Part-B

UNIT-5 DECREASE AND CONQUER APPROACHES, SPACE-TIME TRADEOFFS

RAJESHWARI.J
ASSOCIATE PROFESSOR
DSCE.

TOPICS

1.) Introduction
2.) Insertion Sort
3.) Depth first search
4.) Breadth first search
5.) Topological Sorting

Space time tradeoffs

1.) Introduction
2.) Sorting by Counting
3.) Input Enhancement in String Matching.

INTRODUCTION

The principle idea of decrease & conquer is to solve the given problem by reducing its instance to a smaller one & then extending the obtained solution to get a solution to its original instance.

Difference B/w Divide & conquer AND Decrease & conquer is

There may be several sub problems that need to be solved for Divide & Conquer algorithms.

There may be one sub problem that needs to be solved for Decrease & conquer algorithms.

Decrease & Conquer

Is a approach for solving a problem by:

* Change an instance into one smaller instance of the problem.
* Solve the smaller instance.
* Convert the solution of the smaller instance into a solution for the larger instance.

In this method the problem can be solved using top-down solution (or) using bottom up solution.

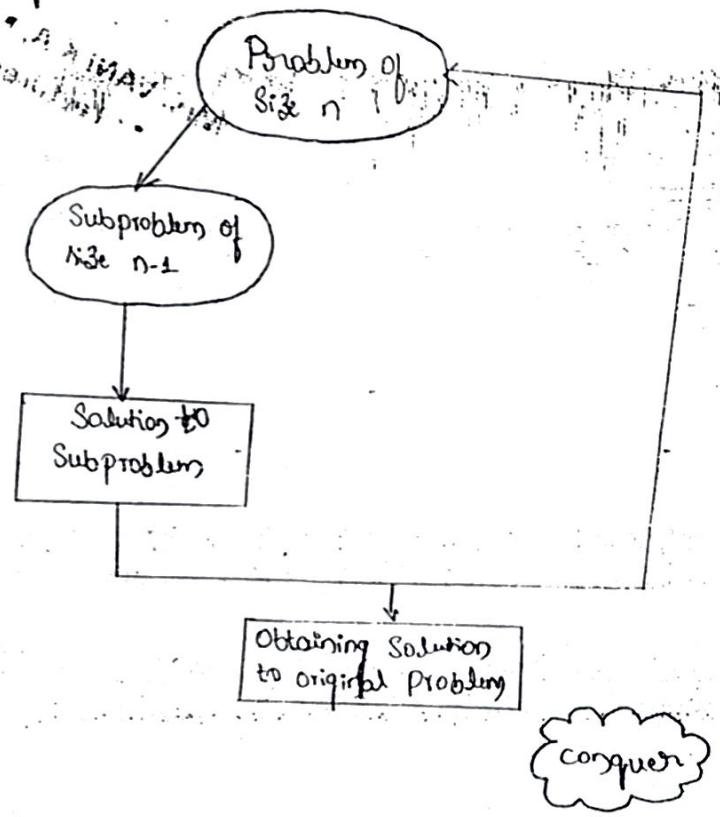
There are 3 major variations:-

a) Decrease by constant

b) Decrease by constant factor

c) Variable size decrease.

(a)* 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.



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

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

If we formulize this eq then we can write it as,

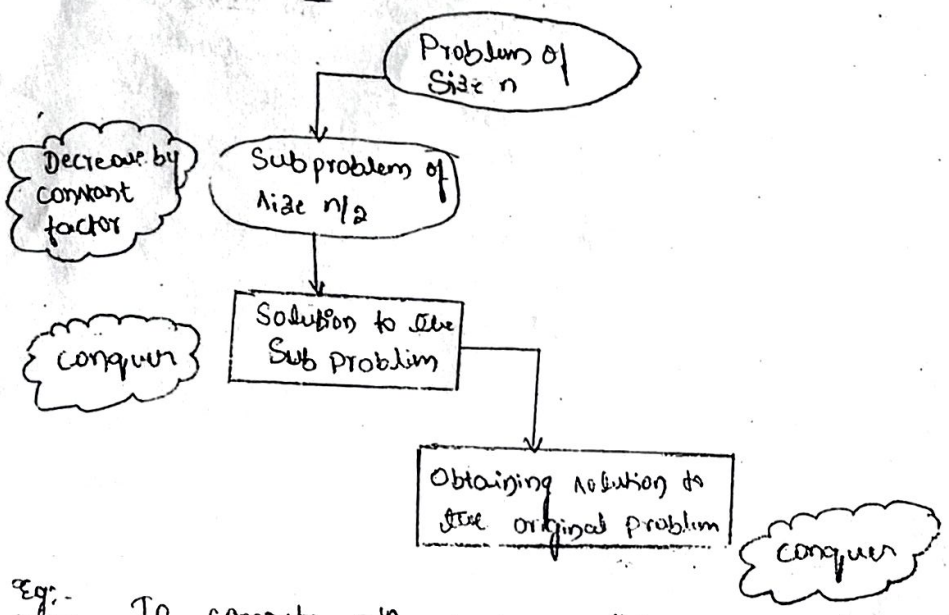
$$a^n = a^{n-1} . a$$

Applications of Decrease & Conquer

This method is used to solve the following problems:-

* Insertion sort
* Graph Searching algorithm
  - Depth First search
  - Breadth First search
  - Topological Sorting.

Decrease by a Constant Factor In this method the size of the instance is reduced by half (or) by some other fraction.



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

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

Applications of Decrease by Constant

*) Binary search

b) Variable size Decrease In this method the size reduction pattern varies from one iteration of an algorithm to another.

Eg:- Finding GCD

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

In finding GCD the 2 no's go on varying until we get the GCD value.

$$\underset{m\ n}{gcd(60,24)} = gcd(24,12)$$
$$= gcd(12,0)$$

60
12×4
48

Mrs. VANI K.A, B.E,M.Tech
Lecturer

(24,12)

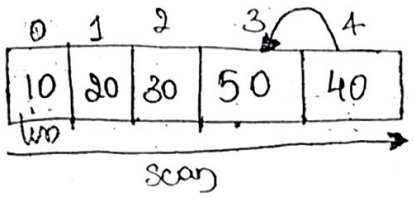
24) 48 (2
48
12

In this method the element is inserted at its appropriate position. Generally the insertion sort maintains a zone of sorted elements. If any unsorted element is obtained then it is compared with the elements in the sorted zone & then it is inserted at the proper position in the sorted zone. This process is continued until we get the complete sorted list. Hence the name of this method is insertion sort.

There are 3 ways by which insertion sort is implemented.

M1) In this method of implementation the sorted sub array is scanned from left to right until the 1st element greater than (or) equal to A[n-1] is encountered & then insert A[n-1] right before that element.

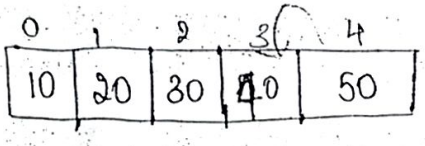
Eg:



40?10
>20

40

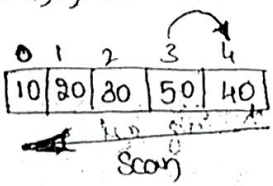
In the above Eg we get element 40 which is lower than 50 so we insert 40 before 50. Hence the list will be sorted.



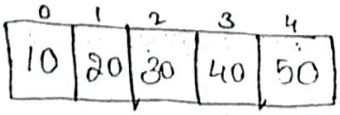
M2) In this method of implementation the sorted sub array is scanned from right to left until the 1st smaller than (or) equal to A[n-1] is encountered & then insert A[n-1] right after that element.

Eg:

40 < 50
40 < 30



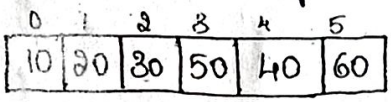
In the above Eg we get element 50 which is greater than 40 so we insert 40 after 50. Hence the list will be.



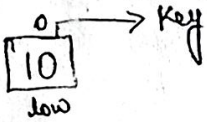
This method is also called as straight insertion sort (or) simply as insertion sort

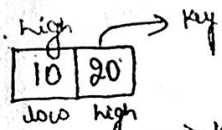
The 3rd method is called as "Binary Insertion sort" because in this method smaller element is searched using Binary search method. (3)

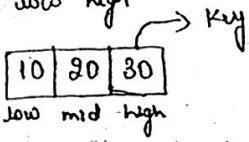
Eg:- Consider the list of elements to be:-

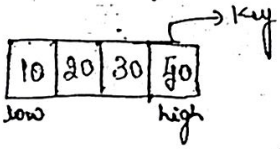


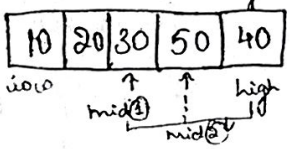
Now we will scan the list.



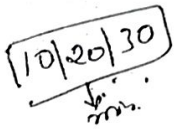




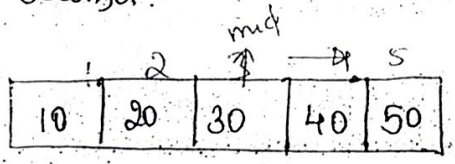


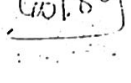


∴ insert key at proper position.



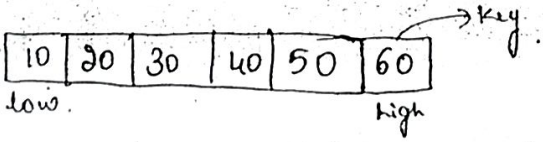
The list becomes.



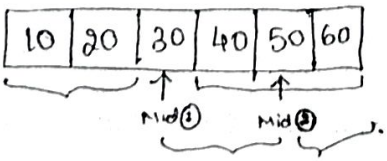


4 5

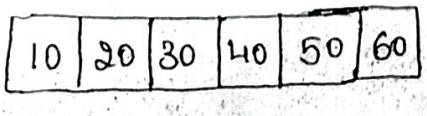
In the further scan we consider.



By Binary search we will check whether a smaller element is lying after any larger element.



An key is at its proper place. We won't insert it any where else.
Thus after binary insertion sort method we get a sorted list as.



Mrs. VANI K.A, B.E,M.Tech
Lecturer

Algorithm :- Insert-Sort (A[0---n-1])

```
// Input : An array of n elements
// Output : Sorted array A[0---n-1] in ascending order
for i ← 1 to n-1 do
{
temp ← A[i]    //mark A[i]th element
j ← i-1        // Set 'j' at previous element of A[i]
while (j >= 0) AND (A[j] > temp) do
{
A[j+1] ← A[j]
j ← j-1
}
A[j+1] ← temp
```

/* comparing all the previous elements of A[i] with A[j]. if any greater element is found then insert it at the proper position */

89 45 68 90 29 34 17

Analysis

Time Efficiency :-

Best case:- It happens for almost sorted array. The no of key comparisions can be

$$C_{best}(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1) \cancel{-1+1}$$

$$= (n-1)$$

$$\therefore C_{best}(n) = \Theta(n)$$

25 75 40 10 20

Thus the best case time complexity of insertion sort is $\Theta(n)$.

Average case:- When an array contains randomly distributed elements then it results in average case time complexity. As compared to worst case atleast half no of comparisons are needed to obtain average case complexity.

$$\therefore C_{avg}(n) = \frac{(n-1)n}{2} \times \frac{1}{2} = \frac{n^2-n}{4}$$

$$= \frac{n^2}{4}$$

Worst Case:- When the list is sorted in descending order

i.e. want to sort such a list in ascending order then it results in worst case behavior of algorithm.

1.) The basic operation of algorithm is A[j] > temp.

2.) The basic operation depends upon the size of list i.e. $n-1$

3.) The no of key comparisions for such I/p is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} i$$

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

$$C_{worst}(n) = \Theta(n^2)$$

$\because \sum_{i}^{u} 1 = u - l + 1$

$\therefore \sum_{j=0}^{i-1} 1 = (i-1) - 0 + 1$

$= 1$

The worst case time complexity of Insertion sort is $\Theta(n^2)$

Advantages of Insertion Sort

1.) Simple to implement

2.) This method is efficient when we want to sort small no of elements. And this method has excellent performance on almost sorted list of elements.

3.) More efficient than most other simple $O(n^2)$ algorithms such as selection sort (or) bubble sort.

4.) This is a stable algorithm in maintaining relative order.

5.) It is called in-place sorting algorithm. [in-place → The I/P is overwritten by o/p to execute the sorting method]

Eg:-

Sorted part			unsorted part			
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

Mrs. VANI K.A, B.E, M.Tech
Lecturer

The systematic follow up of the edges of the graph in some specific order to visit the vertices of the graph is called "Graph Searching"

There are 2 searching algorithms:-

(1) Depth First Search (DFS)
(2) Breadth First Search (BFS)

Rules of DFS:-

1.) Select an unvisited node 'V', visit it, & treat it as the current node.

2.) Find an unvisited neighbour of the current node, visit it, & make it the new current node. If there are more than one unvisited neighbours then resolve the tie by following the alphabetical order.

3.) If the current node has no unvisited neighbours, backtrack to its parent & make it new current node. → Process continues until a deadend

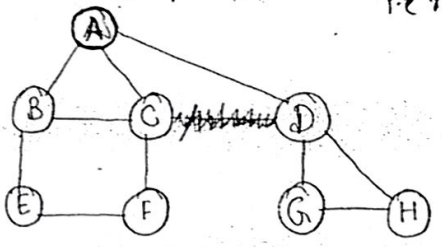
4.) Repeat the steps 2 & 3 until no more nodes can be visited.

5.) Repeat from steps 1 for the remaining nodes.

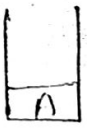
6) Vertex is pushed into the stack when vertex is reached for first time

7) Vertex is poped off the stack when it becomes a deadend i.e no adjacent vertices.

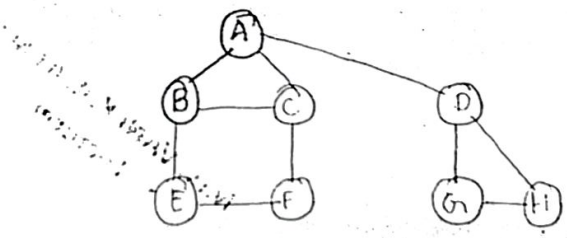
Eg:-

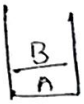


We will start searching in DFS manner from vertex 'A'

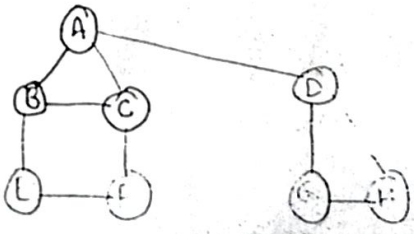


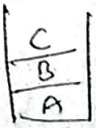
(1,2)



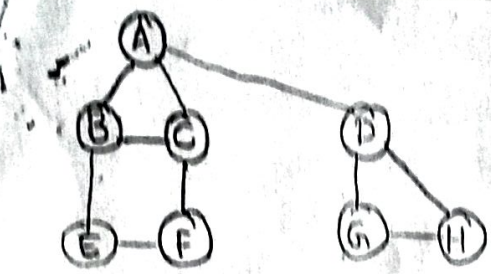


Then visit the neighbouring node 'B'.

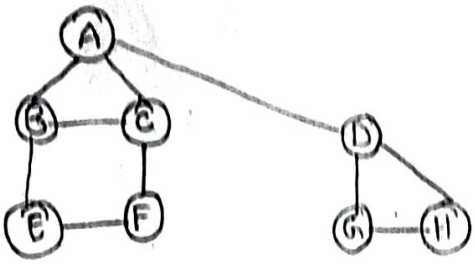




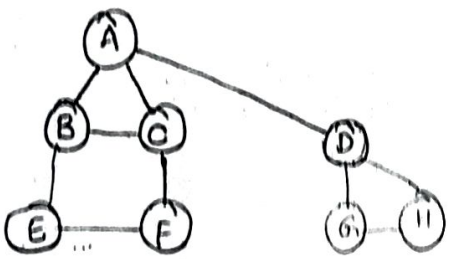
Then visit neighbouring node 'C'.



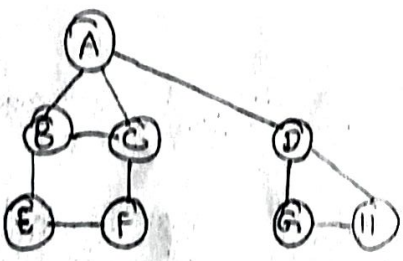
Visit 'F'



Visit 'G',
Now from 'G' there is an edge for 'B', But we've already visited B. So we do not visit it

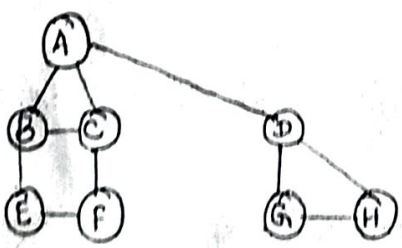


Hence we pop the visited nodes & check if any alternative path exists. From node 'A' we get a path to 'D'. Hence 'D' is visited.

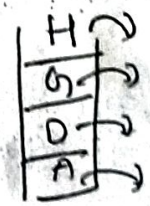


Visit node 'G'

Mrs. VANI K.A, B.E,M.Tech
Lecturer



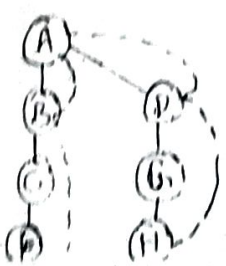
Visit node H. From node 'H' we can visit node 'D' but it is already visited.



As all the nodes are visited in DEPTH first manner we get the DFS sequence as

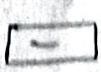
A, B, C, F, E, D, G, H.

The DFS forest can be written as below.



Stack stores all the visited node. To find the alternate edge after

Back edge



Tree edge

DFS:- [Extra]

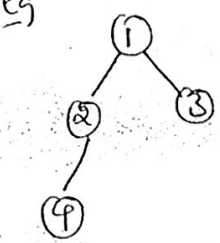
DFS is a method of traversing the graph by visiting each node of the graph in a systematic order.
A name implies dfs means "to search deeper in the graph".

eg:- ① In DFS, a vertex u is picked as source vertex and is visited.
Vertex v adjacent to u is picked and is visited.
Now search begins at vertex v.
There may be still some nodes which are adjacent to u but not visited.
Then the vertex v is completely examined, then only u is examined. The search will terminate when all the vertices have been examined.
Search continues deeper and deeper in the graph until no vertex is adjacent or all the vertices are visited.

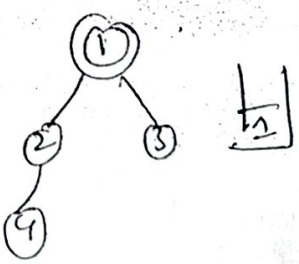
Design:- [Extra]

1) Select node u as start vertex, push u onto stack & mark visited
2) While stack not empty
   for vertex u on top of stack, find next immediate adjacent vertex
   If v is adjacent
      If v is not visited Then
         Push onto stack
         mark as visited
      Else
         Ignore the vertex
      Endif
   Else
      pop vertex from stack
   Endif
End while

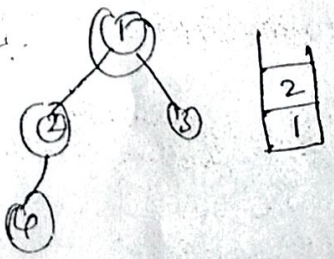
Eg



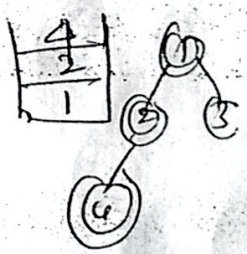
Step1:



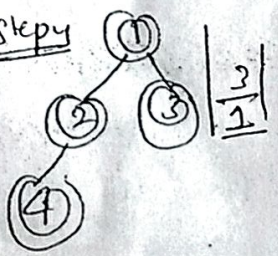
Step2:



Step3:



Step4



Step5 Stack Empty

Tree edge: In a graph 'G' containing an edge (u,v) if a new visited vertex 'v' is reached from the current vertex then edge (u,v) is called tree edge.

Back Edge: In a graph 'G' if a previously visited vertex 'v' is reached from the current vertex 'u' then edge (u,v) is called back edge.

Algorithm

Algorithm DFS(v)

// I/P: The graph 'G' which is a collection of vertices (V) & edges (E)

// O/P:- from which vertex 'v' is selected as a starting vertex.

The sequence of depth first traversal. Initially the vertex 'v' in visited array is marked 1, that means currently we are visiting vertex v.

```
Visit[v] = 1
for (each vertex u adjacent from v) do
{
  if (Visit[u] = 0) then
    DFS(u) //recursive call from next selected vertex u.
}
```

or

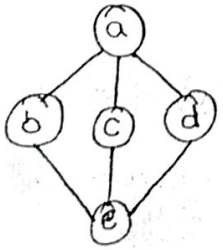
```
{ visited[v] = 1
for(v=1; v<=n; v++)
{
If (cost[u][v]==1 && visited[v]==
    dfs(v);
```

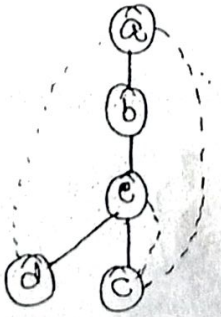
Analysis

Every node is visited once hence the time complexity of DFS is $O(|V|+|E|)$ if the graph is created using adjacency list & it is $O(|V|^2)$ if the graph is created using adjacency matrix.

Applications of Depth First Traversal

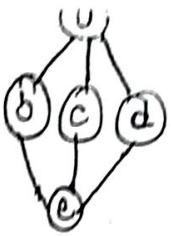
1.) DFS is used for checking connectivity of a graph.
- Start traversing the graph using Depth first method & after an algorithm halts if all the vertices of graph are visited then the graph is said to be a connected graph.

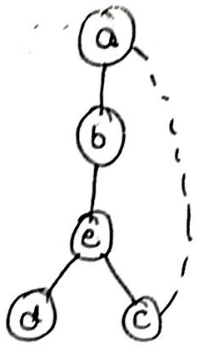




Mrs. VANI K.A, B.E,M.Tech
Lecturer

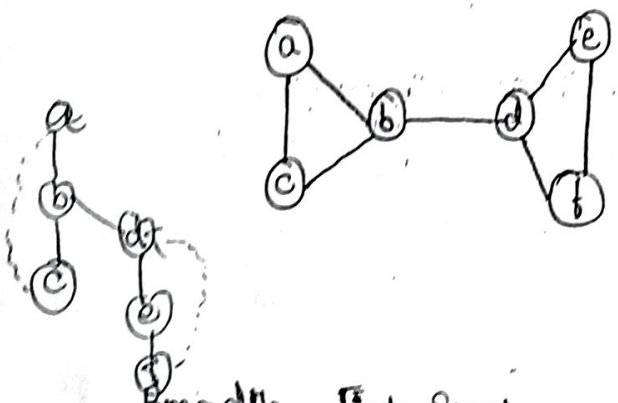
2.) DFS is used for checking acyclicity of graph. If the DFS forest does not have back edge then the graph is said to be





This graph contains cycle, ∵ it contains back edge.

3.) DFS is used to find articulation point.



A vertex of connected graph is said to be its articulation point if its removal with all its incident edges breaks the graph into disjoint pieces.
The vertex 'd' in an articulation point.

Breadth First Search

BFS follows the following rules:-

1.) Select an unvisited node V, visit it. Its level is called the current level.

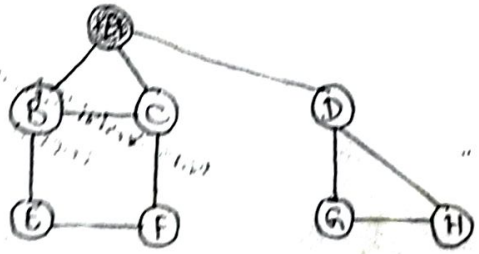
2.) From each node 'X' in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbours of X. The newly visited nodes from this level form a sub level. This new level becomes the next current level.

3.) Repeat step 2 for all the unvisited vertices.

4.) Repeat from step 1 until no more vertices are remaining.

Eg:- Consider the following graph.

Sol:-



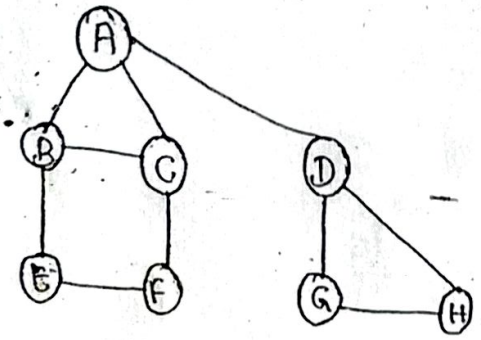
We will start BFS from node A. Hence insert 'A' in the queue Q.

Q:

A	

front/rear

Step 2:- Delete 'A' & print it as traversed node.



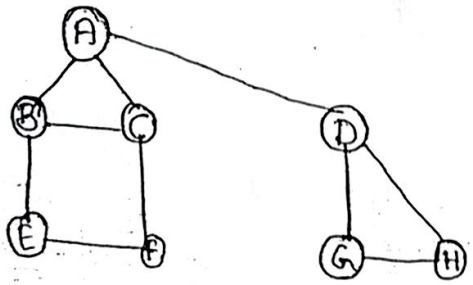
Adjacent to 'A' are nodes B, C, & D. So we'll insert all the adjacent node of 'A' to the queue.

Q

B	C	D

front rear

S3:- Delete 'B' & print it as traversed node.

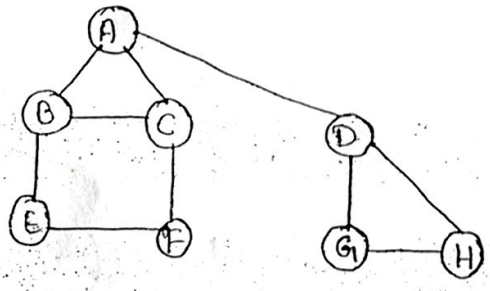


Now find adjacent node of node 'B' i.e. E. & insert it in queue
[Although 'C' is an adjacent node to 'B' but it is already inserted in queue
∴ insert 'E']

C	D	E

front rear

S4:- Delete 'C' & print it as traversed node.



Output

A	B	C

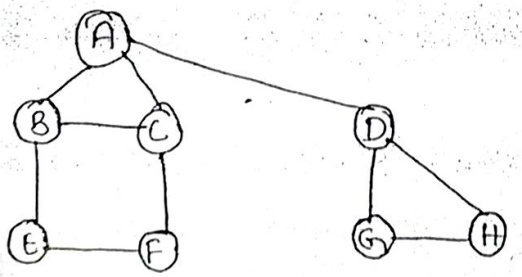
Now insert adjacent unvisited node of 'C' i.e. F.

Q

D	E	F

front rear.

S5:- Delete 'D' & print it as traversed node.



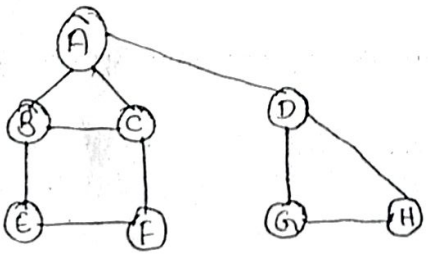
Output

A	B	C	D

Now insert adjacent nodes of 'D' in queue. Q

E	F	G	H

S6:- Delete 'E' & print it as traversed node.

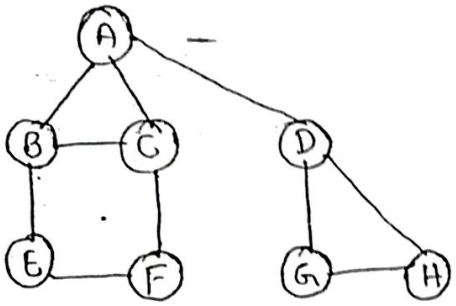


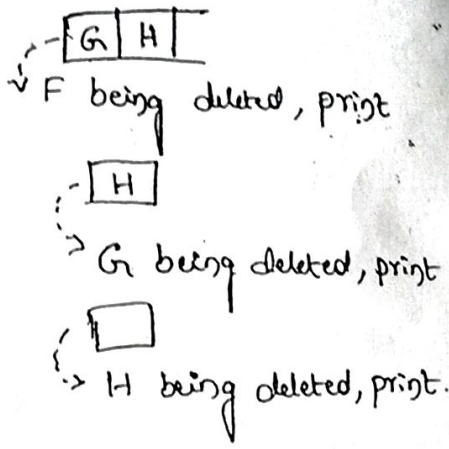
Output

A	B	C	D	E

Now insert adjacent node of 'E' which is unvisited. As all the adjacent nodes 'E' are already inserted in queue. We will not insert any more node in queue. Hence queue will remain. Q

the vertices from queue & print.





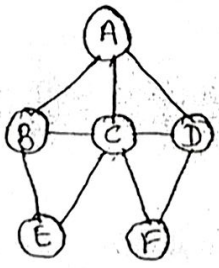
output

← As queue is empty we get display for BFS as

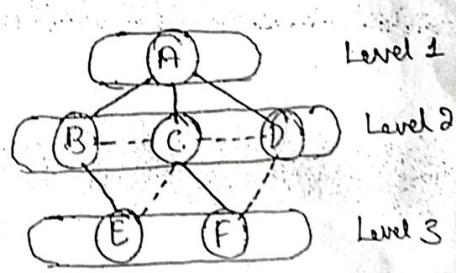
Basic Terminologies in Breadth First Traversal.

Breadth first Forest :- It is a collection of trees in which the traversals starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is visited then it is attached as a child to the vertex from which it is being reached.

Consider following graph



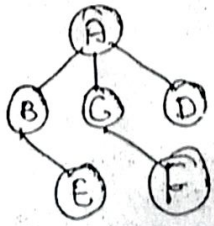
A, B, C, D, E, F



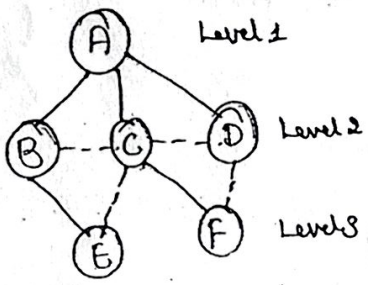
BFS Forest.

Tree edge:- In a graph 'G' containing an edge (u, v), if a new unvisited vertex 'V' is reached from the current vertex then edge (u, v) is called tree edge.

The edges b/w these vertices are tree Edges



Cross edge:- If an edge leading to a previously visited vertex other than its immediate predecessor is encountered then that edge is called Cross edge. It is shown by dotted line in the figure below.



Mrs. VANI K.A, B.E,M.Tech
Lecturer

Algorithm:-

```
Algorithm BFS (V1)
// I/P:- The vertex V1 from which the breadth first searching is done.
// O/P:- The BFS sequence for the given graph.
{
  visit[V1] ← TRUE
  front ← rear ← -1
  Q[++rear] ← V1
  while (front != rear) do
  {
    V1 ← Q[++front];
    write(V1);
    for V2 ← 0 to V2 < n do
    {
      if (g[V1][V2] = TRUE && visit[V2] = FALSE) then
      {
        Q[++rear] ← V2;
        visit[V2] ← TRUE;
      }
    }
  }
}
```

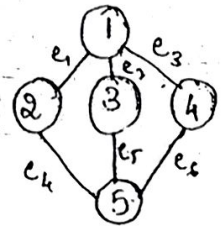
```
Algm bfs(int u)
{
  visited[u]=1;
  q[r]=u;
  while(f<=r)
  {
    u = q[f++];
    for(v=1, v<=n, v++)
    {
      If(a[u][v]==1 && visited[v]==
      {
        visited[v]=1;
        q[++r]=v;
      }
    }
  }
}
```

Analysis:- Every node is visited once hence the time complexity of BFS is $O(|V|+|E|)$ if the graph is created using the adjacency list & it is $O(|V|^2)$ if the graph is created using adjacency matrix.

(1) For finding the connected components in the graph.
(2) For checking if any cycle exist in the given graph.
(3) To obtain shortest path between two vertices.

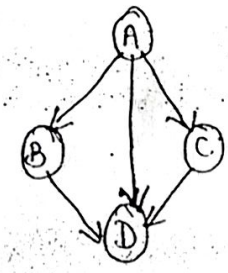
Topological Sorting:-

Graph:- A graph is a collection of vertices (or) nodes, connected by a collection of edges.

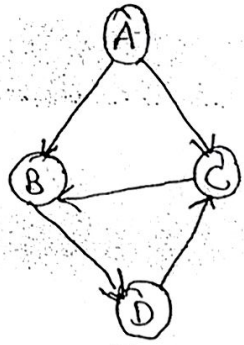


$G = \{V, E\}$
$V = \{1, 2, 3, 4, 5\}$
$E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

DAG:- (Directed Acyclic Graph)
Is a graph with NO cycle. Also a graph is directed.



It is a DAG



It is not DAG

Based on DAG specific ordering of vertices is possible. This Method of arranging the vertices in some specific manner is called Topological sort. (or) Topological Ordering.

Topological sorting is done in two ways:-
(1) Source Removal Method &
(2) DFS Method.

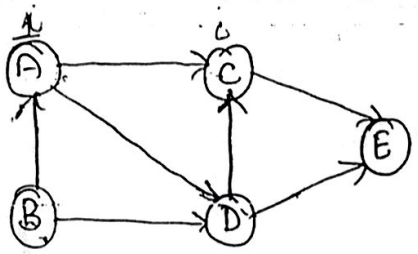
Source Removal Algorithm (SRA)

This is a direct implementation of decrease & conquer method.

Following are the steps to be followed in this algorithm.

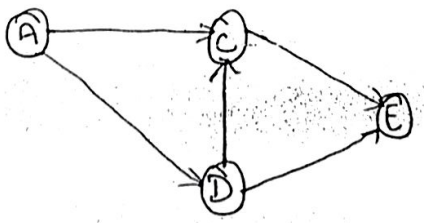
1. From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it. If there are more than one such vertices then break the tie randomly.
2. Note the vertices that are deleted.
3. All these recorded vertices give topologically sorted list.

Eg:- Sort the digraph for Topological Sort using SRA method?



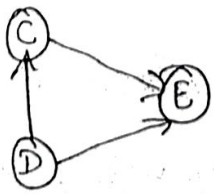
Choose vertex (B) ∵ it has no incoming edge, delete it along with its adjacent edge.

Delete B →

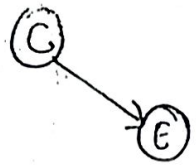


Now delete (A) ∵ it has no incoming edges, delete it long with its adjacent edge.

Delete A →



Now Delete (D) ∵ its incoming edges are 0, delete it along with its adjacent edge.

Delete D → 

Now delete (C) ∵ it has no incoming edge. Along with its adjacent edges.

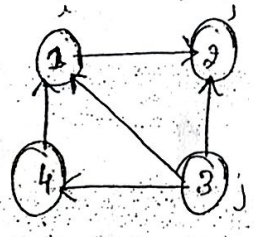
Delete C → (E)

Finally Delete 'E'

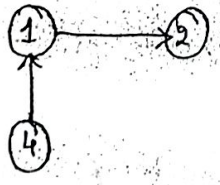
Hence the list after Topological sorting will be

B, A, D, C, E.

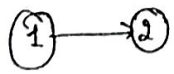
2 Eg.) Consider the following graph.



Delete (3) since it has no incoming vertices.



Now delete (4)



Now delete (1)

(2)

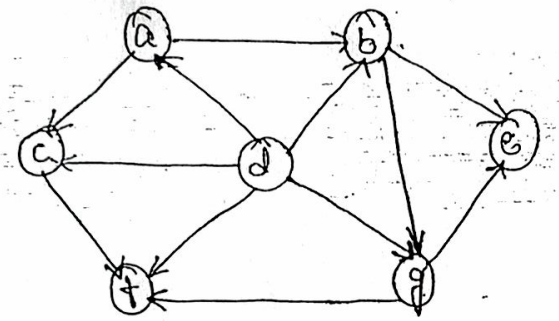
The topological order is 3, 4, 1, 2

s Method

The following steps are used to find topological order.

1.) Select any arbitrary vertex.
2.) When vertex is visited first time it is pushed on to stack.
3.) When vertex becomes a dead end. it is removed.
4.) Repeat above two steps till all nodes are over.
5.) Reverse the order of deleted nodes.

Consider the graph shown below:-



Mrs. VANI K.A, B.E,M.Tech
Lecturer

d, a, c b, g, f, e

Let source be 'a'

Stack	Nodes to be visited	Node deleted. Poped.
[a]	b	—
[a|b]	e	—
[a,b|e]	From 'e' no nodes to visit, so delete 'e'	e
[a|b]	g	—
[a,b,g]	f	—
[a,b,g,f]	From 'f' no nodes to visit, so delete 'f'	f
[a,b,g]	From 'g' no nodes to visit, so delete 'g'	g
[a,b]	From 'b' no nodes to visit, so delete 'b'	b
[a]	c	—
[a, c]	From 'c' no nodes to visit, so delete 'c'	

visit to delete 'a'	a

Now stack is empty, any node left in graph is push on to stack. The left out node is 'd'.

d	from 'd' no node to visit so delete 'd'	d

The node popped sequence is

e, f, g, b, c, a, d

The topological sort is reverse of popped sequence

d, a, c, b, g, f, e

Algorithm:-

```
Algorithm topo(S,n,a)
  // S is array indicates node visited/not
  // a is adjacency matrix
  // n is no of vertices.
S1: for u ← 1 to n do
    if (S[u]=0) call DFS (u,n,a)   // S[u] = Visited
  endfor
S2: for i ← n down to 1 // Print in Reverse order of popped vertex
    Print result[i]
S3: End.

Algorithm DFS (u,n,a)
  begin
    S[u] ← 1   // S[u] = visited
    for v ← 1 to n do
  if (a[u][v]= 1 and S[v]= 0) then   // S[v] = visited
    DFS (v,n,a)
  End if
  End for
  result[j++] = u
End.
```

Efficiency of Topological sorting for DFS

Let T(n) be the time taken by algorithm to sort vertices. The basic (or) key operation is

```
for v ← 1 to n do
    if (a[u][v] = 1 and S[v] = 0) then
        DFS(v, n, a)
```

```
for u ← 1 to n do
    if (S[v] = 0) call
    DFS(u, n, a)
```

$$T(n) = \sum_{u=1}^{n} \sum_{v=1}^{n} 1$$

$$= \sum_{u=1}^{n} n - 1 + 1$$

$$= \sum_{u=1}^{n} n$$

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

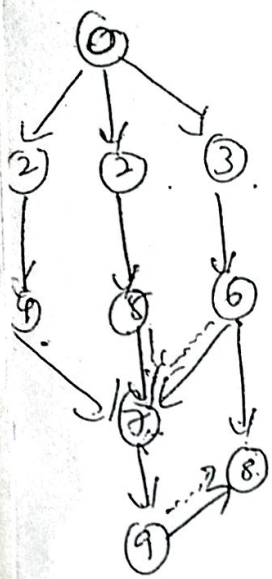
No. time efficiency is $T(n) = \theta(n^2)$

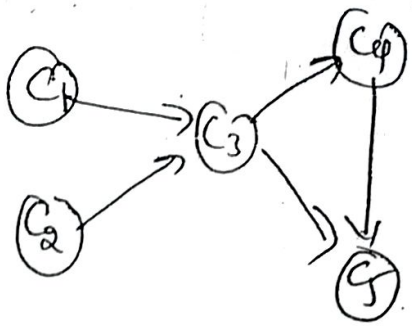
Time efficiency of Topological sorting by source removal Method.

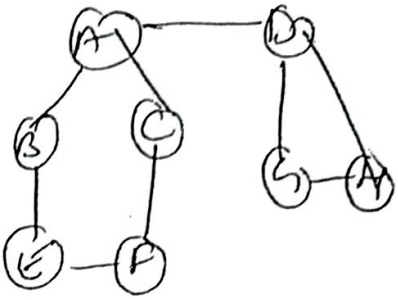
Algorithm Topology (a, n)

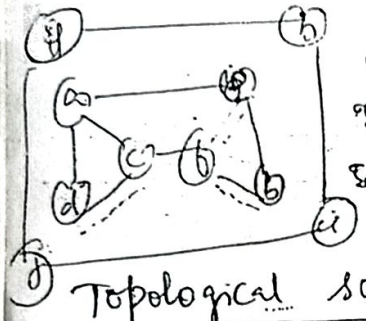
// a is adjacency matrix, 'n' is the no of nodes

```
begin
    for i ← 1 to n do
        for j ← 1 to n do
            if (a[i][j] = 1) sc[j] = sc[j] + 1
        end
    end
    for i ← 1 to n do
    begin
        if (sc[i] = 0) then
```










```
print i
SC[i] = -1
for j ← 1 to n do
begin
if (a[i][j] = 1) then
  sc[j] = sc[j] - 1
End if
i = 0
End for
End if
End for
End.
```

Topological sorting using Source-Removal technique

```
Algm topological sorting (cost[][])
{
    for(i=g;i<=n;i++)
    for(j=1;j<=n;j++)
    Indegree[i]= indegree[i] + cost[j][i];
    count=0;
    while(count<n)
    {
        for(i=1;i<=n;i++)
        If( indegre[i]==0 && !visited[i])
        {
            Printf("%d",i);
            visited[i]=1; //deleting
            count++;
            for(w=1;w<=n;w++)
                If (cost[i][w]==1)
            Indegree[w]--; //Reducing the Indegree
                           Removing the
                           outgoing edges
        }
    }
```

403
404 16
406 18
407 21
408 31
414 28
416 24
28
1 33
5 31
9 35
10 4
12 4
13 41
52
5
5