Graph

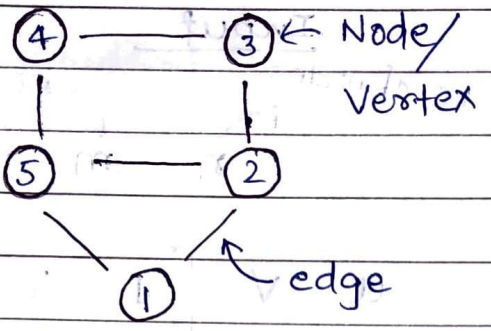
-By Nikhil Kumar

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

GRAPHS

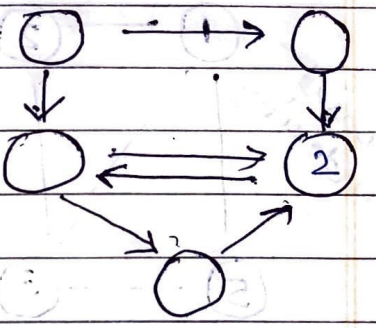
Page no. :
Date : / /

Undirected Graph



Degree (2) = 3

Directed Graph



Indegree (2) = 3

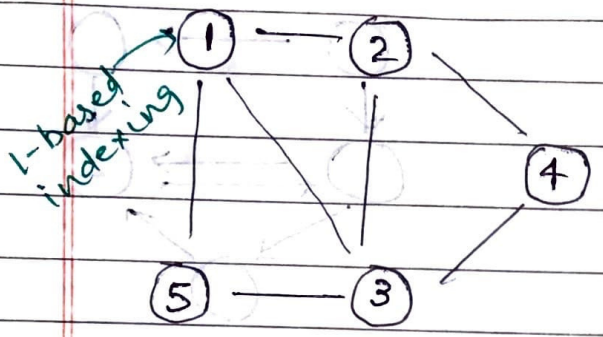
Outdegree (2) = 1

- Total no. of degrees of all the nodes = 2 x no. of edges.

Degree (1) = 2
Degree (2) = 3
Degree (3) = 2
Degree (4) = 2
Degree (5) = 3
12

No. of edges = 6.

* 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.

edges

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

adj [u][v] = 1

adj [v][u] = 1

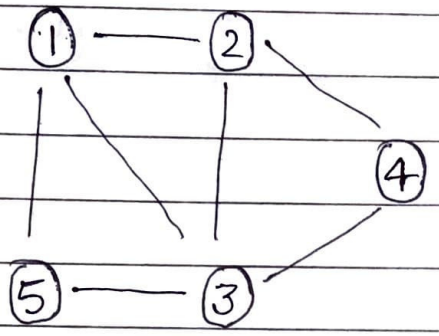
S.C = $O(n^2)$

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

7 edges

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

S.C → O(n + 2E)

↓ no. of nodes

↳ no. of edges.

Program :-

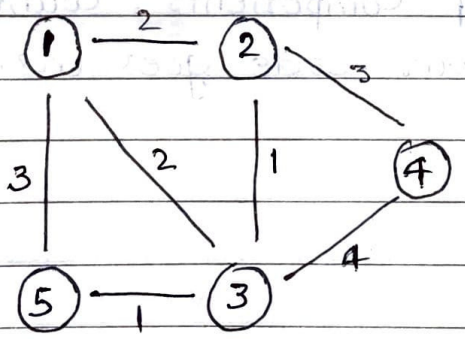
```
int main() {
    int n, m;        // m → no. of edges.
    cin >> n >> m;

    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);
    }
    return 0;
}
```

} in case of directed graph

S.C → O(n + E)

→ 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});
    }

    return 0;
}
```



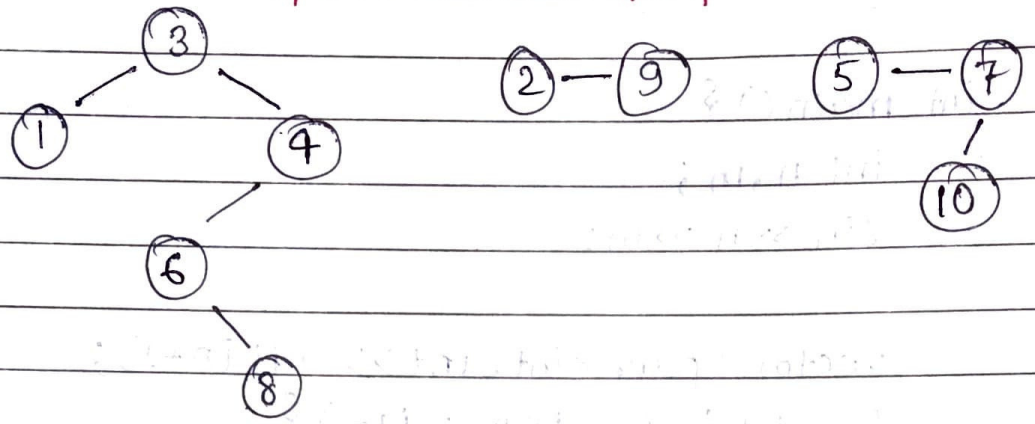
vector<pair<int, int>>adj[6];

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

node, wt.

S.C → O(n + 2E) + 2E

* Connected Components in Graph



1 2 3 4 5 6 7 8 9 10

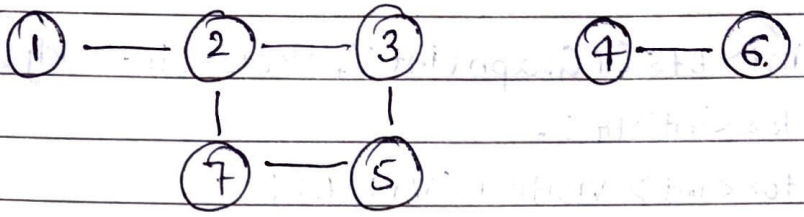
```
for (i=1; i<=10; i++){
    if (!vis[i]){
        dfs/bfs
    }
}
```

vis[i]

1	2	3	4	5	6	7	8	9	10
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.

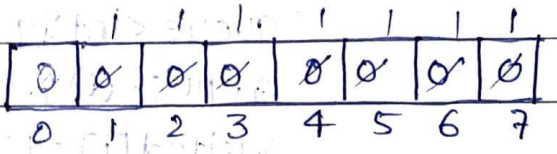
* BFS (Breadth - First Search)

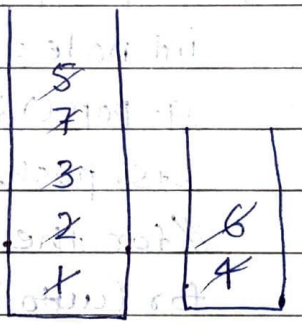


Adjacency list

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

visited





O/P = 1 2 3 7 5 4 6

```
for (i = 1 → n)
{
    if (visited[i] == 0)
        bfs(i);
};
```

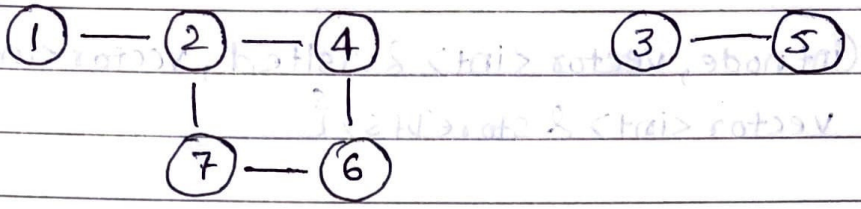
T.C → O(N+E), N is time taken for visiting N nodes and E is for travelling through adjacent nodes overall.

S.C → O(N+E) + O(N) + O(N)
adjacent list + queue + Array (visited)

Program :-

```
vector<int> bfsOfGraph(int V, vector<int> adj[]){
    vector<int> bfs;
    vector<int> visited(V+1, 0);
                    → for marking visited array 0 in start.

    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

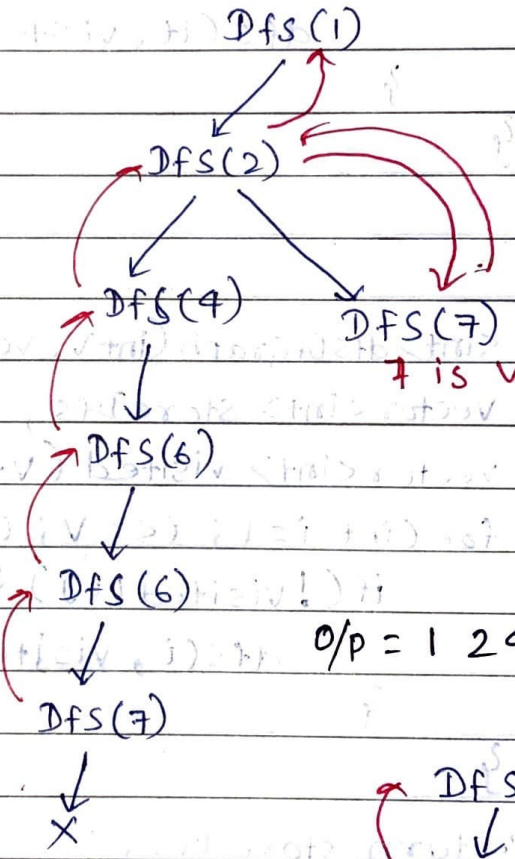


Visited.

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

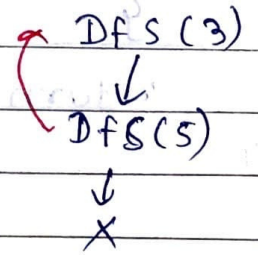
Adjacency list

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




```
for (i=1→7)
{
  if (!visited[i])
      dfs(i)
}
```

O/p = 1 2 4 6 7 3 5



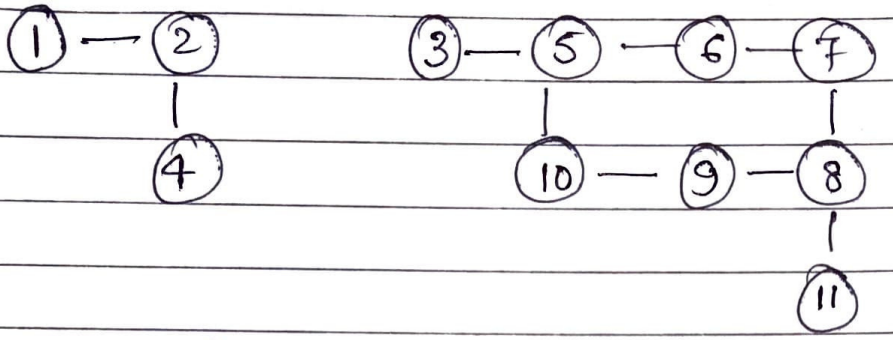
T.C → O(N+E)

S.C → O(N+E) + O(N) + O(N)

Program :-

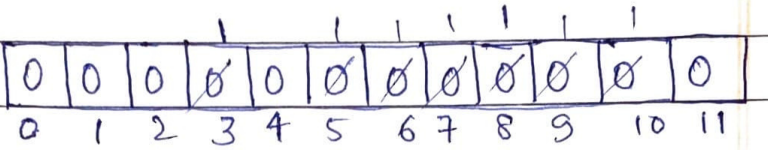
```
void dfs(int node, vector<int> &visited, vector<int> adj[],
         vector<int> &storeDfs){

    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
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



queue

<node, Parent>
<8,7>
<9,10>
~~<7,6>~~
~~<10,5>~~
~~<6,5>~~
~~<5,3>~~
~~<3,-1>~~

Jb <9,10> ka adj. list mein dekhe, toh 8 aur 10 hai, aur dono visited hai, but parent iska '10' hai, mtlb '8' kbi aur node se visited hua hai, mtlb cycle present hai.

```
for(i=1→N)
{
  if(!visited[i])
    if(cyclebfs(i))
      return true;
}
```

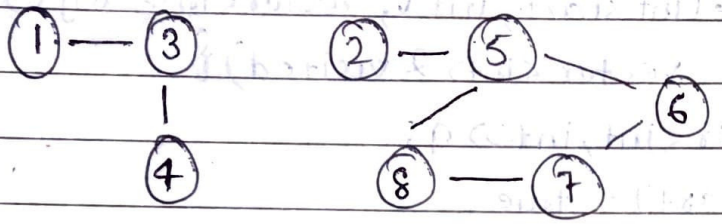
T.C → O(N+E)

S.C → 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;
        }
    }
    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, V, adj, visited))
                return true;
    }
    return false;
}
```

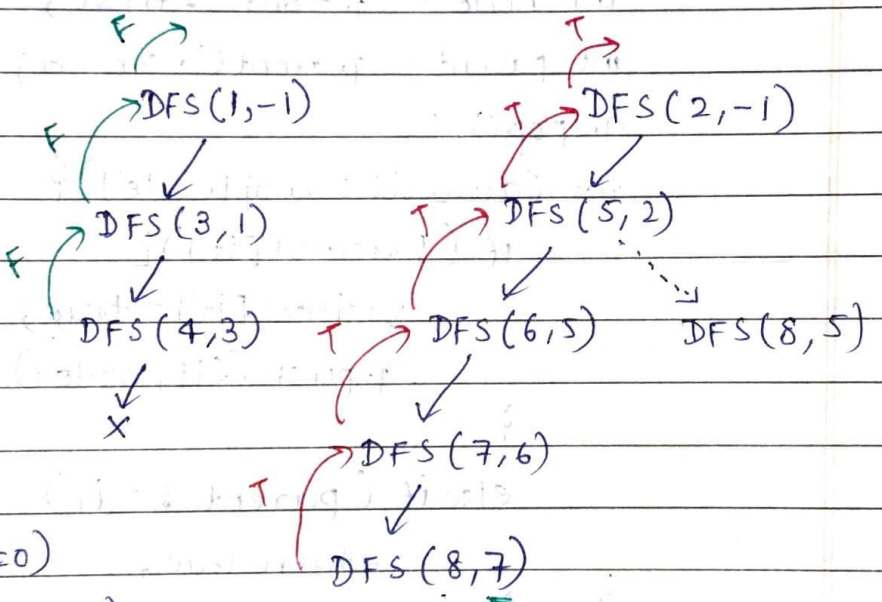
* Detect a Cycle in Undirected Graph (DFS)



adjacency list	
1	3
2	5
3	1, 4
4	3
5	2, 6, 8
6	5, 7
7	6, 8
8	7, 5

visited:

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




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

iske bad ('8') ka adj, 7 aur 5 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 :-

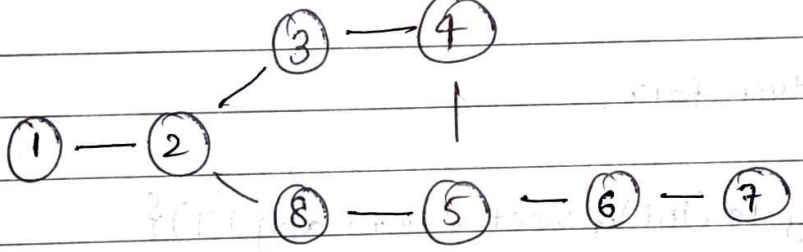
```
bool checkForCycle(int node, int parent, vector<int>& visited,
                   vector<int> adj[]) {
    visited[node] = 1;
    for (auto it : adj[node]) {
        if (visited[it] == 0) {
            if (checkForCycle(it, node, visited, adj))
                return true;
        }
        else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

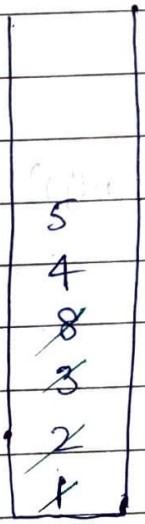
* Bipartite Graph (BFS)

→ 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.
=





color array =

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

~~node = 1~~, adj = 2.

~~node = 2~~, adj = 3, 8 } if -1, then it will be coloured with exact opposite of node '2'.

~~node = 3~~, adj = ~~2~~, 4
↳ already colored & opposite.

~~node = 8~~, adj = ~~2~~, 5

node = 4, adj = 3, 5

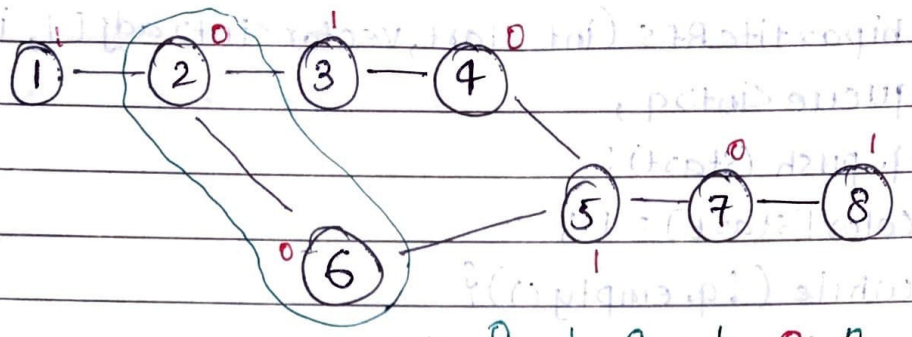
3 → already colored & opposite

5 → already colored but not opposite ⇓ False

Program :-

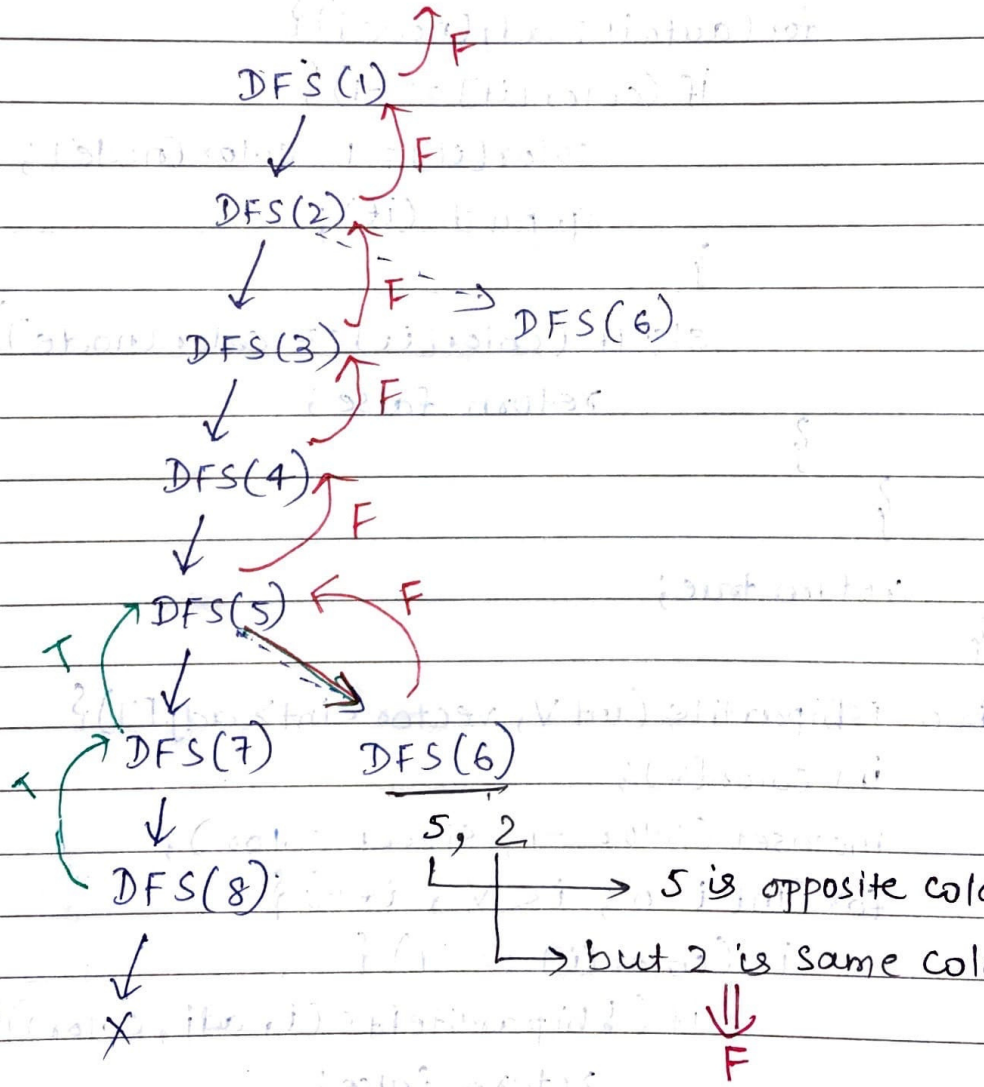
```
bool bipartiteBfs (int start, vector<int> adj [], int color[]){
    queue <int>q ;
    q. push (start);
    color[start] = 1 ;
    while (!q. empty ()){
        int node = q. front ();
        q. pop ();

        for (auto it : adj [node]){
            if (color [it] == -1){
                color[it] = 1 - color [node];
                q. push (it);
            }
            elseif (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



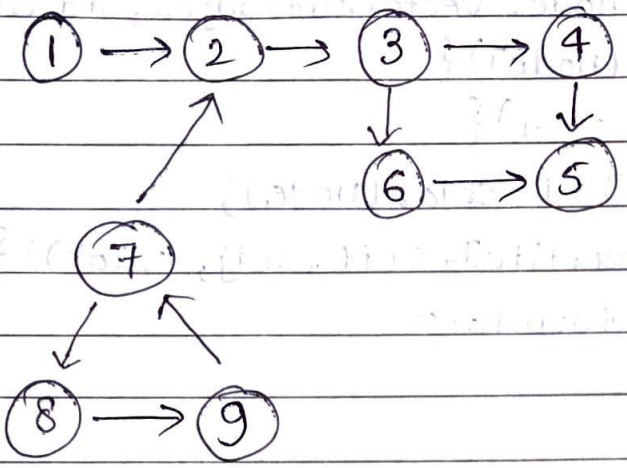
T.C → O(V+E)

S.C → O(V+E) + O(V) + O(V)

Program :-

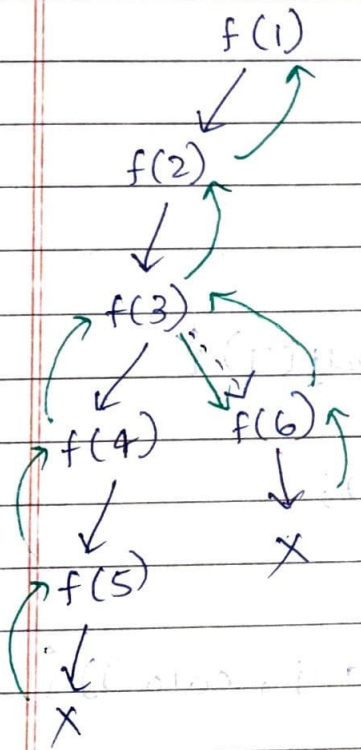
```
bool bipartiteDfs (int node, vector<int>adj[], int color[]){
    for (auto it : adj [node]){
        if (color [it] == -1){
            color [it] = 1 - color [node];
            if ( !bipartiteDfs (it, adj, color)) {
                return false
            }
        }
        else if (color [it] == 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 → 2
2 → 3
3 → 4, 6
4 → 5
5 →
6 → 5
7 → 2, 8
8 → 9
9 → 7

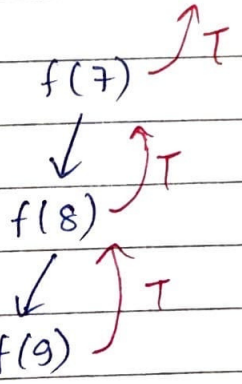


Visited

1	2	3	4	5	6	7	8	9
ø	ø	ø	ø	ø	ø	ø	ø	ø

DfsVisited

1	2	3	4	5	6	7	8	9
ø	ø	ø	ø	ø	ø	ø	ø	ø



T.C → O(V+E)
S.C → O(2V)

f(7) ⇒ Both visited and Dfsvisited is '1'

⇓

True

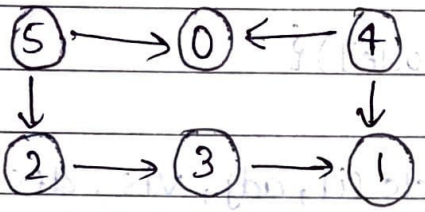
Program :-

```
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 $u \to v$, u appears before v in that ordering.



O/P = 5 4 2 3 1 0

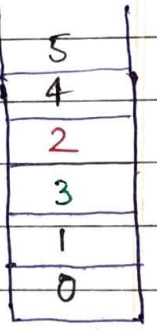
Adjacency list

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 0, 2

```
for(i=0 → 5)
{
  if(!vis[i])
  {
    dfs(i);
  }
}
```

visited

0	1	2	3	4	5
∅	∅	∅	∅	∅	∅



stack

visited[0] = 1

for i=0 :→ ~~dfs(0)~~ ⇒ No adj. nodes

visited[1] = 1

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

for i=1 :→ ~~dfs(1)~~

T.C → O(V+E)

S.C → O(N)+O(N)

Auxiliary S.C → O(N)

for i=2 :→ ~~dfs(2)~~

~~dfs(3)~~

⇒ already visited

for i=3 :→ already visited

for i=4 :→ ~~dfs(4)~~

for i=5 :→ ~~dfs(5)~~

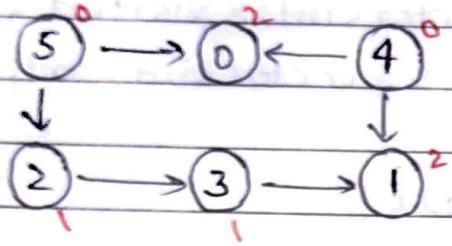
Program :-

```
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

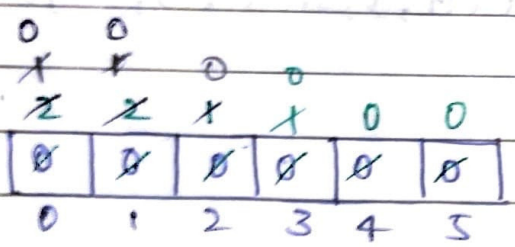
* Topological Sort (BFS)

(DAG)

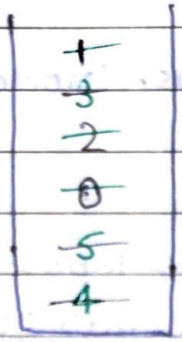


adjacency list

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 0, 2



Visited []



Queue

topo :→ 4 5 0 2 3 1

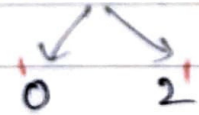
→ Count the indegree of nodes and replace them in visited array.

→ Put nodes with indegree '0' in queue.

node = 4

Reduce indegree of 0 & 1 by 1.

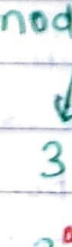
node = 5



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

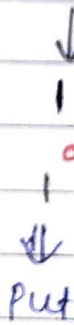
node = 0

node = 2



Put in queue

node = 3



Put in queue.

node = 1

T.C → O(N+E)

S.C → O(N) + O(N)

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 ⇒ Cycle doesn't exist

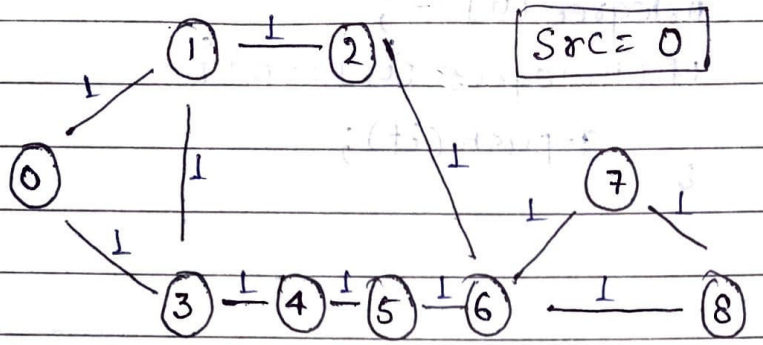
If Topological sort doesn't exist ⇒ Cycle exist

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



adjacency list

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

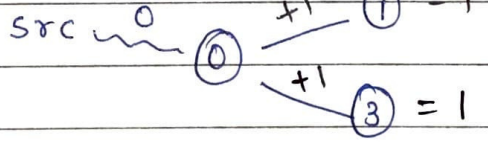
Assume, weight as 1

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

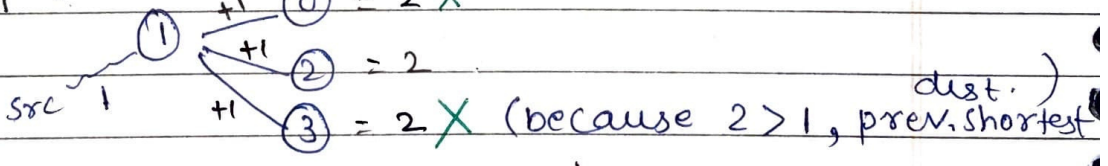
distance []

queue: 8, 7, 5, 6, 4, 2, 3, 1, 0

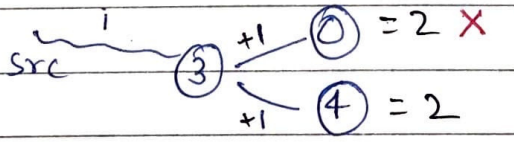
→ node = 0
dis = 0



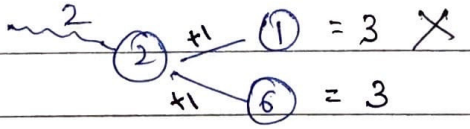
→ node = 1
dis = 1



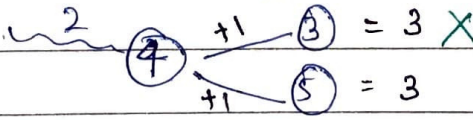
→ node = 3
dis = 1



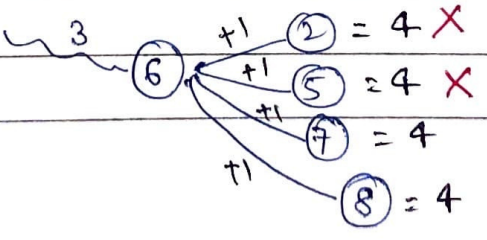
→ node = 2
dis = 2



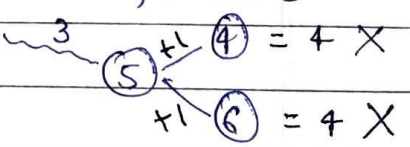
→ node = 4
dis = 2



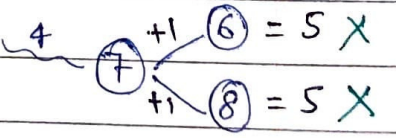
→ node = 6
dis = 3



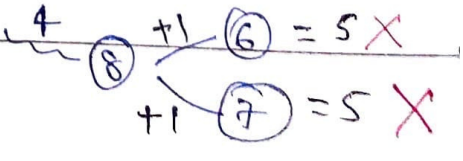
→ node = 5, dis = 3



→ node = 7, dis = 4



→ node = 8, dis = 4



T.C → O(N+E)

S.C → O(N) + O(N)

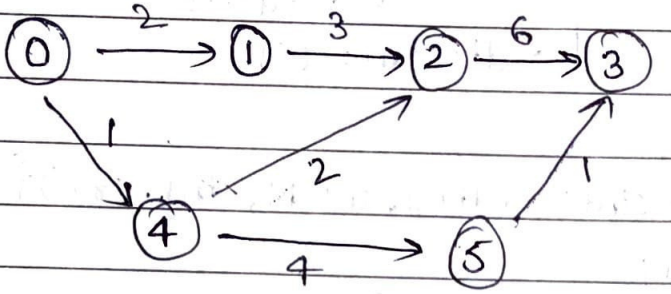
O(N) → distance []

O(N) → queue

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] << " ";
    }
}
```

* Shortest Path in Weighted Directed Acyclic Graph (DAG)



Src = 0

Adjacency list

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

node, weight

Step 1 :→ Find topo sort ⇒

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

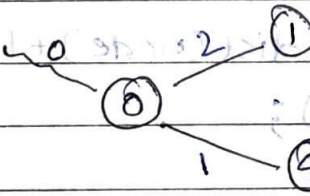
stack

Step 2 :→

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

check, if (dis[node] != ∞), for every node before entering.

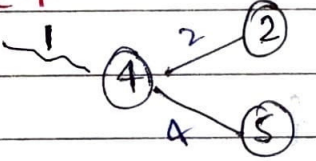
(i) node 0



= 2

= 1

(ii) node 4



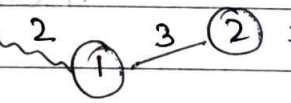
= 3

= 5

(iii) node 5

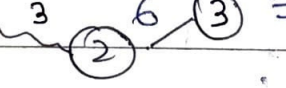
= 6

(iv) node 1



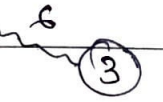
= 5 ✗ (5>3)

(v) node 2



= 9 ✗ (9>6)

(vi) node 3



Program :-

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

void topoSort(int node, int visited[], stack<int> & s,
              vector<pair<int, int>> adj[]){
    visited[node] = 1;
    for(auto it : adj[node]){
        if(! visited[it.first])
            topoSort(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]){
            topoSort(i, visited, s, adj);
        }
    }
    int dist[N];
    for(int i=0; i<N; i++){
        dist[i] = 1e9;
    }
    dist[src] = 0;
```

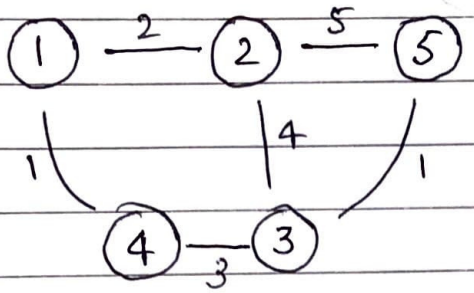
```
    while(!s.empty()){
        int node = s.top();
        s.pop();
        if(dist[node] != INF){
            for(auto it : adj[node]){
                if(dist[node] + it.second < dist[it.first]){
                    dist[it.first] = dist[node] + it.second;
                }
            }
        }
    }
    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 → $O((N+E)\log N) = N \log N$

S.C → $O(N) + O(N)$

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)

node ↑ weight

Src = 1

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

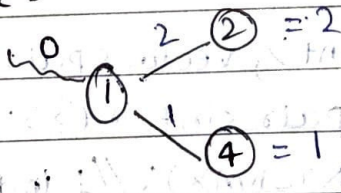
(dist, node)

Priority_queue (min Heap) ⇒ Top se nikalenge

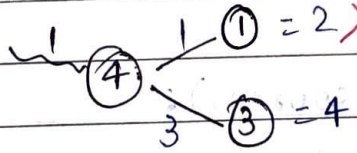
0	1	2	3	4	5
~~0~~	~~2~~	~~4~~	~~1~~	~~7~~ 5	
~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~

distance []

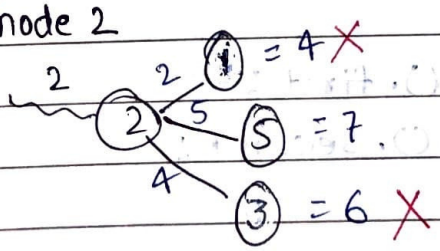
(i) → node 1



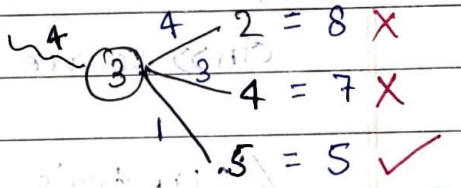
(ii) → node 4



(iii) → node 2

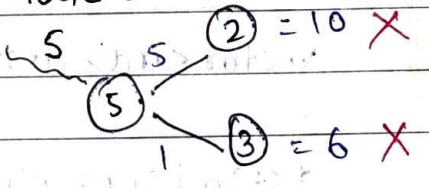


(iv) node 3

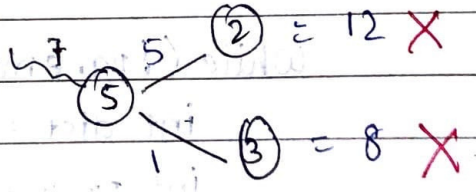


(5 < 7)

(v) node 5



(vi) node 5



Program :→

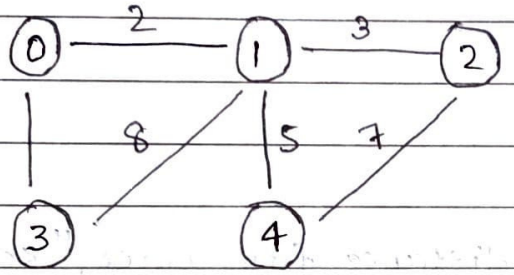
```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n,m, source;
    cin >> n >>m;
    vector<pair<int, int> > g[n+1]; //1-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); //1-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();
```

```
        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;
}
```

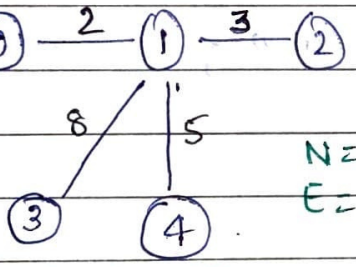
* Minimum Spanning Tree (MST)

⇒ 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. →



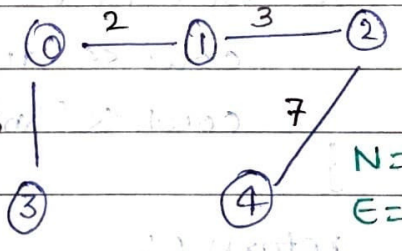
(i)



N=5, E=4

Cost = 18

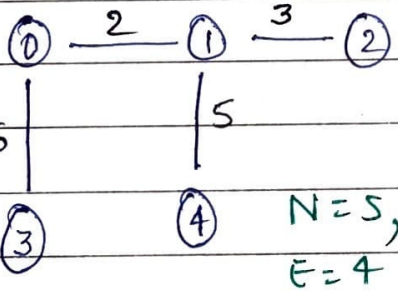
(ii)



N=5, E=4

Cost = 18

(iii)

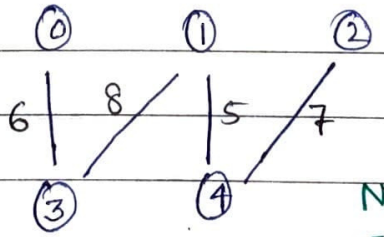


MST.

N=5, E=4

Cost = 16

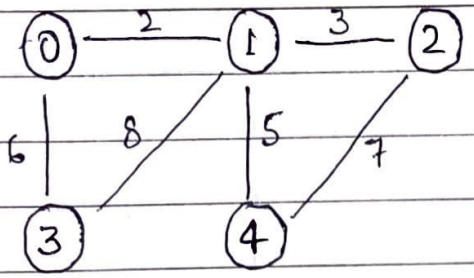
(iv)



N=5, E=4

Cost = 26.

* PRIM'S ALGORITHM (MST)



Key[]

0	1	2	3	4
0	∞	∞	∞	∞
	2	3	6	5

MST[]

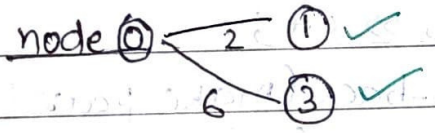
0	1	2	3	4
F	F	F	F	F
T	T	T	T	T

Parent[]

0	1	2	3	4
-1	✗	✗	✗	✗
	0	1	0	1

→ choose min^m Key[] (node of) from Key[] and mark 'T' in MST. where MST is 'F'

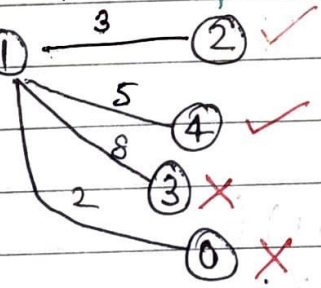
(i) node 0



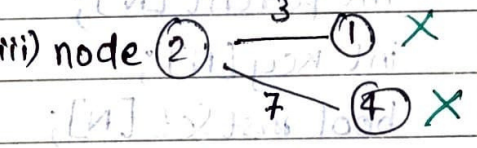
→ Mark their Parents.

Again, Similar steps.

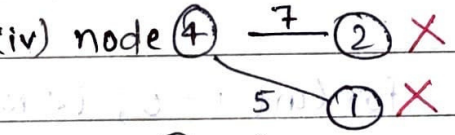
(ii) node 1



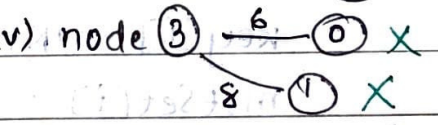
(iii) node 2



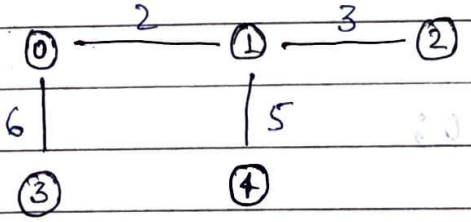
(iv) node 4



(v) node 3



→ 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];

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

    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;
```

Same for Approach 2.

```
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;
        }
    }
}
```

Same for Approach 2:

```
for (int i=1; i<N; i++) {
    cout << parent[i] << " - " << i << "\n";
    return 0;
}
}
```

Approach 2 :→ (using Min-Heap)

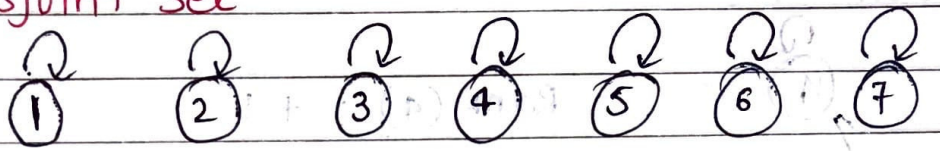
In approach 1, we are going through the key-value again and again to find the minimum edge weight that is not part of the MST.

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

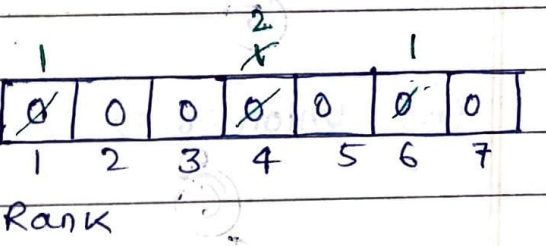
Program :→

```
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});
        }
    }
}
```

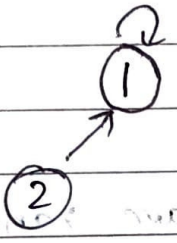
* Disjoint Set



(i) Union (1,2)



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



Rank (1) = +1

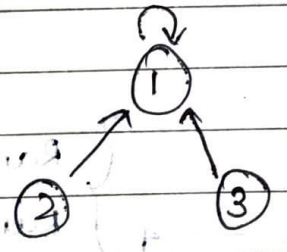
Parent (2) = 1

(ii) Union (2,3)

findpar (2) → 1
findPar (3) → 3
} so, we Connect 1 and 3.

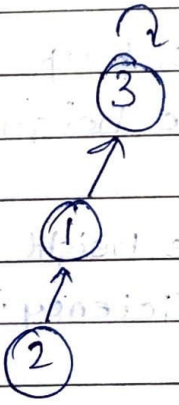
Rank = 1 Rank = 0

~~Rank~~ Whoever has smaller rank, connect that to other.



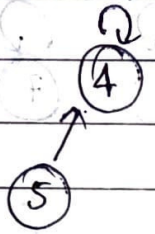
→ There is no need to increase rank, because both have diff. rank, If we get same rank the increase it. (because, depth/height doesn't get increase in diff. rank)

NOTE :→



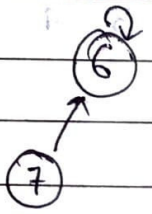
Inorder to reduce T.C, we are not doing like this, because if we connect higher rank to smaller rank, it will increase depth and hence, T.C.

(iii) Union (4, 5)



Rank (4) = +1

(iv) Union (6, 7)

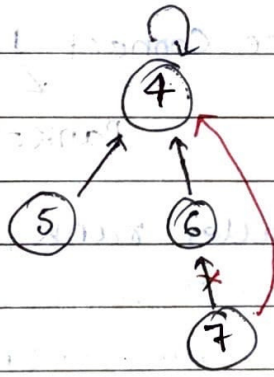


Rank (6) → +1

(v) Union (5, 6)

findPar (5) → 4
findPar (6) → 6

} Both have rank '1', so we connect anyone to other.



Rank (4) = +1

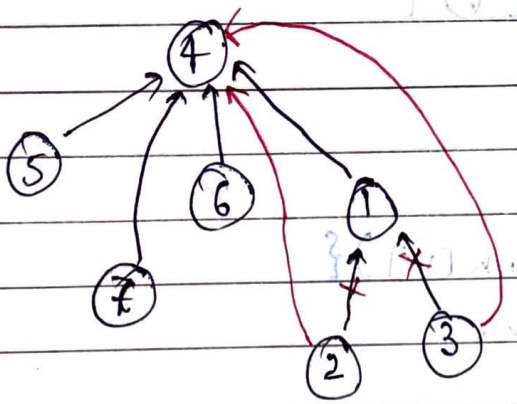
(vi) Union (3, 7)

findPar (3) → 3
findPar (7) → 6 → 4

} Rank (3) → 1
Rank (4) → 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.



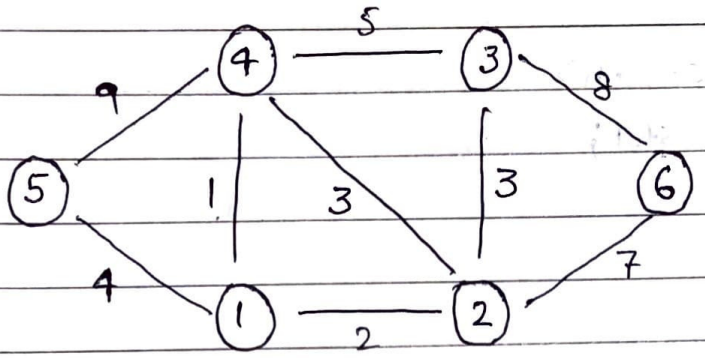
T.C → $O(4\alpha) \simeq O(4)$, for every Union operation.
S.C → $O(N)$, Rank array and Parent array.

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)



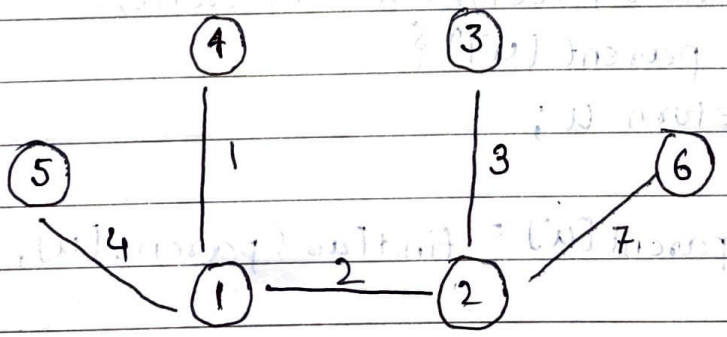
Step 1 :→ Sort all edges w.r.t weight.

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 :-

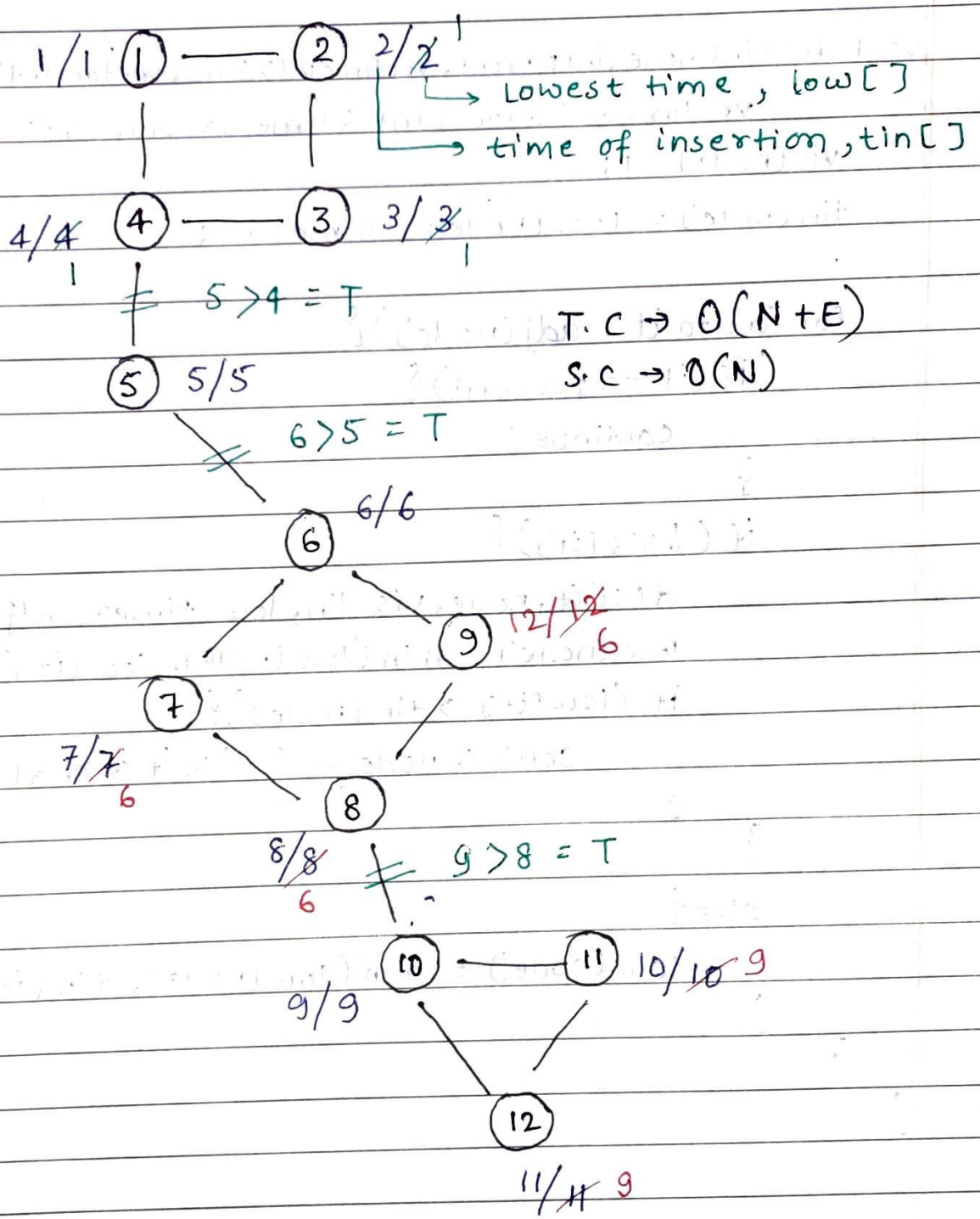
```
#include <bits/stdc++.h>
using namespace std;
Struct node {
    int u;
    int v;
    int wt;
    node (int first, int second, int weight){
        u = first;
        v = second;
        wt = weight;
    }
};

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 union(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 (rank[v] < rank[u])
        parent[v] = u;
    else {
        parent[v] = u;
        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.



Low [it] > tin [node] ⇒ 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]);
        }
    }
}
```

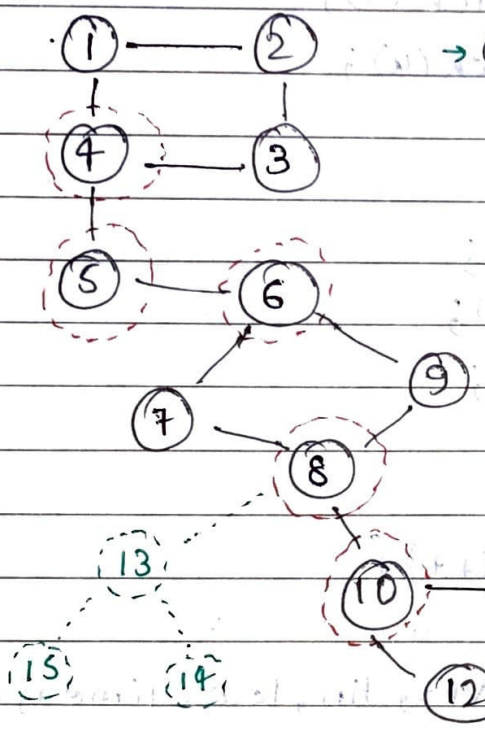
```
int main() {
    int n,m ;
    cin >> n >> m ;
    vector <int> adj [n];
    for (int i=0 ; i<m ; i++) {
        int u,v ;
        cin >> u >> v ;
        adj [u]. push-back (v);
        adj [v]. push-back (u) ;
    }

    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)

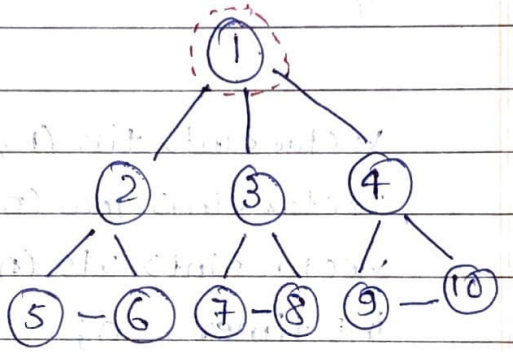
A vertex in an undirected connected graph is an articulation point if removing it (and edges around it) disconnects the graph.

Condition :→

→ low[it] >= tin[node] && parent != -1



→ (child > 1 && parent == -1)



NOTE :→ 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 → O(N+E)

S.C → O(N)

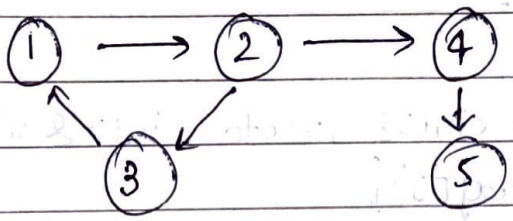
Program :->

```
#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;
    }
}
```

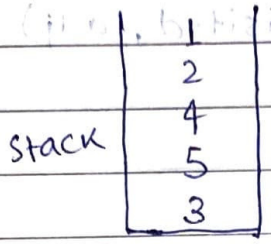
```
int main() {
    int n,m;
    cin >>n >> m;
    vector <int> adj[n];
    for (int i=0; i<m ; i++) {
        int u,v;
        cin >> u >> v;
        adj[u]. push_back (v);
        adj[v]. push_back(u);
    }

    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

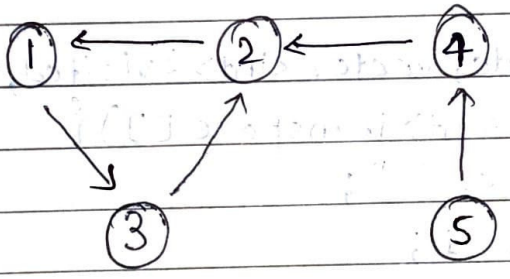


Step 1 :→ Sort all nodes in order of finishing time. (Topo Sort)



T.C → O(N)

Step 2 :→ Transpose the graph



T.C → O(N+E)

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

T.C → O(N+E)

O/P = 1 3 2
4
5

T.C → O(N+E)

S.C → O(N+E) + O(N) + O(N)

↳ stack

↓

visited array of dfs.

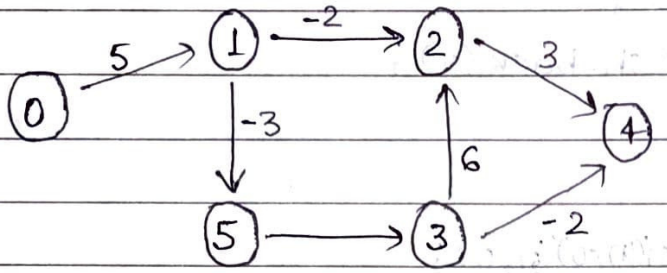
Program :->

```
#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 (dist[u] + wt < dist[v])
    dist[v] = dist[u] + wt.
```

Relaxation 1 :→ Array,

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

dist[3] + 6 < dist[2] → NO
dist[5] + 1 < dist[3] → NO
dist[0] + 5 < dist[1] → YES → update array
dist[1] - 3 < dist[5] → YES → "
dist[1] - 2 < dist[2] → YES → "
dist[3] - 2 < dist[4] → NO
dist[2] + 3 < dist[4] → YES → update array.

After N-1, relaxation, :-

updated Array =

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

T.C → O(N*E)

S.C → 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;
}
```