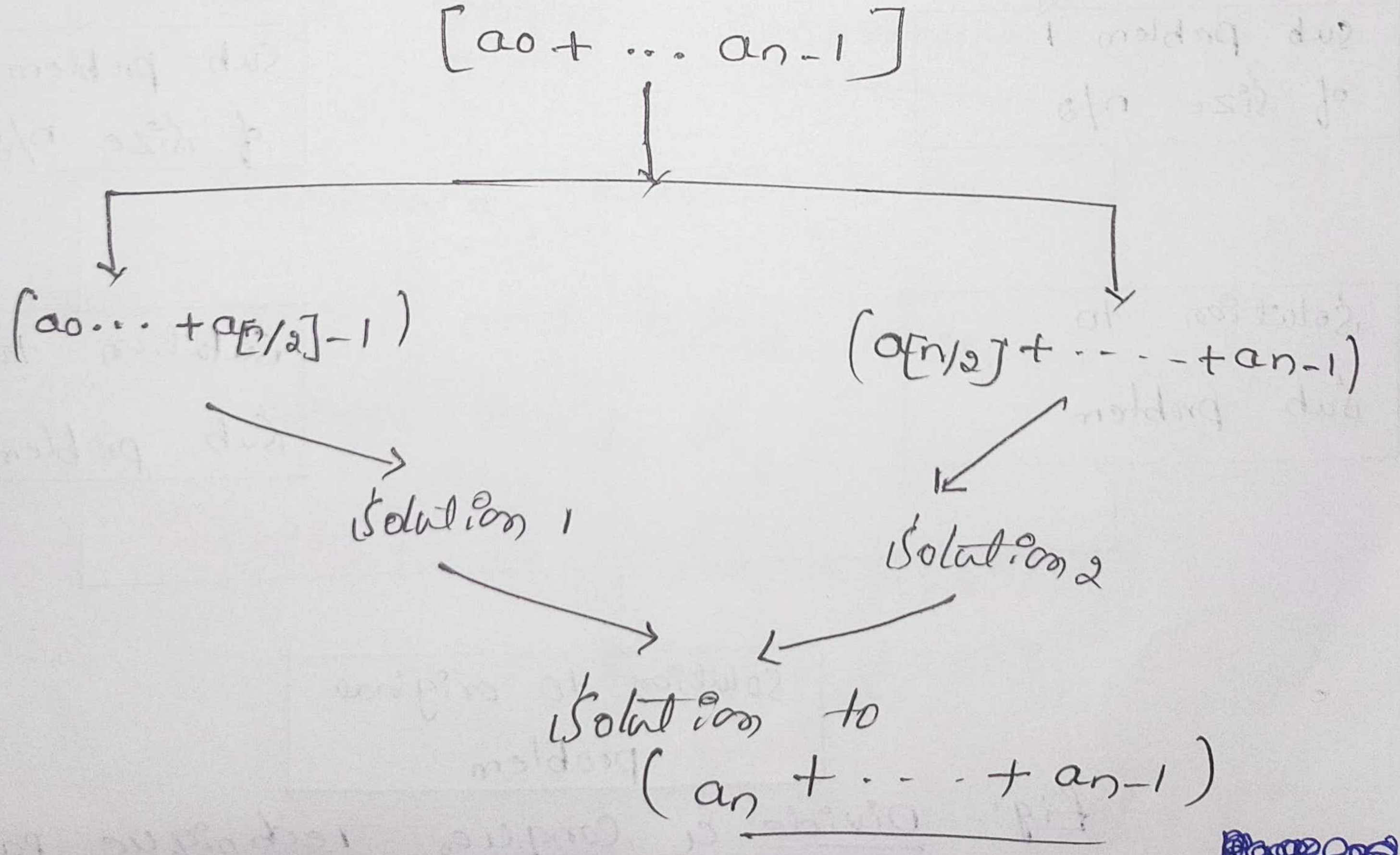
Fig: Divide & Conquer Technique

Topic-02

Recurrence Equation for divide & conquer

- * The Generated sub problems are Usually of same type as the original problem.
- * Hence sometime Recursive Algorithms are Used in divide & conquer strategy.

Ex:- To compute sum of n numbers then by divide & Conquer we can solve the problem as.


$[a_0 + \dots a_{n-1}]$
$(a_0 \dots + a_{\lceil n/2 \rceil - 1})$
$(a_{\lceil n/2 \rceil} + \dots - - + a_{n-1})$
solution 1
solution 2
solution to
$(a_n + \dots + a_{n-1})$

* If we want to divide a problem of size n into a size of n/b taking f(n) time to divide & combine, then we can set up Recurrence Relation for obtaining time for size n is -

$$\underbrace{T(n)}_{\text{Time for size n}} = \underbrace{a}_{\text{Number of subinstances}} \; T(\underbrace{n/b}_{\text{Time for size n/b}}) + \underbrace{f(n)}_{\text{Time required for dividing the problem into subproblem.}}$$

The Above Equation is called general Divide & Conquer Recurrence

Let Recurrence Relation be

Consider $a \geq 1$ and $b \geq 2$. Assume $n = b^k$.

where $k = 1, 2, \ldots$

$$T(b^k) = aT(b^k/b) + f(b^k)$$

$$= a\overbrace{T(b^{k-1})}+ f(b^k)$$

$$= a[aT(b^{k-2}) + f(b^{k-1})] + f(b^k)$$

$$= a^2 T(b^{k-2}) + af(b^{k-1}) + f(b^k)$$

Now substituting $T(b^{k-2})$ by using Back substitution:

$$= a^2 [aT(b^{k-3}) + f(b^{k-2})] + af(b^{k-1}) + f(b^k)$$

~~$a[a^3 T$~~

$$= a^3 T(b^{k-3}) + a^2 f(b^{k-2}) + af(b^{k-1}) + f(b^k)$$

continuing this fashion we get,

$$= a^k T(b^{k-k}) + a^{k-1} f(b^1) + a^{k-2} f(b^2) + \dots + a^0 f(b^k)$$

$$= [a^k T(1) + a^{k-1} f(b) + a^{k-2} f(b^2) + \dots + a^0 f(b^k)]$$

This can also be written as

$$= a^k T(1) + \frac{a^k}{a} f(b) + \frac{a^k}{a^2} f(b^2) + \dots + \frac{a^k}{a^k} f(b^k).$$

Taking a^k as common factor

$$= a^k\left[T(1) + \frac{f(b)}{a} + \frac{f(b^2)}{a^2} + \dots + \frac{f(b^k)}{a^k}\right]$$

$$= a^k\left[T(1) + \sum_{j=1}^{K} \frac{f(b^j)}{a^j}\right]$$

By property of logarithm

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

Hence we can write a^k as

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

we can Rewrite the Equation

$$T(n) = a^k\left[T(1) + \sum_{j=1}^{K} f(b^j)/a^j\right]$$

$$T(n) = n^{\log_b a}\left[T(1) + \sum_{j=1}^{\log_b n} f(b^j)/a^j\right]$$

thus order of growth of T(n) depends upon values of Constants a and b and order of growth of function f(n).

1. If f(n) is $O(n^{\log_b a-\varepsilon})$, then

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

2. If f(n) is $\Theta(n^{\log_b a} \log^k n)$, then

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$$

3. If f(n) is $\Omega(n^{\log_b a+\varepsilon})$, then

$$T(n) \text{ is } = \Theta(f(n))$$

Topic-03

Divide & conquer algorithms & complexity Analysis of finding the Maximum & Minimum.

Finding the ~~Maximum~~ & Minimum Element from the set ~~Algorithm~~ of n numbers.||

Algorithm

```
Algorithm Minimum_val (A[1... n])
{
    // Problem Description : This Algorithm is to
    Find the minimum value from array A
    of n ~~element~~ Elements.

        min ← A[1] //Assuming first Element
                                    as min.
        for (i ← 2 to n) do
        {
            if( min > A[i] ) then
                min ← A[i] // Set new min
                            value.
        }
        return min
}
```

First finding the minimum Element from the set of n numbers

Algorithm

```
Algorithm Maximum_Val( A[1.....n])
{
// Problem Description: This Algorithm is to
    find Maximum value from array A
    of n Elements.

    max ← A[1]
    for (i ← 2 to n) do
    {
        if [A[i] > max ) then
                max ← A[i]
    }
    return max
}
```

Obtain maximum & minimum values from an array simultaneously Using following Algorithm.

```
Algorithm  Max-min (A[1....n], Max, min)
{
    // Problem Description: Finding max and min
                            values
              Max ← min ← A[1]
  for (i ← 2 to n) do
  {
     if (A[i] > max) then
           Max ← A[i] // obtaining maximum
                                value.
     if (A[i] < min) then
           Min ← A[i] // obtaining minimum
                                value.
  }
}
```

Ex:-

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

Step-1

1								
50	40	-5	-9	45	90	65	25	75
↑ min max								

max = 50

min = 40

Step 2:-

Now from index 2 to n we will Compare an Array Element with min & Max value.

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

min = -5

Max = 50

Step-03:-

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

min = -9

max = 50

step 04:

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

min = -9

max = 45

step 05:

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

min = -9
max = 90

step 06:

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

min = -9
max = 90

Step-07

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

Min = -9

Max = 90

Step-08

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

Min =

Max =

Analysis

* The Above Algorithm takes O(n) running time.

* This is Because the Basic Operation of Comparing array Element with min or max value is done within a for loop.

* The Above Algorithm is a Straighforward algorithm of finding Minimum & Maximum.

Topic-04

Binary Search

- * Binary Search is an Efficient Searching Method.
- * while Searching the Elements Using this Method the Most Essential thing is that the Elements in the array should be sorted one

Problem statement

An Element which is to be searched from the list of Elements ~~is~~ is stored in array A[0 . . . . . . n-1] is called Key Element

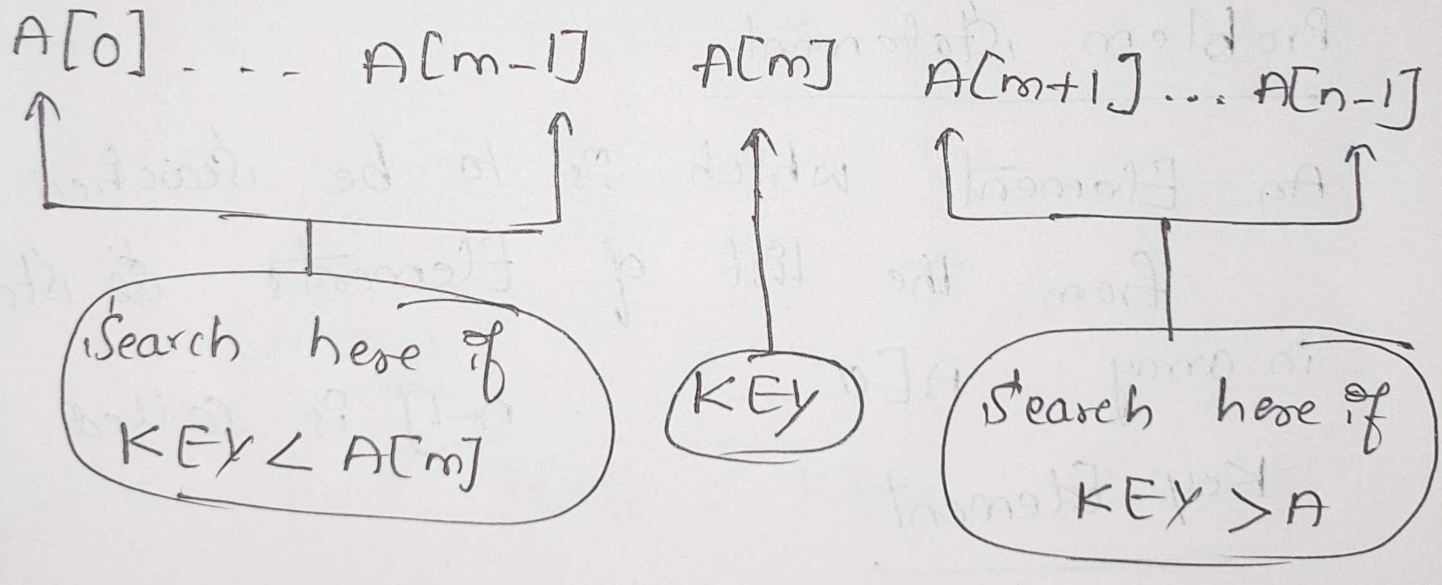
- * Let A[m] be the mid Element of Array.
- * Then There are three Conditions that needs to be tested while searching the Array Using this method.

1. If KEY = A[m] then desired Element is present in the list.

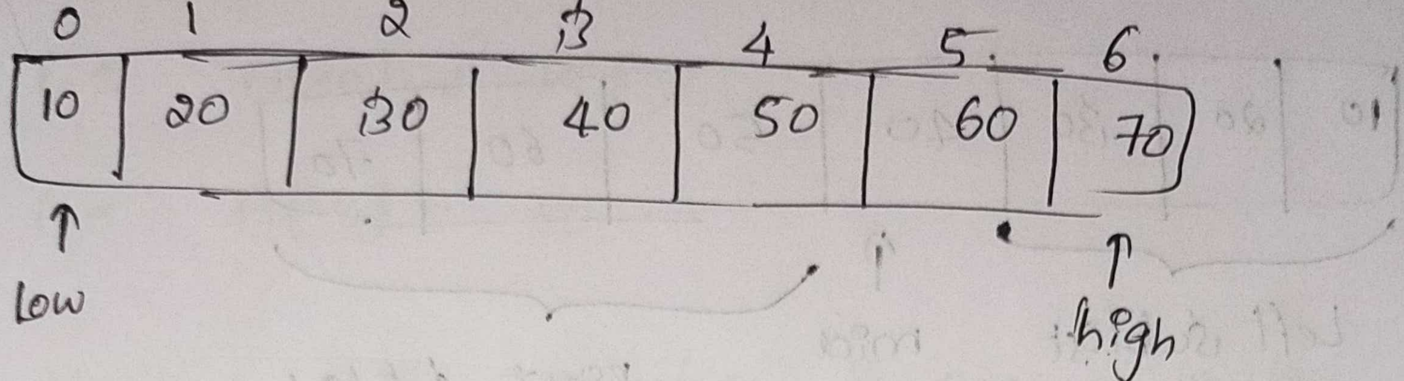
2. otherwise if KEY < A[m] then search the left sub list.

3. otherwise if Key > A[m] ~~then~~ then search the right sub list.

This can be Represented as



Ex:- Consider 10, 20, 30, 40, 50, 60, 70 & search 60.



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

The Key Element (i.e the element to' be searched) is 60.

Now obtain middle Element we will Apply formula

$$m = (low + high)/2$$

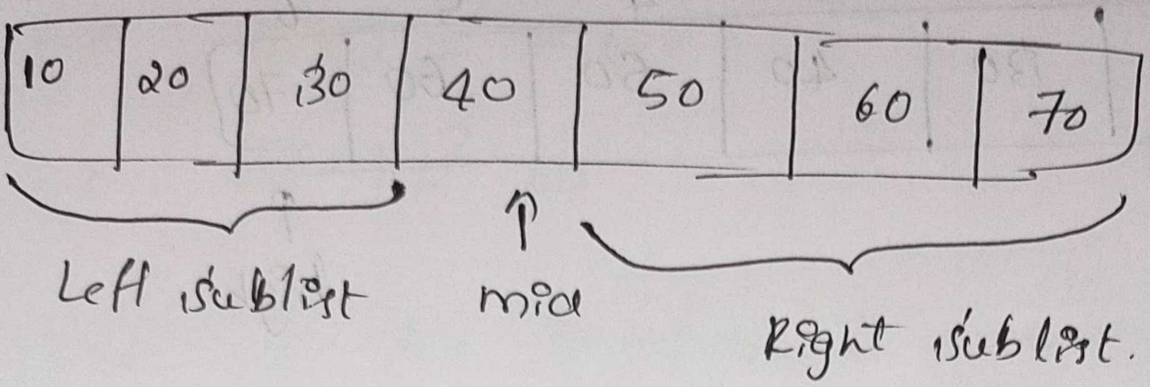
$$m = (0+6)/2$$

$$\underline{m = 3}$$

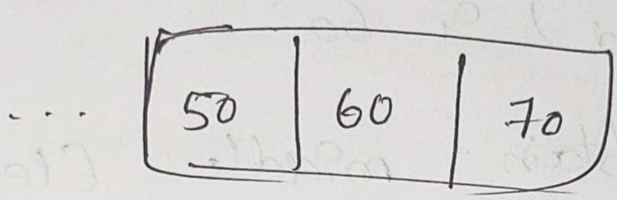
Then check $A[m] \overset{?}{=} KEY$

i.e $A[3] \overset{?}{=} 60$ No $A[3] = 40$ & $40 < 60$

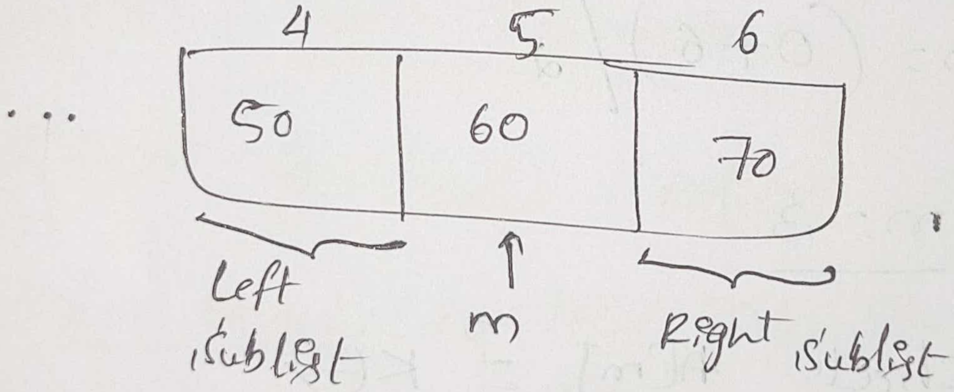
∴ Search the right sublist.



The right sublist is



Now we will again divide this list & check the mid element



$$m = (low + high)/2$$

$$m = (4+6)/2$$

$$\boxed{\therefore m = 5}$$

i:e $A[m] \overset{?}{=} KEY$

i.e $A[5] \overset{?}{=} 60$

Yes i.e the number is present in the list.

Thus we can search the desired number from the list of elements.

Algorithm

Algorithm (Non- Recursive)

```
Algorithm BinSearch (A[0 ... n-1], KEY)
// Problem Description: This Algorithm is for
    searching the 'element ~~the~~ Using
        Binary Search Method.

//Input: An Array A from which the
        KEY element is to be searched.
//output: It Returns the Index of an
    array element if it is
```

equal to KEY otherwise it Returns -1

```
low ← 0
high ← n-1
while (low < high) do
{
    m ← (low + high)/2      // mid of the array is obtained
    if ( KEY = A[m]) then.
        return m
    else if( KEY < A[m] ) Then
        high ← m-1      // search the left sub list
    else
        low ← m+1       // search the right sub list
}
return -1       // if Element is not present in the list.
```

Algorithm (Recursive)

```
Algorithm BinSearch (A, KEY, low, high)
}
//Problem Description : This Algorithm is
        for searching the Element
    Using Binary Search Method.

//Input: A is an Array of Elements in
        which the desired Element is to be
    searched KEY is the Element that
    has ~~no~~ to be searched.

//output: It Returns the Index of the
        array Element if the KEY Element
        is found.

// Initially low=0; and high= n-1 where n-is
    total number of Elements in the list

m= (low+high)/2;        //mid of the array
                         is obtained
if (KEY = A[m]) then
        return m;
```

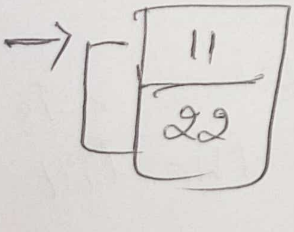
```
else if (KEY < A[m]) then
        BinSearch(A, KEY, low, m-1);
    else
        BinSearch(A, KEY, m+1, high);
}
```

Worst Case

$C_{worst}(n) = C_{worst}(n/2) + 1$ for $n > 1$

Avarage Case

If $n=1$
i.e only Element 11 is there



If $n=2$ & Search key = 22
Two Comparision are made to Search 22.

Topic-05

Merge sort

* Merge sort is a sorting Algorithm that Uses the divide & conquier strategy.

* Merge sort on an Input Array with n elements consists of three steps.

Divide: Partition array into two sub lists s1 & s2 with n/2 Elements each.

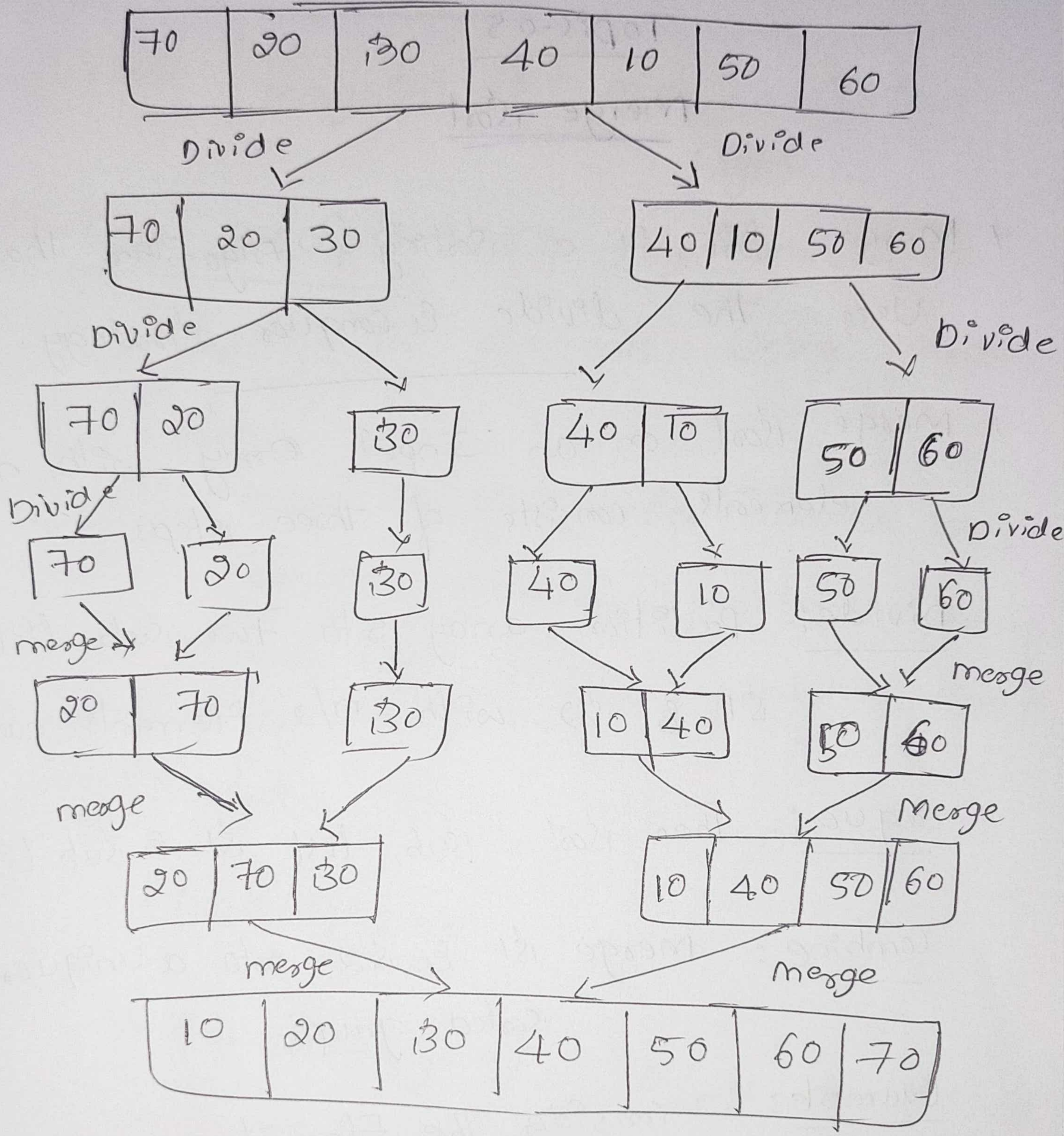
Conquer:- Then sort sub list s1 & sub list s2.

Combine: merge s1 & s2 into a Unique sorted group.

Example: Consider The Elements as

70, 20, 30, 40, 10, 50, 60.

Now we will split this list into Two sublists.


70 | 20 | 30 | 40 | 10 | 50 | 60
Divide
Divide
70 | 20 | 30
40 | 10 | 50 | 60
Divide
Divide
70 | 20
30
40 | 10
50 | 60
Divide
Divide
70
20
30
40
10
50
60
merge
merge
20 | 70
30
10 | 40
50 | 60
merge
Merge
20 | 70 | 30
10 | 40 | 50 | 60
merge
merge
10 | 20 | 30 | 40 | 50 | 60 | 70

Algorithm for merge sort

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 as 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          //split the list at mid.
    mergesort (A, low, mid)       // first, sub list
    mergesort (A, mid+1, high)    //second sublist
  combine (A, low, mid, high)     // merging of two sublists.
}
```

```
Algorithm Combine (A [0, . . . n-1] low, mid,
                        high)
{
    k ← low; //k as index for array temp.
    i ← low; //i as index for left sublist
                    of Array A.
    j ← mid+1 // j as index for right sublist
                    of array A
    while (i<= mid and j <=high) do
    {
      if (A[i] <= A[j]) then // if smaller
              element is present in left
                    sublist.
      {
        //copy that smaller element to temp
            array.
          temp [k] ← A[i]
           i ← i+1
           k ← k+1
       }
else //smaller element is present in right
   sublist
{
     //copy that smaller element to temp
         Array
         temp [k] ← A[j]
            j ← j+1
            k ← k+1
         ~~}~~ }
     }
     //copy Remaining elements of left
         sublist to temp.
  while ( i<= mid) do
  {
     temp [k] ← A[i]
        i ← i+1
        k ← k+1
  }
```

Topic-06

Quick Sort

* Quick sort is a sorting Algorithm that Uses the divide & conquer strategy.

* In this method division is dynamically carried out.

Three steps

1. Divide:- Split the Array into Two Sub Arrays that each Element in the left sub is less than / Equal the middle Element & each Element in the Right sub Array is greater than the middle Element.

2. Conquer:- Recursively sort the Two sub Arrays.

3. Combine:- Combine all the sorted Elements in a group to form a list of sorted Elements.

$$\underbrace{A[0] \ldots A[m-1]}_{\text{These elements are mid less than } A[m]}, \; \underset{\uparrow}{A[m]}, \; \underbrace{A[m+1] \text{ -- } A[n-1]}_{\text{These elements are greater than } A[m].}$$

Ex:

step-1:

low							high
50	30	10	90	80	20	40	70
i/pivot							j

we will now split the Array in two parts
The left sublist will contain the element less than pivot (i. e 50) & right sublist contains elements greater than pivot

step-2

low							
50	30	10	90	80	20	40	70
Pivot	i						j

we will increment i. If A[i] ≤ Pivot, we will continue to Increment it until the element pointed by. i is greater than A[low]

Step-3

Low							
50	30	10	90	80	20	40	70
pivot		i					j

Increment i as $A[i] \le A[Low]$

Step-04

Low							
50	30	10	90	80	20	40	70
pivot			i				j

As $A[i] > A[Low]$, we will stop incrementing i

Step-05

low							
50	30	10	90	80	20	40	70
pivot			i			j	

As $A[j] >$ pivot (i.e $70 > 50$). we will decrement j. we will continue to decrement j untill the element pointed by j is less than $A[low]$.

Step-06

Low							high
50	30	10	90	80	20	40	70
Pivot			j			j	

Now we can not decrement j because 40<50. Hence we will swap A[i] and A[j] i.e 90&40

Step-07

Low							
50	30	10	40	80	20	90	70
pivot			i			j	

As A[i] is less than A[low] and A[j] is greater than A[low] we will continue incrementing i and decrementing j, until the false conditions are obtained

Step-08

50	30	10	40	80	20	90	70
				i	j		

we will stop incrementing i & stop decrementing j. As i is smaller than j we will swap 80 & 20

step-09

Low							High
50	30	10	40	20	80	90	70
pivot				i	j		

As A[i] < A[low] and A[j] > A[low], we will continue incrementing i and decrementing j.

Step-10

Low							High
50	30	10	40	20	80	90	70
pivot					i j		

Low							High
50	30	10	40	20	80	90	70
pivot				j	i		

As A[j] < A[low] and j has crossed i. That is j<i, we will swap A[low] and A[j]

step-11

low							High
20	30	10	40	50	80	90	70
				j	i		

Now we have left sublist

pivot is shifted at its position

~~now~~ ~~is~~ Now we have right sublist

~~Step~~

we will now start sorting left sublist, assuming the ~~list~~ first Element of left sublist as pivot element Thus now new pivot=20

Step-12

Low							High
20	30	10	40	50	80	90	70
pivot	i		j	occupied its position			

Now we will set i and j pointer & then we will start comparing A[i] with A[Low] or A[Pivot]. Similarly comparison with A[j] and A[Pivot].

Step-13

Low							
20	30	10	40	50	80	90	70
pivot	i		j				

As A[i] > A[pivot], hence stop incrementing i. Now as A[j] > A[pivot], hence decrement j.

Step-14

Low							
20	30	10	40	50	80	90	70
pivot	i	j					

Now j cannot be decremented Because 10<20. Hence we will swap A[i] & A[j]

Step-15

Low							
20	10	130	40	50	80	90	70
pivot	i	j					

Now as A[i] > A[low] or ~~A[j] > A[pivot]~~ decrement ~~j~~ i.

Step-16

Low							
20	10	130	40	50	80	90	70
pivot	j	i					

Now as A[i] > A[low], or A[j] > A[pivot] decrement j.

Step-17

low							
20	10	130	40	50	80	90	70
pivot	j	i					

As A[j] < A[low] we cannot decrement j now. we will now swap A[Low] and A[j] as j has crossed i and i>j.

Step-18

low							
10	20	130	40	50	80	90	70
Left sub list	↑ pivot	Right sublist					

As there is only one Element in left sublist hence we will sort right sublist

Step-19

Low							
10	20	30	40	50	80	90	70

Now Consider this sublist for sorting

As left sublist is sorted completely we will sort sublist, assuming first Element of right sublist as pivot

Step-20

10	20	30	40	50	80	90	70
					pivot	i	j

~~As A[i] < A[pivot], increment i.~~

As A[i] > A[Pivot], hence we will stop incrementing i. similarly A[j] < A[Pivot]. Hence we stop decrementing j. Swap A[i] and A[j]

Step-21

10	20	30	40	50	80	70	90
						i	j

As A[i] < A[pivot], increment i.

Step - 22

10	20	30	40	50	80	70	90
					pivot		i,j

As A[i] > A[pivot], decrement j

Step-23

10	20	30	40	50	80	70	90
					pivot	j	i

Now swap A[pivot] and A[j]

Step-24

10	20	30	40	50	70	80	90
						pivot	

The left sublist now contains 70 & right sublist contains only 90. we cannot further subdivide the list

Hence list is

10	20	30	40	50	70	80	90

This is a sorted list

Algorithm for Quick sort

Algorithm Quick (A [0... n-1], low, high)

//Problem Description: This Algorithm performs ~~of~~ sorting of the Elements given in Array A[0... n-1]

//Input: An Array A[0.... n-1] in which Unsorted elements are given The low indicates the leftmost ~~[illegible]~~ Element in the list. and high Indicates the rightmost element in the list.

//output: Creates a sub Array which is sorted in Ascending order.

```
if(low <high) then
//split thes Array into two sub Arrays.
  m← partition (A[low]... high) // m is mid
      of the array.
  Quick (A [low ... m- 1])
  Quick (A[ mid+1 .... high]))
```

Algorithm Partition (A[Low. ... high])

//Problem Description : This Algorithm partitions the subarray Using the first Element as pivot element

// Input :- A subarray A with low as left most Index of the array and high as the rightmost Index of the array

//output : The partitioning of array A is done & pivot occupies its proper position. And the rightmost index of the list is Returned.

```
pivot ← A[Low]
i ← low
j ← high +1
while (i < =1) do
{
    ~~i ← i+1~~ while( A[i] <= pivot) do
        i ← i+1
    while( A[j] >= pivot )do
        j ← j - 1;
    if ( i<= j) then
swap (A[i], A[j]) // swaps A[i] and A[j]
}
swap (A[low], A[j]) // when i crosses j
    swap A[low] and A[j]
    return j // rightmost index of the list.
```

Time complexity of Quick sort

1. Best case: $\Theta(n \log_2 n)$

2. Average case: $O(n \log_2 n)$

3. Worst case: $O(n^2)$

Chapter-02

Decrease & Conquer Approach

Topic-01

Introduction

Decrease and Conquer Approach

Decrease & Conquer is an Approach for solving a problem by.

1. Change an instance into one smaller instance of the problem.

2. Solve the smaller instance

3. Convert the solution of the smaller instance into a solution for the larger instance

* In decrease & conquer method the problem can be solved using top down (Recursive) solution (or) using Bottom-UP (Iterative (or) non Recursive solution.

Variations of Decrease & Conquer

These are Three major variations of decrease & conquer

1. Decrease by constant
2. Decrease by a constant factor
3. Variable size decrease

1. Decrease by Constant

- In this method the size of the instance is Reduced by same constant on each iteration of the Algorithm.
- Generally this constant is equal to one.

Ex:- To compute a^{10} we can write

$$a^{10} = a^{9}.a$$

If we formulize ~~this~~ the Example then we can write it as,

$$a^{n} = a^{n-1}.a$$

Applications of decrease by constant

1. Insertion Sort

2. Graph Searching Algorithm.
   - Depth first search
   - Breadth first Search
   - Topological Sorting

2. Decrease by a Constant Factor

* Decrease by a Constant Factor decreases the instant size by half (or) by some other fraction.

Ex: $a^{10} = a^5 \cdot a^5$

Applications of decrease by constant

1. Binary Search

3. Variable Size Decrease

* In Variable Size decrease method the size Reduction pattern Varies from one iteration of an Algorithm to Another.

Ex:- Finding GCD of Two numbers Using Euclid's Algorithm.

gcd(m, n) = gcd(n, m, mod n)

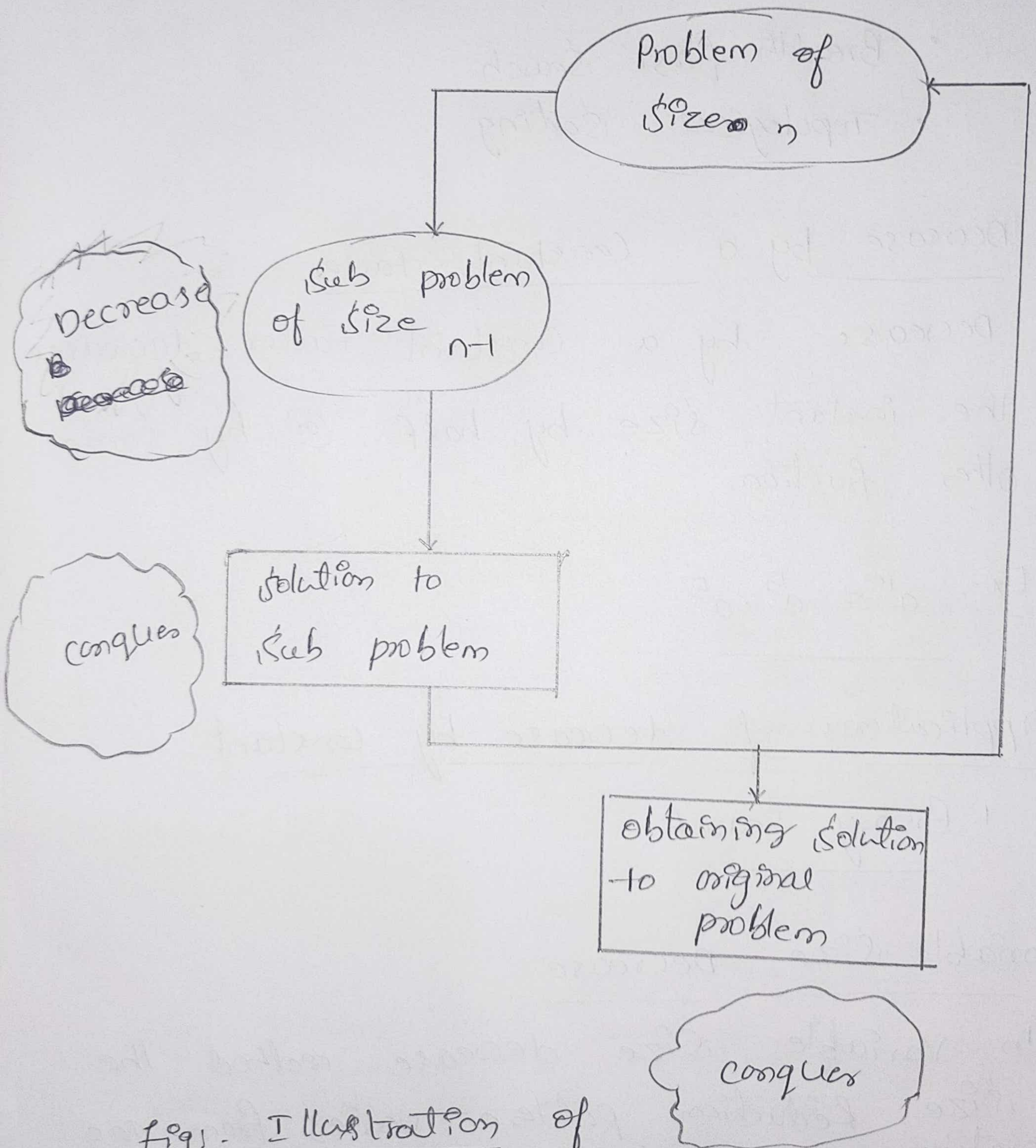


Fig:- Illustration of decrease by one & conquer method

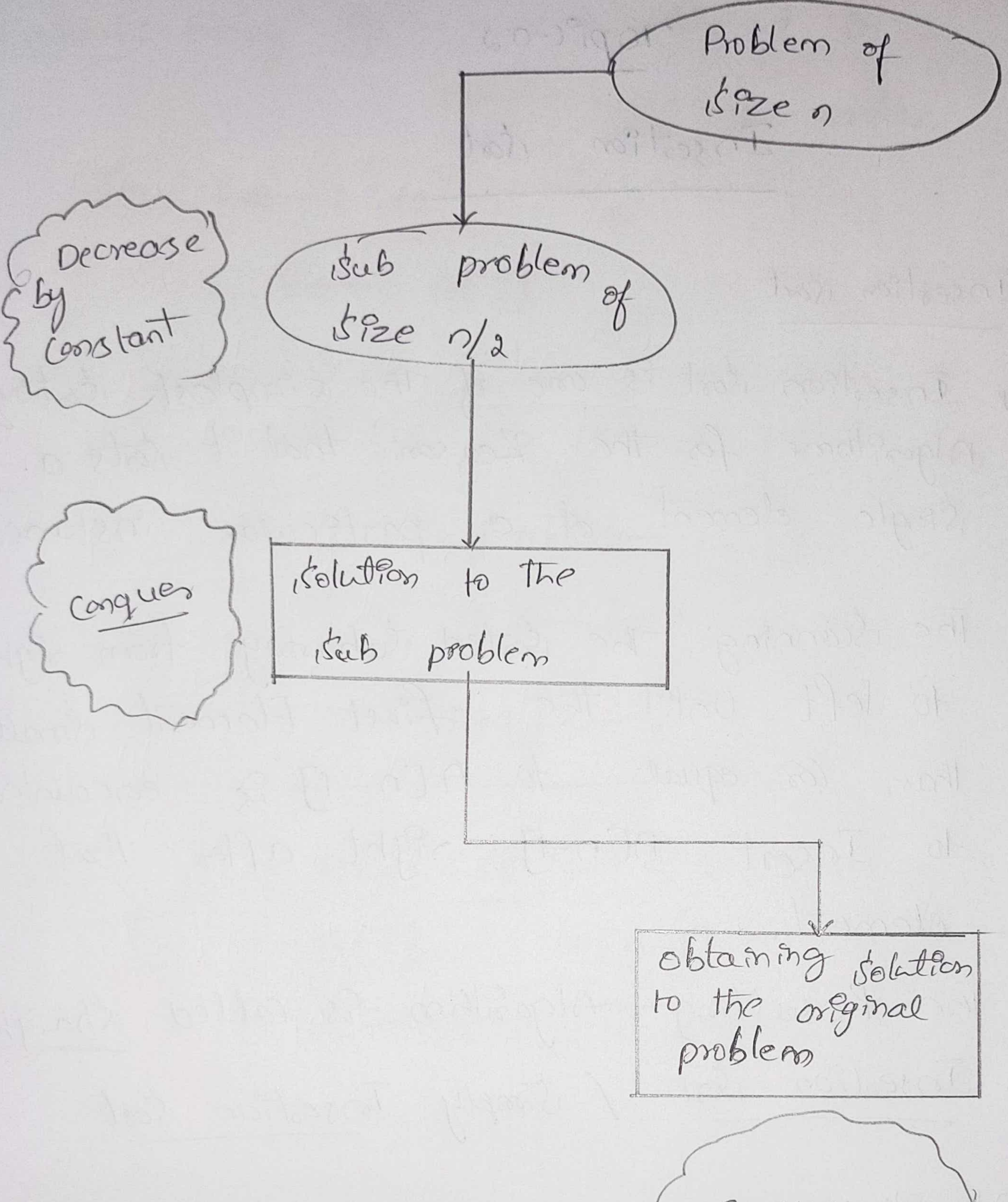


fig: Illustrating decrease by half & conques Method

Topic-02

Insertion sort

Insertion sort

- Insertion sort is one of the simplest sorting Algorithms for the Reason that it sorts a single element at a particular instance.
- The scanning the sorted subarray from right to left until the first Element smaller than (or) equal to A[n-1] is encountered to Insert A[n-1] right after that element.
- The Resulting Algorithm is called Straight Insertion sort / Simply Insertion sort

Algorithm

```
Insertion sort(A[0 --- n-1]
//Sorts a given array by insertion sort
//Input: An Array A[0 --- n-1] of n orderable
                              elements.
```

//output:- Array A[0---n-1] is sorted in nondeecreasing order.

```
for i ← 1 to n−1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] v do
        A[j+1] ← A[j]
        j ← j − 1
    A[j+1] ← v
```

$A[0] \leq \ldots \leq A[j] < A[j+1] \leq \ldots \leq A[i-1]$

$A[i] \ldots A[n-1]$

Ex:

89 | 45 68 90 29 34 17

45 89 | 68 90 29 34 17

45 68 89 | 90 29 34 17

45 68 89 90 | 29 34 17

29 45 68 89 90 | 34 17

29 34 45 68 89 90 | 17

17	29	34	45	68	89	90

Topic-03

Graph Searching Algorithms

Depth - First Search

* Depth First Search starts a graph's Traversal at an arbitrary vertex by marking it as visited.

* On each Iteration, the Algorithm proceeds to an Unvisited vertex That is adjacent to the one it is currently in.

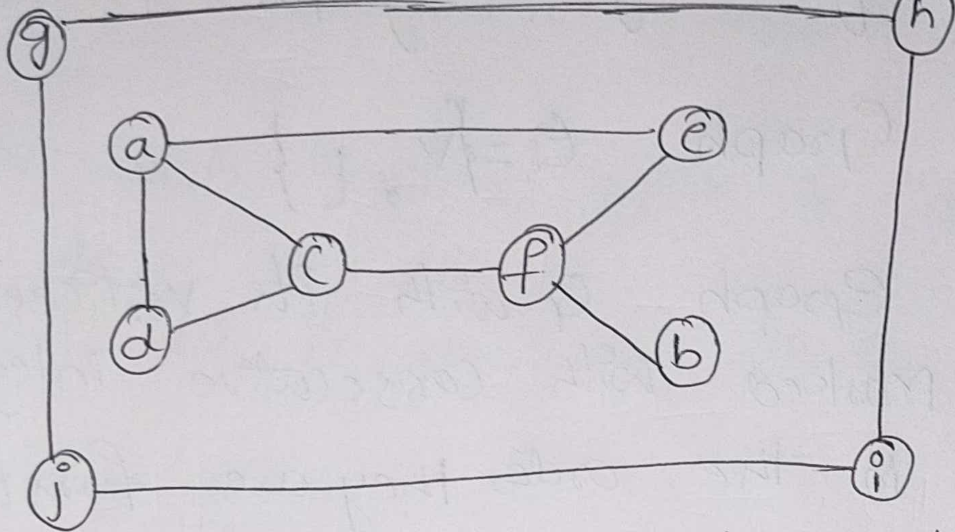


fig:- DFS Graph

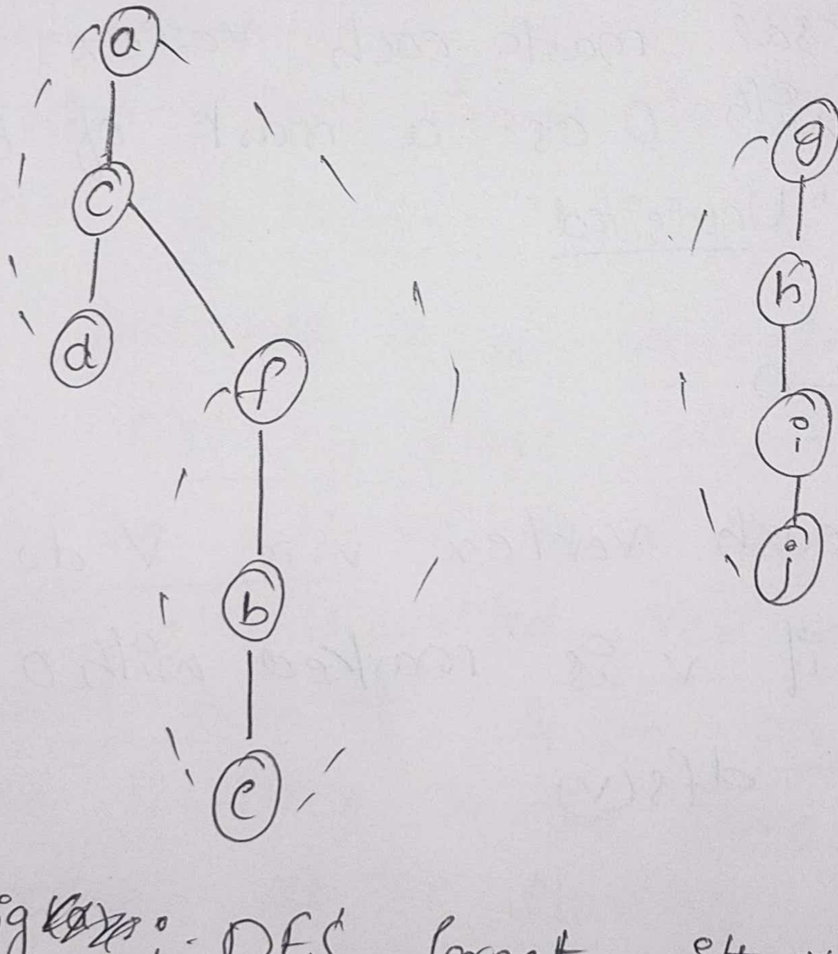


fig:- DFS forest with the Tree & Back Edges shown with solid & Dashed lines.

ALGORITHM

```
//Implements a depth-first search
  Traversal of a given graph.
//Input: Graph G={V, E}
//output: Graph G with its vertices
        Marked with consecutive integers
        in the order they are first
        Encountered by the DFS Traversal
        mark each vertex in V
        with 0 as a mark of Being
        "Unvisited"

count ← 0
    for each vertex v in V do
        if v is marked with 0
            dfs(v)

dfs(v)

//visits Recursively all the Unvisited vertices
  Connected to vertex v by a path
```

and numbers them in the order they are Encountered via global variable count.

```
count ← count + 1; mark v with
                          count
for each vertex w in V adjacent to v
  do
  if w is marked with 0.

     dfs(w)
```

Breadth-First Search

* Breadth-first search is a traversal for the cautious.
* It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges aparts from it, & so on untill all the vertices in the same connected component as the starting vertex are visited.

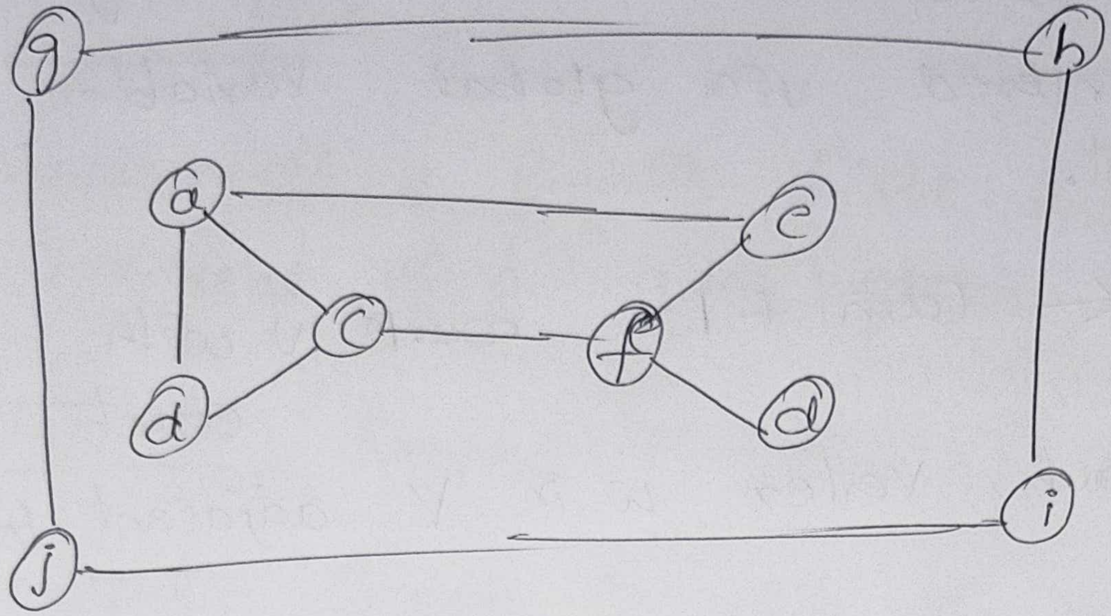


fig: BFS graph

Traversal queue

a_1 c_2 d_3 e_4 f_5 b_6

g_7 h_8 j_9 i_{10}

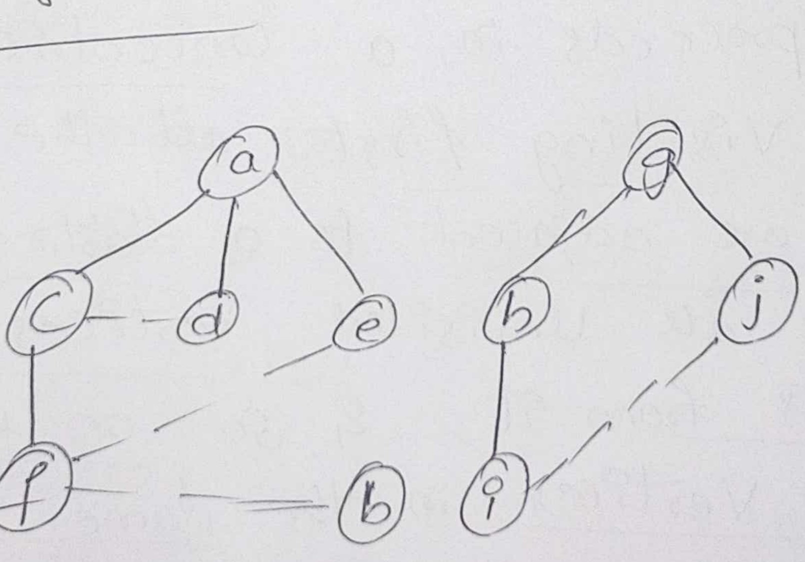


fig: BFS forest with the tree & cross edges shown with solid & ~~dott~~ dotted lines.

Algorithm BFS.

```
//Implements a Breadth - First Search
  Traversal of a given graph.

//Input:-  Graph G = {V, E}

//output:  Graph G with its Vertices
           Marked with Consecutive
           integers in the order they
           are visited by the BFS
         Traversal mark each vertex in
         V with 0 as a mark of Being
         "Unvisited"

         count ← 0

         for each vertex v in V do
             if v is marked with 0
                 bfs(v)

         bfs(v)
     // visits all the unvisited vertices
        connected to vertex v
```

```
//by a path and numbers them in the
  order they are visited
//via global variable count

Count ← Count + 1; mark v with count
                   and Initialize a
                   queue with v
  while the queue is not empty do
  for each vertex w in V adjacent to
  the front vertex do
    if w is marked with 0
      Count ← Count + 1; mark w with
                         count
      add w to the queue
  remove the front vertex from the
  queue.
```

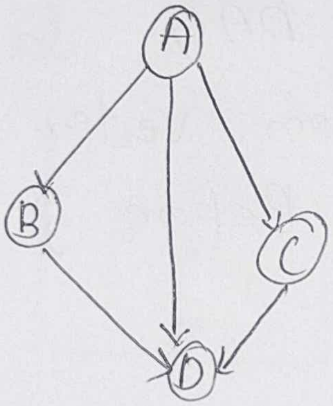
Topic-04

~~Topology~~ Topological sorting.
It's Efficiency Analysis

Definition DAG

A Directed Acyclic Graph is a directed graph with no cycles.

Ex:



* Based on the principle of DAG, specific ordering of vertices is possible.

* This method of Arranging the vertices in some specific manner is called Topological sort

Topological Sorting Techniques

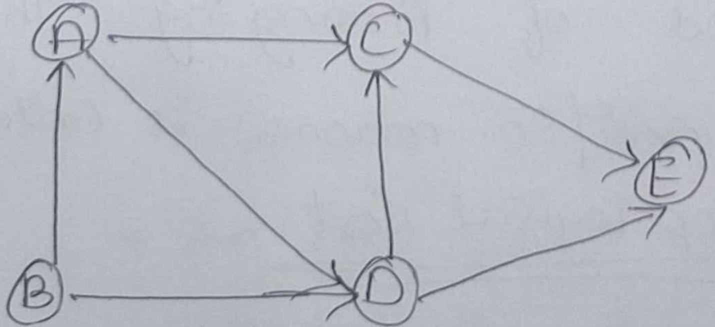
1. DFS Based Algorithm.

2. Source Removal Algorithm.

1. DFS Based Algorithm

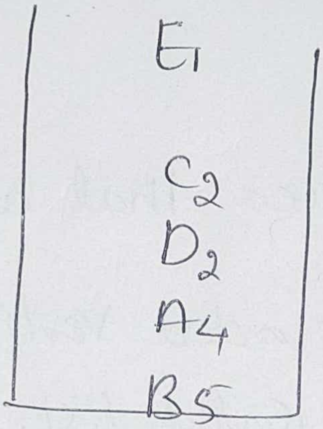
* Topological Sort is ~~[illegible]~~ a process of assigning a linear ordering to the vertices of a DAG, so that if there is an edge from vertex i to vertex j then i appears Before j in the linear ordering.

Ex:- Sort the diagraph for Topological Sort Using DFS Based Algorithm.



Solution:- As the graph contains no cycle i.e The graph is a DAG, the Topological sorting is possible.

Step-01:- First find the Depth first Search & push the visited Vertices in the Stack thus Creates a DFS Traversal Stack.



Step-02:- Now pop-off the contents of the Stack E, C, D, A, B.

Step-03:- Reverse the popped contents.

* The list which are getting is a Topologically Sorted list.

∴ B, A, D, C, E

2. Source Removal Algorithm

* This is a direct Implementation of decrease & conquer method

Algorithm follow these steps

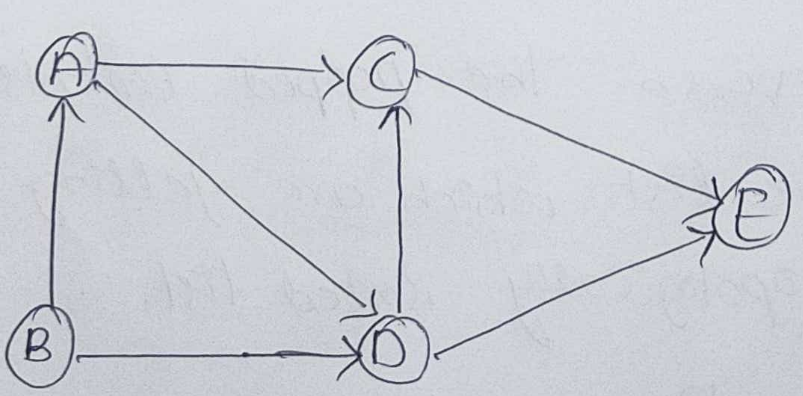
1. From a given graph find a vertex with no Incoming edges.

* Delete it Along with all the edges outgoing from it.

2. Note the vertices that are deleted

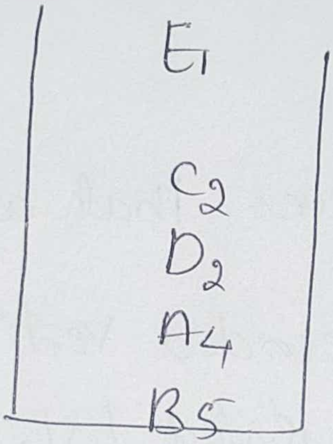
3. All these Recorded vertices give Topologically sorted list.

Example:- Sort The diagraph for Topological sort Using source Removal Algorithm.



Solution:- As the graph contains no cycle i.e The graph is a DAG, the Topological sorting is possible.

Step-01:- First find the Depth first Search & push the visited vertices in the stack thus Creates a DFS Traversal stack.



Step-02:- Now pop-off the contents of the stack E, C, D, A, B.

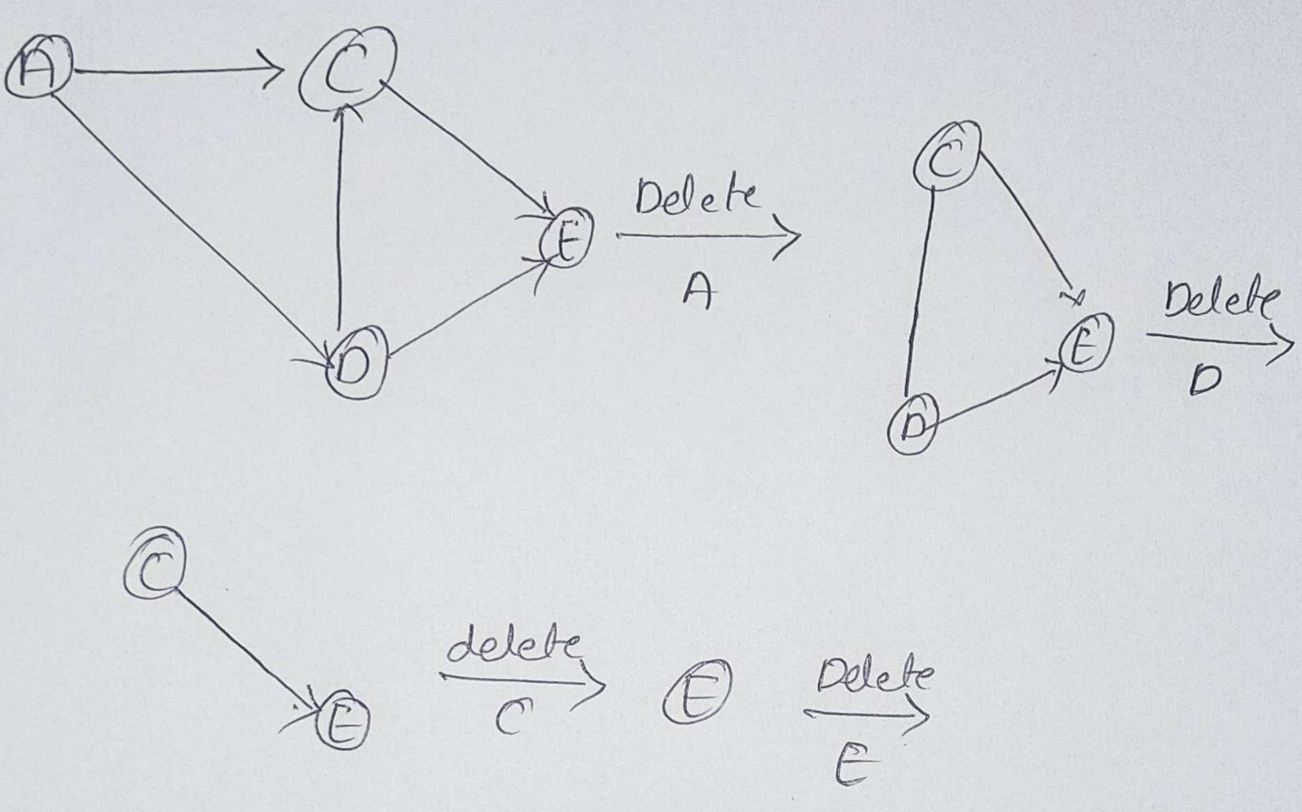
Step-03:- Reverse the popped contents.

* The list which are getting is a Topologically sorted list.

∴ B, A, D, C, E

Solution :- we will follow following steps to obtain Topologically sorted list.

Choose Vertex B Because it has no Incoming edge, delete it along with its adjacent Edges.



B, A, D, C   B - A D, C, E

Hence the list After Topological sorting will be B, A, D, C, E