TUTORIAL-5

Q1) Solution :-

BFS	DFS
It stands for Bredth First Search.	It Stands for Depth first Search.
It uses Queue data Structure.	It uses Stack data structure.
It is more suitable for searching Vertices, which are Closer to given Source.	It is more suitable when there are solutions away from Source.
BFS Considers all neighbours first & therefore not suitable for decision making trees used in games and puzzles.	DFS is more suitable for games or puzzle problems. We make a decision then explore all paths through this decision, And if decision leads to win situation, we stop.
Here siblings are ~~wat~~ visited before the children.	Here, children are visited before the Siblings.
There is no concept p backtracking	It is a recursive algorithm that uses backtracking.
It requires more memory	It requires less memory.

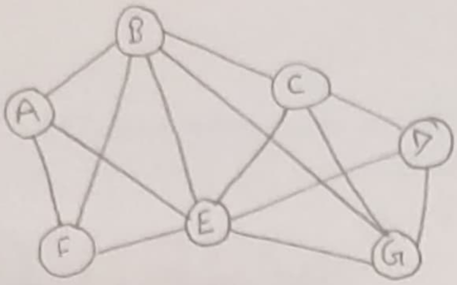
→ Application :-

- BFS → Bipartite graph and shortest path, peer to peer networking, Crawlers in Search Engines & GPS navigation systems
- DFS → Acyclic graph, topological order, Schaduling problems, Sudoku puzzle.

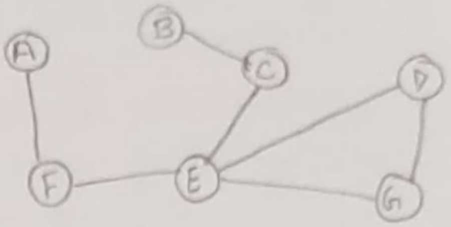
Q2) Solution :- For implementing BFS we need a queue data structure for finding shortest path between any node. We use queue because things don't have to be processed immediately, but have to processed in FIFO order like BFS. BFS searches for nodes level wise, i.e. it searches nodes with respect to their distance from root (source). For this queue is better to use in BFS.

For implementing DFS we need a Stack data structure as it traverse a graph in depthword motion and uses Stack to remembers to get the next index to start a search, when a dead end occurs in any iteration.

Q3> Solution:- Dense graph is a graph in which no. q edges is close to maximal no. q edges.
Space graph is graph in which no. q edges is ~~close to maximal~~ very less.



Dense Graph
(many Edges b/w nodes)



Sparse graphs (few edges b/w nodes)

- For sparse graph it is preferred to use Adjacency list.
- For dense graph it is preferred to use Adjacency Matrix.

Q4> Solution:- For detecting cycle in a graph using BFS we need to use kahn's algorithm for Topological sorting.

The steps invalved are:

1> Compute in-degree (no. of incoming Edges) for each q vertex present in graph and initialize count of visited nodes as 0.

2> Pick all vertices with in-degree as 0 and add then in queue.

3> Remove a vertex from queue and then.
- Increment count q visited nodes by 1.
- Decrease in-degree by 1 for all its neighbouring nodes.
- If in-degree q neighbouring nodes is reduced to use then add to queue.

4> If count q visited nodes is not Equal to no. of nodes in graph has cycle, Ether wise not.

For dectecting cycle in grap using DFS we need to do following:
DFS for a connected graph produces a tree. There is cycle in graph if there is a back Edge present in the graph. A back Edge in an Edge that is from a node to itself (self-loop) or one q its ancestors in the tree produced by DFS. For a disconnected graph, get DFS first as output. To detect cycle, check for a cycle in individual trees by checking back edges. To detect a back edge, back track of ~~[illegible]~~ vertices currently in recursive track for DFS traversal. If a vertex is reached that is already in recursion stack, then there is a cycle.

Q5> Solution :- A disjoint set is a data structure that keeps of set of elements partioned into several disjoint subsets. In other words, a disjoint set is a group of sets where no item can be more than one set.

3 Operations !

- Find → Can be implemented by recursively ~~[illegible]~~ traversing 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]);
        }
    }
```

- Union → It takes 2 elements as input. And find representatives of this sets using the find operation and finally puts either one of the trees under root node of other tree, effectively merging the trees and sets.

Eg :-

```
mid union (int i, intj) {
    int irep = this. Find (i);
    int jrep = this. Find (j);
     this. parent [irep] = jrep;
    }
```

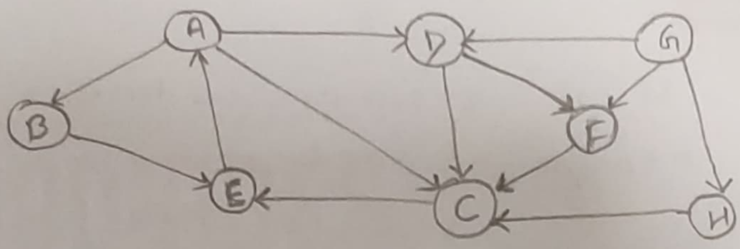
- Union by Rank → We need a new array rank [ ], size of array same as parent array If i is representative of set, rank [i] is height of tree. We need to minimize height of tree. If we are iterating two trees, we call them left and right, then it all depends on rank of left and right.
  - If rank of left is less than right then it's best to move left under right and vice versa.
  - If ranks are equal, rank of result will always be one greater than rank of trees.

Eg :-

```
Void union ( int i, int j ) {
     int irep = this. find (i);
      int jrep = this. find (j);
           if (irep == jrep) return;
            irank = Rank [irep];
            jrank = Rank [jrep];
          if ( irank < jrank )
                this.parent [irep] = jrep;
```

```
        else if (jrank < irank)
              this.parent [jrep] = irep;
        else
              this.parent [irep] = jrep;
              Rank [jrep]++;
    }
```

Q6) Solution:-



BFS

Child	G	H	D	F	C	E	A	B
Parent		G	G	G	H	C	E	A

Path → G → H → C → E → A → B.

DFS

~~G~~
D
H
~~F~~
~~C~~
~~E~~
~~A~~
~~G~~
} NODES VISITED

G
F
C
E
A
B
} STACK

Path → G → F → C → E → A → B

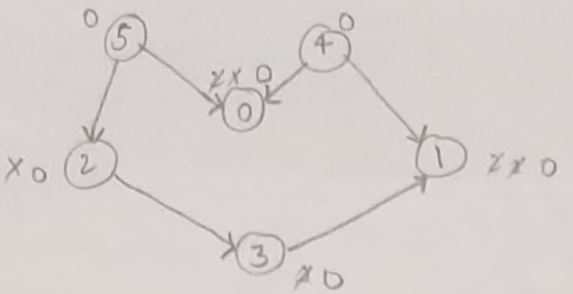
Q7) Solution:- V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

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

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

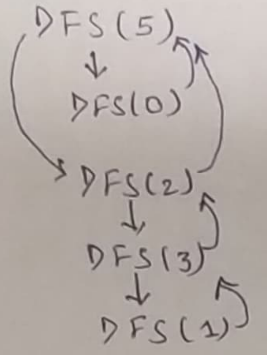
No. of Connected Components = 3

Q8> Solution :-



We take source node as 5.

Applying Topological Sort



DFS(4)
↓
Not possible

q: 5/4 ; pop 5 and decrement indegree of it by 1.

q: 4/2 ; pop 4 and decrement indegree and push 0.

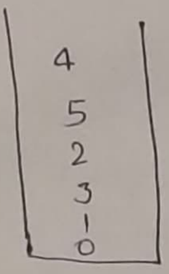
q: 2/0 pop 2 and decrement indegree & push 3.

q : 0/3 pop 0, pop 3 push 1

q : 1 ; pop 1

Answer : $\underbrace{5\ 4\ 2\ 0\ 3\ 1}_{\text{Topological Sort}}$

DFS



Stack

4 → 5 → 2 → 3 → 1 → 0

Q9> Solution :- Yes, heap data structure can be used to implement priority queue. It will take O(log N) time to insert and delete each element in priority queue. Based on heap structure, priority queue has two types max-priority queue based on max heap and min priority queue based on min-heap. Heaps provide better performance comparision to array. ~~and~~

The Graph like Dijkatra's shortest path algorithm, prim's Minimum spanning tree use priority Queue.

- <u>Dijkatra's Algorithm</u> :- When graph is sorted in form of adjacency list or matrix, priority queue is used to extract minimum efficiently when implementing the algorithm.
- <u>Prim's Algorithm</u> :- It is used to store keys of nodes and Extract minimum key node at every step.

Q10> Solution:-

Min - Heap	Max - heap
• In min-heap, key present at root node must be less than or Equal to among keys present at all of its Children.	• In max-heap the key present at root no must be greater than or Equal to am Keys present at all of its Children.
• The minimum key Element is present at the root.	• The maximum key Element is present at the root.
• It uses ascending priority.	• It uses descending priority.
• The smallest Element has priority while Construction of min-heap.	• The largest element has priority while Construction of Max-heap.
• The smallest Element is the first to be popped from the heap.	• The largest element is the first to be popped from the heap.