Tutorial-5

Solution-1:)

BFS	DFS
(i) BFS stands for Breadth first Search.	(i) DFS stands for Depth first Search.
(ii) BFS uses queue to find the shortest path.	(ii) DFS uses stack to find the shortest path.
(iii) BFS is better when target is close to source. As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	(iii) DFS is more suitable for decision tree. As with one decision we need to traverse further to argument the decision. If we reach the conclusion.
(iv) BFS is slower than DFS.	(iv) DFS is faster than BFS.

Application of BFS →

→ If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree.

→ We can detect cycle in a graph using DFS.

→ If we get one back-edge during BFS then there must be one edge.

→ Using DFS we can find path b/w two given vertices u & v.

Application of DFS →

→ Like DFS, BFS may also used for detecting cycles in a graph.

→ Finding Shortest path and minimal spanning tree in unweighted graph. (2)

→ Finding a route through GPS navigation system with minimum number of crossings.

→ In networking finding a route for packet trans-mission.

Solution- 2:)

BFS (Breadth First Search) use queue data structure and DFS (Depth First Search) uses Stack data Structure.

A queue (FIFO - first in first out) data structure is used by BFS. You mark any node in the graph as root and start traversing the data from it. BFS traverse asl the nodes in the graph and keep dropping them as completed BFS visits an adjacent unvisited node, mark it as done, and insert it into a queue.

DFS algorithm traverse a graph in depthwardmotion and uses a Stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Solution-3:)

(i) Sparse Graph → A graph in which the number of edges is much less than the possible number of edges.

(ii) Dense Graph → A dense graph is a graph in which the number of edges is close to the maximal no. of edges.

If the graph is sparse we should store it as a list of edges. Alternatively, if the graph is dense, we should store it as a adjacency Matrix.

Solution: 4)

The existence of a cycle is directed and undirected graph can be determined by whether depth first search (DFS) finds an edge that points to an ancestors of the current vertex (it contains a back edge). All the back edge which DFS skips over are part of cycles.

* Detect cycle in a directed Graph.

DFS can be used to detect a cycle in a graph. DFS for a connected graph produce a tree. There is a cycle in a graph only if there is a back edge that is from a node to itself (self-loops). or one of its ancestors in the tree produced by DFS, then the for a disconnected graph, Get the DFS for as output to detect cycle check a cycle in individuals tree by checking back edge.

To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS traversal. If a vertex is reached that is already in the recursion stack, then there is a cycle in the tree. The edge that connects the current vertex to the vertex in the recursion stack is a back edge. Use rec stack [] array to keep track of vertices in the rec. stack.

* Detect cycle in an undirected Graph.

Run a DFS from every unvisited node. DFS can be used to detect a cycle in a graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is joining a node to itself or one of its ancestors in the tree produced by DFS.

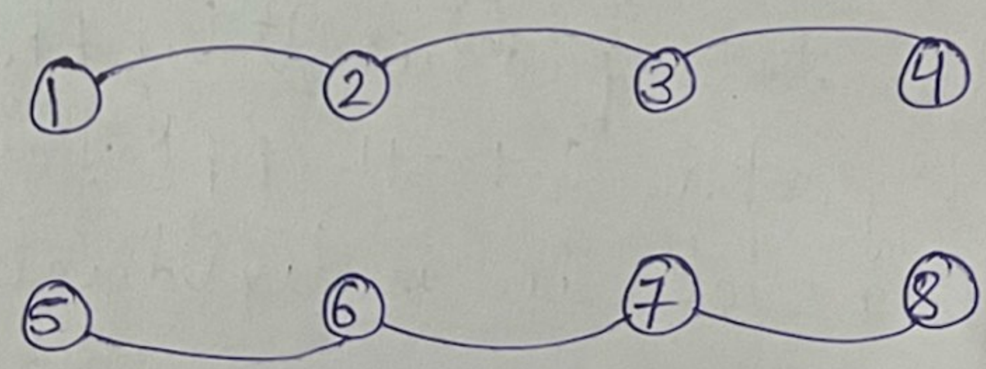
Solution: 5)

Disjoint Set Data Structure :→

It allows to find out whether the two element are in the same set or not efficiently.

The disjoint set can be defined as subset where there is no common element b/w the two sets.

Eg → $S_1 = \{1, 2, 3, 4\}$
$S_2 = \{5, 6, 7, 8\}$



Operations performed →

(i) Find: can be implemented by recursively transfer the parent array until we hit a node who is parent to itself.

```
int find (int i)
{
  if(parent [i] == i)
    return i ;
```

```
}
else
{
    return find(parent[i]);
}
}
```

(ii) <u>Union</u> :→ It takes, as input, two elements. And finds the representation of their sets usings the find operation, and finally puts either one of trees (representing the set) under the root node of the other tree, effectively merging the trees & the sets.

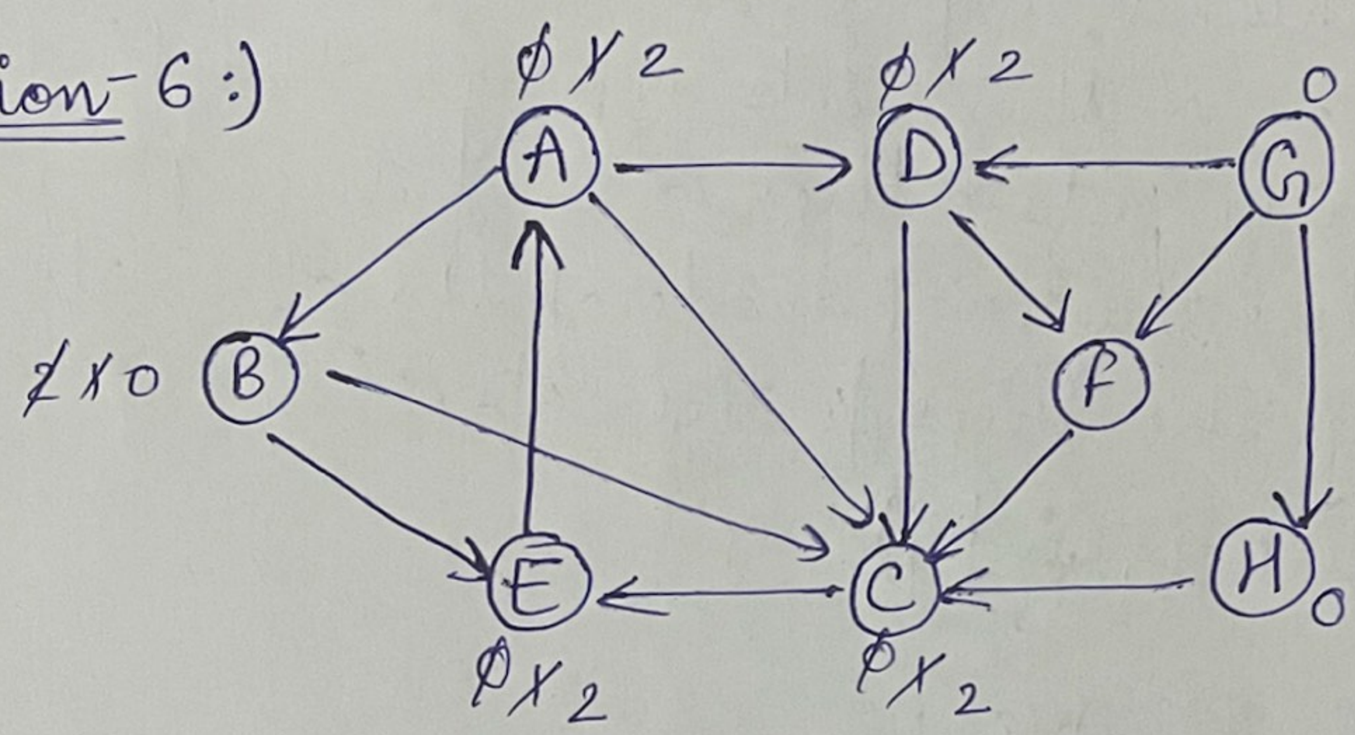
```
void union (int i, int j)
{
  int irep = this.find(i);
  int jrep = this.find(j);
  this.parent [irep] = jrep ;
}
```

(iii) <u>Path Compression</u> (Modificat^n to find ()) :→ It speeds up the data structure by compressing the height of the trees. It can be acheived by inserting a small caching mechanism into find operation.

```
int find (int i)
{
  if (Parent [i] == i)
  {
    return i ;
  }
  else
```

```
{
  int result = find(Parent [i]);
  Parent [i] = result;
  return result;
  }
}
```

<u>Solution</u> - 6 :)

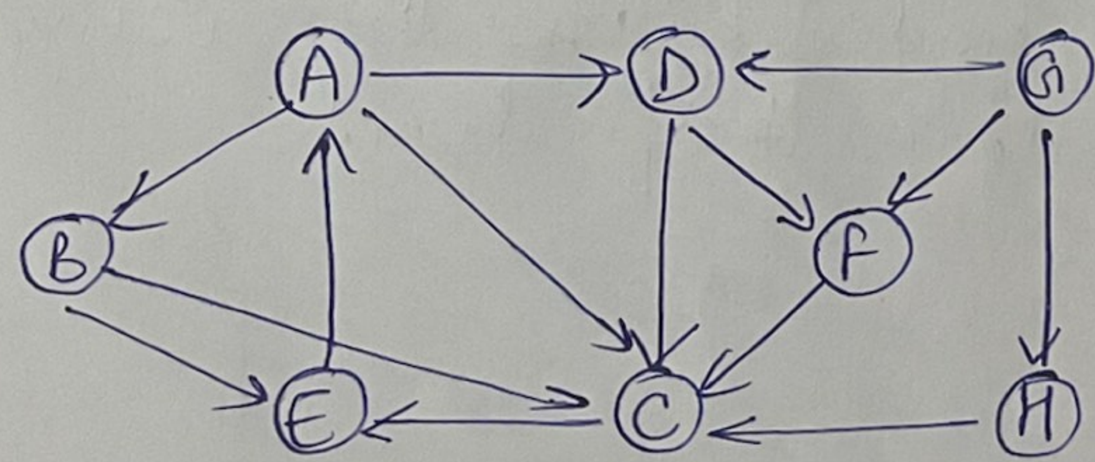


<u>BFS</u> → Node	B	E	C	A	D	F
Parent	–	B	B	E	A	D

Unvisited nodes :→ G and H

Path → B → E → A → D → F

<u>DFS</u> →



Node processed	B	B	C	E	A	D	F
Stack	B	CE	EE	AE	DE	FE	E

Path :→ B → C → E → A → D → F

Solution-7:)

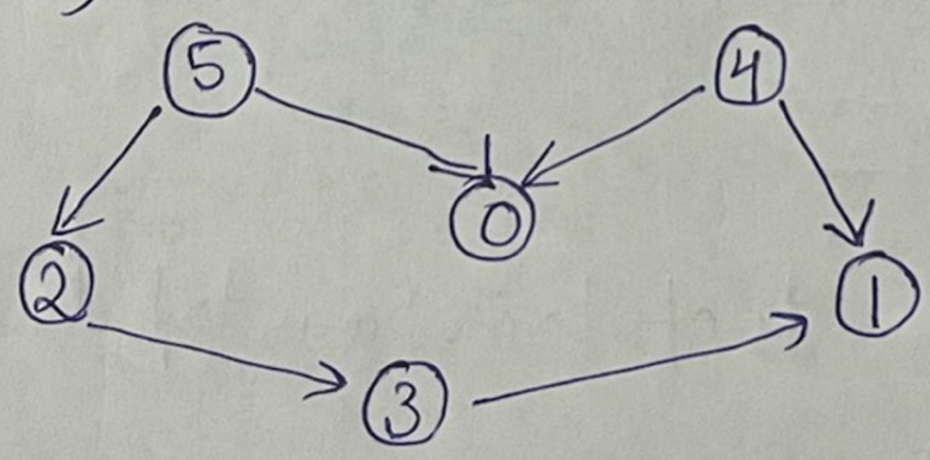
V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

E = {a, b}, {a, c}, {b, c}, {b, d}, {e, f}, {e, g}, {h, i}, {j}

(a, b)	{a, b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(a, c)	{a, b, c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}	
(b, c)	{a, b, c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}	
(b, d)	{a, b, c, d}	{e}	{f}	{g}	{h}	{i}	{j}		
(e, f)	{a, b, c, d}	{e, f}	{g}	{h}	{i}	{j}			
(e, g)	{a, b, c, d}	{e, f, g}	{h}	{i}	{j}				
(h, i)	{a, b, c, d}	{e, f, g}	{h, i}	{j}					

No. of connected components = 3 ans.

Solution-8:)



Adjacent list
0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

Visited :→

0	1	2	3	4	5
false	false	false	false	false	false

Stack (empty)

Step 1 → Topological sort (0), visited [0] = true
list is empty, No more recursion call

Stack [0]

Step 2:) Topological sort (1), visited [1] = true
list is empty. No more recursion call.

Stack | 0 | 1 |

Step -3) Topological sort [2], visited [2] = true
list is empty. No more recursion call.

Stack | 0 | 1 | 3 | 2 |

Topological sort [3], visited [3] = true
'1' is already visited, No more recursion call.

Step -4) Topological sort [4], visited [4] = true
'0', '1' are already visited. No more recursion call.

Stack | 0 | 1 | 3 | 2 | 4 |

Step -5) Topological Sort [5], visited [5] = true
'2', '0' are already visited. No more recursion call.

Stack | 0 | 1 | 3 | 2 | 4 | 5 |

Step -6) Print all the elements of Stack from top to bottom.

5, 4, 2, 3, 1, 0.

Solution-9:) We can use heaps to implement the priority queue. It will take O (log N) time to insert & delete each element in the priority queue. Based on heap structure, priority queue also has 2 types - max and min priority queue.

Some algo. where we need to use priority queue are :→

(i) Dijkstra's shortest path Algo. using priority queue :→
→ When the graph is stored in the form of adjacency list or matrix, priority queue can be used extract minimum efficiently when implementing Dijkstra's Algo.

(ii) Prim's Algo. :→ It is used to implement Prim's Algo. to store keys of nodes & extract min. key node at every step.

(iii) Data Compression :→ It is used in Huffman's code which is used to compress data.

Solution - 10 :)

Min-heap	Max-heap
(i) In a Min-heap, the key present at the root must be less than or equal to among the keys present at all of its children.	(i) In a max-heap, the key is present at the root node must be greater than or equal to among the keys present at all of its children.
(ii) The minimum key element present at the root.	(ii) The maximum key element present at the root.
(iii) Uses the ascending priority.	(iii) Uses descending priority.