Tutorial-5

Solution:-1.

BFS	DFS
BFS stands for Breadth first search.	DFS stands for Depth first search
BFS uses queue to find the shortest path.	DFs uses stack to find the shortest path.
Bfs is better when target is close to source.	DFs is more suitable for decision tree. As with one decision, we need to traverse further to argument that the decision. If we reach the conclusion.
As BFs considers all neighbour so it is not suitable for decision tree used in puzzle games.	
BFS is slower than DFS.	DFS is faster than BFs.

Application of DFS:-

If we perform DFS on unweighted graph, then it will create minimum spaning tree for all pair shortest path tree.
We can detect cycles in a graph using DFS If we get one back-edge during BFS then there must be one edge.
Using BFS we can find path between two given vertices u^2qv.

Application of BFS:-

1. Like DFS, BFS may also used for detecting cycles in a graph.
2. Finding shortest path and minimal spanning tree in unweighted graph.
3. Finding a route through GPS navigation system with minimum number of crossings.
4. In networking finding a route for packet transmission
5. In building the index of search engine transist

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 all 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 depthward motion 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

1. Sparse Graph :- A graph in which the number of edges is much less than the possible number of edges.

2. Dense Graph :- A dense graph is a graph in which the number of edges is close to the maximal number of edges.

If the graph is spare 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 forest as output to detect cycle check a cycle in individuals trees 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 recursion 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 ancestor in the tree produced by DFS. To find the back edge to any of its ancestor keep a visited array, and if there is a back edge to any visited node then there is a loop and return true.

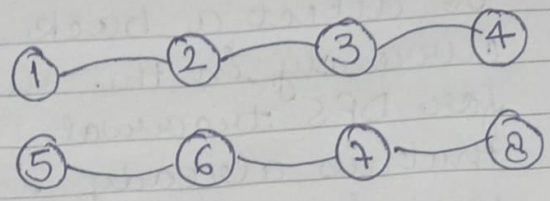
Solution:-5

Disjoint Set Data Structure:-
It allow to find out whether the two element are in the same set or not efficiently.

Date

The disjoint set can be defined as the 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 traverse the parent array until we hit a node who is present to itself.

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

ii) Union : It takes, as input, two elements. And finds the representation of their sets using the find operation, and finally puts either one of the trees under the root node of the other tree, effectively merging the tree and the sets.

```
void union (int i, int j) {

    int irep = this.find (i);
    int jrep = this.find (j);
```

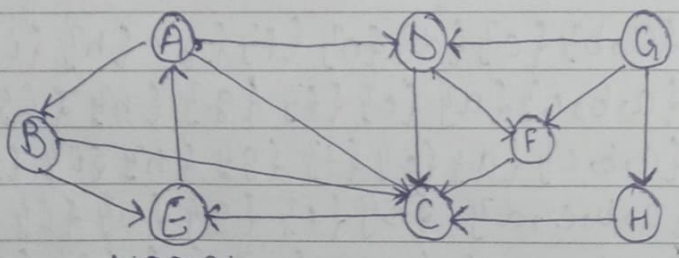
Date

```
    this.parent[irep] = jrep;
}
```

3.) Path compression (Modification to find ());
It speeds up the data structure by compressing the height of the trees. It can be achieved 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;
}
}.
```

Solution:- 6



NODE:-

Bfs:- (B) (E) (C) (A) (D) (F)

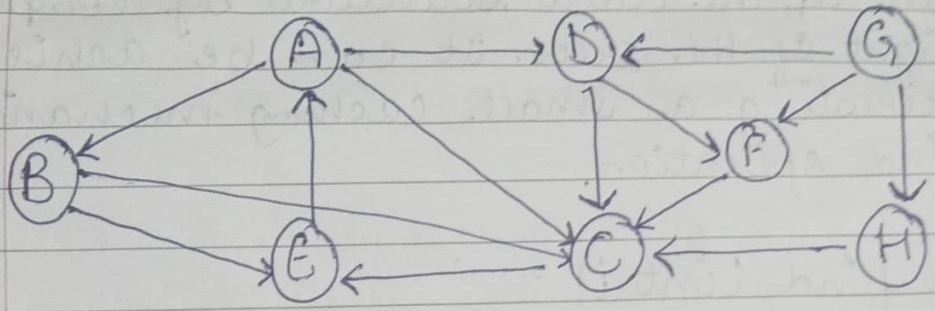
PARENT:-

B B E A D.

Date

unvisited nodes :- G and H.
Path → B → E → A → D → F.

Depth first Search :-



Node processed :- B B C E A D E
Stack :- B CE EE AE DE FE E
Path :- B → C → E → A → D → E.

Solution :- 7

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

$E = \{a,b\}\{a,c\}\{b,c\}\{b,d\}\{e,f\}\{e,g\}$
$\{h,i\}$.

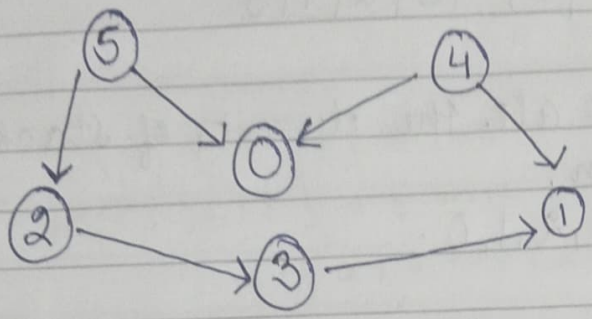
(a, b)	{a,b}{c}{d}{e}{f}{g}{h}{i}{j}
(a, c)	{a,b,c}{d}{e}{f}{g}{h}{i}{j}.
(b, c)	{a,b,c}{d}{e}{f}{g}{h}{i}{j}.
(b, d)	{a,b,c,d}{e}{f}{g}{h}{i}{j}.
(e, f)	{a,b,c,d}{e,f}{g}{h}{i}{j}.
(e, g)	{a,b,c,d}{e,f,g}{h}{i}{j}.
{h, i}	{a,b,c,d}{e,f,g}{h,i}{j}.

Number of connected component = 3 ans.

Date

Solution :- 8

Topological Sort.



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

visited :-

0	1	2	3	4	5
f	f	f	f	f	f

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.
↓
Topological Sort (3) visited [3] = true
'1' is already visited, No more recursion
Stack | 0 | 1 | 3 | 2 |

Step 4 :- Topological Sort (4), visited [4] = true
'0' '1' are already visited. No more recursion C.
Stack | 0 | 1 | 3 | 2 | 4 |

Step 5:- Topological Sort(5), visited(5) = true '2', '0' are already visited. No more recursive call. Stack | 0 | 1 | 3 | 2 | 4 | 5 |

Step 6:- Print all the elements of stack from top to bottom.
5, 4, 3, 2, 1, 0.

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

i) Dijkstra shortest path Algorithm :- When the path is sorted in the form of adjacency list or matrix, priority queue can be used extract minimum effeciently when implementing Dijkstra's algorithm.

ii) Prim's algorithm :- It is used to implement prim's algorithm to store key of nodes and extract minimum key node at every step.
Data compression :- It is used in Huffman's code which is used to compress data.

Solution :- 10

Min-Heap	Max Heap
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.	In a max-heap the key present at the root node must be greater than one equal to among the keys present all of its children
→ use ascending priority	use descending priority.