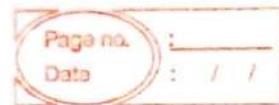


Graph

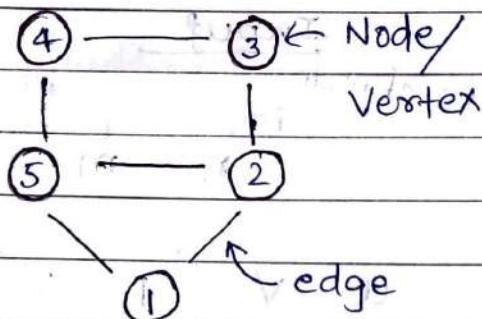
-By Nikhil Kumar

<https://www.linkedin.com/in/nikhilkumar0609/>

GRAPHS



Undirected Graph



$$\text{Degree}(2) = 3$$

Total no. of degrees
of all the nodes
 $= 2 \times \text{no. of edges}$

$$\text{Degree}(1) = 2$$

$$\text{Degree}(2) = 3$$

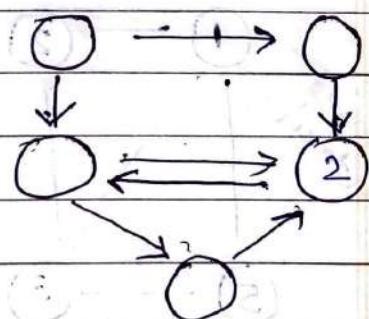
$$\text{Degree}(3) = 2$$

$$\text{Degree}(4) = 2$$

$$\text{Degree}(5) = \frac{3}{12}$$

$$\text{No. of edges} = 6.$$

Directed Graph

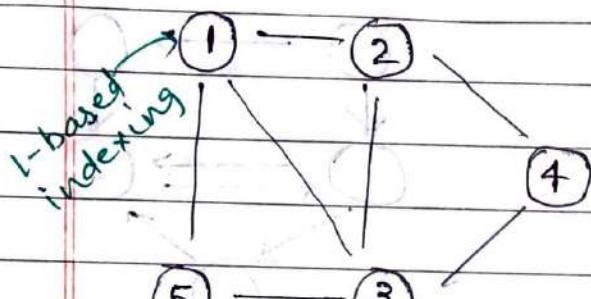


$$\text{Indegree}(2) = 3$$

$$\text{Outdegree}(2) = 1$$

QUESTION

* Graph Representation



Input
no. of nodes : n no. of edges : m

u	v
1	2
2	4
2	3
1	3
3	4
1	5
5	3

Approach 1: Adjacency Matrix

As, no. of nodes = 5.

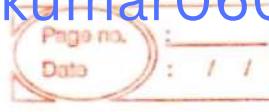
so, we will take matrix of (6×6) . because
it is 1-based indexing.
nodes

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

$$\text{adj}[u][v] = 1$$

$$\text{adj}[v][u] = 1$$

$$S.C = O(n^2)$$



Program :-

```
#include <bits/stdc++.h>
using namespace std;
```

```
int main () {
```

```
    int n, m;
```

```
    cin >> n >> m;
```

```
// declare the adjacent matrix
```

```
int adj[n+1][n+1];
```

```
// take edges as input
```

```
for (int i = 0; i < m; i++) {
```

```
    int u, v;
```

```
    cin >> u >> v;
```

```
    adj[u][v] = 1;
```

```
    adj[v][u] = 1;
```

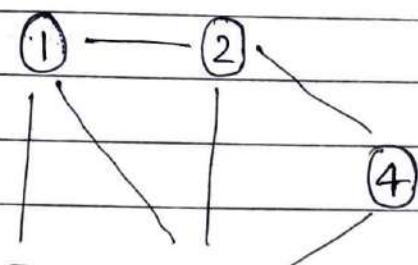
```
}
```

```
return 0;
```

```
}
```

NOTE:- We can use adjacency matrix for the smaller value of the no. of nodes.

Approach 2: Adjacency list



$n = 5$

vector<int> adj[6]

u	v	edges
1 - 2		
1 - 5		
1 - 3		
3 - 5		
2 - 3		
2 - 4		
3 - 4		

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

S.C $\rightarrow O(n + 2E)$

no. of nodes

no. of edges

Program :-

```
int main() {
    int n, m; // m → no. of edges.
    cin >> n >> m;
```

```
vector<int> adj[n+1];
```

```
for (int i=0; i<m; i++) {
```

```
    int u, v;
```

```
    cin >> u >> v;
```

```
    adj[u].push_back(v);
```

```
// adj[v].push_back(u);
```

} in case of directed

graph

}

```
return 0;
```

S.C $\rightarrow O(n + E)$

Y

→ Weighted Graph

```
int main() {
```

```
    int n, m;
```

```
    cin >> n >> m;
```

```
    vector<pair<int, int>> adj[n+1];
```

```
    for (int i = 0; i < m; i++) {
```

```
        int u, v, wt;
```

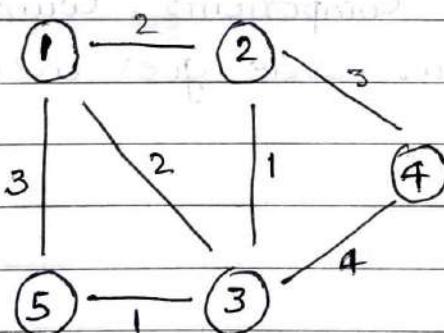
```
        cin >> u >> v >> wt;
```

```
        adj[u].push_back({v, wt});
```

```
        adj[v].push_back({u, wt});
```

```
    }
```

```
}
```

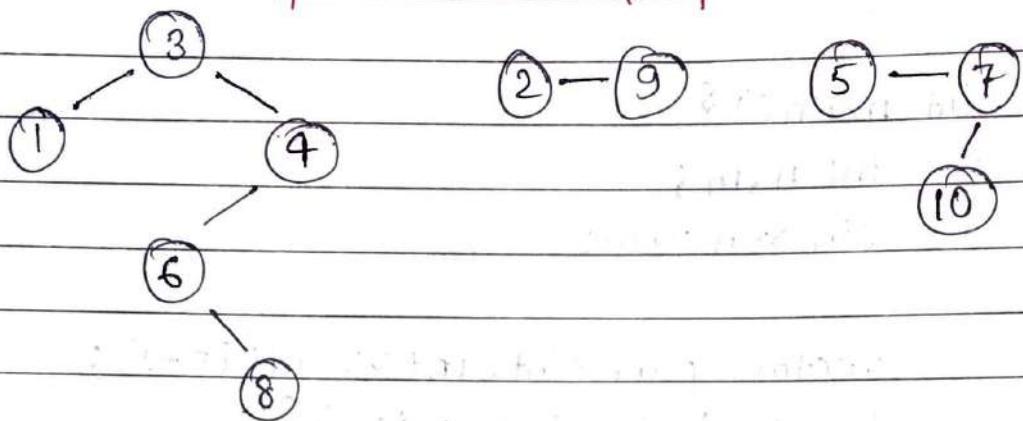


```
vector<pair<int, int>> adj[6];
```

node	wt.
0	
1	{(2, 2),
2	{(1, 2),
3	
4	
5	

S.C $\rightarrow O(n + 2E) + 2E$

* Connected Components in Graph



~~3 4 8 & 9 8 9 10~~

```
for (i=x; i<=10; i++) {
    if (!vis[i]) {
```

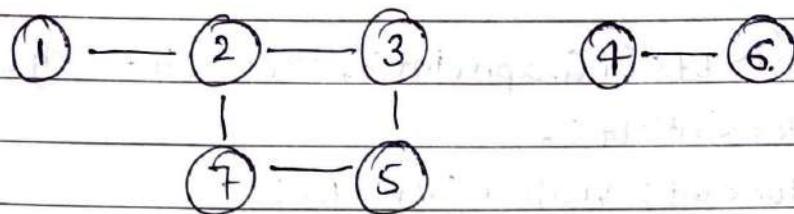
vis[i]

dfs
bfs

	1	2	3	4	5	6	7	8	9	10
vis[i]	0	0	0	0	0	0	0	0	0	0

→ To find the no. of components, Count the number of time your code goes inside the if statement.

* BFS (Breadth-First Search)



Adjacency list

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

$Q_p = 1 \ 2 \ 3 \ 7 \ 5 \ 4 \ 6$

queue (for 1) queue (for 4)

for ($i=1 \rightarrow n$)

{

if (visited[i] == 0)

bf(s(i));

};

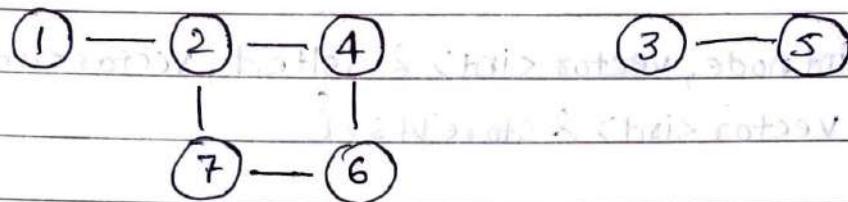
T.C $\rightarrow O(N+E)$, N is time taken for visiting N nodes, and E is for travelling through adjacent nodes overall.

S.C $\rightarrow O(N+E) + O(N) + O(N)$
 adjacent queue Array
 list (visited)

Program :-

```
vector<int> bfsOfGraph(int V, vector<int> adj[]) {
    vector<int> bfs;
    vector<int> visited(V+1, 0);
    for(int i=1; i<=V; i++) {
        if (!visited[i]) {
            queue<int> q;
            q.push(i);
            visited[i] = 1;
            while (!q.empty()) {
                int node = q.front();
                q.pop();
                bfs.push_back(node);
                //for the adjacent node
                for (auto it: adj[node]) {
                    if (!visited[it]) {
                        q.push(it);
                        visited[it] = 1;
                    }
                }
            }
        }
    }
    return bfs;
}
```

* Depth - First Search



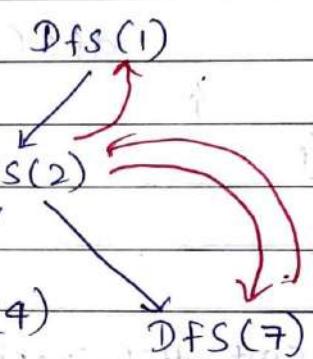
Adjacency list

1	2
2	1 4 7
3	5
4	2 6
5	3
6	4 7
7	2 6

Visited:

0	∅	∅	0	∅	0	∅	∅
0	1	2	3	4	5	6	7

DFS(1)



↑ is visited.

for ($i = 1 \rightarrow 7$)

{

if (!visited[i])

dfs(i)

}

O/P = 1 2 4 6 7 3 5

T.C $\rightarrow O(N+E)$

S.C $\rightarrow O(N+E) + O(N) + O(N)$

X

DFS(3)

X

DFS(5)

Program :-

```
void dfs(int node, vector<int> &visited, vector<int> adj[],  
        vector<int> &storeDfs) {
```

```
    storeDfs.push_back(node);
```

```
    visited[node] = 1;
```

```
    for (auto it : adj[node]) {
```

```
        if (!visited[it]) {
```

```
            dfs(it, visited, adj, storeDfs);
```

```
}
```

```
}
```

```
vector<int> dfsOfGraph(int V, vector<int> adj[]) {
```

```
    vector<int> storeDfs;
```

```
    vector<int> visited(V + 1, 0);
```

```
    for (int i = 1; i <= V; i++) {
```

```
        if (!visited[i]) {
```

```
            dfs(i, visited, adj, storeDfs);
```

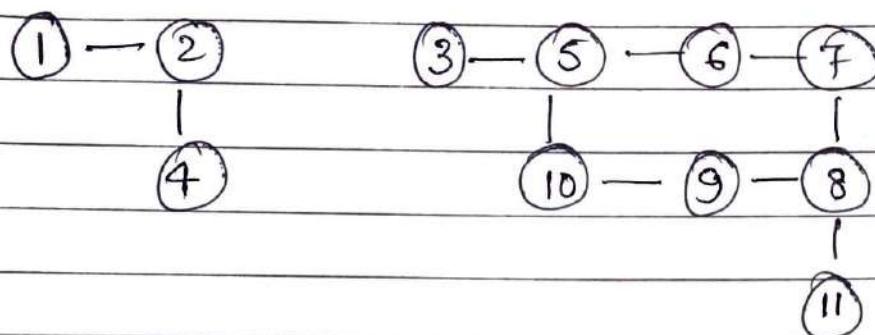
```
}
```

```
}
```

```
    return storeDfs;
```

```
}
```

* Detect a Cycle in undirected Graph (BFS)



adjacency list

1	2	0 0 0 0 0 0 0 0 0 0 0 0
2	1, 4	0 1 2 3 4 5 6 7 8 9 10 11
3	5	
4	2	
5	3, 10, 6	
6	5, 7	
7	6, 8	
8	7, 9, 11	<8, 7>
9	10, 8	<9, 10>
10	5, 9	<7, 6>
11	8	queue <10, 5> <6, 5> <5, 3> <3, -1>

for($i=1 \rightarrow N$)

{

 if(!visited[i])

 if(CycleBFS(i))

 return true;

T.C $\rightarrow O(N+E)$

S.C $\rightarrow O(N+E) + O(N) + O(N)$

}

Program:-

```
bool checkForCycle(int start, int V, vector<int> adj[],  
                    vector<int>& visited){
```

```
    queue<pair<int, int>> q;
```

```
    visited[start] = true;
```

```
    q.push({start, -1});
```

```
    while (!q.empty()) {
```

```
        int node = q.front().first;
```

```
        int parent = q.front().second;
```

```
        q.pop();
```

```
        for (auto it : adj[node]) {
```

```
            if (!visited[it]) {
```

```
                visited[it] = true;
```

```
                q.push({it, node});
```

```
}
```

```
            else if (parent != it)
```

```
                return true;
```

```
}
```

```
    }
```

```
    bool isCycle(int V, vector<int> adj[]) {
```

```
        vector<int> visited(V + 1, 0);
```

```
        for (int i = 1; i <= V; i++) {
```

```
            if (!visited[i]) {
```

```
                if (checkForCycle(i, V, adj, visited))
```

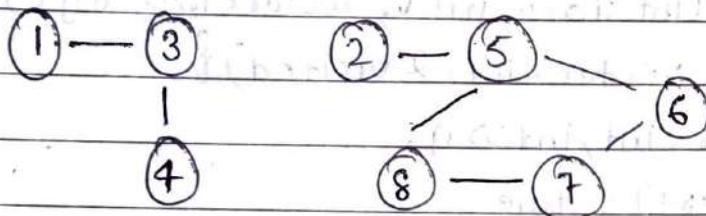
```
                    return true;
```

```
}
```

```
    return false;
```

```
}
```

* Detect a Cycle in Undirected Graph (DFS)



adjacency list

1	3	[0 0 1 0 1 0 1 0 1 0 1]
2	5	0 1 2 3 4 5 6 7 8
3	1,4	
4	3	
5	2,6,8	
6	5,7	
7	6,8	
8	7,5	

```

for (i=1→8)
{
    if (vis[i]==0)
        if (cycleDFS(i))
            return true;
}
  
```

iske bad ('8') ka adj, 7 aur s
hai, aur 7 parent hai, but 5
parent nhi hai aur visited
hai, mtlb. cycle exist krta hai

T.C → O(N+E)

S.C → O(N+E) + O(N) + O(N)

Program :-

```

bool checkForCycle(int node, int parent, vector<int>& visited,
vector<int> adj[]) {
    visited[node] = 1;
    for (auto it : adj[node]) {
        if (visited[it] == 0) {
            if (checkForCycle(it, node, visited, adj))
                return true;
        } else if (it != parent)
            return true;
    }
    return false;
}

bool isCycle(int V, vector<int> adj[]) {
    vector<int> visited(V + 1, 0);
    for (int i = 1; i <= V; i++) {
        if (!visited[i]) {
            if (checkForCycle(i, -1, visited, adj))
                return true;
        }
    }
    return false;
}

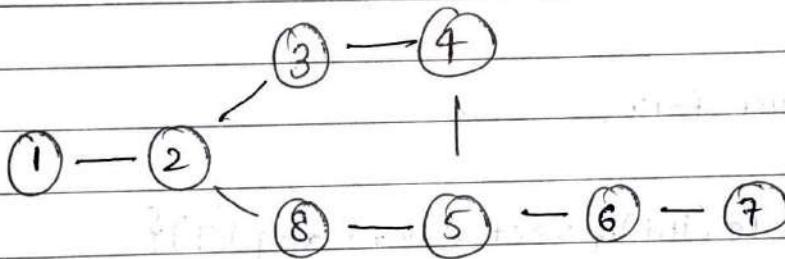
```

* Bipartite Graph (BFS)

→ A Graph that can be coloured using 2 colors such that no two adjacent nodes have same color.

- Graphs having odd number of nodes in a Cycle is a non-bipartite graph.
- Graphs which doesn't have cycle with odd no. of nodes is a bipartite graph.

e.g.



			0	1	0	1	1	0
			-1	-1	-1	-1	-1	-1
			0	1	2	3	4	5

color array = [-1, -1, -1, -1, -1, -1, -1, -1]

5
4
8
3
2
1
queue

node = 1, adj = 2. if -1, then it
node = 2, adj = 3, 8 } will be coloured with
node = 3, adj = 8, 4 exact opposite of node 2.
↳ already colored & opposite.

node = 8, adj = 8, 5

node = 4, adj = 3, 5

already colored
& opposite

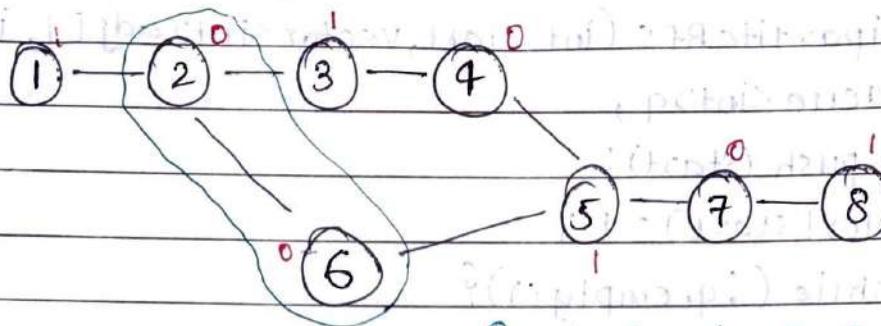
already Colored but
not opposite

↓
False

Program :-

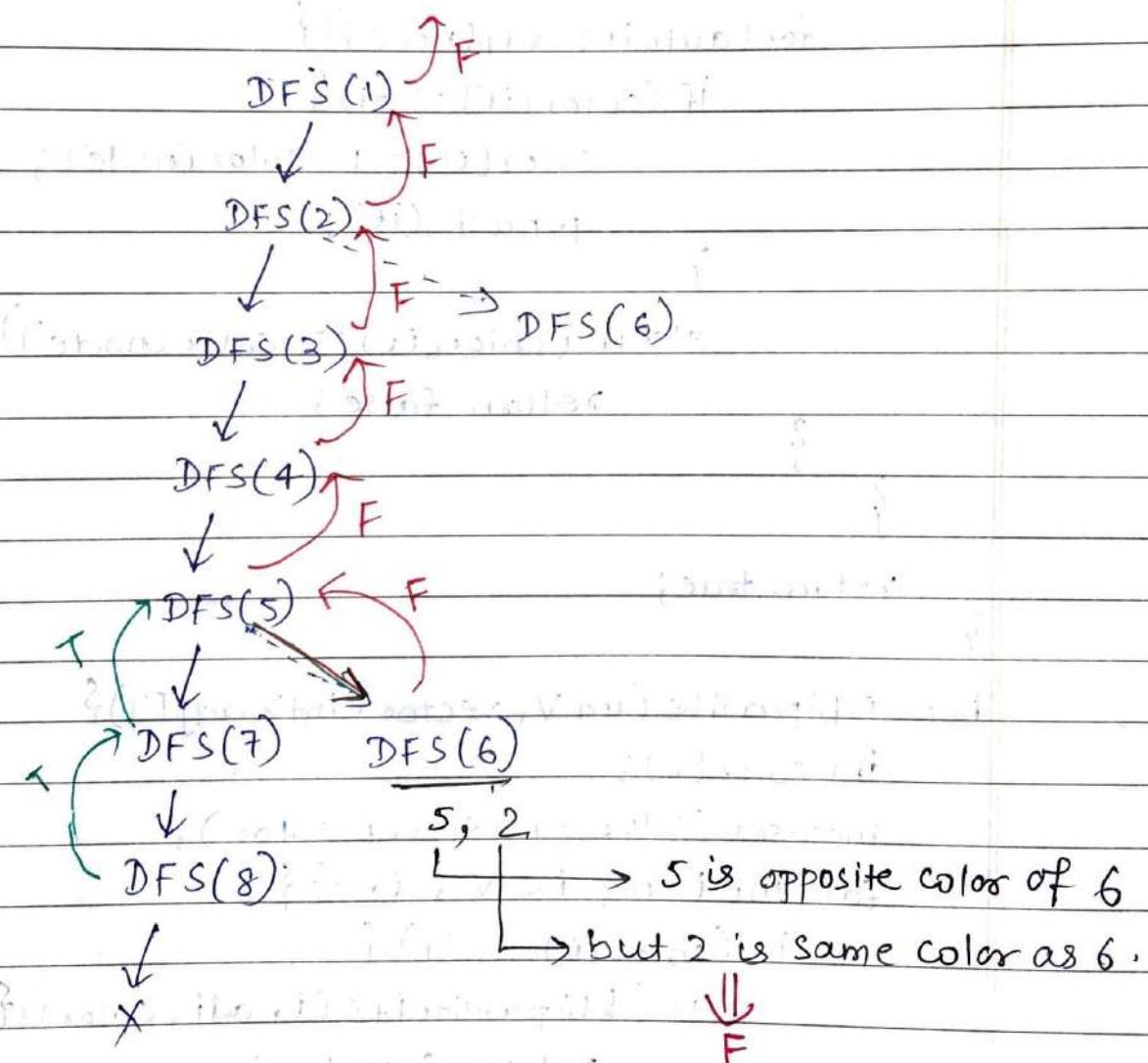
```
bool bipartiteBfs (int start, vector<int> adj[], int color[]) {  
    queue<int> q;  
    q.push (start);  
    color[start] = 1;  
    while (!q.empty ()) {  
        int node = q.front ();  
        q.pop ();  
  
        for (auto it : adj[node]) {  
            if (color[it] == -1) {  
                color[it] = 1 - color[node];  
                q.push (it);  
            }  
            else if (color[it] == color[node])  
                return false;  
        }  
    }  
    return true;  
}  
  
bool isBipartite (int V, vector<int> adj[]) {  
    int color[V];  
    memset (color, -1, sizeof color);  
    for (int i = 0; i < V; i++) {  
        if (color[i] == -1) {  
            if (!bipartiteBfs (i, adj, color))  
                return false;  
        }  
    }  
    return true;  
}
```

* Bipartite Graph (DFS)



color array =

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



$$T.C \rightarrow O(V+E)$$

$$S.C \rightarrow O(V+E) + O(V) + O(V)$$

Program :-

```

bool bipartiteDfs (int node, vector<int>adj[], int color[]) {
    for (auto it : adj[node]) {
        if (color[it] == -1) {
            color[it] = 1 - color[node];
            if (!bipartiteDfs(it, adj, color)) {
                return false;
            }
        } else if (color[it] == color[node])
            return false;
    }
    return true;
}

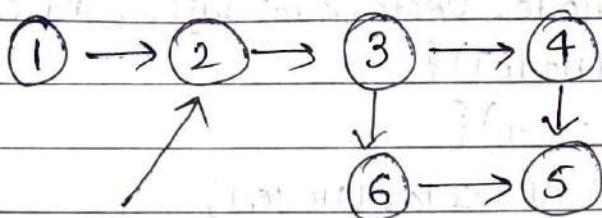
```

```

bool isBipartite (int V, vector<int>adj[]) {
    int color[V];
    memset(color, -1, sizeof color);
    for (int i = 0; i < V; i++) {
        if (color[i] == -1) {
            if (!bipartiteDfs(i, adj, color))
                return false;
        }
    }
    return true;
}

```

* Detect a Cycle in Directed Graph (DFS)



Adjacency list

$1 \rightarrow 2$

$2 \rightarrow 3$

$3 \rightarrow 4, 6$

$4 \rightarrow 5$

$5 \rightarrow$

$6 \rightarrow 5$

$7 \rightarrow 2, 8$

$8 \rightarrow 9$

$9 \rightarrow 7$

$f(1)$

$f(2)$

$f(3)$

$f(4)$

$f(5)$

X

		visited								
1	2	3	4	5	6	7	8	9		
∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅

DFS Visited

		DFS Visited								
1	2	3	4	5	6	7	8	9		
r	x	x	x	x	x	x	1	1	1	1

$f(7) \uparrow T$

T.C $\rightarrow O(V+E)$

$f(8) \uparrow T$

S.C $\rightarrow O(2V)$

$f(9) \uparrow T$

$f(7) \Rightarrow$ Both visited and DFSvisited is '1'

↓
True

Program :-

```

bool checkCycle(int node, vector<int> adj[], int vis[],
                int dfsVis[]){

    vis[node] = 1;
    dfsVis[node] = 1;

    for (auto it : adj[node]) {
        if (!vis[it]) {
            if (checkCycle(it, adj, vis, dfsVis))
                return true;
        }
        else if (dfsVis[it])
            return true;
    }

    dfsVis[node] = 0;
    return false;
}

bool isCyclic(int N, vector<int> adj[]) {
    int vis[N], dfsVis[N];
    memset(vis, 0, sizeof vis);
    memset(dfsVis, 0, sizeof dfsVis);

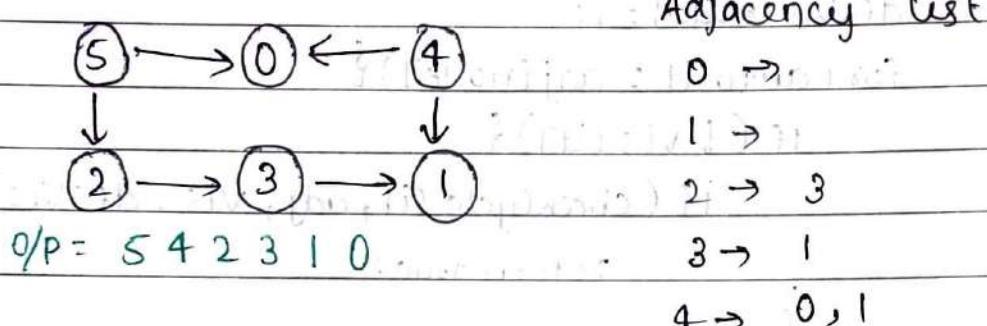
    for (int i = 0; i < N; i++) {
        if (!vis[i]) {
            if (checkCycle(i, adj, vis, dfsVis))
                return true;
        }
    }

    return false;
}

```

* Topological Sort (DFS)

(DAG) • Linear ordering of vertices such that if there is an edge $u \rightarrow v$, u appears before v in that ordering.



for ($i = 0 \rightarrow 5$)

{

 if (!vis[i])

 {

 dfs(i);

 }

 }

 }

 for ($i = 0 \rightarrow 5$) : $\neg \text{visited}[i] = 1$

 dfs(0) \Rightarrow No adj. nodes

 when dfs(0) is over then put it in stack

 for ($i = 1 \rightarrow 5$) : $\neg \text{dfs}(i)$

 for ($i = 2 \rightarrow 5$) :

$\neg \text{dfs}(2)$

$\neg \text{dfs}(3)$

$\neg \text{dfs}(4)$

$\neg \text{dfs}(5)$

$\neg \text{dfs}(6)$

$\neg \text{dfs}(7)$

$\neg \text{dfs}(8)$

$\neg \text{dfs}(9)$

$\neg \text{dfs}(10)$

$\neg \text{dfs}(11)$

$\neg \text{dfs}(12)$

$\neg \text{dfs}(13)$

$\neg \text{dfs}(14)$

$\neg \text{dfs}(15)$

$\neg \text{dfs}(16)$

$\neg \text{dfs}(17)$

$\neg \text{dfs}(18)$

$\neg \text{dfs}(19)$

$\neg \text{dfs}(20)$

$\neg \text{dfs}(21)$

$\neg \text{dfs}(22)$

$\neg \text{dfs}(23)$

$\neg \text{dfs}(24)$

$\neg \text{dfs}(25)$

$\neg \text{dfs}(26)$

$\neg \text{dfs}(27)$

$\neg \text{dfs}(28)$

$\neg \text{dfs}(29)$

$\neg \text{dfs}(30)$

$\neg \text{dfs}(31)$

$\neg \text{dfs}(32)$

$\neg \text{dfs}(33)$

$\neg \text{dfs}(34)$

$\neg \text{dfs}(35)$

$\neg \text{dfs}(36)$

$\neg \text{dfs}(37)$

$\neg \text{dfs}(38)$

$\neg \text{dfs}(39)$

$\neg \text{dfs}(40)$

$\neg \text{dfs}(41)$

$\neg \text{dfs}(42)$

$\neg \text{dfs}(43)$

$\neg \text{dfs}(44)$

$\neg \text{dfs}(45)$

$\neg \text{dfs}(46)$

$\neg \text{dfs}(47)$

$\neg \text{dfs}(48)$

$\neg \text{dfs}(49)$

$\neg \text{dfs}(50)$

$\neg \text{dfs}(51)$

$\neg \text{dfs}(52)$

$\neg \text{dfs}(53)$

$\neg \text{dfs}(54)$

$\neg \text{dfs}(55)$

$\neg \text{dfs}(56)$

$\neg \text{dfs}(57)$

$\neg \text{dfs}(58)$

$\neg \text{dfs}(59)$

$\neg \text{dfs}(60)$

$\neg \text{dfs}(61)$

$\neg \text{dfs}(62)$

$\neg \text{dfs}(63)$

$\neg \text{dfs}(64)$

$\neg \text{dfs}(65)$

$\neg \text{dfs}(66)$

$\neg \text{dfs}(67)$

$\neg \text{dfs}(68)$

$\neg \text{dfs}(69)$

$\neg \text{dfs}(70)$

$\neg \text{dfs}(71)$

$\neg \text{dfs}(72)$

$\neg \text{dfs}(73)$

$\neg \text{dfs}(74)$

$\neg \text{dfs}(75)$

$\neg \text{dfs}(76)$

$\neg \text{dfs}(77)$

$\neg \text{dfs}(78)$

$\neg \text{dfs}(79)$

$\neg \text{dfs}(80)$

$\neg \text{dfs}(81)$

$\neg \text{dfs}(82)$

$\neg \text{dfs}(83)$

$\neg \text{dfs}(84)$

$\neg \text{dfs}(85)$

$\neg \text{dfs}(86)$

$\neg \text{dfs}(87)$

$\neg \text{dfs}(88)$

$\neg \text{dfs}(89)$

$\neg \text{dfs}(90)$

$\neg \text{dfs}(91)$

$\neg \text{dfs}(92)$

$\neg \text{dfs}(93)$

$\neg \text{dfs}(94)$

$\neg \text{dfs}(95)$

$\neg \text{dfs}(96)$

$\neg \text{dfs}(97)$

$\neg \text{dfs}(98)$

$\neg \text{dfs}(99)$

$\neg \text{dfs}(100)$

$\neg \text{dfs}(101)$

$\neg \text{dfs}(102)$

$\neg \text{dfs}(103)$

$\neg \text{dfs}(104)$

$\neg \text{dfs}(105)$

$\neg \text{dfs}(106)$

$\neg \text{dfs}(107)$

$\neg \text{dfs}(108)$

$\neg \text{dfs}(109)$

$\neg \text{dfs}(110)$

$\neg \text{dfs}(111)$

$\neg \text{dfs}(112)$

$\neg \text{dfs}(113)$

$\neg \text{dfs}(114)$

$\neg \text{dfs}(115)$

$\neg \text{dfs}(116)$

$\neg \text{dfs}(117)$

$\neg \text{dfs}(118)$

$\neg \text{dfs}(119)$

$\neg \text{dfs}(120)$

$\neg \text{dfs}(121)$

$\neg \text{dfs}(122)$

$\neg \text{dfs}(123)$

$\neg \text{dfs}(124)$

$\neg \text{dfs}(125)$

$\neg \text{dfs}(126)$

$\neg \text{dfs}(127)$

$\neg \text{dfs}(128)$

$\neg \text{dfs}(129)$

$\neg \text{dfs}(130)$

$\neg \text{dfs}(131)$

$\neg \text{dfs}(132)$

$\neg \text{dfs}(133)$

$\neg \text{dfs}(134)$

$\neg \text{dfs}(135)$

$\neg \text{dfs}(136)$

$\neg \text{dfs}(137)$

$\neg \text{dfs}(138)$

$\neg \text{dfs}(139)$

$\neg \text{dfs}(140)$

$\neg \text{dfs}(141)$

$\neg \text{dfs}(142)$

$\neg \text{dfs}(143)$

$\neg \text{dfs}(144)$

$\neg \text{dfs}(145)$

$\neg \text{dfs}(146)$

$\neg \text{dfs}(147)$

$\neg \text{dfs}(148)$

$\neg \text{dfs}(149)$

$\neg \text{dfs}(150)$

$\neg \text{dfs}(151)$

$\neg \text{dfs}(152)$

$\neg \text{dfs}(153)$

$\neg \text{dfs}(154)$

$\neg \text{dfs}(155)$

$\neg \text{dfs}(156)$

$\neg \text{dfs}(157)$

$\neg \text{dfs}(158)$

$\neg \text{dfs}(159)$

$\neg \text{dfs}(160)$

$\neg \text{dfs}(161)$

$\neg \text{dfs}(162)$

$\neg \text{dfs}(163)$

$\neg \text{dfs}(164)$

$\neg \text{dfs}(165)$

$\neg \text{dfs}(166)$

$\neg \text{dfs}(167)$

$\neg \text{dfs}(168)$

$\neg \text{dfs}(169)$

$\neg \text{dfs}(170)$

$\neg \text{dfs}(171)$

$\neg \text{dfs}(172)$

$\neg \text{dfs}(173)$

$\neg \text{dfs}(174)$

$\neg \text{dfs}(175)$

$\neg \text{dfs}(176)$

$\neg \text{dfs}(177)$

$\neg \text{dfs}(178)$

$\neg \text{dfs}(179)$

$\neg \text{dfs}(180)$

$\neg \text{dfs}(181)$

$\neg \text{dfs}(182)$

$\neg \text{dfs}(183)$

$\neg \text{dfs}(184)$

$\neg \text{dfs}(185)$

$\neg \text{dfs}(186)$

$\neg \text{dfs}(187)$

$\neg \text{dfs}(188)$

$\neg \text{dfs}(189)$

$\neg \text{dfs}(190)$

$\neg \text{dfs}(191)$

$\neg \text{dfs}(192)$

$\neg \text{dfs}(193)$

$\neg \text{dfs}(194)$

Program :-

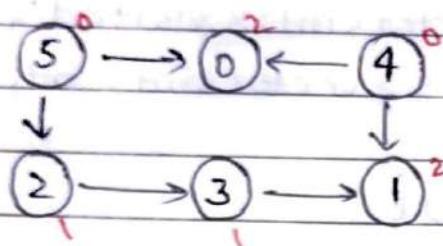
```
void findTopoSort(int node, vector<int>& visited,
                  stack<int>& s, vector<int> adj[]) {
    visited[node] = 1;
    for (auto it : adj[node]) {
        if (!visited[it]) {
            findTopoSort(it, visited, s, adj);
        }
    }
    s.push(node);
}
```

```
vector<int> topoSort(int N, vector<int> adj[]) {
    stack<int> s;
    vector<int> visited(N, 0);
    for (int i = 0; i < N; i++) {
        if (visited[i] == 0) {
            findTopoSort(i, visited, s, adj);
        }
    }
    vector<int> topo;
    while (!s.empty()) {
        topo.push_back(s.top());
        s.pop();
    }
    return topo;
}
```

→ KAHN'S ALGORITHM

* Topological Sort (BFS)

(DAG) ↗



adjacency list

0 →

1 →

2 → 3

3 → 1

4 → 0, 1

5 → 0, 2

0	0				
X	X	X	X	0	0
2	2	2	2	0	0
0	1	2	3	4	5

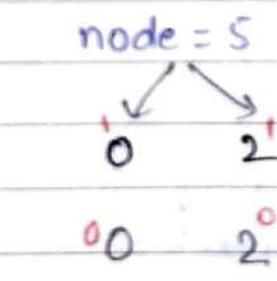
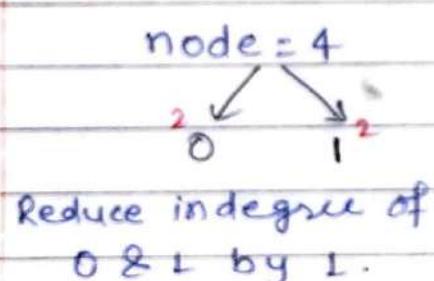
Visited[]

+	topo : →
3	4
2	5
0	0
5	2
4	3

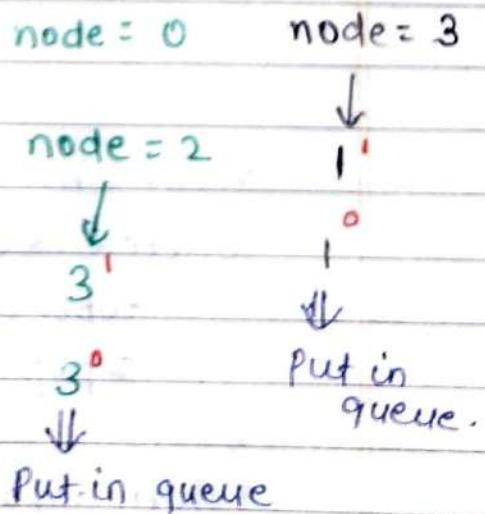
queue

→ Count the indegree of nodes and replace them in visited array.

→ Put nodes with indegree '0' in queue.



indegree=0
hence, put '0' &
'2' in queue



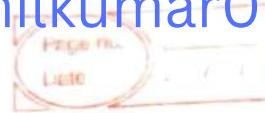
Put in queue

T.C → O(N+E)
S.C → O(N) + O(N)

node = 1

Program :-

```
vector<int> topoSort (int N, vector<int> adj[]) {  
    queue<int> q;  
    vector<int> indegree (N, 0);  
    for (int i=0; i<N; i++) {  
        for (auto it: adj[i]) {  
            indegree [it]++;  
        }  
    }  
    for (int i=0; i<N; i++) {  
        if (indegree [i] == 0) {  
            q.push (i);  
        }  
    }  
    vector<int> topo;  
    while (!q.empty ()) {  
        int node = q.front ();  
        q.pop ();  
        topo.push_back (node);  
        for (auto it: adj [node]) {  
            indegree [it] --;  
            if (indegree [it] == 0) {  
                q.push (it);  
            }  
        }  
    }  
    return topo;  
}
```



* Detect a Cycle in Directed Graph (BFS)

If Topological sort exists \Rightarrow Cycle doesn't exist

If Topological sort doesn't exist \Rightarrow Cycle exist

Program :-

```
bool isCyclic (int N, vector<int> adj[]) {
```

```
    queue<int> q;
```

```
    vector<int> indegree(N, 0);
```

```
    for (int i = 0; i < N; i++) {
```

```
        for (auto it : adj[i]) {
```

```
            indegree[it]++;
```

```
}
```

```
}
```

```
    for (int i = 0; i < N; i++) {
```

```
        if (indegree[i] == 0) {
```

```
            q.push(i);
```

```
}
```

```
}
```

```
    int count = 0;
```

```
    while (!q.empty()) {
```

```
        int node = q.front();
```

```
        q.pop();
```

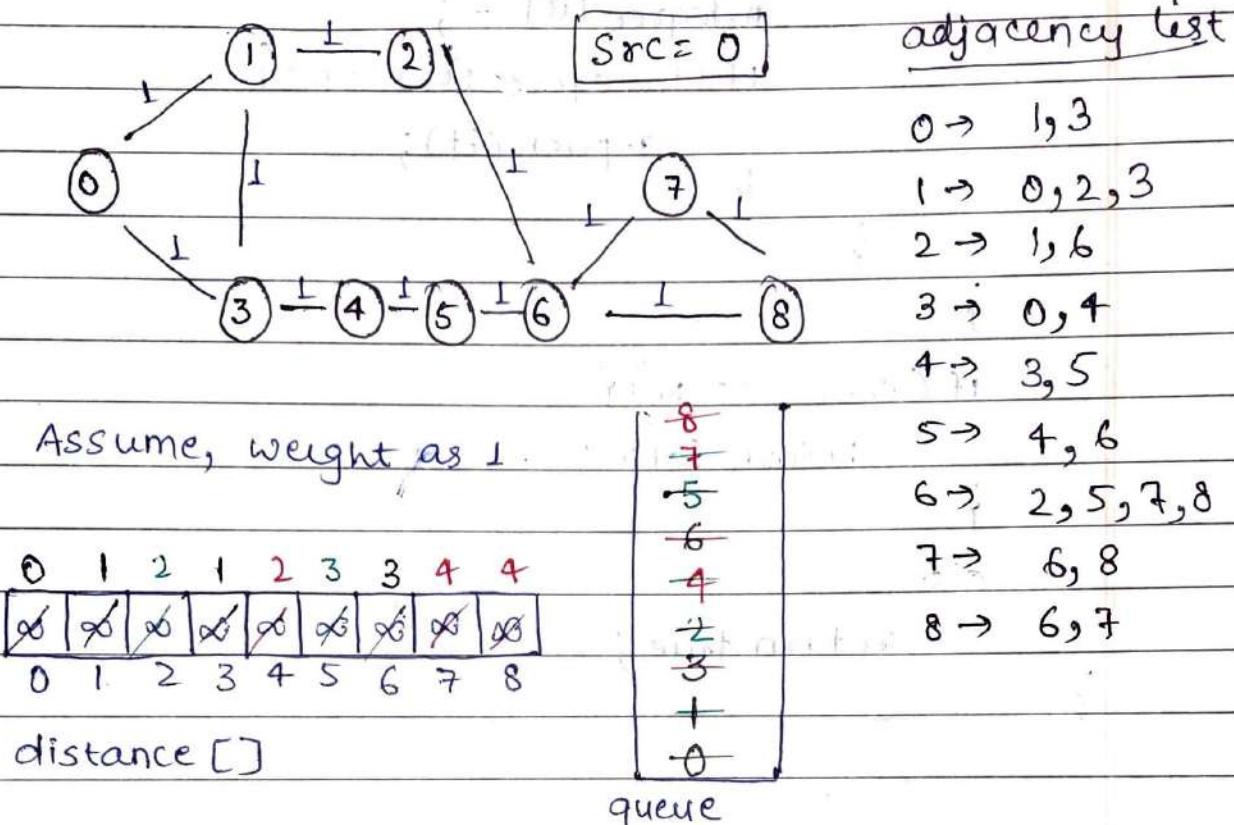
```
        count++;
```



```
for (auto it : adj[node]) {
    indegree[it] -- ;
    if (indegree[it] == 0) {
        q.push(it);
    }
}
if (count == N) {
    return false;
}
else
    return true;
}
```



* Shortest Distance in Undirected Graph



$\rightarrow \text{node} = 0$
 $\text{dis} = 0$

$\text{src} \rightsquigarrow 0 \xrightarrow{+1} 1 = 1$

$\xrightarrow{+1} 3 = 1$

$\rightarrow \text{node} = 1$

$\text{dis} = 1$

$\text{src} \rightsquigarrow 1 \xrightarrow{+1} 0 = 2 \times$

$\xrightarrow{+1} 2 = 2$

$\xrightarrow{+1} 3 = 2 \times$ (because $2 > 1$, prev. shortest dist.)

$\rightarrow \text{node} = 3$

$\text{dis} = 1$

$\text{src} \rightsquigarrow 1 \xrightarrow{+1} 0 = 2 \times$

$\xrightarrow{+1} 4 = 2$

$\rightarrow \text{node} = 5, \text{dis} = 3$

$\rightsquigarrow 3 \xrightarrow{+1} 4 = 4 \times$

$\xrightarrow{+1} 6 = 4 \times$

$\rightarrow \text{node} = 2$

$\text{dis} = 2$

$\rightsquigarrow 2 \xrightarrow{+1} 1 = 3 \times$

$\xrightarrow{+1} 6 = 3$

$\rightarrow \text{node} = 7, \text{dis} = 4$

$\rightsquigarrow 4 \xrightarrow{+1} 6 = 5 \times$

$\xrightarrow{+1} 8 = 5 \times$

$\rightarrow \text{node} = 4$

$\text{dis} = 2$

$\rightsquigarrow 4 \xrightarrow{+1} 3 = 3 \times$

$\xrightarrow{+1} 5 = 3$

$\rightarrow \text{node} = 8, \text{dis} = 4$

$\rightsquigarrow 4 \xrightarrow{+1} 6 = 5 \times$

$\xrightarrow{+1} 8 = 5 \times$

$\rightarrow \text{node} = 6$

$\text{dis} = 3$

$\rightsquigarrow 6 \xrightarrow{+1} 2 = 4 \times$

$\xrightarrow{+1} 5 = 4 \times$

$\rightarrow \text{node} = 8, \text{dis} = 4$

$\rightsquigarrow 4 \xrightarrow{+1} 6 = 5 \times$

$\xrightarrow{+1} 7 = 5 \times$

T.C $\rightarrow O(N+E)$

S.C $\rightarrow \underbrace{O(N)} + \underbrace{O(N)}$

queue
distance []

Program :-

```
void BFS(vector<int> adj[], int N, int src){
```

```
    int dist[N];
```

```
    for (int i=0; i<N; i++) {
```

```
        dist[i] = INT_MAX;
```

```
}
```

```
queue <int> q;
```

```
dist[src] = 0;
```

```
q.push(src);
```

```
while (!q.empty()) {
```

```
    int node = q.front();
```

```
    q.pop();
```

```
    for (auto it : adj[node]) {
```

```
        if (dist[node]+1 < dist[it]) {
```

```
            dist[it] = dist[node]+1;
```

```
            q.push(it);
```

```
}
```

```
{}
```

```
for (int i=0; i<N; i++) {
```

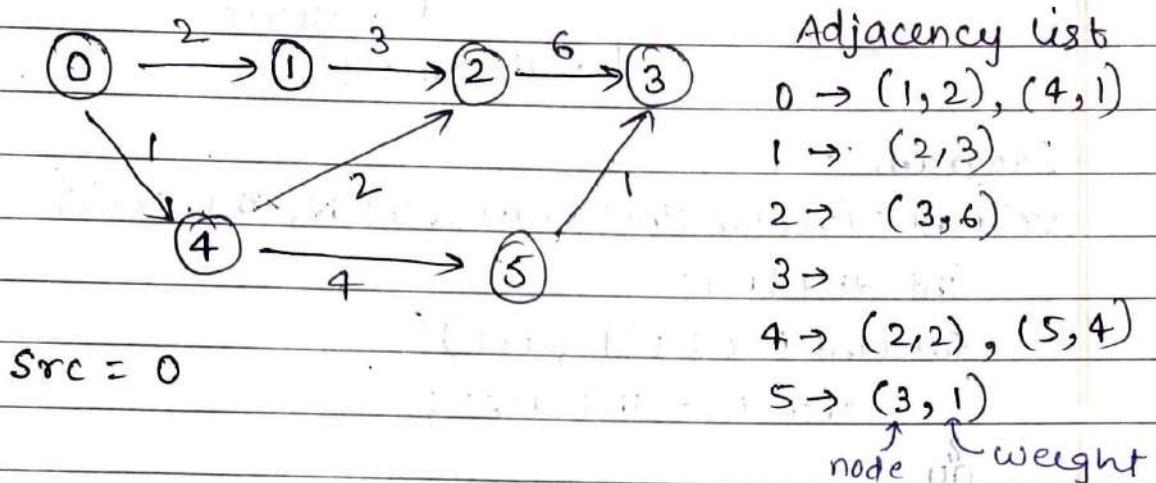
```
    cout << dist[i] << " ";
```

```
}
```

```
{}
```

Weighted

* Shortest Path in [↑] Directed Acyclic Graph (DAG)



Step 1: → Find topo sort ⇒

0	(i)
4	(ii)
5	(iii)
1	(iv)
2	(v)
3	(vi)

stack

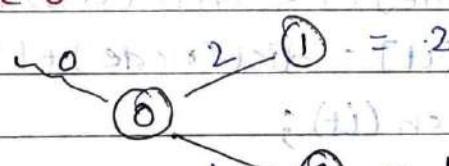
Step 2: →

0	2	3	6	1	5
0	1	2	3	4	5

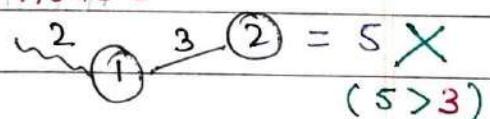
topo sort

for
each node before entering.

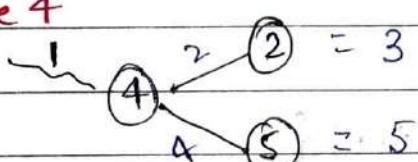
(i) node 0



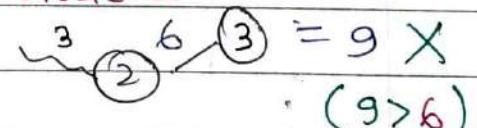
(iv) node 1



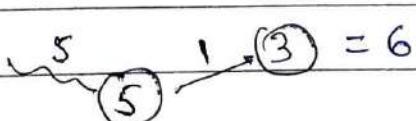
(ii) node 4



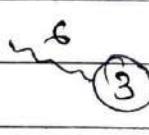
(v) node 2



(iii) node 5



(vi) node 3



Program:-

```
#include <bits/stdc++.h>
#define INF INT_MAX
using namespace std;

void topoSrt(int node, int visited[], stack<int>& s,
             vector<pair<int, int>> adj[]) {
    visited[node] = 1;
    for (auto it : adj[node]) {
        if (!visited[it.first])
            topoSrt(it.first, visited, s, adj);
    }
    s.push(node);
}
```

```
void shortestPath(int src, int N, vector<pair<int, int>> adj[]) {
    int visited[N] = {0};
    stack<int> s;
    for (int i = 0; i < N; i++) {
        if (!visited[i])
            topoSrt(i, visited, s, adj);
    }
    int dist[N];
    for (int i = 0; i < N; i++) {
        dist[i] = 1e9;
    }
    dist[src] = 0;
```



```
while (!S.empty()) {
    int node = S.top();
    S.pop();
    if (dist[node] != INF) {
        for (auto it : adj[node]) {
            if (dist[node] + it.second < dist[it.first]) {
                dist[it.first] = dist[node] + it.second;
            }
        }
    }
    for (int i=0; i<N; i++) {
        (dist[i] == 1e9) ? cout << "INF" : cout << dist[i] << " ";
    }
}
```

```
int main() {
    int n, m;
    cin >> n >> m;
    vector<pair<int, int>> adj[n];
    for (int i=0; i<m; i++) {
        int u, v, wt;
        cin >> u >> v >> wt;
        adj[u].push_back({v, wt});
    }
}
```

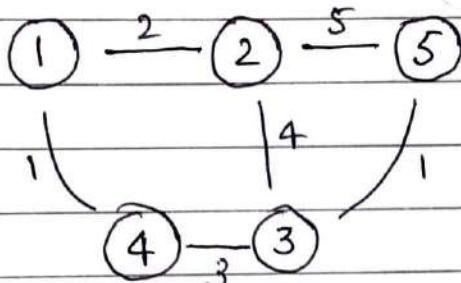
```
shortestPath(0, n, adj);
return 0;
```

$$T.C \rightarrow O((N+E)\log N) = N \log N$$

$$S.C \rightarrow O(N) + O(N)$$

Page no. _____
Date : / /

* Dijkstra's Algorithm (shortest path in Undirected 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)

$SRC = 1$ node weight

- (i) (1, 4)
- (iv) (5, 5)
- (ii) (2, 2)
- (iii) (4, 3)
- (v) (7, 5)
- (vi) (0, 1)
- (vii) (1, 1)

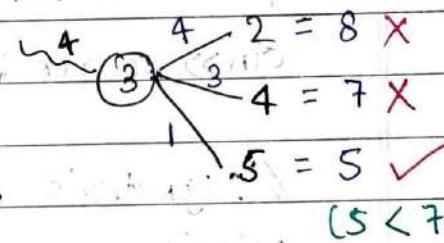
	0	2	4	1	5
	∞	∞	∞	∞	∞

distance []

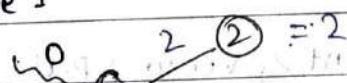
(iv) node 3

Priority-queue

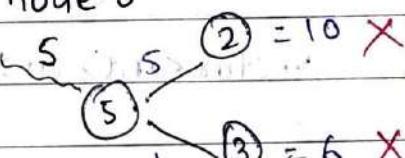
(min Heap) \Rightarrow Top se nikalenge



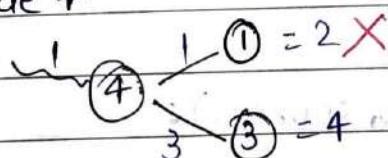
(i) node 1



(v) node 5

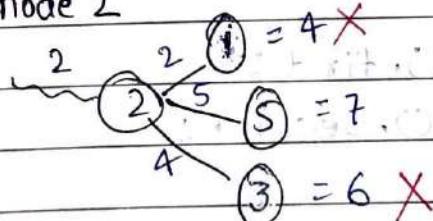


(ii) node 4



(vi) nodes 5

(iii) node 2



Program :-

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, m, source;
    cin >> n >> m;
    vector<pair<int, int>> g[n+1]; // i-index adj. list for graph.

    int a, b, wt;
    for (int i=0; i<m; i++) {
        cin >> a >> b >> wt;
        g[a].push_back(make_pair(b, wt));
        g[b].push_back(make_pair(a, wt));
    }

    cin >> source;

    // Dijkstra's Algorithm
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> pq;
    vector<int> distTo(n+1, INT_MAX); // i-index array

    distTo[source] = 0;
    pq.push(make_pair(0, source));

    while (!pq.empty()) {
        int dist = pq.top().first;
        int prev = pq.top().second;
        pq.pop();

        for (auto &adj : g[prev]) {
            if (dist + adj.second < distTo[adj.first]) {
                distTo[adj.first] = dist + adj.second;
                pq.push(make_pair(distTo[adj.first], adj.first));
            }
        }
    }
}
```

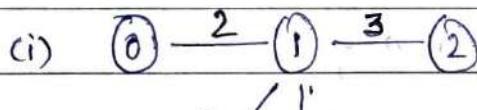
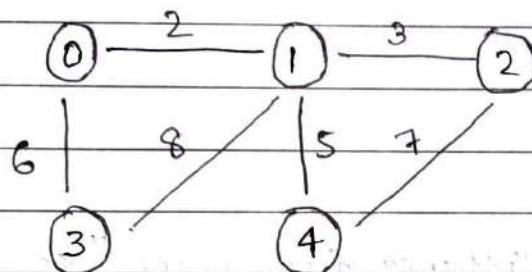
```
vector<pair<int,int>>::iterator it ;  
for (it = g[prev].begin(); it != g[prev].end(); it++) {  
    int next = it->first ;  
    int nextDist = it->second ;  
  
    if (distTo[next] > distTo[prev] + nextDist) {  
        distTo[next] = distTo[prev] + nextDist ;  
        pq.push (make_pair (distTo[next], next)) ;  
    }  
}
```

```
cout << "The distance from source, " << source << ", are: \n";  
for (int i=1; i<=n; i++) {  
    cout << distTo[i] << " ";  
    cout << "\n";  
}  
return 0;  
}
```

* Minimum Spanning Tree (MST)

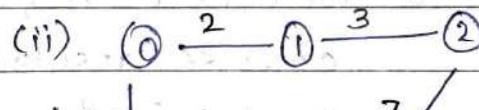
→ If we can draw a tree from a graph consisting of 'N' nodes and 'N-1' edges such that every node is reachable to every other node in that tree and the cost of the edge weight in that tree is minimal then we call that tree as minimum spanning tree.

e.g. →



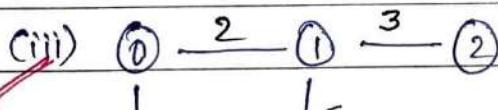
$$N=5, E=4$$

$$\text{Cost} = 18$$



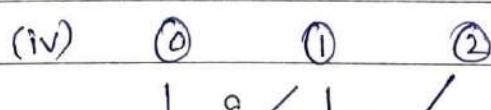
$$N=5, E=4$$

$$\text{Cost} = 18$$



$$N=5, E=4$$

$$\text{Cost} = 16$$



$$N=5, E=4$$

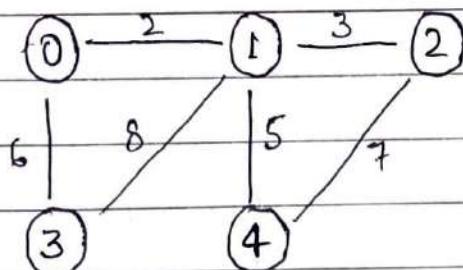
$$\text{Cost} = 26$$

MST





* PRIM'S ALGORITHM (MST)



	2	3	6	5
Key[]	0	∞	∞	∞
0 1 2 3 4	0	1	2	3

T	T	T	T	T
F	F	F	F	F
0 1 2 3 4	0	1	2	3

Parent[]	0	1	0	1
-1	X	X	X	X
0 1 2 3 4	0	1	2	3

→ choose \min^m Key[] from Key[], and mark 'T' in MST.

(i) node 0 → 2 ✓ where MST is 'F'

→ Mark their Parents.

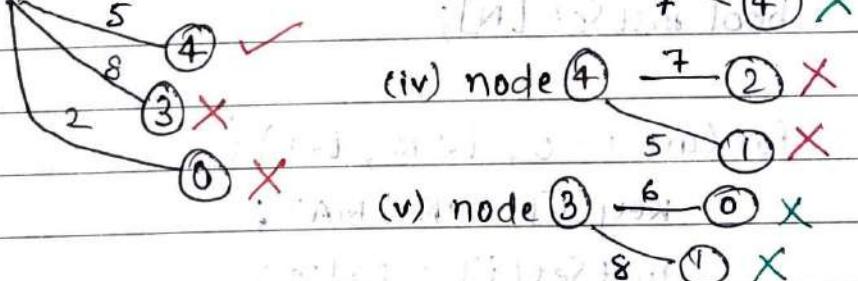
Again, Similar steps.

(ii) node 1 → 3 ✓

(iii) node 2 → 1 X

5
2
8
4
3
0 X

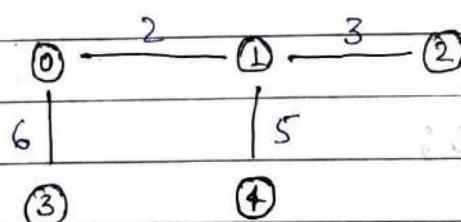
(iv) node 4 → 7 X



(v) node 3 → 6 X

8
1 X

→ Now, using Parent we create a tree



Approach 1: Brute-Force

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int N, m;
    cin >> N >> m;
    vector<pair<int, int>> adj[N];
    for (int i = 0; i < m; i++) {
        int a, b, wt;
        cin >> a >> b >> wt;
        adj[a].push_back(make_pair(b, wt));
        adj[b].push_back(make_pair(a, wt));
    }
}
```

Source for Approach 2.

```
int parent[N];
int key[N];
bool mstSet[N];
for (int i = 0; i < N; i++) {
    key[i] = INT_MAX;
    mstSet[i] = false;
}
```

```
key[0] = 0;
parent[0] = -1;
int ansWeight = 0;
```

```
for (int count = 0; count < N - 1; count++) {
```

```
    int mini = INT_MAX, u;
```

```
    for (int v = 0; v < N; v++) {
```

```
        if (mstSet[v] == false && key[v] < mini) {
```

```
            mini = key[v], u = v;
```

```
}
```

```
    mstSet[u] = true;
```

```
    for (auto it : adj[u]) {
```

```
        int v = it.first;
```

```
        int weight = it.second;
```

```
        if (mstSet[v] == false && weight < key[v]) {
```

```
            parent[v] = u;
```

```
            key[v] = weight;
```

```
}
```

```
}
```

```
for (int i = 1; i < N; i++) {
```

```
    cout << parent[i] << " -><" << i << "\n";
```

```
return 0;
```

same for
Approach 2.

Approach 2: (using Min-Heap)

In approach 1, we are going through the key-value again and again to find the minimum edge weight that is not part of the MST.

→ Min-Heap would contain the weight required to reach a node along its index.

Program :-

```
priority_queue<pair<int,int>, vector<pair<int,int>>,  
greater<pair<int,int>> >pq ;
```

```
key[0] = 0;
```

```
parent[0] = -1;
```

```
pq.push({0,0});
```

```
while (!pq.empty()) {
```

```
    int u = pq.top().second;
```

```
    pq.pop();
```

```
mstSet[u] = true;
```

```
for (auto it : adj[u]) {
```

```
    int v = it.first;
```

```
    int weight = it.second;
```

```
    if (mstSet[v] == false && weight < key[v]) {
```

```
        parent[v] = u;
```

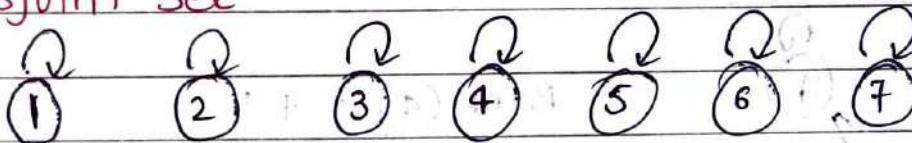
```
        key[v] = weight;
```

```
        pq.push({key[v], v});
```

```
}
```

```
}
```

* Disjoint Set

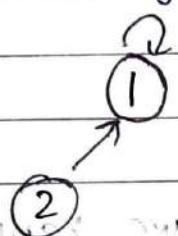


(i) Union (1,2)

1	2	3	4	5	6	7
∅	∅	0	∅	0	∅	0

Rank

Both have rank '0', So we can attach anyone to other.



Rank(1) = +1

Parent(2) = 1

(ii) Union (2,3)

findpar(2) \rightarrow 1 } so, we connect 1 and 3.

findPar(3) \rightarrow 3

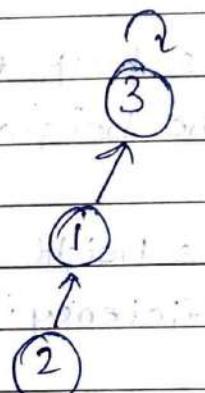
Rank=1

Rank=0

Rank Whoever has smaller rank, connect that to others.

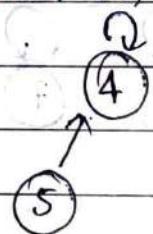
\rightarrow There is no need to increase rank, because both have diff. rank, If we get same rank the increase it (because, depth/height doesn't get increase in diff. rank)

NOTE: \rightarrow



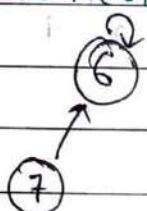
In order to reduce T.C., we are not doing like this, because if we connect higher rank to smaller rank, it will increase depth and hence, T.C.

(iii) Union (4, 5)



$$\text{Rank}(4) = +1$$

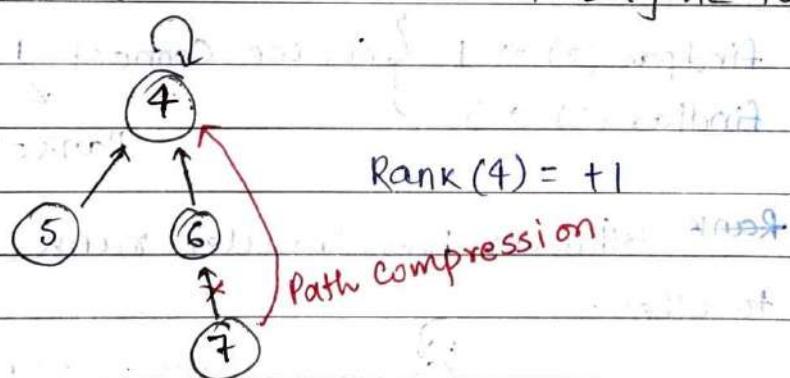
(iv) Union (6, 7)



$$\text{Rank}(6) \rightarrow +1$$

(v) Union (5, 6)

findPar(5) $\rightarrow 4$ } Both have rank 'L', so we
findPar(6) $\rightarrow 6$ connect anyone to other.



$$\text{Rank}(4) = +1$$

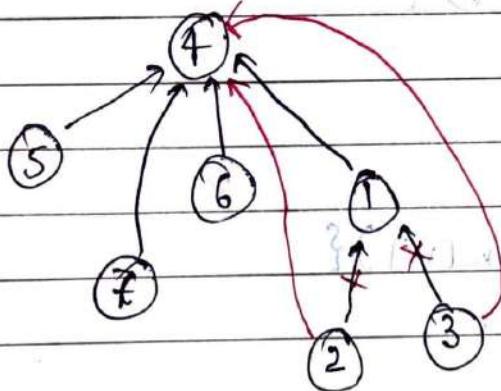
(vi) Union (3, 7)

findPar(3) $\rightarrow 1$ } Rank(1) $\rightarrow 1$
findPar(7) $\rightarrow 6 \rightarrow 4$ } Rank(4) $\rightarrow 2$

Path Compression :- It speeds up the data

structure by compressing the height of tree.

We need to minimize the height
in order to improve the efficiency.



T.C $\rightarrow O(4\alpha) \approx O(4)$, for every Union operation.
S.C $\rightarrow O(N)$, Rank array and Parent array.

Program :-

```
int parent[100000];  
int rank[100000];
```

```
void makeSet() {  
    for (int i = 1; i <= n; i++) {  
        parent[i] = i;  
        rank[i] = 0;  
    }  
}
```

```
int findPar(int node) {  
    if (node == parent[node]) {  
        return node;  
    }  
    // Path Compression.  
    return parent[node] = findPar(parent[node]);  
}
```

```

void union(int u, int v) {
    u = findPar(u);
    v = findPar(v);

    if (rank[u] < rank[v]) {
        parent[u] = v;
    } else if (rank[v] < rank[u]) {
        parent[v] = u;
    } else {
        parent[v] = u;
        rank[u]++;
    }
}

```

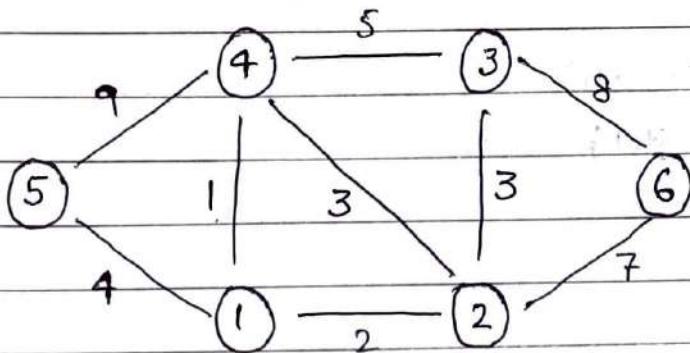
```

void main() {
    makeSet();
    int m;
    cin >> m;
    while (m--) {
        int u, v;
        union(u, v);
    }

    // if 2 and 3 belong to same component or not
    if (findPar(2) == findPar(3)) {
        cout << "Different Component";
    } else {
        cout << "same";
    }
}

```

* Kruskal Algorithm (MST)



Step 1: Sort all edges

w.r.t weight

wt, 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)

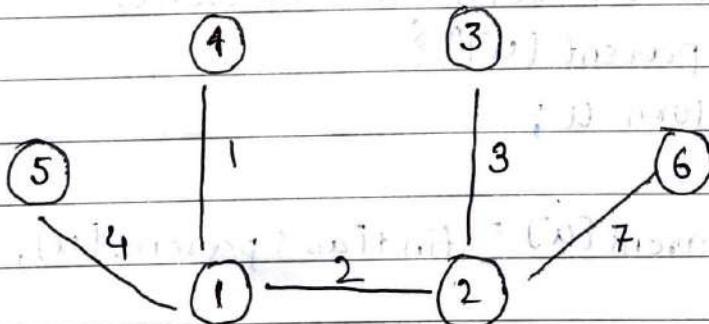
✗ (9, 4, 5)

✓ → We will take it in MST

because it belongs to different component.

✗ → We will not take it in MST

because it belongs to same component. (makes a cycle)





Program :-

```
#include <bits/stdc++.h>
using namespace std;

Struct node {
    int u;
    int v;
    int wt;
    node (int first, int second, int weight) {
        u = first;
        v = second;
        wt = weight;
    }
};

bool comp (node a, node b) {
    return a.wt < b.wt;
}

int findPar (int u, vector<int> &parent) {
    if (u == parent[u]) {
        return u;
    }
    return parent[u] = findPar (parent[u], parent);
}

void unionn (int u, int v, vector<int> &parent, vector<int> &rank) {
    u = findPar (u, parent);
    v = findPar (v, parent);
    if (rank[u] < rank[v])
        parent[u] = v;
}
```

else if ($\text{rank}[v] < \text{rank}[u]$)

$\text{parent}[v] = u;$

else {

 if ($\text{parent}[v] = u;$

$\text{rank}[u]++;$

}

}

int main() {

 int N, m;

 cin >> N >> m;

 vector<node> edges;

 for (int i=0; i<m; i++) {

 int u, v, wt;

 cin >> u >> v >> wt;

 edges.push_back(node(u,v,wt));

}

 sort(edges.begin(), edges.end(), comp);

 vector<int> parent(N);

 for (int i= 0; i<N ; i++) {

 parent[i] = i;

}

 vector<int> rank(N, 0);

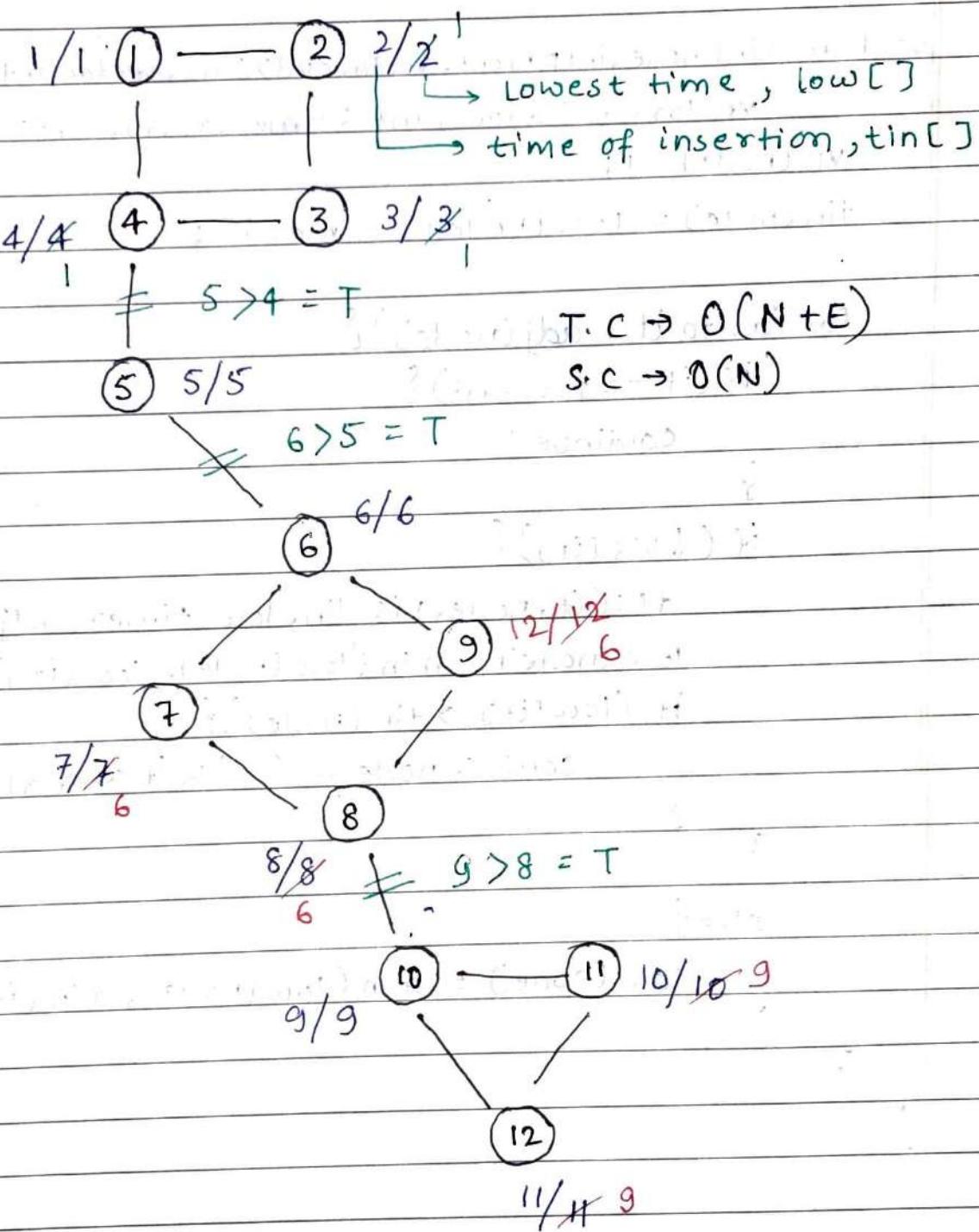
 int cost = 0;

 vector<pair<int, int>> mst;

```
for (auto it : edges) {  
    if (findPar(it.v, parent) != findPar(it.u, parent)) {  
        cost += it.wt;  
        mst.push_back({it.u, it.v});  
        unionn(it.u, it.v, parent, rank);  
    }  
}  
  
cout << cost << endl;  
for (auto it : mst) {  
    cout << it.first << "-" << it.second << endl;  
}  
return 0;
```

* Bridges in a graph

An edge in an undirected connected graph is a bridge if removing it disconnects the graph.

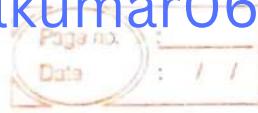


$\text{low}[i] > \text{tin}[\text{node}] \Rightarrow \text{bridge}$
 adjacent

Program :-

```
#include<bits/stdc++.h>
using namespace std;

void dfs(int node, int parent, vector<int>& vis, vector<int>& tin,
         vector<int>& low, int & timer, vector<int>& adj[]) {
    vis[node] = 1;
    tin[node] = low[node] = timer++;
    for (auto it : adj[node]) {
        if (it == parent) {
            continue;
        }
        if (!vis[it]) {
            dfs(it, node, vis, tin, low, timer, adj);
            low[node] = min(low[node], low[it]);
            if (low[it] > tin[node]) {
                cout << node << " " << it << endl;
            }
        } else {
            low[node] = min(low[node], tin[it]);
        }
    }
}
```



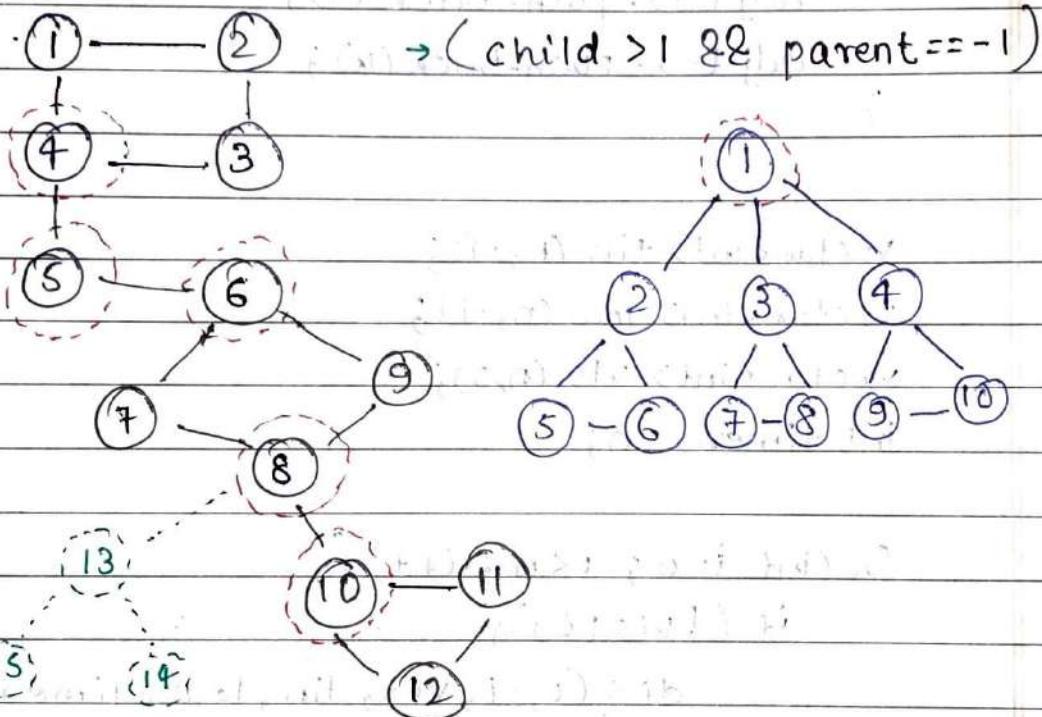
```
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> tin(n, -1);
    vector<int> low(n, -1);
    vector<int> vis(n, 0);
    int timer = 0;
    for (int i = 0; i < n; i++) {
        if (!vis[i]) {
            dfs(i, -1, vis, tin, low, timer, adj);
        }
    }
    return 0;
}
```

* Articulation Points (Cut Vertices)

A vertex in an undirected connected graph is an articulation point if removing it (and edges around it) disconnects the graph.

Condition : \rightarrow

$\rightarrow \text{low}[it] \geq \text{tin}[\text{node}] \text{ & parent} != -1$



NOTE : \rightarrow In this case, 8 will be articulation point for 10 as well as 13, so if we go on printing articulation point, 8 will be printed twice. Hence, instead of printing we will use Hashmap.

T.C $\rightarrow O(N+E)$

S.C $\rightarrow O(N)$

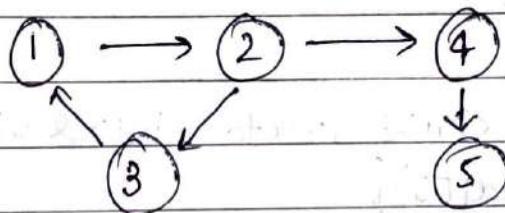
Program :-

```
#include<bits/stdc++.h>
using namespace std;
void dfs(int node, int parent, vector<int>& vis,
         vector<int>& tin, vector<int>& low, int & timer,
         vector<int> adj[], vector<int>& isArticulation) {
    vis[node] = 1;
    tin[node] = low[node] = timer++;
    int child = 0;
    for (auto it : adj[node]) {
        if (it == parent)
            continue;
        if (!vis[it]) {
            dfs(it, node, vis, tin, low, timer, adj, isArticulation);
            low[node] = min (low[node], low[it]);
            child++;
            if (low[it] >= tin[node] && parent != -1) {
                isArticulation[node] = 1;
            }
        } else {
            low[node] = min (low[node], tin[it]);
        }
    }
    if (parent == -1 && child > 1)
        isArticulation[node] = 1;
}
```

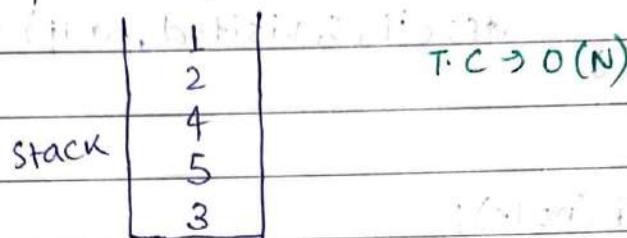
```
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];
    for (int i=0; i<m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> tin(n, -1);
    vector<int> low(n, -1);
    vector<int> vis(n, 0);
    vector<int> isArticulation(n, 0);
    int timer = 0;
    for (int i=0; i<n; i++) {
        if (!vis[i]) {
            dfs(i, -1, vis, tin, low, timer, adj, isArticulation);
        }
    }
    for (int i=0; i<n; i++) {
        if (isArticulation[i] == 1) {
            cout << i << endl;
        }
    }
    return 0;
}
```

* Kosaraju's Algorithm

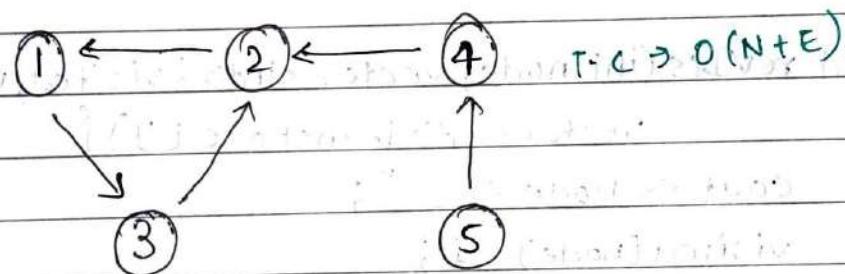
→ Strongly Connected Component



Step 1: → Sort all nodes in order of finishing time
(Topo Sort)



Step 2: → Transpose the graph



Step 3: → DFS according to finishing time (stack)

T.C \rightarrow O(N+E)

O/P = 1 3 2 4 5

4

5

T.C \rightarrow O(N+E)

S.C \rightarrow O(N+E) + O(N) + O(N)

↓ ↴ stack
visited array of dfs.

Program :-

```
#include <bits/stdc++.h>
using namespace std;

void dfs(int node, stack<int>&s, vector<int>& visited,
         vector<int> adj[]) {
    visited[node] = 1;
    for(auto it : adj[node]) {
        if(!visited[it]) {
            dfs(it, s, visited, adj);
        }
    }
    s.push(node);
}
```

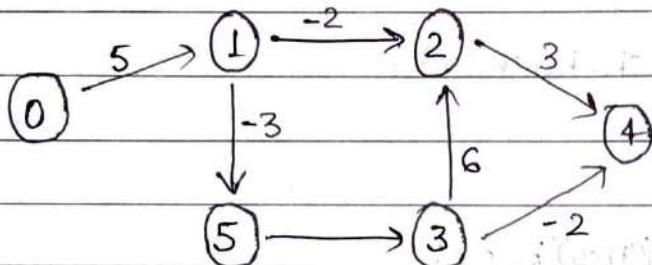
```
void revDfs(int node, vector<int>& visited,
            vector<int> transpose[]) {
    cout << node << " ";
    visited[node] = 1;
    for(auto it : transpose[node]) {
        if(!visited[it]) {
            revDfs(it, visited, transpose);
        }
    }
}
```

```
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];
```

```
for (int i=0 ; i<m ; i++) {  
    int u,v;  
    cin >> u >> v;  
    adj[u].push_back(v);  
}  
  
stack<int> s;  
vector<int> visited(n, 0);  
for (int i=0 ; i<n ; i++) {  
    if (!visited[i]) {  
        dfs(i, s, visited, adj);  
    }  
}  
  
vector<int> transpose[n];  
for (int i=0 ; i<n ; i++) {  
    visited[i] = 0;  
    for (auto it : adj[i]) {  
        transpose[it].push_back(i);  
    }  
}  
  
while (!s.empty()) {  
    int node = s.top();  
    s.pop();  
    if (!visited[node]) {  
        cout << "SCC: ";  
        revDfs(node, visited, transpose);  
        cout << endl;  
    }  
}  
return 0;  
}
```

* Bellman - Ford Algorithm

→ Shortest distance with negative edge.



Approach : → Relax all edges $(N-1)$ times.

If $(\text{dist}[u] + \text{wt} < \text{dist}[v])$

$$\text{dist}[v] = \text{dist}[u] + \text{wt}.$$

Relaxation : → Array,

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

$\text{dist}[3] + 6 < \text{dist}[2] \rightarrow \text{NO}$

$\text{dist}[5] + 1 < \text{dist}[3] \rightarrow \text{NO}$

$\text{dist}[0] + 5 < \text{dist}[1] \rightarrow \text{YES} \rightarrow \text{update array}$

$\text{dist}[1] - 3 < \text{dist}[5] \rightarrow \text{YES} \rightarrow$

$\text{dist}[1] - 2 < \text{dist}[2] \rightarrow \text{YES} \rightarrow$

$\text{dist}[3] - 2 < \text{dist}[4] \rightarrow \text{NO}$

$\text{dist}[2] + 3 < \text{dist}[4] \rightarrow \text{YES} \rightarrow \text{update array}$

After $N-1$, relaxation, :-

updated Array =

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

T.C $\rightarrow O(N * E)$

S.C $\rightarrow O(N)$

Program :-

```
#include<bits/stdc++.h>
using namespace std;
struct node{
    int u;
    int v;
    int wt;
    node(int first, int second, int weight){
        u = first;
        v = second;
        wt = weight;
    }
};

int main(){
    int N, m;
    cin >> N >> m;
    vector<node> edges;
    for (int i=0; i<m; i++){
        int u, v, wt;
        cin >> u >> v >> wt;
        edges.push_back(node(u, v, wt));
    }
    int src;
    cin >> src;
    int inf = 10000000;
    vector<int> dist(N, inf);
    dist[src] = 0;
```



```
for (int i=1; i<= N-1; i++) {  
    for (auto it : edges) {  
        if (dist[it.u] + it.wt < dist[it.v]) {  
            dist[it.v] = dist[it.u] + it.wt;  
        }  
    }  
}
```

```
int flag = 0; // for negative cycle (flag=1)  
for (auto it : edges) {  
    if (dist[it.u] + it.wt < dist[it.v]) {  
        cout << -1;  
        flag = 1;  
        break;  
    }  
}
```

```
if (!flag) {  
    for (int i=0; i< N; i++) {  
        cout << dist[i] << " ";  
    }  
}  
return 0;
```