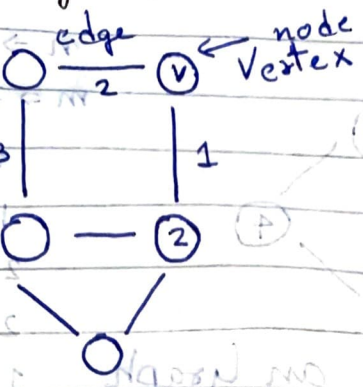
lyapth

~ Arbind

Graph



If edge have a graph it a weight graph.

Undirected Cyclic graph

u v
v u

Degree (2) = 3

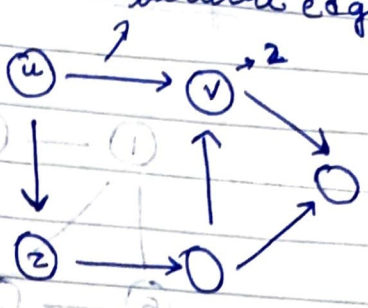
Total degree of all the nodes = 2 * edges

Path: 1 2 3
(can't visit the same node twice)

If we traverse from 1 we can again reach back 1 then its a cyclic graph

⇒ If weight of edge is not given consider '1'

directed edges



Directed graph

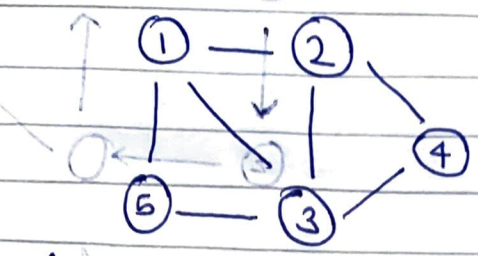
u v

In degree (2) = 2
Out degree (2) = 1

⇒ There can be path from u to z but not z to u

⇒ If we start from u, v, z we can't reach back the node again so its acyclic graph

Graph representation in C++



$n \Rightarrow 5$ (nodes)
$m \Rightarrow 7$ (edges)

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

2 Ways to store an Graph

SC → $O(n^2)$ (can use when value of n is smaller)

[•] Adjacency Matrix for

⇒ mark every thing with '0' first

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

Code:

```
int main() {
    int n, m;
    cin >> n >> m;
    int adj[n+1][n+1];
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adj[v][u] = 1;
    }
}
```

[•] Adjacency List

N = 5 → pair<int, int> → in case of weights

vector <int> adj [6]

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

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

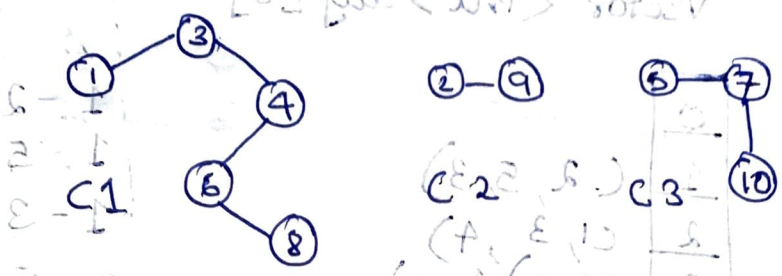
} 7 edges

SC → O(N+2E)

Code:

```
int n, m;
cin >> n >> m;
                    → pair[wt, wt] (in case of weights)
vector <int> adj [n+1];
for(int i=0 ; i < m ; i++){
    int u, v;   w
    cin >> u >> v; >> w
    adj[u] .push_back (v) ; .push_back({v,w})
    adj [v] . push_back (u) ; .push_back({u,w})
}
return 0;
```

Connected Components in a Graph



Disconnected graph with 3 diff. component.

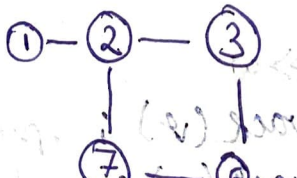
(1)

↓

even a single node can be called graph

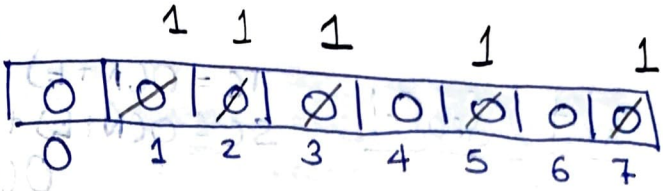
Breath - First Search

(Since graph can have multiple component we can run a for loop)

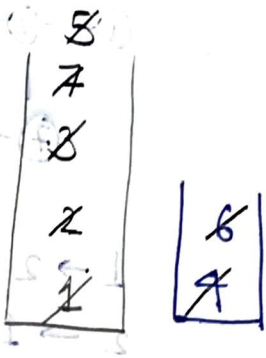


1	2
2	1 3 7
3	2 5
4	6
5	3 7
6	4
7	2 5

1 is not inserted as vis[1]=1;




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



$TC \rightarrow O(N+E)$

$SC \rightarrow O(N) + O(N) + O(N+E)$

⇒ N → time taken for visiting N nodes

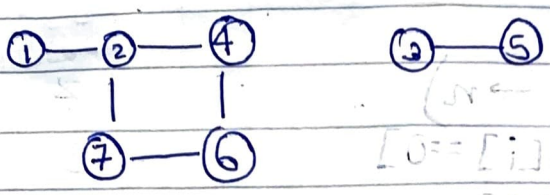
⇒ E → time taken for travelling through adjacent Node

Code

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

Depth First Search

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



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

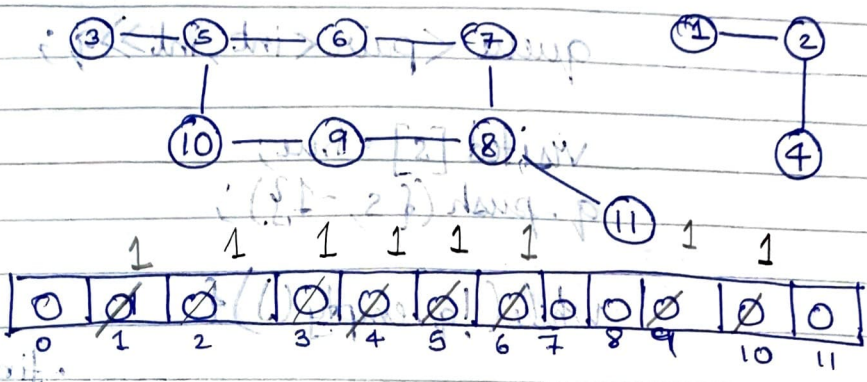
Code:

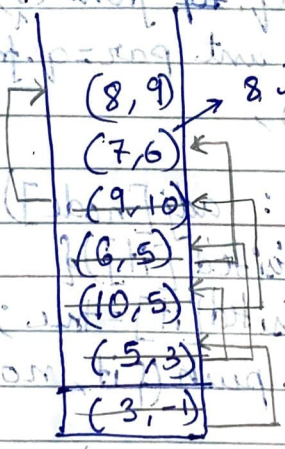
```
void dfs (int node, vector<int> vis, vector<int> adj[],
vector <int> ans){
    ans.push_back(node);
    vis[node] = 1;
    for(auto it : adj[node]){
        if (! vis[it]) dfs(it, vis, adj, ans);
    }
}

vector <int> dfsOfGraph(int V, vector <int> adj[]){
    vector <int> ans;
    vector <int> vis (V+1, 0);
    for (int i = 1; i <= V ; i++){
        if(! vis[i]){
            dfs(i, vis, adj, ans);
        }
    }
    return ans;
}
```

$TC = O(N+E)$

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

Detect a cycle in undirected graph (BFS).





8 is already visited & is not a parent

adj list

1	2
2	1 4
3	5
4	2
5	3 10 6
7	6 8
8	7 9 11
9	10 8
10	5 9
11	8

Code :

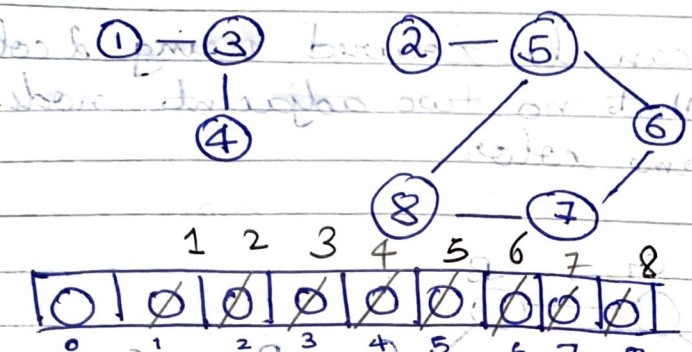
```
bool isCycle(int V, vector<int> adj[]){
    vector<int> vis(V+1, 0);
    for(int i=1; i<=V; i++){
        if(!vis[i])
            if(checkforcycle(i, V, adj, vis))
                return true;
    }
    return false;
```

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

    queue<pair<int,int>> q;

    visited[s] = true;
    q.push({s, -1});

    while(!q.empty()){

        int node = q.front().first;
        int par = q.front().second;
        q.pop();

        for(auto it: adj[node]){
            if(!visited[it]){
                visited[it] = true;
                q.push({it, node});
            }
            else if(par != it)
                return true;
        }
    }
    return false;
}
```

TC = O(N+E)

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

Cycle Detection in Undirected Graph



adj list	
1	3
2	5
3	1 4
4	3
5	2 6 8
6	5 7
7	6 8
8	7 5

DFS(2,-1)
|
DFS(5,2)
|
DFS(6,5)
|
DFS(7,6)
|
DFS(8,7)

for(i = 1 → 8)
{
if(!vis[i])
{
if(cycle DFS(i))
return true;
}
}

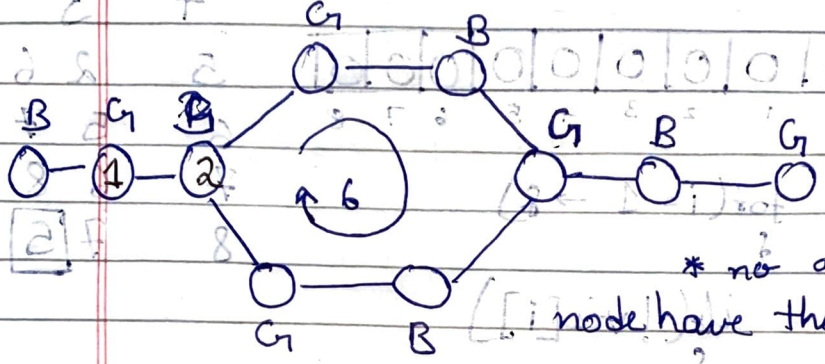
DFS(i, -i)
node parent

Code:

```
bool checkCycle(int node, int parent, vector<int> &vis, vector<int> adj[]){
    vis[node] = 1;
    for(auto it : adj[node]){
        if(!vis[i])
            if(checkCycle(it, node, vis, adj)) return true;
        else if(it != parent)
            return true;
    }
    return false;
}
public:
bool isCycle(int V, vector<int> adj[]){
    vector<int> vis(V+1, 0)
    for(int i = 1 ; i <= V ; i++)
        if(!vis[i])
            if(checkCycle(i, -1, vis, adj))
                return true;
returns true;
```

Bipartite Graph (BFS)

⇒ That can be colored using 2 colors such that no two adjacent nodes have same color



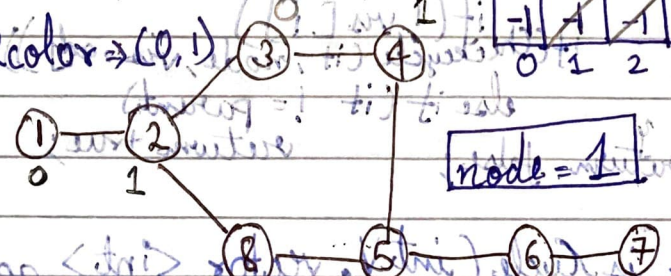
* no adjacent node have the same color

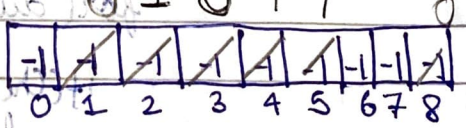
⇒ Any graph with even length cycle is Bipartite Graph (or) no cycle

[Graph without odd length cycle]

⇒ We are considering single component graph but it can be implemented for multiple also.

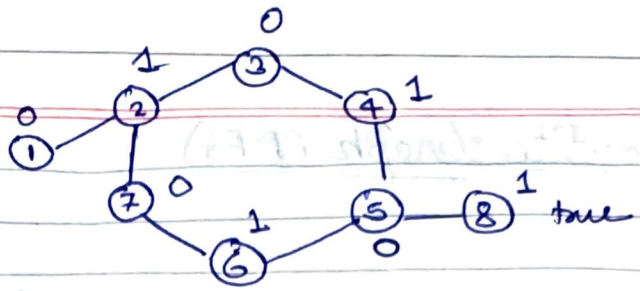
2color ⇒ (0,1)





node = 1

Queue



Code:

```
bool bipartiteBFS(int src, vector<int> adj[], int color[]) {
    queue<int> q;
    q.push(src);
    color[src] = 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 checkBipartite(vector<int> adj[], int n)
{
    int color[n];
    memset(color, -1, sizeof color);
    for(int i=0; i<n; i++) {
        if(color[i] == -1)
            if(!bipartiteBFS(i, adj, color))
                return false;
    }
    return true;
}
```

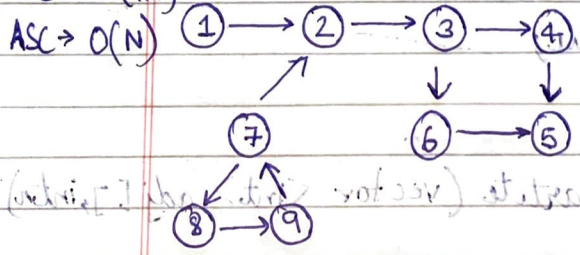
Bipartite Graph (DFS)

Code

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

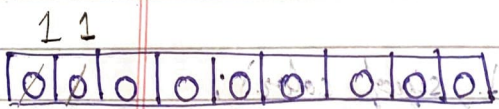
TC → O(N+E)
SC → O(2N)
ASC → O(N)

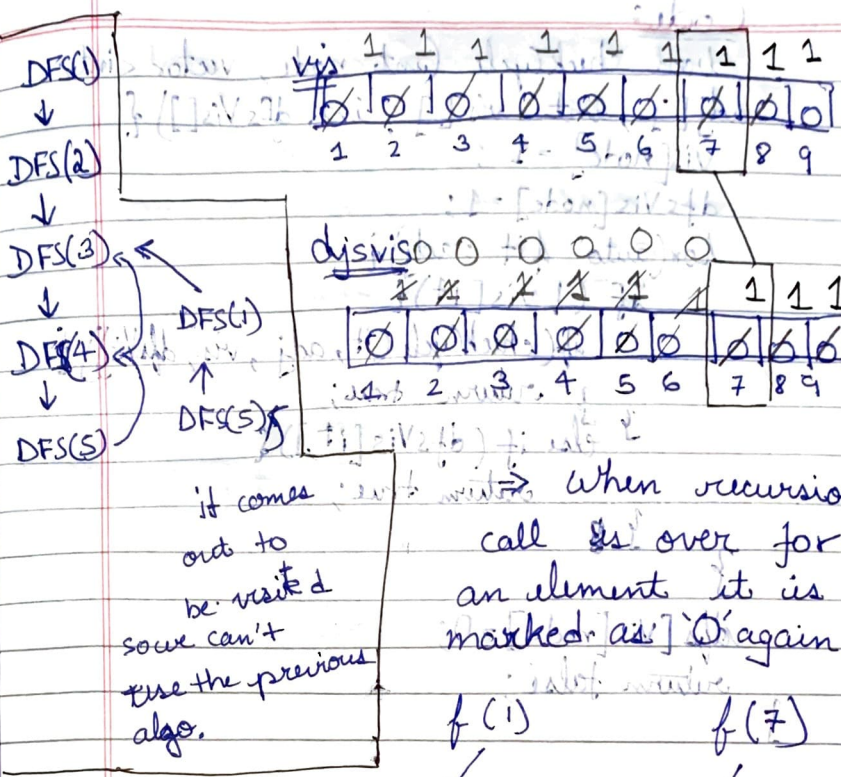
Detect a cycle in Directed Graph DFS



adjacent list

1 → 2
2 → 3
3 → 4 6
4 → 5
5 →
6 → 5
7 → 2 8
8 → 9
9 → 7





⇒ If the adj node is not a parent & is marked as 1 in both the stores then the graph can have a cycle within it.

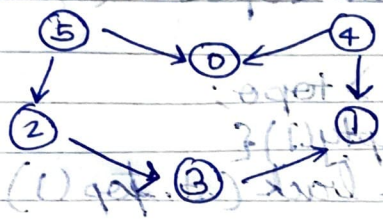
Code:

```
bool checkCycle (int node, vector<int> adj[], int vis[], int dfsVis[]) {
    vis[node] = 1;
    dfsVis[node] = 1;
    for(auto &it : adj){
        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[node] = 1;
    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;
}
```

TC → O(N+E)
SC → O(N) + O(N)
ASC → O(N)

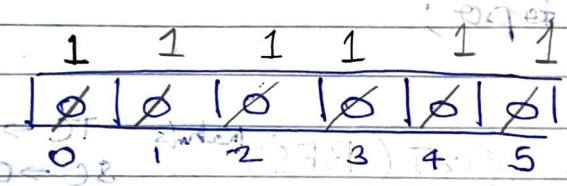
Topological Sorting (DFS)

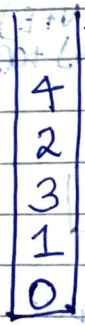
Linear ordering of vertices such that if there is an edge $u \to v$, u appears before v in that ordering.



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

⇒ 5 4 2 3 1 0





LIFO

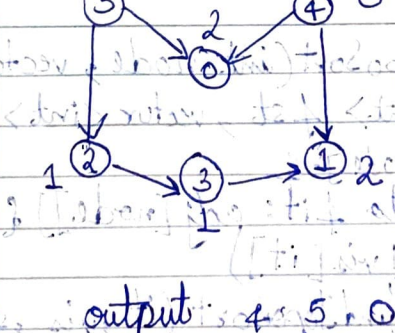
Code:

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

```
vector<int> topoSort(int N, vector<int>
    stack<int> st;
    vector<int> vis(N,0);
    for(int i=0; i<N; i++){
        if(vis[i]==0)
            findTopoSort(i, vis, st, adj);

    vector<int> topo;
    while(!st.empty()){
        st.push_back(st.top());
        st.pop;
    }
    return topo;
}
```

Topological Sort (BFS) Kahn's

TC → O(N+E)
SC → O(N) + O(E)



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

In degree.

output: 4 5 0 2 3 1

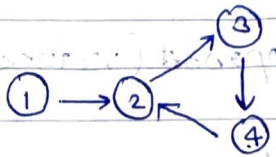
Q: 1 3 2 0 5 4

adj[][]

Code

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

Detect Cycle in Directed Graph



Kahn's Algo

↓

Topological Sort

→ If topological sort can't be generated then it is cyclic

Code

```
while(!q.empty()){
    .
    .
    cnt++;
    .
    .
}
```

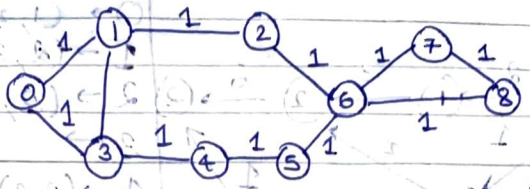
if(cnt == N) ← we are able to generate topological sort

    return false;

return true;

TC → O(N+E)
SC → O(N) + O(N)

Shortest Distance in Undirected Graph



src = 0

adj

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

0 — 6 ⇒ 3

src ↑ des ↑

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

Code:

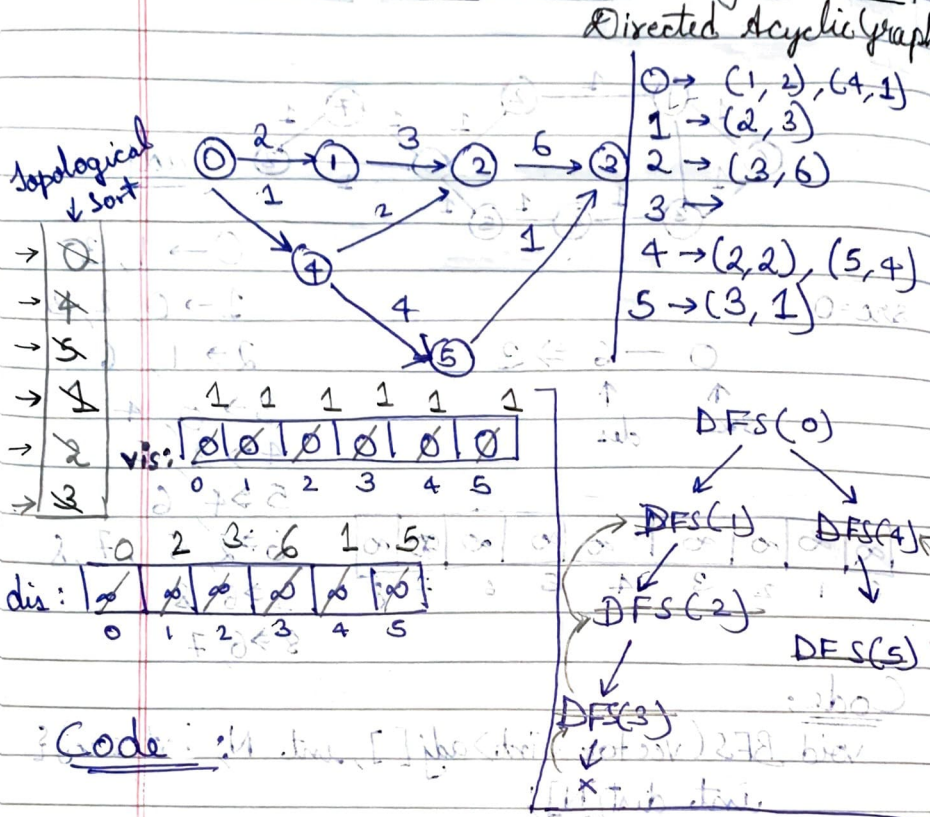
```
void BFS(vector<int> adj[], int N, int src){
    int dist[N];
    for(int i=0; i<N; i++)
        dist[i] = INT_MAX;
    queue<int> q;
    dist[src] = 0;
    q.push(src);
    while(!q.empty()){
        int node = q.front();
        q.pop();
        for(auto it : adj[node]){
            if(dist[node]+1 < dist[it]){
                dist[it] = dist[node]+1;
            } q.push(it);
    // print dist array }
```

TC → O(N + E) × 2 ASC → for DFS

SC → O(2N)

Shortest Path in a weighted DAG

Directed Acyclic Graph



Code :

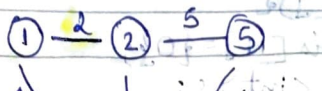
```
void findTopoSort(int node, int vis[], stack<int> &st, vector<pair<int,int>> adj[]) {
    vis[node] = 1
    for(auto it : adj[node])
        if(!vis[it.first])
            findTopoSort(it.first, vis, st, adj);
    st.push(node);
}
```

```
void shortestPath(int src, int N, vector<pair
<int,int>> adj[]){
        int vis[N] = {0};
        stack <int> s;
        for(a int i=0 ; i<N; i++){
            if (!vis[i])
                findTopoSort(i,vis,st,adj));

        int dist[N];
        for(int i=0 ; i<N ; i++)
            dist [i] = 1e9

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

TC = O(N+E) log N
~ O(N log N)

Dijkstra's Algo SC = O(N) + O(N)

adj 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

0	2	4	1	7	5
∞	∞	∞	∞	∞	∞
0	1	2	3	4	5

this will never give optimal answer so, we can use

(dis, node) min heap

set<pair<int,int>>

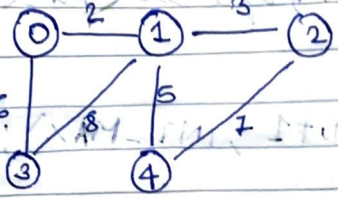
Code:

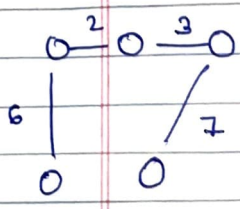
```
int main(){
    int n, m, source;
    cin >> n >> m;
    vector <pair <int, int>> g [n+1];

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

    cin >> source;
```

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

vector<int> dist(n+1, INT_MAX);

dist[source] = 0;
pq.push({0, source});        // {dist, node}

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
        ind     nextDist = it->second;
        if(dist[next] > dist[prev] + nextDist){
            dist[next] = dist[prev] + nextDist;
            pq.push({dist[next], next});
        }

    for(int i = 1; i<=n; i++)
        cout<<dist[i]<<" ";

return 0;
}
```

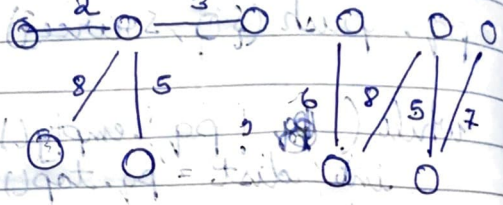
convert graph with 'N' nodes 'N-1' edges

Minimum Spanning Tree

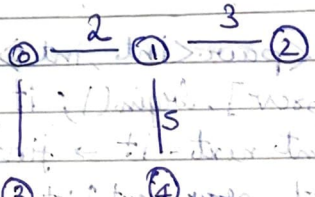




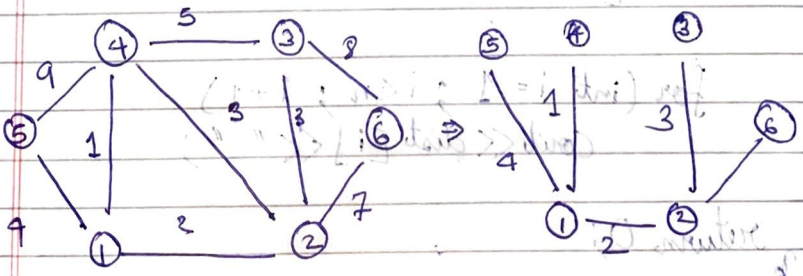
= 18



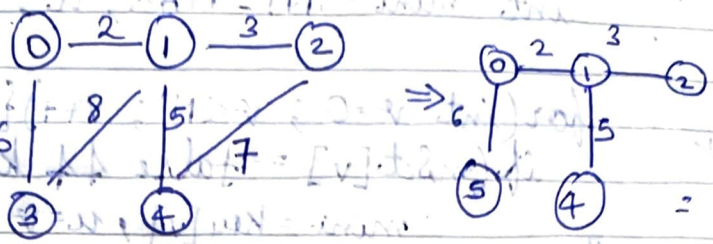
= 18 = 26

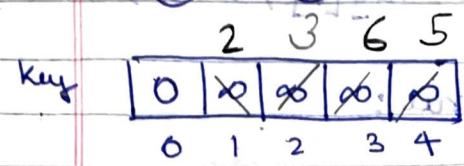


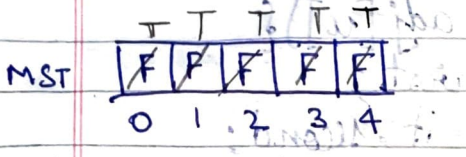
= 16 → (min. weight)



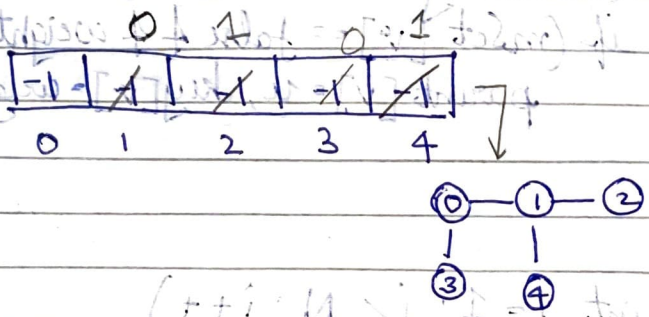
Prim's Algo (MST)







⇒ mein in key list



Code (brute force)

```
int parent[N], key[N], mSet[N];

for (int i=0; i<N ; i++)
    key[i] = INT_MAX , mSet[i] = false ;

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

```
for(int i=0; i<N-1; i++){
    int mini = INT_MAX, u;

    for(int v=0; v<N; v++){
        if(mSet[v]==false && key[v]<mini)
            mini = key[v], u=v;
    }
    mSet[u] = true;

    for(auto &it : adj[u]){
        int v = it.first;
        int weight = it.second;
        if(mSet[v]==false && weight<key[v])
            parent[v] = u, key[v] = weight;
    }
}

for(int i=1; i<N; i++)
    cout << parent[i] << " - " << i << endl;

return 0;
}
```

can use min heap

⇒ (optimised) $O(N+E)+O(n\log n)$ ⇒ time
$O(n\log n)$ ⇒ space

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

key[0] = 0;
parent[0] = -1;
pq.push({0,0});
```

index

```
while (! pq.empty())
{
    int u = pq.top().second;
    pq.pop();
    mSet[u] = true;
    .
    .
    .
    if (mSet[v] = false && weight < key[v])
    {
        parent[v] = u        key[v] = weight;
        pq.push({key[v], v});
    }
    .
    .
    .
}
```

Disjoint Set → if two node belong to same component

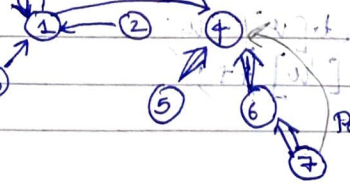
→ findPar(): → parent of a set

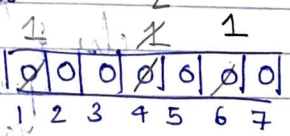
→ Union() → join to set

Time: $O(4\alpha) \approx O(4)$

Space: $O(n)$

Union(1,2) ✓
" (2,3) ✓
" (4,5) ✓
" (6,7) ✓
" (5,6) ✓
" (3,7) ✓





Path Compression

Code

```
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
    return parent[node] = findPar(parent[node]);
}
```

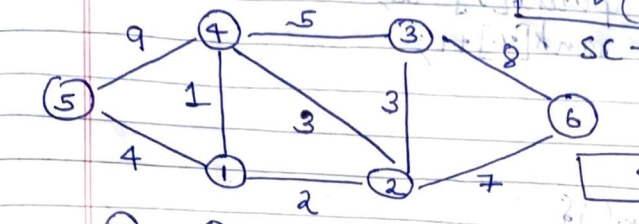
path compress

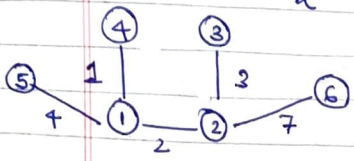
```
void union(int u, int v){
    u = findPar(u);
    v = findPar(v);
    if(rank[u] < rank[v]){
        parent[u] = v;
    else if
        parent[v] = u;
    else{
        parent[v] = u;
        rank[u]++;
    }
```

→ Level increases when same rank sets are attached

Krushal's Algo

TC → O(M log M)
+ O(M × O(4α))
≈ (M log M)
SC → O(M)
+ O(N)
+ O(N) disjoint set
≈ O(N)





Code

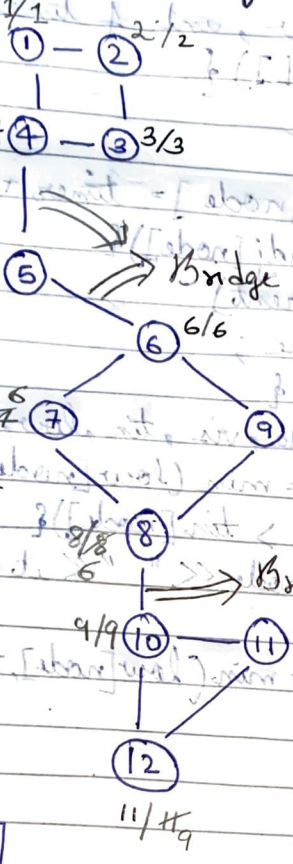
```
struct node {
    int u;
    int v;
    int wt;
};

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 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(rank[v] < rank[u])
        parent[v] = u;
```

```
        else{
            parent[v] = u;
            rank[u]++;
        }
    }

int main(){

    vector <node> edges;

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

An edge is called bridge on whose removal the graph is broken on 2 or more component.

Bridges in Graph

time of insertion / minimal time



low[it] -> tin[it]

↳ its a bridge

TC ⇒ O(N+E)

SC ⇒ O(2N) ~ O(N)

ASC in case of recursion

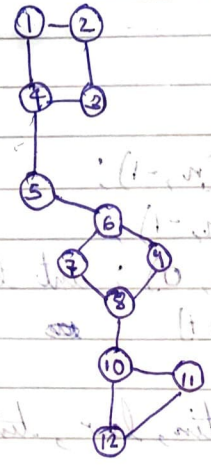
parent);

Code :

```
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);
```

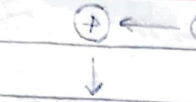
```
void dfs(int node, int parent,
vector<int>& vis, vector<int>& tin,
vector<int>& low, int& timer,
vector<int> adj[]) {

    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]);
    }
}
```

Articulation Point

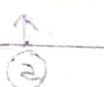


low[it] >= tin[node]
&& parent != -1

SCC → Strongly Connected Components

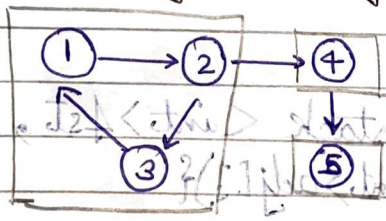


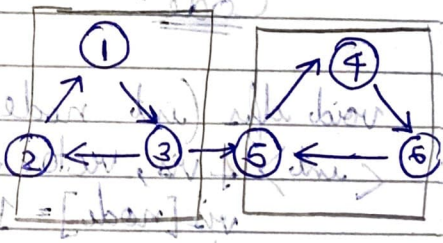
```
low[node] = min(low[node], low[it]);
child++
if(low[it] >= tin[node] && parent != -1)
        isArticulation[node] = 1
```



```
if(parent == -1 && child > 1){
        isArticulation[node] = 1;
```

Kosaraju's Algorithm for SCC



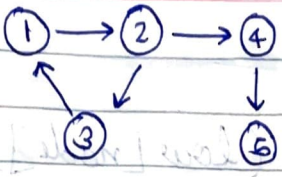


→ Every node is reachable to every other node

5
4
3 2 1

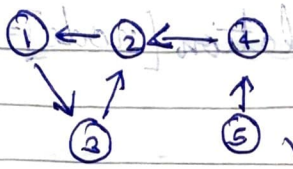
6,5 4
3, 2 1

(1) Sort all nodes in order of finishing time → O(N)
(2) Transpose the graph → O(N+E)
(3) DFS according to finishing time → O(N+E)



Time : O(N+E)
Space : O(N+E) + O(N)
↓ Transpose graph
↳ visit

1
2
4
5
3



DFS(4)

DFS(5)

DFS(1)
DFS(3)
DFS(2)

Code:

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

void revDfs (int node, vector <int> & vis, vector
<int> transpose [] ){
    cout << node << " ";
    vis [node] = 1;
```

N) +O(N)

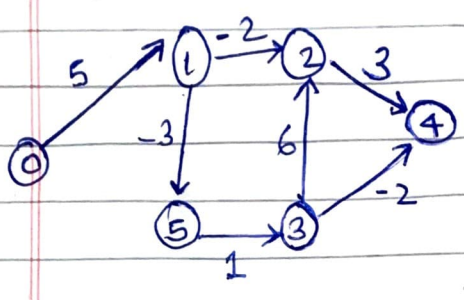
d stack

```
for (auto it : transpose[node]){
    if (!vis[it])
        revdfs(it, vis, transpose);
}
}

int main(){
    stack<int> st;
    vector<int> vis(n,0);
    for(int i=0; i<n; i++){
        if(!vis[i])
            dfs(i, st, vis, adj);
    }
    vector<int> transpose[n];
    for(int i=0; i<n; i++){
        vis[i]=0;
        for(auto it : adj[i])
            transpose[it].push_back(i);
    }

    while(!st.empty()){
        int node = st.top();
        st.pop();
        if(!vis[node])
            revDfs(node, vis, transpose);
    }
    return 0;
}
```

Time: $O(N-1) \times O(E)$
Space: $O(N)$

Bellman Ford Algo (Directed graph)

⇒ Can't work in case of negative cycle



src = 0

Repeat this n-1 times

u	v	t
(3,	2)	6
(5,	3)	1
(0,	1)	5
(1,	5)	-3
(1,	2)	-2
(3,	4)	-2
(2,	4)	3

[·] Relax all the edges N-1 times

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

	5	3	-1	6	-2
dist[] →	0	∞	∞	∞	∞	∞
	0	1	2	3	4	5

⇒ dist[0] + 5 < dist[1]

∴ 0 + 5 < ∞

dist[] →	0	5	3	3	1	2

Code

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

```
int main(){
    .
    .
    .
int src;
cin >> src;
vector <int> dist(N, INT_MAX);

dist[src] = 0;

for(int i=1; i<=N-1; i++){
      for(auto &it : edges)
       if(dist[it.u] + it.w < dist[it.v]){
             dist[it.v] = dist[it.u] + w;
       }
}
int fl = 0;
for(auto &it : edges){
     if(dist[it.u] + it.w < dist[it.v]){
           cout << "Negative Cycle";
           fl = 1;
           break;
     }
}

if(!fl)
   for(int i=0; i<N; i++)
         cout << i << " - " << dist[i] << endl;

return 0;
```