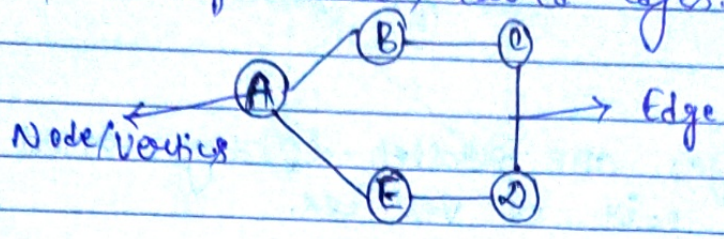
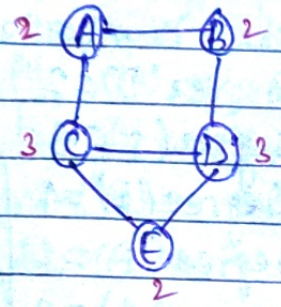
GRAPH

"A graph is a pair (V,E), where V is a set of nodes, called vertices, and E is a collection of pairs of vertices, called edges."



Types of Graph :-

(1.) Undirected Graph

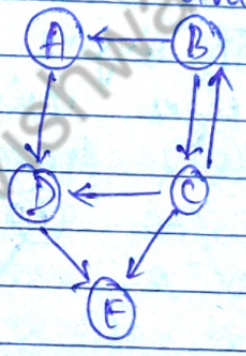


Degree(D) = 3
Degree(A) = 2

Total sum of degree of all vertices = (2 x No. of edges)
= (2 x 6)
= 12

Ex: Flight Network

(2) Directed Graph

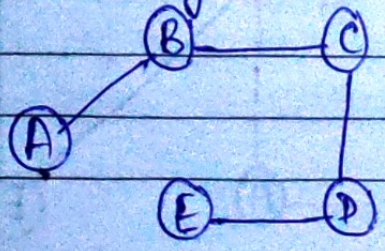


Indegree(C) = 1
Outdegree(C) = 3

Ex: Route Network

Some Important Points :-

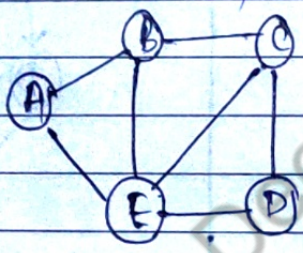
① A graph with no cycles is called a tree. A tree is an acyclic connected path.



(2) A self loop is an edge that connects a vertex to itself.

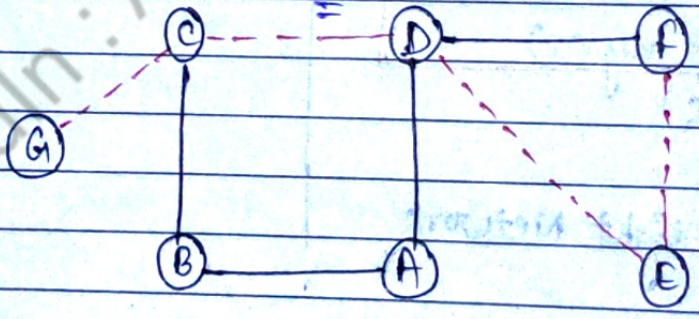
(3) 2 edges are parallel if they connect the same pair of vertices.

(4) The degree of vertex is the number of edges incident on it.

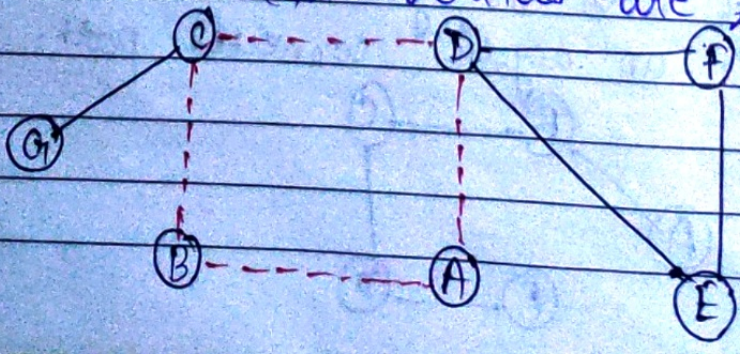


Degree(D) = 2
Degree(C) = 3
Degree(E) = 4
Degree(A) = 2
Degree(B) = 3

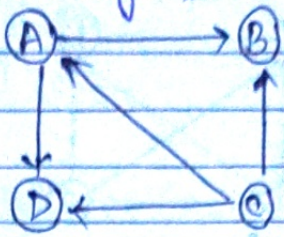
(5) Path :- A path in a graph is a sequence of vertices with no repeated vertices in it. Ex :- Path G to E.



(6) Cycle :- A cycle is a path where first and last vertices are same.

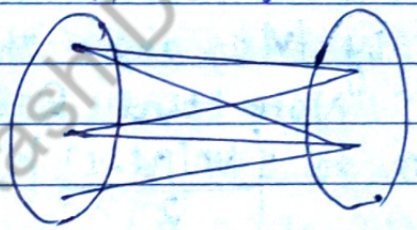


(7) A directed acyclic graph (DAG) is a directed graph with no cycles.

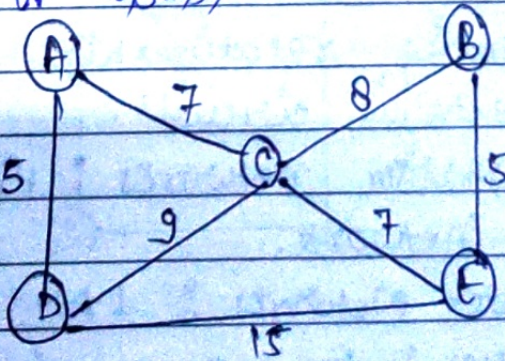


(8) A spanning tree is a connected graph is a subgraph that contains all of that graph's vertices and is a single tree.

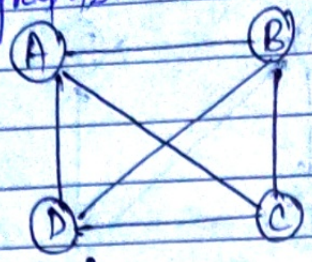
(9) A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.



(10) In weighted graphs integers (weights) are assigned to each edge to represent (distances or costs).



(11) Graphs with all edges present are called complete graphs.



Sparse Graph :-

(12) Graphs with relatively few edges (generally if it edges $< |V| \log |V|$), are called Sparse Graph.

(13) Dense Graph :- Graphs with relatively few of the possible edges missing, are called Dense Graph.

(14) We will denote the no. of vertices in a graph by $|V|$, and the no. of edges by $|E|$. Note that E can range anywhere from 0 to $\left\{\frac{|V|(|V|-1)}{2}\right\}$ (In undirected graph)

Because each node can connecte to every other node.

Applications of Graphs:

(1.) Representing relationships b/w components in electronic circuits.
(2.) Transportation networks : Highway network, flight network.
(3.) Computer Networks : LAN, Internet, Web
(4.) Databases : for representing ER diagrams in databases, for representing

Graph Representation :-

1.) Adjacency matrix
2.) Adjacency list
3.) Adjacency set

(1) Adjacency matrix :- In this representation, we use a 2D matrix of dimension (n x n) where n represents number of nodes present in graph.

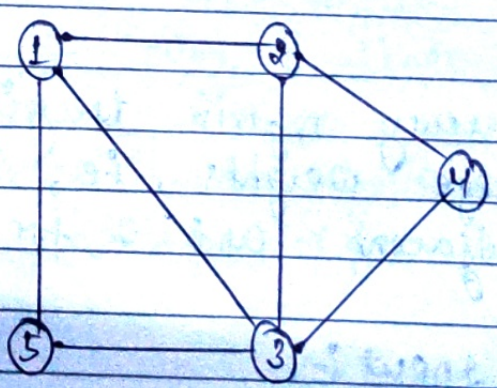
We put true (1) if there is an edge b/w the nodes else put false (0).

No. of nodes → No. of edges

Input : n = 5, m = 7

Edges = { {1,2}, {2,3}, {2,4}, {3,4}, {5,3}, {1,3}, {1,5} }

```
int[][] adjMat = new int[n+1][n+1]; // Because nodes starting from 1 so taking (n+1).
for(int i=1; i<n+1; i++){
    for(int j=1; j<n+1; j++){
        adjMatrix[i][j] = sc.nextInt();
    }
}
```



Case ① An adjacency matrix is given input as it is, i.e, 1 for adjacent and 0 for non-adjacent.

Sample Input :

→ The first line contains a single integer n - The number of nodes.

→ Each of the next n lines contains n space separated integers. The jth integer in the ith row denotes a[i][j].

n = 3

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

```
boolean [][] adjMatBool = new boolean[n][n];
for( int i=0 ; i<n ; i++){
    for( int j=0; j<n ; j++) {
        adjMatBool[i][j] = sc.nextInt() == 1;
    }
}
```

Case ② An adjacency matrix is given as input with edge weights, i.e, some value for adjacent and 0 for non-adjacent.

Sample Input :-

→ The first line contains a single integer n - the number of nodes.

→ Each of the next n lines contains n space separated integers.

n=3

0	10	5
2	7	9
3	2	0

```
int[][] adjMatrix = new int[n][n];
for( int i=0 ; i<n ; i++) {
    for( int j=0 ; j<n ; j++)
        adjMatrix[i][j] = sc.nextInt();
}
```

Case ③ A weighted edge list is given input with multiple edges. i.e, A single pair of nodes can have multiple edges between them and we have to keep only minimum edge weight.

Sample Input:

→ The first line of each test case contains two space-separated integers N (total vertices) & M (total edges).

→ Each of the next M lines contains 3 space-separated integers u, v and w denoting vertices (u & v) are connected by an edge having weight w.

N=3, M=5

0 3 10

1 5 2

3 7 -2

0 3 2

2 4 6

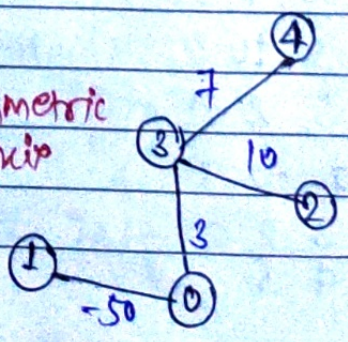
```
public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt(); // nodes
    int m = sc.nextInt(); // edges
    int[][] adjMatrix = new int[n][n];
    // initialize matrix with infinity
    for(int[] matrix : adjMatrix){
        Arrays.fill(matrix, Integer.MAX_VALUE);
    }

    for(int i=0; i<m; i++){
        int from = sc.nextInt();
        int to = sc.nextInt();
        int cost = sc.nextInt();

        adjMatrix[from][to] = Math.min(
            adjMatrix[from][to], cost);
        // if undirected : add following
        adjMatrix[to][from] = adjMatrix[from][to];
    }
}
```

⇒ Some important point regarding Adjacency matrix :-

(1.) for undirected graph, we get symmetric adjacency matrix while for directed graph, we get asymmetric (not symmetric) adjacency matrix.

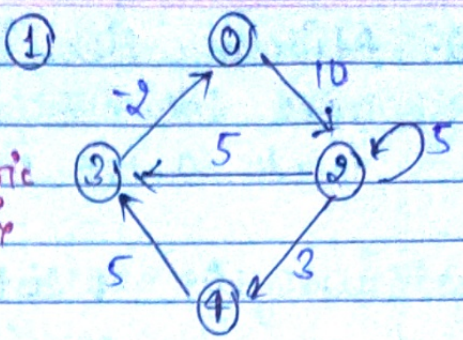
Nodes	0	1	2	3	4
0	∞	-50	∞	3	∞
1	-50	∞	∞	∞	∞
2	∞	∞	∞	10	∞
3	3	∞	10	∞	7
4	∞	∞	∞	7	∞

Symmetric Matrix



nodes	0	1	2	3	4
0	∞	∞	10	∞	∞
1	∞	∞	∞	∞	∞
2	∞	∞	5	5	3
3	-2	∞	∞	∞	∞
4	∞	∞	∞	5	∞

Not Symmetric matrix



(2.) If number of nodes > 10,000, then this representation is not suitable because $(10^5 \times 10^5) = (10)^{10}$ will exceeds the Integer range.

(3.) Adding and removing edges in this representation can be done in constant time.

(4.) When given graph is dense then we should go for adjacency matrix representation.

(5.) Checking whether an edge is a part of a graph, then we can find it in O(1) time from adjacency matrix representation.

(2.) Adjacency lists :- Adjacency lists are much more intuitive to implement and are used a lot more than adjacency matrices. Here, we use list of lists.

Case ① An adjacency list is given input as it is. ie, line number denotes vertices, and number on those lines denote adjacent nodes.

Sample Input :

→ The first line contains a single integer n - The no. of nodes.

→ Each of the next contains first number J and followed by J space separated integers. The ith row denote ith node jth integer in the ith row denotes an adjacent to that node.

```
n = 3
2 1 2
1 2
2 0 1
```

```
ArrayList<ArrayList<Integer>> adjList = new ArrayList<>();
int n = sc.nextInt();
for(int i=0; i<n; i++){
    adjList.add(new ArrayList<>());
    int countOfNodes = sc.nextInt();
    for(int j=0; j<countOfNodes; j++){
        int nodeValue = sc.nextInt();
        adjList.get(i).add(nodeValue);
    }
}
```

Case (b) An adjacency matrix is given input as it is. ie, 1 for adjacent and 0 for non-adjacent.

Sample Input :-

→ The first line contains a single integer n- The number of nodes.

→ Each of the next n lines contains (n*2) space separated integers. They are pairs of adjavents & weight.

n = 3

2 (1, 13) (2, 4)

1 (2, 9) (3, 4)

2 (0, -7) (1, 8)

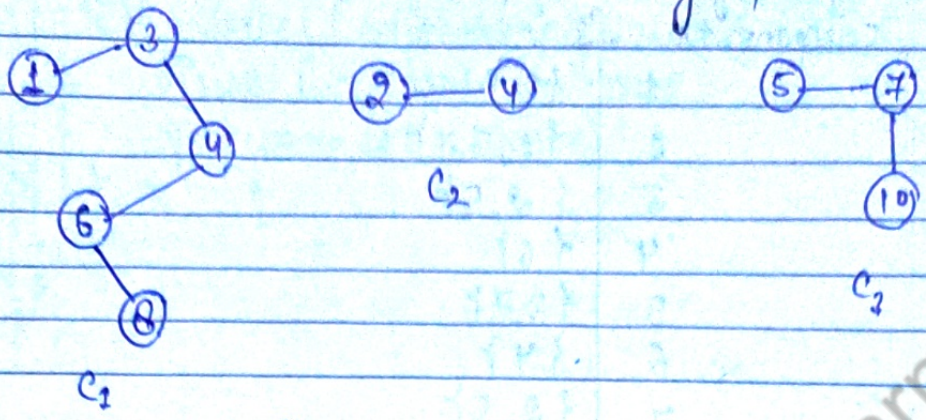
Code :-

```
public class AdjacencyList_withWeights {
    static class Edge {
        int destNode, edgeWeight;
        public Edge(int destNode, int edgeWeight) {
            this.destNode = destNode;
            this.edgeWeight = edgeWeight;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ArrayList<ArrayList<Edge>> adjList = new
        ArrayList<>();
        sout(" Enter No. of Nodes: ");
        int n = sc.nextInt();
```

```
for(int i=0; i<n; i++){
    adjList.add(new ArrayList<>());
    Sout("Enter count:");
    int count = sc.nextInt();
    for(int j=0; j<count; j++){
        Sout("Enter destination node:");
        int destNode = sc.nextInt();
        Sout("Enter edge weight:");
        int edgeWeight = sc.nextInt();
        adjList.get(i).add(new Edge(destNode,
            edgeWeight));
    }
}
printAdjacency(adjList);
}

public static void printAdjacency(ArrayList
<ArrayList<Edge>> adjList){

    for(int i=0; i<adjList.size(); i++){
        Sout(i+"->");
        for(Edge k: adjList.get(i)){
            Sout(k.destinode + " " + k.edgeWeight);
        }
        Sout();
    }
}
```

Representation	Space	Checking edge b/w v & u	Iterate over edges incident to v.
List of edges	E	O(E)	O(E)
Adj. Matrix	V^2	O(1)	O(V)
Adj. List	E+V	O(V)	O(V)
Adj. Set	E+V	O(log(O(V)))	O(V)

Connected components of a graph :-

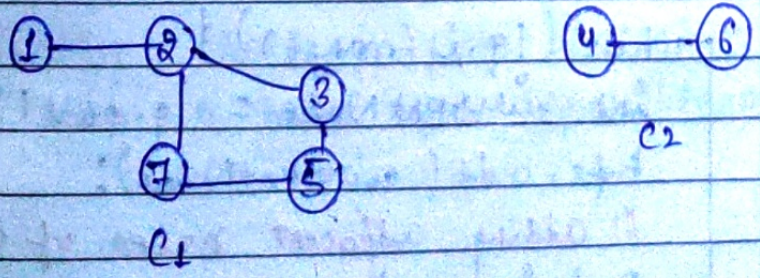


These are 3 disconnected components of a graph.

// If graph has multiple components then we have to write this loop

```
for( int i=0; i<NoOfNodes ; i++){
    if( !visited[i] ){
        bfs() / dfs()
    }
}
```

BFS (Breadth-first search) :- The BFS algorithm works similar to level order traversal of the trees. It also uses the Queue data structure same as in trees.



Perform BFS traversal on these 2 disconnected components of a graph.

Adjacency list representation of given components is :-

1	{2}
2	{1,3,7}
3	{2,5}
4	{6}
5	{3,7}
6	{4}
7	{2,5}

Code :-

```
public static ArrayList<Integer> bfs_Graph
(int numberOfNodes, ArrayList<ArrayList<
Integer>> adjList){
// for storing resultant bfs traversal
ArrayList<Integer> bfs = new ArrayList<>();
boolean[] visited = new boolean[numberOf
                    Nodes+1];

// Running loop for all nodes
for(int i=1; i<numberOfNodes; i++){
// checking if node is already visited or not
if(!visited[i]){
Queue<Integer> q = new LinkedList<>();
q.add(i);
visited[i] = true;
while(!q.isEmpty()){
int currentNode = q.poll();
bfs.add(currentNode);
// adding adjacent nodes of currentNode
for(int adjNode : adjList.get(currentNode)){
```

```
// checking adjacent node is not already visited
if( !visited[adjNode]){
    visited[adjNode] = true;
    queue.add(adjNode);
}
}
}
}
}
return bfs;
```

$TC = O(N+E)$ (N is time taken to visit N nodes, and E is for traveling through adjacent nodes overall.)

$SC = O(N+E) + O(N) + O(E)$

- $O(N+E)$ — for final data structure
- $O(N)$ — for Boolean visited array
- $O(E)$ — for queue

Applications of BFS traversal :-

1.) Finding all connected components in a graph.
2.) Finding all nodes within one connected component.
3.) Finding shortest path b/w two nodes.
4) Testing a graph for bipartiteness.

dfs (Depth first Search) :-

```
public static ArrayList<Integer> dfsGraph(
  int n, ArrayList<ArrayList<Integer>> adjList){
   ArrayList<Integer> dfs = new ArrayList<>();
   boolean[] vis = new boolean[n+1];
   // Run loop for all nodes
   for(int i=1; i<=n; i++){
      if(!vis[i]){
        dfsTravergal(i, adjList, vis, dfs);
      }
   }
   return dfs;
}
public static void dfsTraversal(int node,
  ArrayList<ArrayList<Integer>> adjList, boolean[]
  vis, ArrayList<Integer> dfs){
    dfs.add(node);
    visited[node] = true;
    // check for adjacent nodes to current node
    for(int adjNode : adjList.get(node)){
       if[!vis[adjNode]){
          dfsTraversal(adjNode, adjlist, vis, dfs);
       }
    }
}
```

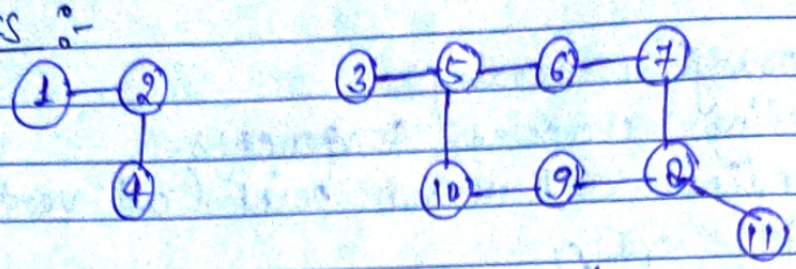
TC = O(N+E)

SC = O(N+E) + O(N) + O(E)

Applications of Dfs :

1.) Topological sorting
2.) finding connected components.
3.) finding articulation point (cut vertices) of the graph.
4.) finding strongly connected components.
5.) solving puzzles such as Maze.

Cycle detection in undirected graph using BFS :-




```
for(i = 1 to N){
    if(! vis[i]){
        if( cycleBFS(i)){
            return true;
        }
    }
}
```

1 {2}
2 {1,4}
3 {5}
4 {2}
5 {3,10,6}
6 {5,7}
7 {6,8}
8 {7,9,11}
9 {10,8}
10 {5,9}
11 {8}

- In Queue data structure, store the node as pair of (node, previousNode).

	0	1	2	3	4	5	6	7	8	9	10	11
vis	f	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	~~f~~ T	f

cycleBFS(1) (✓)
cycleBFS(2) (visited)
cycleBFS(3) (✓)

At this point, we will get a cycle in a graph. So return true

Queue (node, prevNode)	
~~(0,9)~~	n=8, pn=9
~~(7,6)~~	n=7, pn=6
~~(9,10)~~	n=9, pn=10
~~(6,5)~~	n=6, pn=6
~~(10,5)~~	n=10, pn=5
~~(5,3)~~	n=5, pn=3
~~(3,-1)~~	n=3, pn=(-1)
~~(4,2)~~	n=4, pn=2
~~(2,1)~~	node=2, prevNode=1
~~(1,-1)~~	node=1, prevNode=(-1)

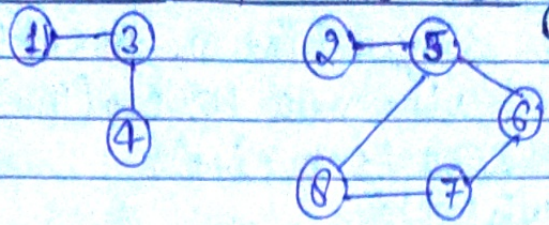
Code :-

```
class Pair {
    int node, parentNode;
    public Pair(int node, int parentNode) {
        this.node = node;
        this.parentNode = parentNode;
    }
}

class Solution {
    public boolean isCycle(int n, ArrayList<ArrayList<Integer>>
    adjList) {
        boolean[] vis = new boolean[n+1];
        for(int i=1; i<=n; i++) {
            if(!vis[i]) {
                if(cycleCheck(i, adjList, vis)) {
                    return true;
                }
            }
        }
        return false;
    }

    public boolean cycleCheck(int node, ArrayList<ArrayList<Integer>
    adjList, boolean[] vis) {
        Queue<Pair> q = new LinkedList<>();
        q.offer(new Pair(node, -1));
        visited[node] = true;

        while(!q.isEmpty()) {
            int currNode = q.peek().node;
            int currNodesParent = q.peek().parentNode;
            q.poll();
```

```
// checking for adjacent nodes of current node
for(int adjacentNode : adjList.get(currNode)){
    if(!vis[adjacentNode]){
        q.offer(new Pair(adjacentNode,
                currNode));
        vis[adjacentNode] = true;
    }
    // if the adjacent node is already visited,
    but it is not equal to the parent of the
    current node so how it is visited before
    that means, it's a cycle.
    else if(currNode!=parentNode && adjacentNode)
        return true;
    }
}
return false;
}
}
```

$$\begin{cases} TC = O(N+E) \\ SC = O(N+E) + O(N) + O(E) \end{cases}$$

Cycle detection in undirected graph using Dfs :-



1 - {3}
2 - {5}
3 - {1,4}
4 - {3}
5 - {2,6,8}
6 - {5,7}
7 - {6,8}
8 - {7,5}

```
for(int i -> 1 to n){
    if( !vis[i] ){
        if( cycleDFS(i)){
            return true;
        }
    }
}
```

vis

0	1	2	3	4	5	6	7	8
F	F	F	F	F	F	F	F	F

T T Node T parentNode

DFS(1, -1)

DFS(3, 1)

DFS(4, 3)

if (parentNode != adjacentNode){ then call Recursive fun@ }

DFS(2, -1)

DFS(5, 2)

DFS(6, 5)

DFS(7, 6)

DFS(8, 7)

DFS(5, 8) → 5 is already visited, that means there is a cycle.

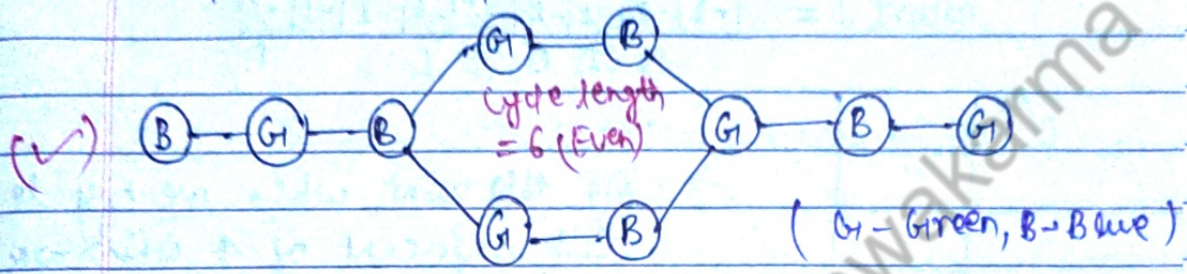
TC = O(N+E)
SC = O(N+E) + O(N) + O(N)

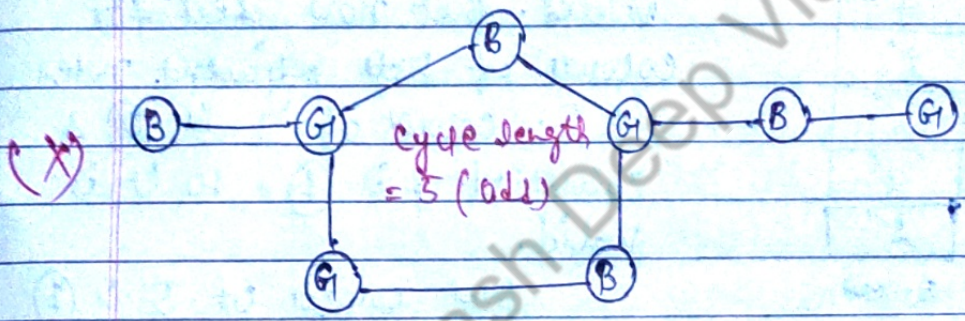
Code :

```
boolean isCycle(int n, AL<AL<Integer>> adj){
    boolean[] vis = new boolean[n+1];
    for(int i=1; i<=n; i++){
        if(!vis[i]){
            if(checkForCycle(i, -1, vis, adj)){
                return true;
            }
        }
    }
    return false;
}

boolean checkForCycle(int node, int parent,
boolean[] vis, AL<AL<Integer>> adj){
    vis[node] = true;
    for(int adjNode: adj.get(node)){
        if(!vis[adjNode]){
            if(checkForCycle(adjNode, node, vis,
                adj)){
                return true;
            }
        }
        else if(adjNode != parent){
            return true;
        }
    }
    return false;
}
```

Bipartite Graph (BFS) : Graph colouring :-

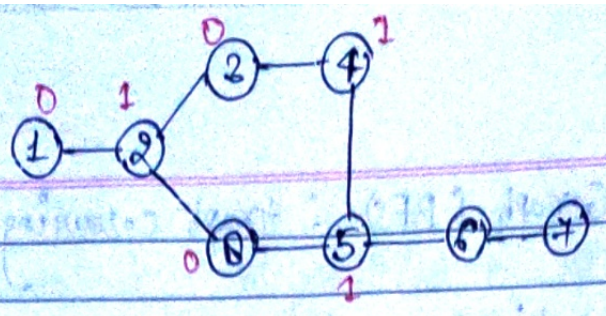
"A graph which can be coloured with 2 colours such that no two adjacent nodes have same colour, is known as Bipartite Graph."





So, if (Graph has odd length cycle) {
    Not Bipartite Graph
} else {
    Bipartite Graph
}

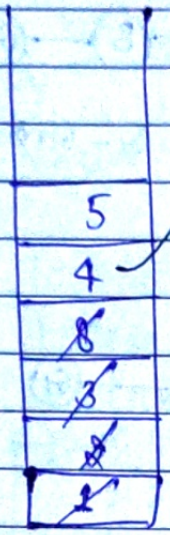
"A bipartite graph is a graph whose vertices can be divided into independent sets, U & V such that every edge (u,v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u,v), either u belongs to U & V belongs to V or vice-versa. We can say that there is no edge that connects vertices of same set."



→ We will colour this graph using 2 colours 0 & 1.

color[] =

0	1	2	3	4	5	6	7	8
~~-1~~ 0	~~-1~~ 1	~~-1~~ 0	~~-1~~ 1	~~-1~~ 1	~~-1~~	-1	-1	~~-1~~ 0



At this point, when we try to visit adjacent of 4 which are {3,5}. Both are already visited. But now check for colours of both adjacent nodes 3 & 5. Colour of 3 is (0) & colour of 4 is (1) so it is valid.

But colour of 5 is (1) & colour of 4 is also (1). So, it is invalid. So, simply return false.

⇒ We will choose 1 as the color of first node and use the formula for filling colors for adjacent nodes.

adjacent Node's color = (1 − node's color)

{If node's color = 0 then adjNode's color = 1
" = 1 then " = 0}

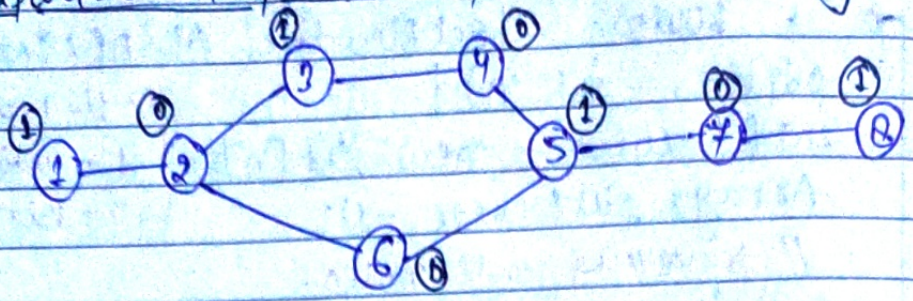
{TC = O(N+E)
SC = O(N+E) + O(N) + O(N)}

Code :-

```
public boolean checkBipartite( ArrayList<ArrayList<Integer>>
adjList, int n) {
int[] color = new int[n];
Arrays.fill(color, -1);
// traversing all nodes.
for(int i=1; i<=n; i++){
  if(color[i] == -1){
    if(!isBipartite(i, color, adjList)){
      return false;
    }
  }
}
return true;
}

public boolean isBipartite(int node, int[] color,
ArrayList<ArrayList<Integer>> adjList) {
Queue<Integer> q = new LinkedList<>();
q.offer(node);
color[node] = 1;
while(!q.isEmpty()){
  int currNode = q.poll();
  int currNodesColor = color[currNode];
  for(int adjNode : adjList.get(currNode)){
    if(color[adjNode] == -1){
      color[adjNode] = (1 - currNodesColor);
      q.offer(adjNode);
    }
    else if(color[adjNode] == currNodesColor){
      return false;
    }
  }
}
return true;
}
```

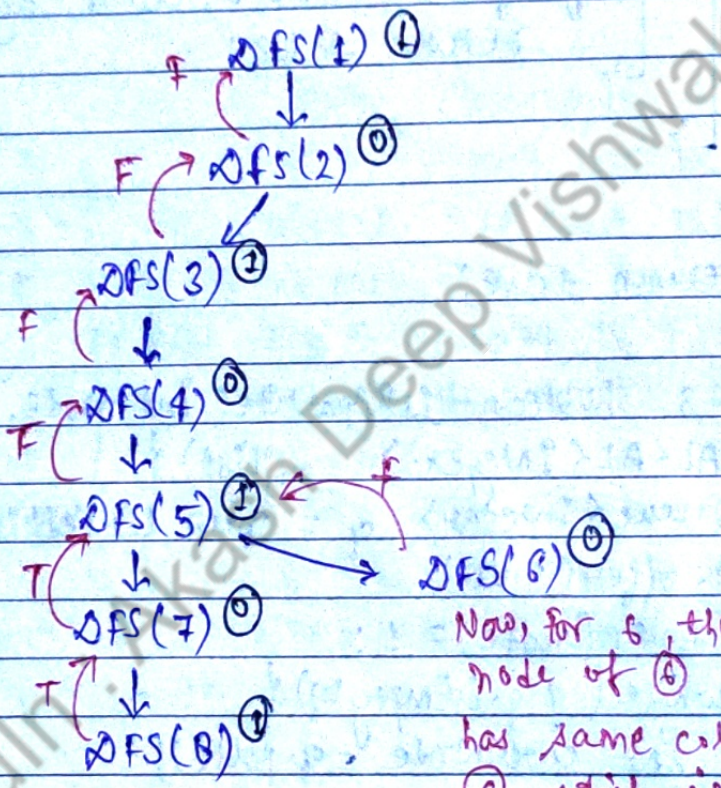
-1 - Not visited
1 - White
0 - Black

Bipartite Graph (DFS) | Graph colouring :



color[] =

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	-1
	1	0	1	0	1	0	0	1



Now, for 6, the adjacent node of 6 which is 2 has same colour as of 6 which is 0 colour. So, it is not a Bipartite graph.

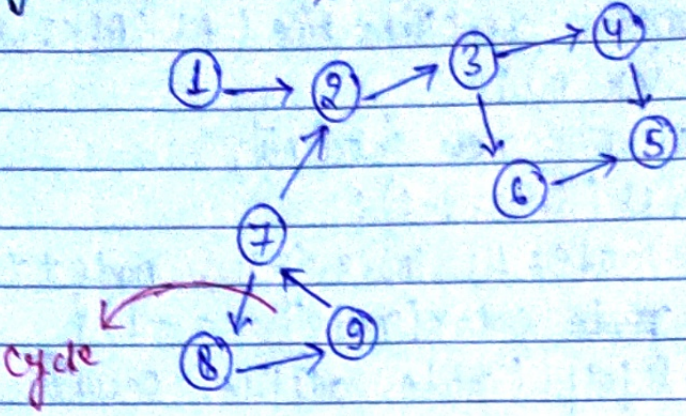
{ TC = O(N+E)
SC = O(N+E) + O(N) + O(E) }

Code :-

```
boolean checkBipartite(ArrayList<ArrayList<Integer>> adjList, int n){
    int[] color = new int[n+1];
    Arrays.fill(color, -1);
    for(int node=1; node<=n; node++){
        if(color[node] == -1){
            if(!dfs(node, adjList, color)){
                return false;
            }
        }
    }
    return true;
}

boolean dfs(int node, ArrayList<ArrayList<Integer>> adjList, int[] color){
    if(color[node] == -1){ // start filling the 1st node's color as 1
        color[node] = 1;
    } // Now, check for adjacent nodes
    for(int adjNode : adjList.get(node)){
        if(color[adjNode] == -1){ // fill opposite color of node
            color[adjNode] = 1 - color[node];
            if(!dfs(adjNode, adjList, color)){
                return false;
            }
        } // If color of adj is same as color of given node
        else if(color[adjNode] == color[node]){ then return false.
            return false;
        }
    }
    return true;
}
```

(Recursion + Backtracking)

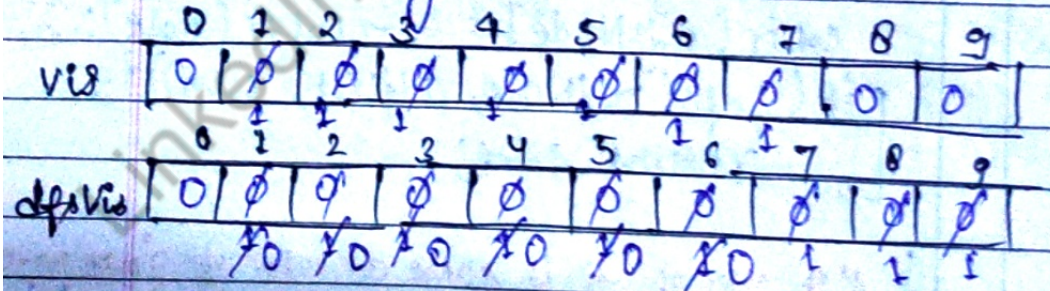
Cycle detection in Directed Graph using DFS :-



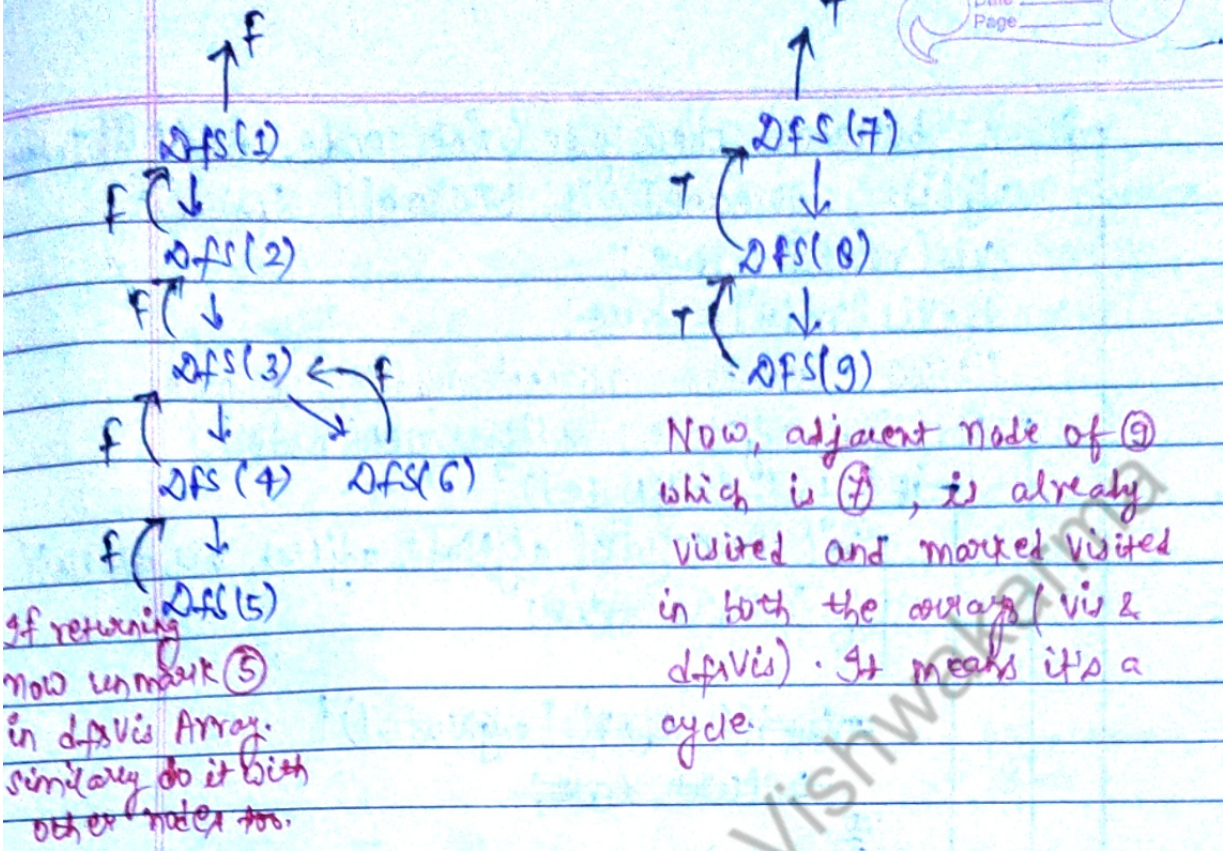
adjacency list
1 → 2
2 → 3
3 → 4, 6
4 → 5
5 → X
6 → 5
7 → 2, 8
8 → 9
9 → 7

→ Here, normal DFS traversal won't work. So we have to modify that. Here, we will use self DFS along with visited DFS.

→ Here, we will use 2 visited arrays (vis and dfsVis). In dfsVis array, we will perform backtracking by marking unmarked the visited node if it is not a cycle.




```
for(int i=1 to 9){
    if(!vis[i]){
        if(cycleCheck(i)){
            return true;
        }
    }
}
```



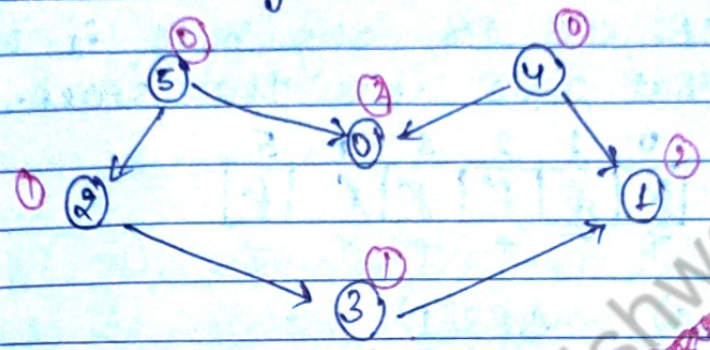
{ TC = O(N+E)
SC = O(2N)
Auxiliary SC = O(N) }

Code :-

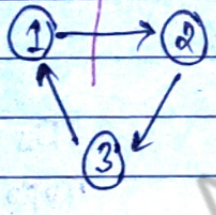
```
public boolean isCyclic(ArrayList<ArrayList<Integer>>
    adjList, int n) {
boolean[] vis = new int[n+1];
boolean[] dfsVis = new int[n+1]; // In this
    array we will perform backtracking

// Now run loop for all nodes
for(int node=1; node<=n; node++) {
    if(!vis[node]) {
        if(checkcycle(node, adjList, vis, dfsVis)) {
            return true;
        }
    }
}
return false;
```

```
boolean checkCycle(int node, ArrayList<ArrayList<Integer>> adjList, boolean[] vis, boolean[] dfsVis){
    vis[node] = true;
    dfsVis[node] = true;
    //Now check for adjacent nodes of node
    for(int adjNode : adjList.get(node)){
        if(!vis[adjNode]){
            if(checkCycle(adjNode, adjList, vis, dfsVis)){
                return true;
            }
            else if(dfsVis[adjNode]){
                return true;
            }
        }
        // If adjacent node is already visited in dfsVis, that means it is visited in the same dfs call so, it forms a cycle.
        else if(dfsVis[adjNode]){
            return true;
        }
    }
    dfsVis[node] = false; // Backtracking step
    return false;
}
```

Topological sort (DFS) :- Topological sorting for DAG (Directed Acyclic Graph) is a linear ordering of vertices such that for every directed edge (u→v), vertex u comes before v in the ordering.
(DAG → No cycle)



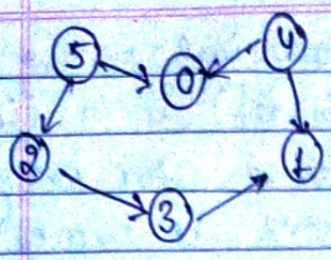
Topological sorting : ① 5 4 2 3 1 0
② 4 5 2 3 1 0
→ first vertex should be one whose indegree is 0.



{If there is a cycle, then no topological sorting can be possible.}

- Here, we will use modified dfs for the topological sorting.
- We will require one visited array and one stack data structure.

```
for(int i=1 to n){
    if(!vis[i]){
        dfs(i);
    }
}
```

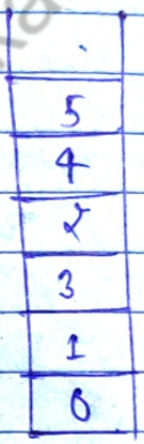


adjacency list :-
0 - { }
1 - { }
2 - {3}
3 - {1}
4 - {0, 1}
5 - {0, 2}

→ When DFS call for any node is over then put that node into the stack.

	0	1	2	3	4	5
vis	F	F	F	F	F	F
	T	T	T	T	T	T

DFS(0) DFS(1) DFS(2) DFS(4) DFS(5) DFS(3)



Stack

↑ This contains Topological sorting

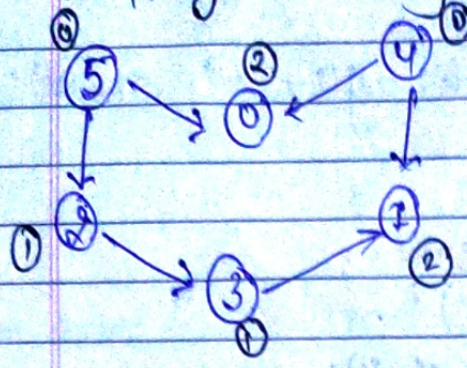
$$\begin{cases} TC = O(N+E) \\ SC = O(N) \\ ASC = O(N) \end{cases}$$

Code :-

```
P & V helperMethod ( AL<AL<Integer>>
    adjList , int n) {
  boolean [] vis = new boolean [n+1];
  Stack<Integer> stack = new stack<>();
  //Run loop for all nodes
```

```
for(int node=1 ; node<=n ; node++){
    if(!vis[node]){
        topologicalSort(node, adjList, vis, stack);
    }
}
// printing topological sorting
while(!stack.isEmpty()){
    sout(stack.pop() + " ");
}
}

p s void topologicalSort(int node, AL<AL<Integer
>> adjList, boolean[] vis, Stack<Integer> stack){
    vis[node] = true;
    // check for adjacent nodes
    for(int adjNode : adjList.get(node)){
        if(!vis[adjNode]){
            topologicalSort(adjNode, adjList, vis,
                            stack);
        }
    }
    stack.add(node);
}
```

Topological sorting (BFS) : Kahn's Algorithm



adjacency list :-

0 - { }
1 - { }
2 - {3}
3 - {1}
4 - {0,1}
5 - {0,2}

⊙ Data structures used :-

1.) An array for storing the indegree of all nodes.
2.) Queue data structure for storing final Topological sorting.

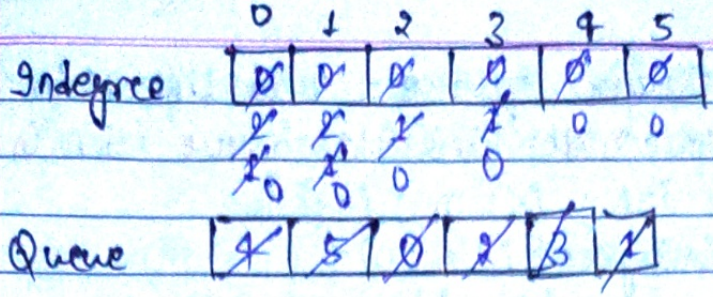
⊙ Steps :-

1.) Push the nodes into the queue whose indegree is 0.
2.) Now, pop one element from queue & decrease the indegree of adjacent nodes of popped node. Print the popped node.

If the indegree of any adjacent node becomes 0, then add that node into the queue.

3.) Repeat the step (2) until all nodes are visited.

{ TC = O(N+E)
SC = O(N) + O(N) }



node = (4) {0, 1} Reduce indegrees of 0 & 1 by 1.
node = (5) {0, 2} Reduce indegree of 0 & 2 by 1.
Now, indegree of 0 & 2 is 0
So, push them into the queue.

node = (0) { }
node = (2) {3} Reduce indegree of (3) by 1.
Now, indegree of (3) is 0 so
add it to the queue.
node = (3) {1} Reduce indegree of (1) by 1. Now
add (1) into the queue.
node = (1) { }

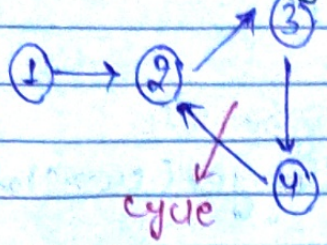
∴ Topological sorting is : {4, 5, 0, 2, 3, 1}

Code :-

```
public void topologicalSort(ArrayList<ArrayList<Integer>> adjList, int n) {
    int[] inDegree = new int[n+1]; // for storing indegree of nodes
    // calculating indegree of nodes
    for(ArrayList<Integer> adjNodesList : adjList) {
        for(int adjNode : adjNodesList) {
            inDegree[adjNode]++;
        }
    }
```

```
Queue<Integer> queue = new LinkedList<>();
// put the nodes into the queue who in degree is 0
for(int i=1; i<=n; i++){
    if(inDegree[i]==0){
        queue.offer(i);
    }
}

ArrayList<Integer> finalList = new ArrayList<>();
while(!queue.isEmpty()){
    int currNode = queue.poll();
    for(int adjNode : adjList.get(currNode)){
        inDegree[adjNode]--; // degree decreases by 1
        if(inDegree[adjNode]==0){ // if indegree is 0 then add it to the queue.
            queue.offer(adjNode);
        }
    }
    finalList.add(currNode);
}
sout(finalList);
```

Cycle detection in DAG (Directed Acyclic Graph) using BFS (Kahn's Algorithm) :-



1-{2}
2-{3}
3→{4}
4→{2}

We know that, topological sort is only possible in DAG (No cycle). If a graph has cycle, so then we will not able to generate the topological sorting of that graph. So, if we are not able to generate topological sort, then it has cycle.

	1	2	3	4
indegree	0	2	1	1

queue {1,}

HINT : If count of nodes present in queue after the BFS Algo is equal to N (No. of Nodes) then there is no cycle, else it contains cycle.

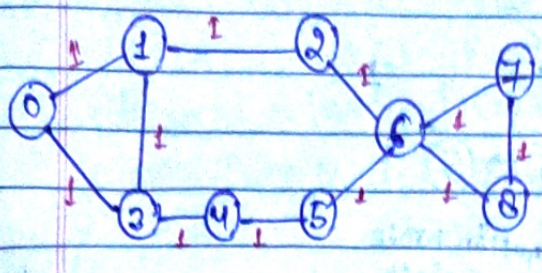
Code :-

```
public boolean isCyclic(AL<AL<Integer>> adjList, int n){
    int[] indegree = new int[n+1];
    for(AL<Integer> row : adjList){
        for(int node : row){
            indegree[node]++;
        }
    }
    Queue<Integer> q = new LinkedList<>();
    for(int node=1; node<=n; node++){
        if(indegree[node]==0){
            q.offer(node);
        }
    }
```

```
int visitedNodes = 0;
while( !q.isEmpty()) {
    int currNode = q.poll();
    visitedNodes++;
    // Adjacent Nodes of currNode
    for( int adjNode : adjList.get(currNode)) {
        indegree[adjNode]--;
        if( indegree[adjNode] == 0) {
            q.offer(adjNode);
        }
    }
}
// If visited Nodes == Total No. of nodes ( So, there is No cycle).
if( visitedNodes == n) {
    return false;
}
return true;
}
```

{TC = O(N+E)
SC = O(N) + O(N)}

Shortest path in undirected graph with unit weights :- (Modified BFS)



0 → {1,3}
1 → {0,2,3}
2 → {1,6}
3 → {0,4}
4 → {3,5}
5 → {4,6}
6 → {2,5,7,8}
7 → {6,8}
8 → {6,7}

→ We have given a source node, now we have to find the shortest path from source node to all other nodes.

	0	1	2	3	4	5	6	7	8
distance	∞	∞	∞	∞	∞	∞	∞	∞	∞
	0	1	2	1	2	3	3	4	4

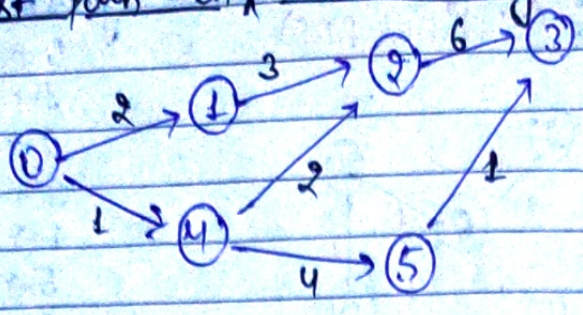
← Initialize the array with ∞.

queue: 0 1 3 2 4 6 5 7 8

Code :-

```
p s v bfs( AL<AL<Integer>> adjList, int source,
  int[] distance) {
Queue<Integer> q = new LinkedList<>();
q.offer(source);
distance[source] = 0;
~~for( int adjNode :~~
while( !q.isEmpty()) {
  int currNode = q.poll();
  for( int adjNode : adjList.get(currNode)) {
    if( (distance[currNode]+1) < distance[adjNode])
    {  distance[adjNode] = (distance[currNode]+1);
       q.offer(adjNode);
    }
  }
}
```

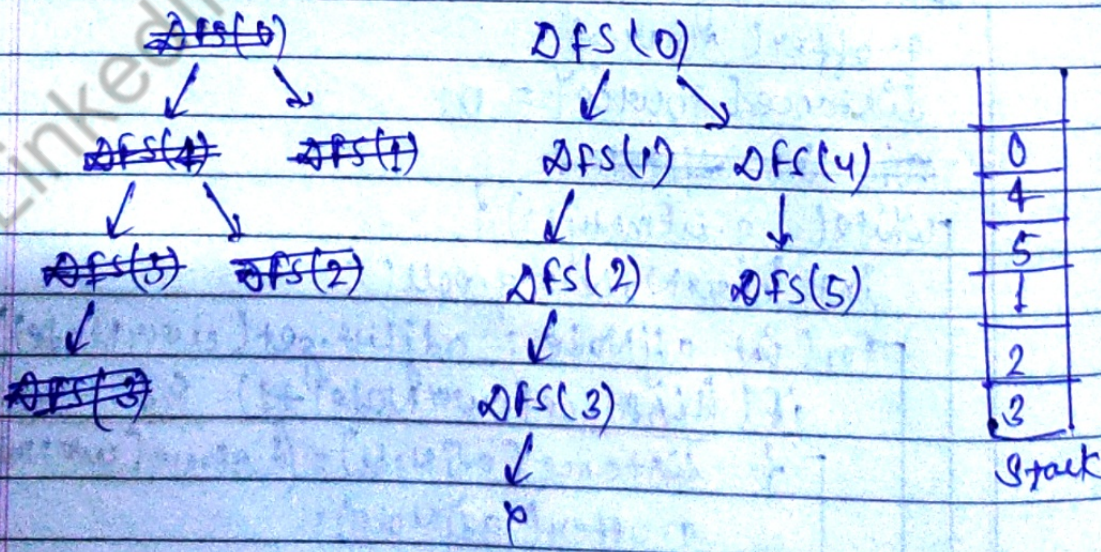
Shortest path in weighted Directed Acyclic Graph (DAG) :-



adjacency list : destination Node, edgeweight

0 → (1,2), (4,1)
1 → (2,3)
2 → (3,6)
3 → X
4 → (2,2), (5,4)
5 → (3,1)

* Step (1) : find the topological sorting of the given graph and store it in stack. Now, perform traversing on nodes through this stack. (Refer Kahn's algo for topological sorting through BFS). or DFS.



* Step (2) :- Create a distance array & fill it with ∞. Now, we have given a source node so fill (distance[source] = 0).

	0	1	2	3	4	5
distance	∞	∞	∞	∞	∞	∞

Now, put 0 in distance[source].

	0	1	2	3	4	5
distance	0	~~∞~~ 2	~~∞~~ 3	~~∞~~ 6	~~∞~~ 1	~~∞~~ 5

⇒ if(dis[node] != ∞) then traverse for adjacent nodes & perform action.

{ TC = O(N+E) × 2
SC = O(2N)
ASC = }

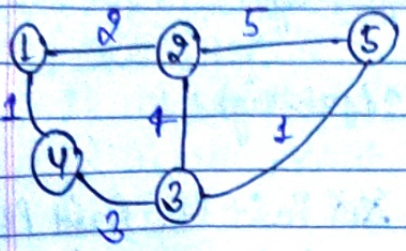
Code :-

```
p s void shortestPath( int source, AL<AL<graphNode>> adjList , int n) {
    Stack<Integer> topoStack = new Stack<>();
    // for storing topological sort of the graph
    int[] distance = new int[n+1];
    Arrays.fill( distance, Integer.MAX_VALUE);
    distance[source] = 0;
    boolean[] vis = new boolean[n+1];
    // Step 1: Topological sort method calling
    for(int node = 1 ; node <= n ; node++) {
        if ( !vis[node]) {
            topoSort( node, topoStack, adjList, vis);
        }
    }
```

```
while( ! topoStack.isEmpty()){
  int currNode = topoStack.pop();
  if(distance[currNode] != Integer.MAX_VALUE){
    for(graphNode adjNode : adjList.get(currNode)){
      if((distance[currNode] + adjNode.getEdgeWeight()
        < distance[adjacentNode.getNode()]){
        distance[adjNode.getNode()] = (distance[
          currNode] + adjNode.getEdgeWeight());
      }
    }
  }
  // print distance array
  for(int i=1; i<=n; i++){
    Sout(distance[i] + " ");
  }
}

class graphNode{
  private int node;
  private int edgeWeight;
  public graphNode(int node, int weight){
    this.node = node;
    this.edgeWeight = weight;
  }
}
```

→ Greedy Algorithm

Dijkstra's Algorithm (Shortest path in undirected graph) :- Given a graph and a source node, find the shortest paths from source to all nodes in the graph.



⇒ Adjacency list :-
1 → (2, 2), (4, 1)
2 → (1, 2), (5, 5), (3, 4)
3 → (2, 4), (4, 3), (5, 1)
4 → (1, 1), (3, 3)
5 → (2, 5), (3, 1)

⇒ Required Data Structure : Min Priority Queue (We will store (distance, Node) pair into the priority queue and will sort the priority queue according to the minimum distance). Now, we have to simply perform BFS.

	0	1	2	3	4	5
distance	∞	∞	∞	∞	∞	∞
	0	2	4	1	5	

{ Put 0 in the distance[source] }

{ TC = O((N+E)·LogN) }
{ SC = O(N) + O(N) }

In Priority Queue there is no remove method. So, we can also use Set data Structure.

Same node, but different distances, But Min PQ will only give priority to Minimum distance

Min PQ (distance, node)
(5, 5) ⑤
(7, 5) ⑤
(4, 3) ④
(1, 4) ②
(2, 2) ③
(0, 1) ①

Code:-

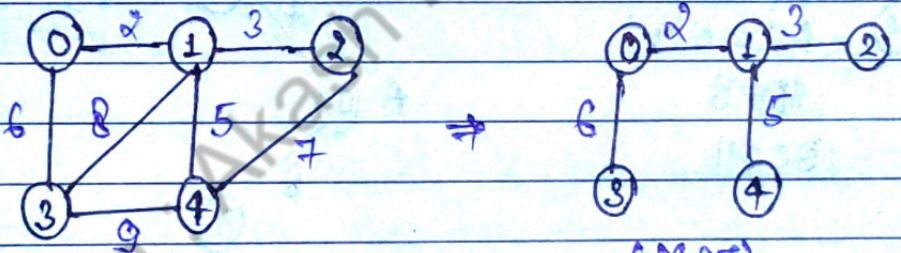
```
class Pair implements Comparator<Pair> {
    int node, edgeWeight;
    Pair() { }
    Pair(int node, int edgeWeight) {
        this.node = node;
        this.edgeWeight = edgeWeight;
    }
    public int compare(Pair P1, Pair P2) {
        return Integer.compare(P1.edgeWeight,
            P2.edgeWeight);
    }
}

p s v djkstraAlgo(AL<AL<Pair>> adjList,
int[] distance, int source, int n) {
// min priority queue declaration
PriorityQueue<Pair> minPQ = new PriorityQueue<>
(n, new Pair());
minPQ.add(new pair(source, 0));
while(!minPQ.isEmpty()) {
    Pair currentPair = minPQ.poll();
    int currentNode = currentPair.node;
    for(Pair adjacentPair : adjList.get(currentNode)
    if((distance[currentNode] + adjacentPair.edgeWeig)
        < distance[adjacentPair.node]) {
    distance[adjacentPair.node] = (distance[
    currentNode] + adjacentPair.edgeWeight);
    minPQ.add(new Pair(adjacPair.node,
    distance[adjacentPair.node]));
```

Minimum Spanning Tree (MST) :- "A MST for a weighted, connected, undirected graph is a spanning tree having a weight less than or equal to the weight of every other possible spanning tree."

⇒ Properties :-

1) It must not form a cycle.
2) There must be no other spanning tree with lesser weight.

If G(V,E) is a graph then every spanning tree of graph G consists of (V-1) edges, where V is the number of vertices in graph. and E is the number of edges in the graph. So, (E-V+1) edges are not a part of the spanning tree.

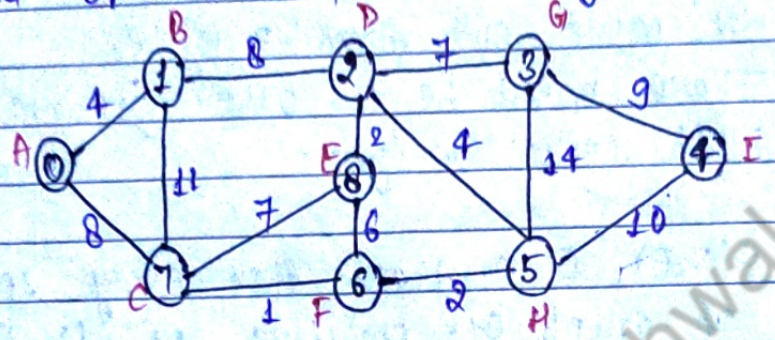


There are 2 algorithms for finding MST :-

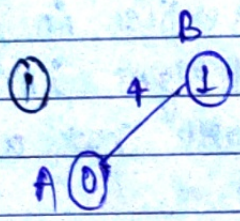
1) Prim's Algorithm
2) Kruskal's Algorithm

Prim's Algorithm for MST :- (Greedy Algorithm)

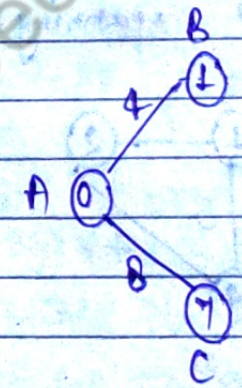
Prim's algorithm always starts with with a single node and it moves through several adjacent nodes, in order to explore all of the connected edges along the way.



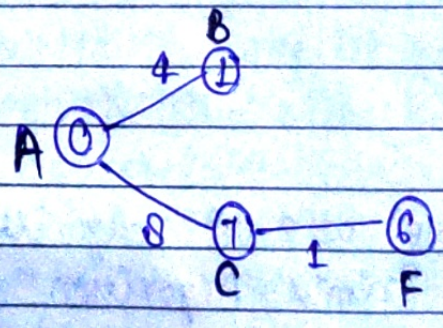
(1) AB = 4 (✓)
AC = 8



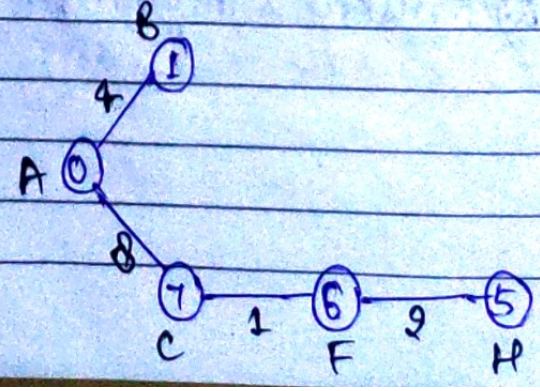
(2) AC = 8 (✓)
BD = 8
BC = 11



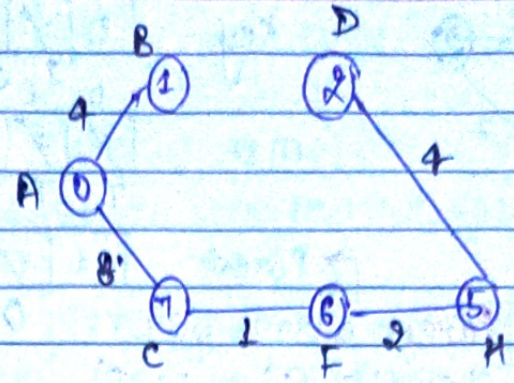
(3) BD = 8
BC = 11
CE = 7
CF = 1 (✓)



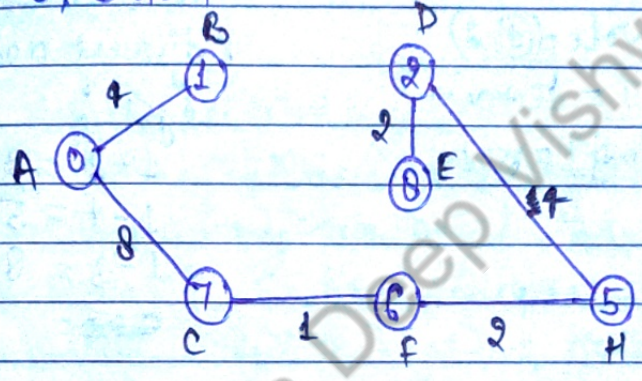
(4) BD = 8
BC = 11
CE = 7
FE = 6
FH = 2 (✓)



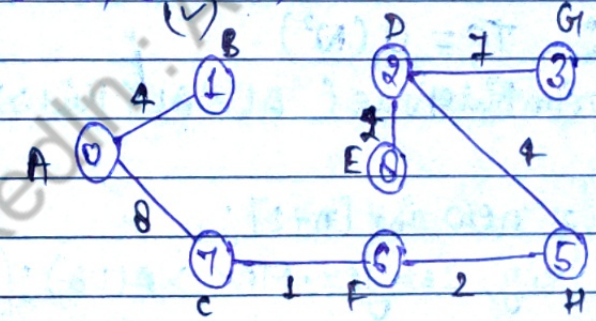
(5) BD = 8
BC = 11
CE = 7
FE = 6
HD = 4 (✓)
HG = 14
HI = 10



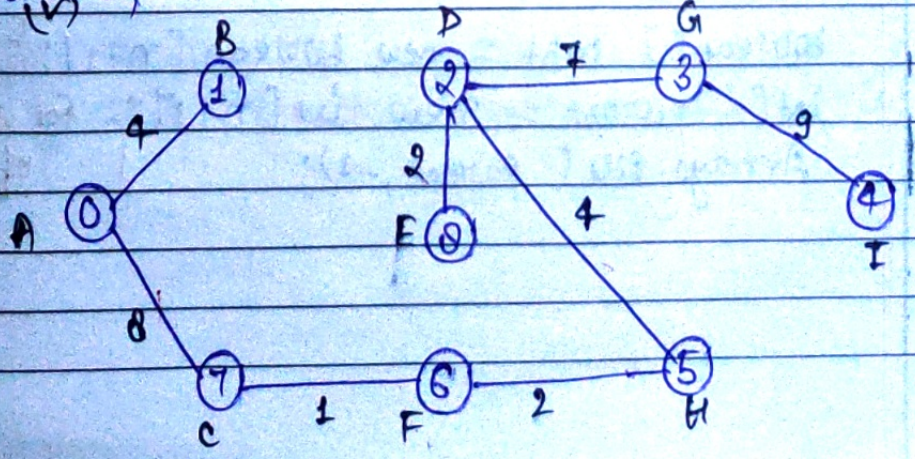
(6) BD = 8, BC = 11, CE = 7, FE = 6, HG = 14, HI = 10, DE = 2 (✓)
DB = 8, DG = 7

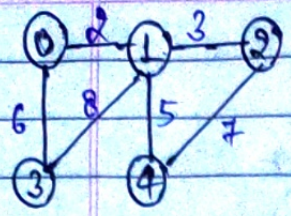


(7) BD = 8, BC = 11, CE = 7, FE = 6, HG = 14, HI = 10,
DB = 8, DG = 7, (✓)



(8) GI = 9, HI = 10 (✓)





key

0	1	2	3	4
0	∞	∞	∞	∞
	2	3	6	5
	1	2	3	4

// for storing min distance

mst

0	1	2	3	4
F	F	F	F	F
T_0	T_1	T_2	T_3	T_4

// for marking visited

Parent

0	1	2	3	4
-1	-1	-1	-1	-1
	0	1	0	1

// for storing the parent of the node.

(Parent array will be the final MST)

① node = 0
mst[node] = True
Adjacent Node = (1, 3)
Key[adj] = distance
Parent[adj] = node

② node = 1
mst[node] = true
Adjacent nodes = (2, 4, 3)
edges:
1-2 = 3
1-4 = 5
1-3 = 8

Code :-

① Brute force : TC = $O(N^2)$

```
P s V primBruteforce( AL<AL<Pair>> adjList,
  int n) {
  int[] Key = new int[n+1];
  Arrays.fill(Key, Integer.MAX_VALUE);
  Key[1] = 0;
```

for storing minimum distance

```
  boolean[] mst = new boolean[n+1];
  int[] parent = new int[n+1];
  Arrays.fill(parent, -1);
```

} visited Array

} for storing parent of the nodes

```
// Run loop till (n-1) because there are (n-1) edges
   in MST where n -> no. of nodes
for(int i=1; i<=(n-1); i++){
  int minDistance = Integer.MAX_VALUE;
  int node=0;
  for(int j=1; j<=n; j++) { // Searching for min Distance node
    if(!mst[j] && key[j]<minDistance){
      minDistance = key[j];
      node = j;
    }
  }
  mst[node]=true; // marking visited
  for(pair adjNode: adjList.get(node)) { // adjacent nodes
    if(!mst[adjNode.node] && (adjNode.edge-
      Weight < key[adjNode.node])) {
      parent[adjNode.node] = node;
      key[adjNode.node] = adjNode.edgeWeight;
    }
  }
}
// printing MST (parent Array)
Sout("\n\n MST of the given graph is :");
for(int node=1; node<=n; node++){
  if(parent[node] != (-1)){
    Sout("parent of node" +node+ " is :"
      +parent[node]);
  } else {
    Sout("parent of node" +node+ " is : 1");
  }
}
}
```

Optimize this block of code by using min priority Queue

② Optimised Approach :- TC = O(N * logN)

```
p s v primOptimised( AL<AL<Pair>> adjList,
  int n){
PriorityQueue<Pair> minPQ = new PriorityQueue<>
  (n, new Pair());
minPQ.add(new Pair(1, key[1]));

for(int i=1; i<=n ; i++){
  int node = minPQ.poll().node;
  mst[node] = true;
  for(pair adjNode : adjList.get(node)){
    if(!mst[adjNode] && adjNode.edgeWeight<
      key[adjNode,node]){
      parent[adjNode.node] = node;
      key[adjNode.node] = adjNode.edgeWeight;
      minPQ.add(new Pair(adjacentNode.node,
        key[adjNode.node]));
    }
  }
}
}
```

L-22 (takeforward) Graph

Disjoint set Data structure | Union by Rank and path compression :

"A data structure that stores non-overlapping or disjoint subset of elements is called disjoint set data structure."

It has following operations :-

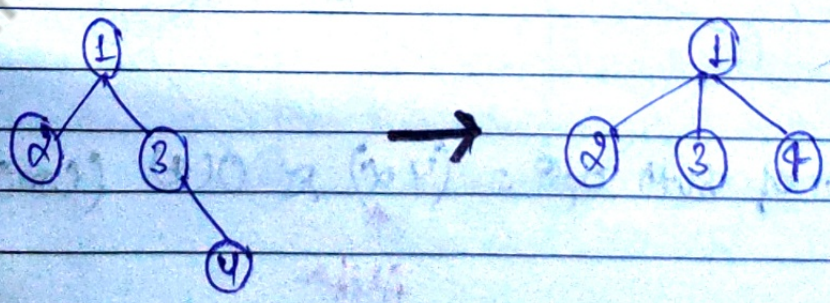
1.) Adding new sets to the disjoint set.
2.) Merging disjoint sets to a single set using Union operation.
3.) finding representative of a disjoint set using find operation.
4.) check if two sets are disjoint or not.

⇒ Operations :-

1.) Union () : for merging 2 elements
2.) find () : for checking if the elements belongs to same set or not.

⇒ Path compression (Modification to find()) :-

It speeds up the data structure by compressing the height of the tree.



Parent[4] = Parent[3] = 1
So, simply connect (4) directly to node (1)
It will compress the height & reduces time complexity.

```
for(int i=1; i<=n; i++){
    parent[i] = i;
    rank[i] = 0;
}
```
} Initialization of rank & parent Arrays

① find() operation :-

```
public int find(int node){
    if(node == parent[node]){ // If node is itself parent (sign node).
        return node;
    }
    return parent[node] = find(parent[node]);
}
```
↑ Path compression

② union() operation :-

```
public void union(int u, int v){
    u = find(u); // Finding parent of u
    v = find(v); //      "      "   v
    if(rank[u] < rank[v]){ // If rank of u is smaller then
        parent[u] = v;      make v as parent of u.
    }
    else if(rank[v] < rank[u]){
        parent[v] = u;
    }
    else{ // If rank of u & v are same, then
        parent[v] = u; simply make any one
        rank[u]++;     parent of other & increase
    }                  the rank by 1.
}
```

T.C. of both opⁿ = $(4\alpha) \approx O(4)$ (Constant)
↑ Alpha

SC = O(N)

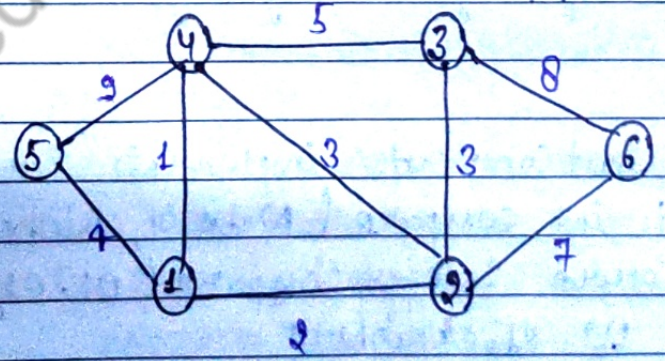
⇒ Application of Disjoint set :-

1) Kruskal's Algorithm for MST
2) Job sequencing problem
3) Cycle Detection
4) Number of pairs
5) city with the smallest number of Neighbours at a threshold distance.

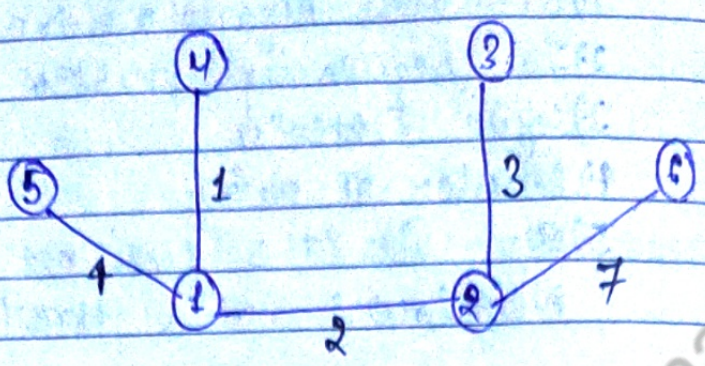
Kruskal's Algorithm :- It is used to find MST.

⇒ Steps :-

1) Sort all the edges in non-decreasing order of their weight.
2) Pick the smallest edge. check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3) Repeat step (2) until there are (V-1) edges in the spanning tree.



weight	u	v
1	1	4 (✓)
2	1	2 (✓)
3	2	3 (✓)
3	2	4 (✗)
4	1	5 (✓)
5	3	4 (✗)
7	2	6 (✓)
8	3	6 (✗)
9	4	5 (✗)



$$\begin{cases} TC = O(E \log E) + O(E \cdot O(4\alpha)) \approx O(E \log E) \\ SC = O(E) + O(EN) + O(N) \end{cases}$$

Code :-

```
class Node {
    int u, v, edgeWeight;
    public Node(int u, int v, int edgeWeight) {
        this.u = u;
        this.v = v;
        this.edgeWeight = edgeWeight;
    }
}

class SortComparator implements Comparator<Node>
    public int compare(Node o1, Node o2) {
        return Integer.compare(o1.edgeWeight,
            o2.edgeWeight);
    }
}

public class KruskalAlgo {
    // find() operation
```

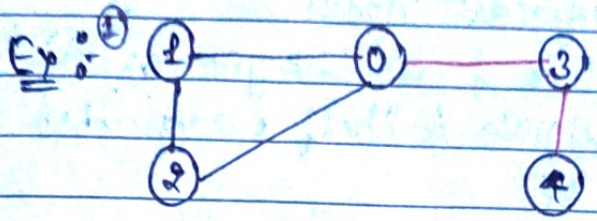
```
public static int find(int node, int[] parent){
    if(node == parent[node]){
        return node;
    }
    return parent[node] = find(parent[node], parent);
}
// union() operation
public static void union(int u, int v, int[]
    parent, int[] rank){
    u = find(u, parent);
    v = find(v, parent);
    if(rank[u] < rank[v]){
        parent[u] = v;
    }
    else if(rank[v] < rank[u]){
        parent[v] = u;
    }
    else{
        parent[v] = u;
        rank[u]++;
    }
}

public static void KruskalAlgorithm(ArrayList
    <Node> arraylist, int n){
    Collections.sort(arraylist, new sortComparator);
    // Sorting arraylist into acending order of
    their edge weights.
    int[] parent = new int[n];
    int[] rank = new int[n];
    // Initializing parent & rank array.
    for(int i=0; i<n; i++){
        parent[i] = i;
        rank[i] = 0;
```

```
int costMst=0; // for storing final weight/cost of MST
ArrayList<Node> Mst = new ArrayList<>();
// traversing arraylist
for(Node node : arraylist){
  // If both node's parents are not same so, it means they are in different components so union them and add them into mst data structure.
  if( find(node.u, parent) != find(node.v, parent)){
    costMst += node.edgeWeight;
    mst.add(node);
    union(node.u, node.v, parent, rank);
  }
}
// printing MST
Sout("\n\n Cost of MST :" + costMst);
for( Node node : mst){
  Sout( node.u + " - " + node.v);
}
}
p s v main(String[] args){
int n=5;
ArrayList<Node> adj = new ArrayList<Node>();
adj.add(new Node(0,1,2));
        (0,3,6));
        (1,3,8));
        (1,2,3));
        (1,4,5));
        (2,4,7));
KruskalAlgorithm( adj, n);
}
}
```

Bridges in Graph :- "An edge in an undirected graph is a bridge if removing it disconnects the graph." (OR)

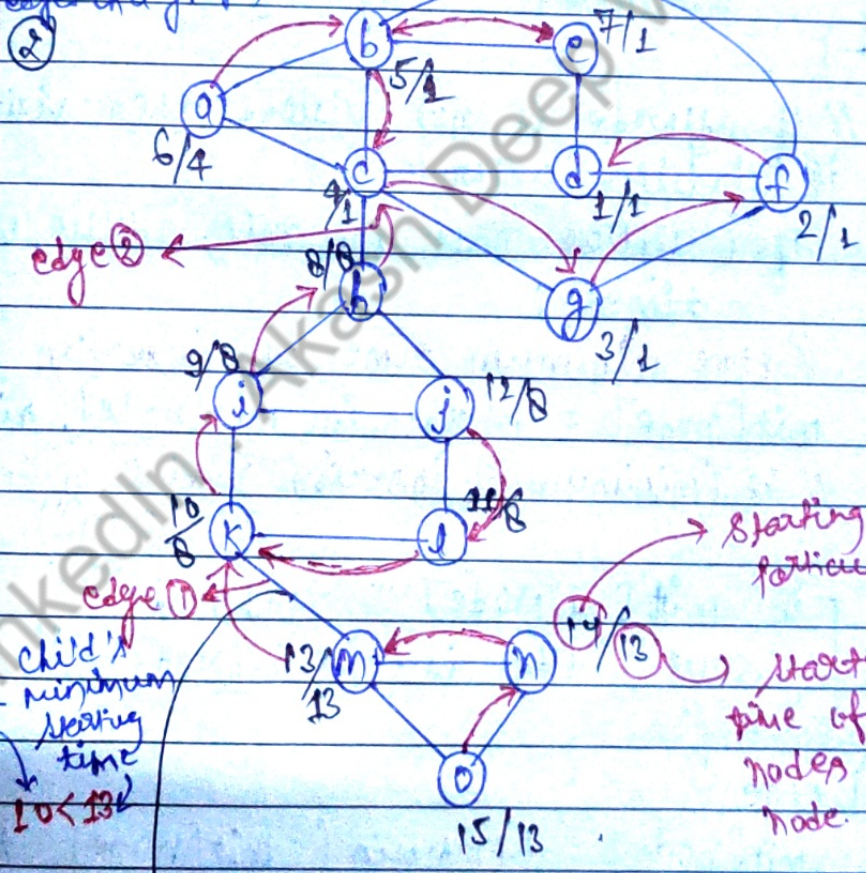
"for a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components."

Ex :-



Bridges are (0,3) & (3,4)

JENNY'S (Lec 6.12 (finding all edges in a graph) (BY DFS)



Starting of that particular node

Starting (minimum) time of adjacent nodes of that node.

parent's starting time > child's minimum starting time

10 < 13

If (starting time of any node < Minimum starting time of other node) {
then it's a bridge;
}

{TC = O(N+E)
SC = O(2N)} {toi - time of insertion
mit - minimum insertion time

Code:-

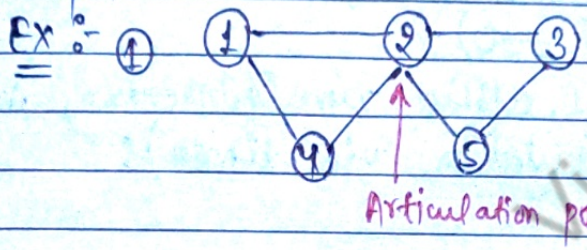
```
public void dfs(int node, int parentNode, int[] toi,
int[] mit, ArrayList<ArrayList<Integer>> adjList, boolean[]
visited, int timer) {
    visited[node] = true; // node is visited
    toi[node] = mit[node] = timer++; // store the time
                                     of insertion

    // visit the adjacent nodes
    for(int adjNode : adjList.get(node)) {
        // if the adjNode is itself parent then no need
           to call DFS
        if(adjNode == parentNode) {
            continue;
        }

        // if adjNode is not visited then visit it
        if(!visited[adjNode]) {
            dfs(adjNode, node, toi, mit, adjList, visited,
                timer);
            // store minimum time of insertion
            mit[node] = Math.min(mit[node], mit[adjNode]);
            // if (minimum of insertion > time of insertion of
                                          parent)
            if(mit[adjNode] > toi[node]) {
                sout("Edge is :" + adjNode + "->" + node);
            }
        } else {
            mit[node] = Math.min(mit[node], toi[adjNode]
```

{ TC = O(N+E)
SC = O(N) }

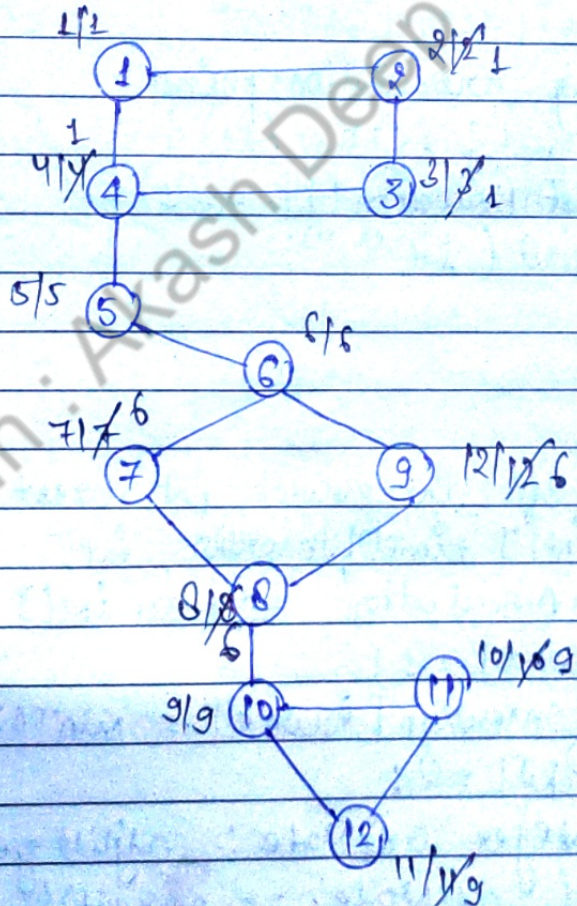
Articulation Point :- A vertex (Node) in an undirected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph.

Articulation points represent vulnerabilities in a connected network - a single points whose failure would split the network into 2 or more components.

Ex :-



Ex 2



(low[adjacentNode] >= tin[node]) && (parent != -1)

```
p s v printArticulation ( AL<AL<Integer>> adjList,
  int n){
int[] vis = new int[n+1];
int[] timeOfInsertion = new int[n+1];
int[] minTOI = new int[n+1];
int[] isArticulation = new int[n+1];
int timer =0;
// calling DFS for all nodes
for( int i=1; i<=n ; i++){
   if( vis[i] ==0) {
      dfs( i, -1, adjList, timeOfInsertion, minTOI,
         isArticulation, vis, timer);
      }
}

// printing articulation points
for( int i=1; i<=n; i++) {
   if ( isArticulation[i] == 1){
      sout( i+" ");
   }
}
}

p s v dfs( int source, int parent, AL<AL<Integer>>
adjList, int[] timeOfInsertion, int[] minTOI,
int[] isArticulation, boolean int[] vis, int timer){
   vis[source] = 1;
   timeOfInsertion[source] = minTOI[source]= timer++
   int child =0;
   for( Integer adjNode : adjList.get(source)){
      if( adjNode == parent){
         continue;
      }
```

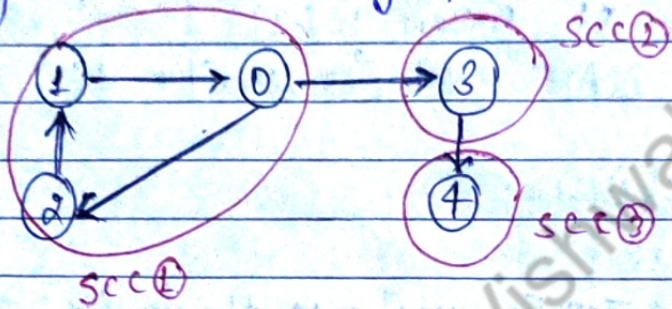
```
if( vis[adjNode] == 0){
  dfs( adjNode, source, adjList, timeOfInsertion,
    minTOI, isArticulation, vis, timer);
  minTOI[source] = Math.min( minTOI[source],
                    minTOI[adjNode]);
  // cond@ for Articulation point
  if( minTOI[adjNode] >= timeOfInsertion[source]
      && parent != (-1)) {
    isArticulation[source] = 1;
  }
  child++;
} else {
  minTOI[source] = Math.min (minTOI[source],
                    timeOfInsertion[adjNode]);
}
// If a node has more than one child
if( parent != (-1) && child > 1) {
  isArticulation[source] = 1;
}
}
```

↳ Run the DFS from the last node. So that we can divide the group of nodes separately.

Kosaraju's Algorithm for Strongly connected component :-

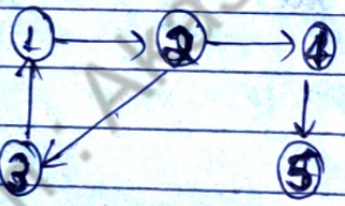
A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph.

Ex :-

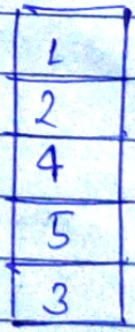


⇒ Steps :-

(1) Sort all the nodes according to their finishing time. (Topological sort).

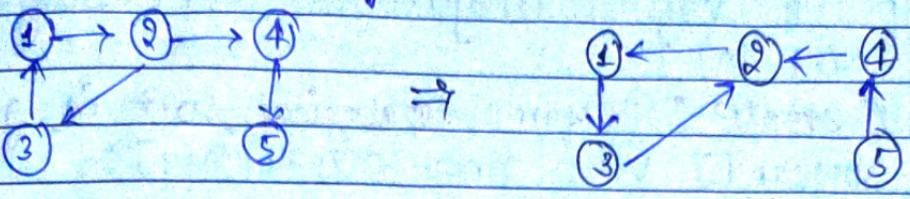


DFS (1)
DFS (2)
DFS (3)
DFS (4)
DFS (5)

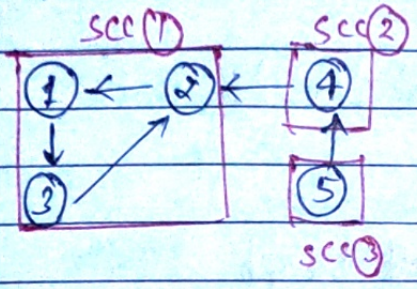


Stack (topo sort)

(2.) Transpose the graph. (Reverse the dir^{n} of edges)



(3.) Run DFS according to the finishing time.



DFS(1) → DFS(3) → DFS(2)
SCC 1

DFS(4)
SCC 2

DFS(5)
SCC 3

$TC = O(N+E)$

$SC = O(N+E) + O(N) + O(N)$

- $O(N+E)$ ↑ for transpose of graph
- $O(N)$ ↑ for vis Array
- $O(N)$ ↑ for stack

Code :-

```
p s void KosarajuAlgo( AL<AL<Integer>> adjList,
  int n) {
// Step(1) : Perform topological sort in given graph
boolean [] vis = new boolean[n+1];
Stack<Integer> stack = new Stack<>();
for( int i=1; i<=n; i++) {
  if( ! vis[i]) {
    dfs( i, stack, adjList, vis);
  }
}
// Step(2) : Transpose of graph
AL<AL<Integer>> transpose = new AL<>();
for( int i=1 ; i<=n+1 ; i++) {
  transpose.add( new AL<>());
}
for ( int node=1 ; node<=n ; node++) {
  vis[node] = ~~true~~ false;
  for( Integer adjNode : adjList.get(node)) {
    transpose.get( adjNode).add( node); // Reverse
                                        dir@ of edge
  }
}
// Step(3) : Run reverse DFS A/Q topological sort
while( ! stack.isEmpty()) {
  int currNode = stack.pop();
  if( ! visited [currNode]) {
    sout("scc");
    reverseDFS( currNode, transpose, vis);
    sout();
  }
}
}
```

```
p ^ v dfs(int node, Stack<Integer> stack,
ArrayList<ArrayList<Integer>> adjList, boolean[] vis) {
    vis[node] = true;
    for(Integer adjNode : adjList.get(node)) {
        if(!vis[adjNode]) {
            dfs(adjNode, stack, adjList, vis);
        }
    }
    stack.push(node);
}

p ^ v reverseDFS(int currNode, ArrayList<ArrayList<Integer>>
transpose, boolean[] vis) {
    vis[currNode] = true;
    Sout(currNode + " ");
    for(Integer adjNode : transpose.get(currNode)) {
        if(!vis[adjNode]) {
            reverseDfs(adjacentNode, transpose, vis);
        }
    }
}
```

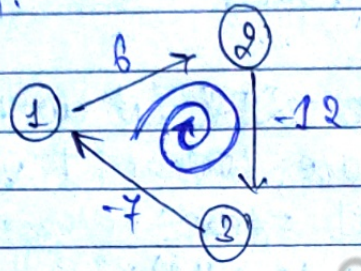
⇒ **Application** :- In social media networks, a group of people are generally strongly connected (for ex- students of a class or any other common place).

(Dijkstra's Algo works only for +ve edge weights)

Bellman-ford Algorithm :- It is used to find the shortest path from source to all nodes in the given graph. The graph may contain -ve weights.

→ It will detect the negative cycle present in the graph.

→ Bellman-ford Algorithm will not work when there is a negative cycle in the graph.

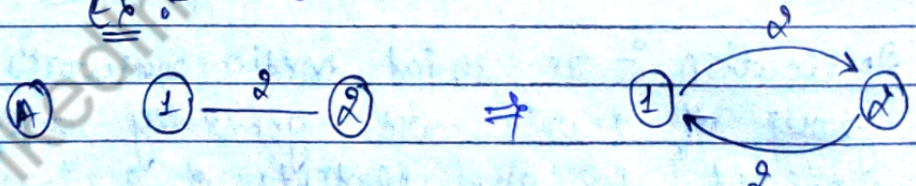


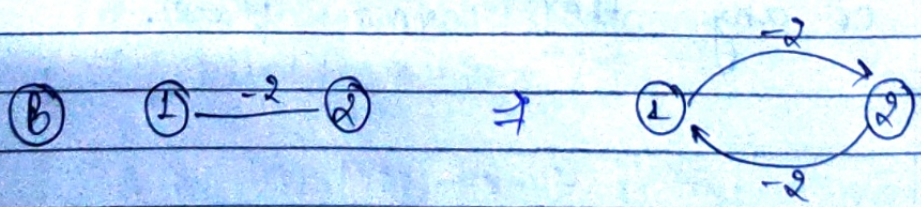
⇒ Bellman-ford Algorithm works :-

(1) Directed graph ⇒ In both cases when edges are +ve or -ve.

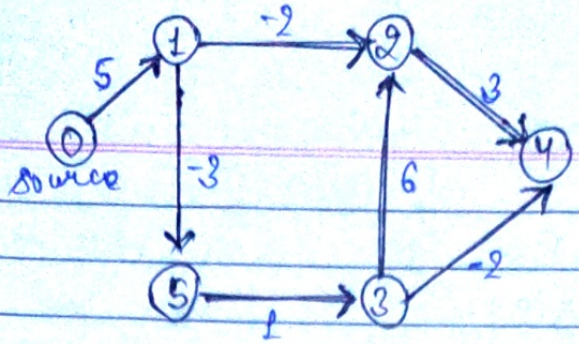
(2) Undirected graph ⇒ Convert the undirected graph into directed graph using Bi-directional arrows.

Ex :-

(A) 

(B) 

{ TC = O(N-1) * O(E)
SC = O(N) }



u	v	wt
3	2	6
5	3	1
0	1	5
1	5	-3
1	2	-2
3	4	-2
2	4	3

Steps :-

1) Relax all the edges (N-1) times. (No. of Nodes)

```
if (distance[u] + weight < distance[v]){
    distance[v] = distance[u] + weight;
}
```

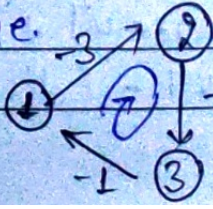
	0	1	2	3	4	5
distance →	0	~~∞~~ 5	~~∞~~ 3	~~∞~~ 3	~~∞~~ 1	~~∞~~ 2

(I)	(II)
(✗) dis[3]+6 < dis[2]	dis[3]+6 < dis[2] (✗)
(✓) dis[5]+1 < dis[3]	dis[5]+1 < dis[3] (✓)
(✓) dis[0]+5 < dis[1]	dis[0]+5 < dis[1] (✗)
(✓) dis[1]+(-3) < dis[5]	dis[1]-3 < dis[5] (✗)
(✓) dis[1]+(-2) < dis[2]	dis[1]+(-2) < dis[2] (✗)
(✗) dis[3]+(-2) < dis[4]	dis[3]+(-2) < dis[4] (✓)
(✓) dis[2]+3 < dis[4]	dis[2]+3 < dis[4] (✗)

Similarly, after the (V) relaxation we will get the minimum distance array as following -

distance	0	5	3	3	1	2

NOTE :- If we relax the edges one more time after (N-1) times & the distance array reduces again then it means it has negative cycle.



Code :-

```
class Node {
    int u, v, e;
    public Node() { }
    public Node(int u, int v, int e){
        this.u = u;
        this.v = v;
        this.e = e;
    }
}

class bellmanford {
    p s v main(-) {
    int n = 6;
    ArrayList<Node> adj = new ArrayList<>();
    adj.add(new Node(3,2,6));
    adj.add(new Node(5,3,1));
    adj.add(new Node(0,1,5));
    adj.add(new Node(1,5,-3));
    adj.add(new Node(1,2,-2));
    adj.add(new Node(3,4,-2));
    adj.add(new Node(2,4,3));
    bellmanford(adj, n, 0);
}

void bellmanford(AL<Node> edges, int n, int src){
    int[] dis = new int[n];
    Arrays.fill(dis, 1000000000);
    dis[src] = 0;
    // Run the loop for (N-1) times
    for(int i=1; i<=n-1; i++){
        for(Node node : edges){
            if(dis[node.u] + node.e < dis[node.v])
                dis[node.v] = (dis[node.u] + node.e);
        }
    }
```

```
// Now check for -ve cycle
// Run the same loop for one more time
// If any distance decreased then it means
// that there is a -ve cycle int fl=0;
for( Node node : edges){
  if( dis[node.u] + node.e < dis[node.v]){
    dis[node.v] = (dis[node.u] + node.e);
    fl = 1;
    Sout(" -ve cycle");
    break;
  }
} // If there is no -ve cycle, then print the array
if (fl==0) {
  for(int distance : dis){
    Sout(distance);
  }
}
```