Nand
Date
Page

Striver's Graph Series

NOTES:

By: Rohit Bindal

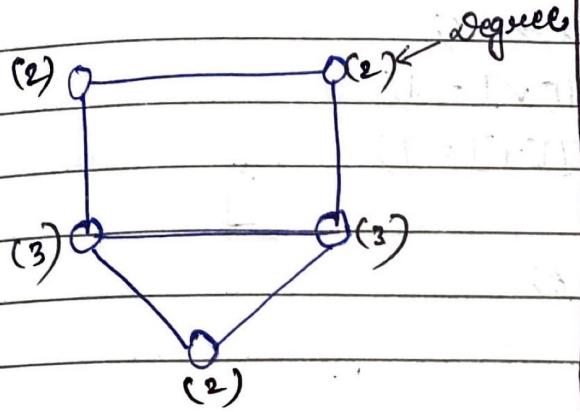
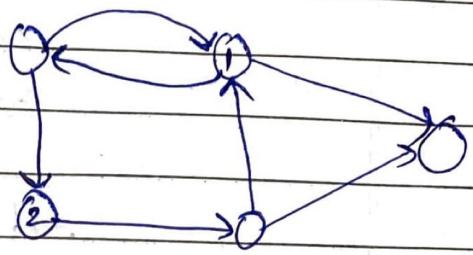
take U forward

♡TUF♡

Graph

Nand
Date
Page

→ Graph is set of nodes/~~edge~~ vertices & edges.

Undirected	Directed
	
Total degree = 2 * no. of edges (12 = 2*6)	in degree(1) = 2 in degree(2) = 1 out degree(1) = 2

- Path: set of nodes without repetition.
- Graph representation in c++:

① Adjacency Matrix:-

→ if no. of nodes are n then create n×n matrix (assuming 0-based indexing of nodes)
↳ A[i][j] = 1 means there is an edge between i & j.

```
//General code: (to take input)
int main() {
    int n, M;    //n=nodes, M=edges
    cin >> n >> m;

    int adj[n+1][m+1]; // for 1-based indexing
```

```
// 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;
}
```

disadv: 1) adjacency matrix can be used only when value of n is not large. (S.C. = $O(N^2)$)

(2) Adjacency list :- (S.C. = $O(N+2E)$)

```
vector<int> adj[n];
if u & v are connected:
    adj[u].push_back(v)
    adj[v].push_back(u)
```

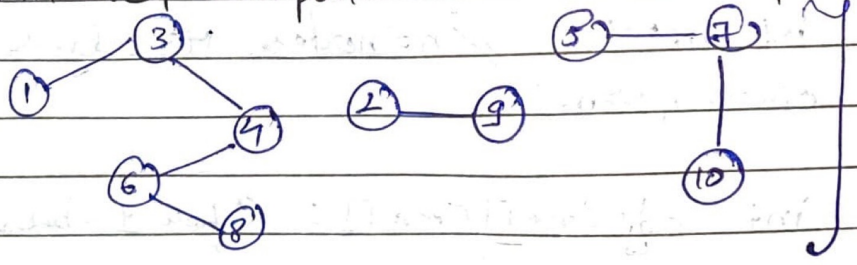
if graph is weighted:

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

(int: adjacent node; int: weight of edge)

- Connected Components in a Graph :-

eg:



A graph with 3 connected components

Nand
Date
Page

1. Breadth - First Search (BFS):-

→ traverse the adjacent nodes first then move ahead.

steps 1) take a queue & a visited array.
2) push initial node in queue & mark it as visited
3) while queue is not empty:
   a) pop the front node, & print it
   b) push its adjacents in queue if they are not visited, & mark them visited

→ repeat these 3 steps for each component.

T.C. = S.C. ≈ O(n)

Code:

```
vector<int> bfsOfGraph(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;
}
```

Nand
Date
Page

- Depth - First Search (DFS):-

→ keep going until adjacent unvisited node is there

→ its a recursive approach

→ TC ≈ SC ≈ O(N)

Code:

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

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

Nand
Date
Page

• cycle Detection in Undirected Graph using BFS:

Idea: do BFS, if from a node 'u', you visit a node 'v' & if v is already visited and also if v is not parent of u (i.e. we do not went to u from v) then we can say, cycle is there

T.C. = S.C. = O(n)

code:

```
bool cycle(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;
}
```

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

if any component has cycle return true

- Cycle Detection in undirected graph using DFS:-

Idea: If adjacent node (next node) is already visited then cycle is there.

Code:

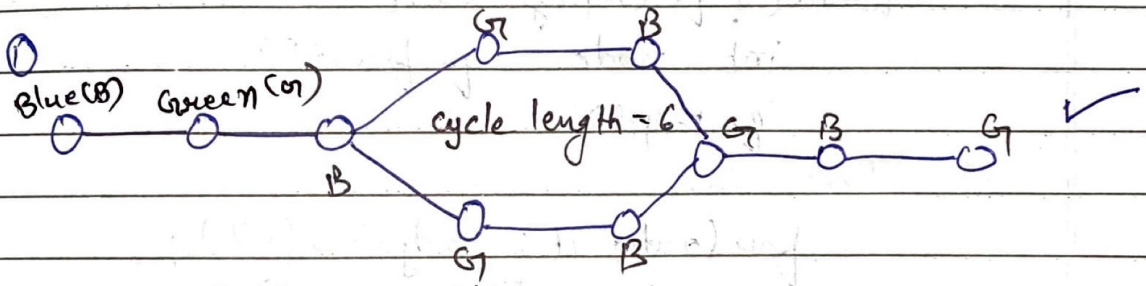
```
bool cycle(int node, int parent, vector<int> &vis,
           vector<int> adj[]) {
    vis[node] = 1;
    for(auto it: adj[node]) {
        if(vis[it] == 0) {
            if(cycle(it, node, vis, adj))
                return true;
        }
        else if(it != parent) return true;
    }
    return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

bool isCycle(int V, vector<int> adj[]) {
    vector<int> vis(V+1, 0);
```

```
for(int i=1; i<=V; i++) {
    if(!vis[i]) {
        if(cycle(i,-1,vis,adj)) return true;
    }
}
return false;
}
```

* Bipartite Graph (BFS) / Graph Coloring :-

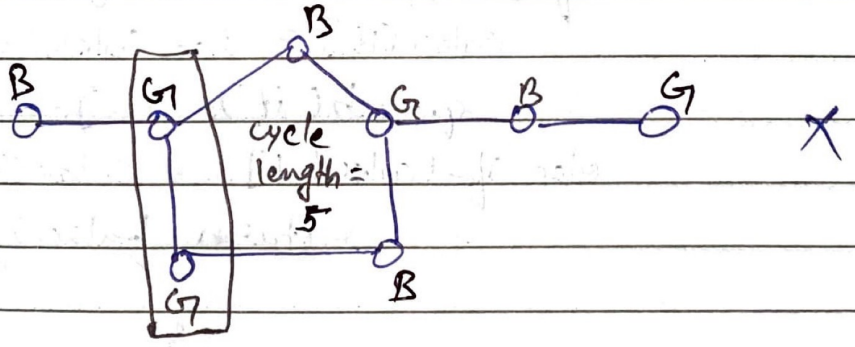
→ A graph that can be colored using 2 colors such that no two adjacent nodes have same color is called as Bipartite Graph.

eg: ①



✓

②



✗

→ A Graph containing odd length cycle can not be a bipartite graph :)

Idea: do bfs, maintain a color array

→ push root (starting node) in queue & mark its

color as 0

→ then keep visiting adjacent nodes & mark them colored with opposite color of their parent.

→ at any point if you visit a node which is already colored & the color is same as color of source node then return false

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;
}
```

* Checking for Bipartite Graph using DFS:-

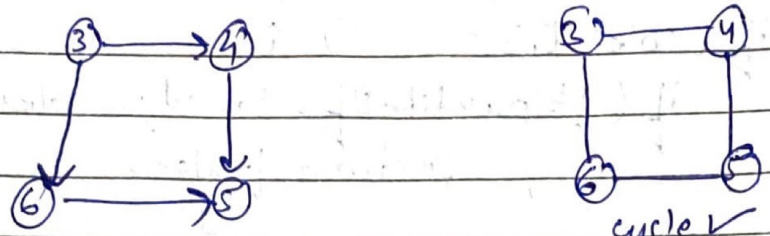
Idea: same as of BFS, but here we will do it recursively.

T.C. = S.C. = O(n)

code:
```
bool bipartiteDfs(int node, vector<int> adj[], int color[])
{
  if(color[node] == -1) color[node] = 1;

  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;
}
```

Nand
Date
Page

Cycle Detection in Directed Graph using DFS:

eg 

→ So we can't use the idea of cycle detection in undirected graph.

Idea: If in recursion stack, a node comes two times then cycle is there.

→ So we can maintain two arrays, visited & dfs call array.

→ If you go to a node then mark it in both arrays but when you go back from that node then unmark it from dfs-visited array only.

→ If you go to a node whose dfs-visited array is already marked then cycle is there

code:

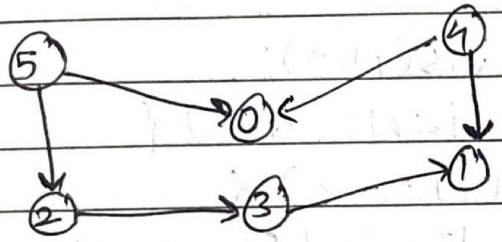
```
bool checkCycle(int node, vector<int> adj[],
                int vis[], int dfsVis[]) {
    vis[node] = 1;
    dfsVis[node] = 1;
    for(auto it : adj[node]) {
        if(!vis[it]) {
            if(checkCycle(it, adj, vis, dfsVis))
                return true;
        }
```

```
        else if(dfsVis[it]==1) return true;
        }
        dfsVis[node] = 0;
        return false;
    }
```

Topological sort using DFS:-

→ Topological sorting is linear ordering of vertices such that if there is an edge $u \to v$, then u appears before v in that ordering.

eg:



one of the topological sort: 5 4 2 3 1 0
(edge will be from left to right only)

→ Topological sort is possible only for Directed Acyclic Graphs (DAGs).

Idea: 1) maintain a visited array & a stack to store the topological sort.

2) if adjacents of a node are done then push the node into stack. (that means adjacent nodes are already in stack)

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); //Backtracking
}

vector<int> topoSort(int N, vector<int> adj[]) {
    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()) {
        topo.push_back(st.top());
        st.pop();
    }
    return topo;
}
```

Topological Sort using BFS (Kahn's Algorithm):-

Idea: node with lesser indegree will come before than the node with greater indegree in the topological sort.

→ so just get the indegree of each node & store in an array.
→ push the nodes in queue with indegree = 0.
→ while queue is not empty:
    1) print the front node & pop it
    2) decreement indegree of its adjacents
    3) if indegree becomes 0 then push that adjacent node in the queue.

code:
```
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;
}
```

Nand
Date
Page

✡ Cycle Detection in Directed Graph using BFS (Kahn's Algorithm):

Idea: If you are not able to generate a topological sort, then graph has cycle :)

→ So, code is same as that of topo sort, just count the no. of nodes you are getting in topo sort vector,

```
if (count == n) return false; // no cycle
else return true;
```

✓

✡ Shortest Path in Undirected Graph with Unit Weights:-

Find shortest distance from given source node to every other node in the graph

Idea:
1) perform BFS
2) maintain a distance array, initially filled with ∞
3) Mark source's distance as 0
4) If you visit neighbours of a node u (whose distance is d) then distance of neighbours will be min(d+1, dist(neighbour)).
5) TC = SC = O(N)

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);
        }
    }
}

for(int i=0; i<N; i++)
    cout << dist[i] << " ";
}
```

* Shortest path in weighted DAG :-

Given a src node, find shortest path length from src to every other node.

Idea:
1) Store topological sort in stack using dfs.
2) make a distance array, mark src as 0 & all other nodes as ∞
3) While stack is not empty:
   a) pop (let say node u is popped)
   b) see the adjacents nodes of u, dist. of adjacent node = min(dist[adjacent node], dist[u] + weight on edge of u & adj. node

Nand
Date
Page

TC = SC = O(N)

code:

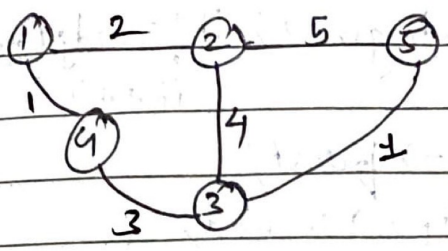
```
void shortestPath(int src, int N, vector<pair<int,int>>
                                        adj[]) {
    int vis[N] = {0};
    stack<int> st;
    for(int i=0; i<N; i++)
        if(!vis[i])
            findTopoSort(i, vis, st, adj);

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

    while(!st.empty()) {
        int node = st.top();
        st.pop();
        // if the node has been reached previously
        if(dist[node] != 1e9) {
            for(auto it: adj[node]) {
                if(dist[node] + it.second <
                        dist[it.first]) {
                    dist[it.first] = dist[node] + it.second;
                }
            }
        }
    }
```

Nand
Date
Page

Dijkstra's Algorithm :-

Given a weighted undirected graph, find shortest path from src node to every other node.

eg:



source = 1
(1-2): 2
(1-3): 4
(1-4): 1
(1-5): 5

Idea: ① Maintain a min heap of pair {distance, node}
② Make a distance array, mark source as 0 & every other node as ∞
③ While min heap is not empty:
a) take the top pair of heap
b) check for adjacents of top node
c) if you find better distance, update it in visited array & push into heap

TC ≈ O(nlogn)
SC ≈ O(n)

code:
```
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(make_pair(0, source)); // {dist, from}

while(!pq.empty()) {
    int dist = pq.top().first;
```

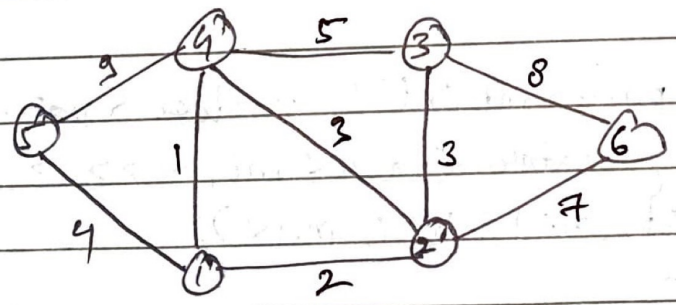
```
int prev = pq.top().second;
pq.pop();

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

* Minimum Spanning Tree (MST) :-

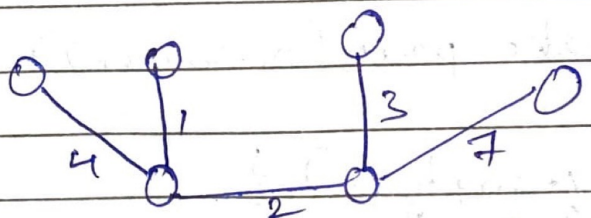
→ If a graph has n nodes then spanning tree will have n nodes and n-1 edges

→ MST is a spanning tree with minimum cost.

eg.



MST:



cost = 17

Nand
Date
Page

① Prim's Algorithm :-
→ used to find MST

Idea : 1) take any one node for tree
2) now check the adjacent edges of all the nodes taken till now in the tree & take the edge with minimum weight.
3) repeat step 2 n-1 times.

steps ① maintain three arrays :
a) key : it will store weights as we move ahead, so from this array we will get the edge with minimum weight
b) MST : it will be a bodean array, mst[u] = true means u node has already taken in MST.
c) parent : in the end, from this array we can construct the MST

→ as we want minimum weight's index (edge) each time, so we can use min heap instead of key array.

Code : int parent[N], key[N], mstSet[N];
for(int i=0; i<N; i++)
key[i] = INT_MAX, mstSet[i] = false;

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

```
key[0] = 0;
parent[0] = -1;
pq.push({0,0}); // {key, index}
// Iterating n-1 (no. of edges) times
for(int count=0; count<N-1; count++){
    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;
            pq.push({key[v], v});
            key[v] = weight;
        }
    }
}
```

TC ≈ O(nlogn)

```
// printing MST:
for(int i=1; i<N; i++)
    cout << parent[i] << " - " << i << "\n";
```

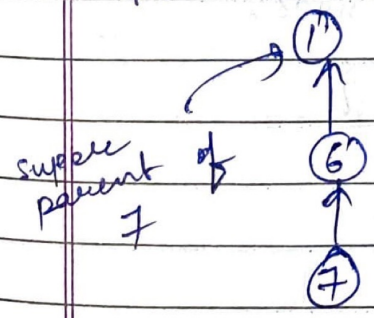
Nand
Date
Page

Disjoint Set :-

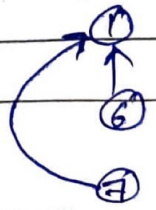
→ this data structure is generally useful when we are given two nodes & we want to find whether they belongs to same component or not.
→ we assume all the vertices to be in different component initially (i.e. all are parents of themselves)
→ If there is an edge between two nodes, then we combined them in a single component by making one of them as parent of both ⇒ this is called as Union operation of two nodes

→ "Path Compression" is used to implement it efficiently.
→ also we connect less rank node to the node with higher rank to reduce the height of tree & to get the parent of a node quickly.
(If rank is same, then connect any node to other, also increement the rank of the node to which other node is connected as height of the tree will be increased by 1) ⇒ "Union By Rank".

path Compression:
→ let say 1 is parent of 6 & 6 is parent of 7.



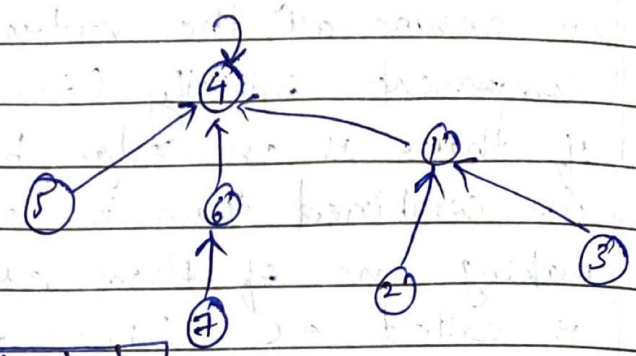
Now if we want parent of 7 then the path is 7 → 6 → 1
so, we can compress this path as:



Nand
Date
Page

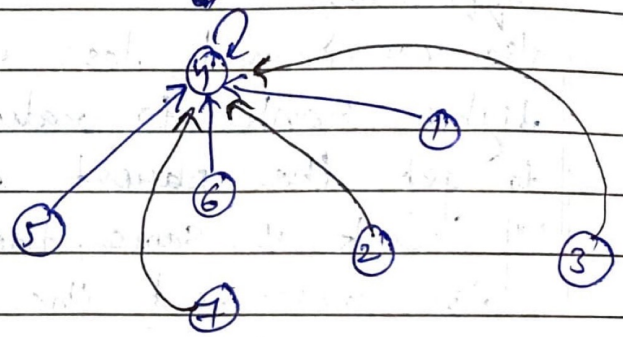
eg no. of nodes = 7
1)union(1,2), 2)union(2,3), 3)union(4,5)
4)union(6,7), 5)union(5,6), 6)union(3,7)

final graph:



Rank:	1	0	0	2	0	1	0
	1	2	3	4	5	6	7

after Path Compression



T.C. of findParent() & Union() $\approx O(4\alpha) \approx O(4)$

Code:

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

void makeSet() {                    // no. of nodes
    for(int i=1; i<=n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}
```

Nand
Date
Page

```
int findPar(int node) {
    if(node == parent[node])
        return node;

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

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

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

→ if parent of u & parent of v are same then they belongs to same component

Nand
Date
Page

(2) Kruskal's Algorithm :-
→ used to find MST

Idea: 1) Store all edges in an array & sort them acc. to their weights.
2) Keep taking the edge with lesser weight if corresponding nodes does not belong to same component. (to avoid cycle)

TC = O(n log n)
SC = O(n)

code: struct node {
int u, v, 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 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);

int cost = 0;
vector<pair<int, int>> mst;
for(auto it: edges) {
    if (findPar(it.v) != findPar(it.u)) {
        cost += it.wt;
        mst.push_back({it.u, it.v});
        union(it.u, it.v);
    }
}

cout << cost << endl;
for(auto it: mst)
    cout << it.first << " - " << it.second << endl;
    return 0;
}
```

☆ Bridges in Graph :-

→ An edge is called as bridge if its removal increases the no. of components in the graph (i.e. it disconnects the graph)

→ bridge is also called as cut-edge.

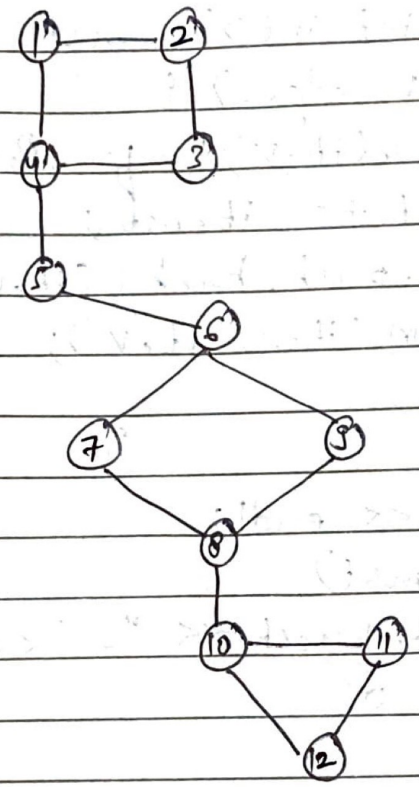
Idea: Maintain two arrays, time of insertion & lowest time of insertion.

→ lowest time of insertion of a node is lowest time of insertion of it & its adjacents.

→ if lowest time of insertion of adjacent node is greater than

its time of insertion then the edge is not a bridge

I/p:



O/p: (8-10)
(5-6)
(4-5)

T.C. ≈ O(N+E)
SC ≈ O(N)

code:

```
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);
```

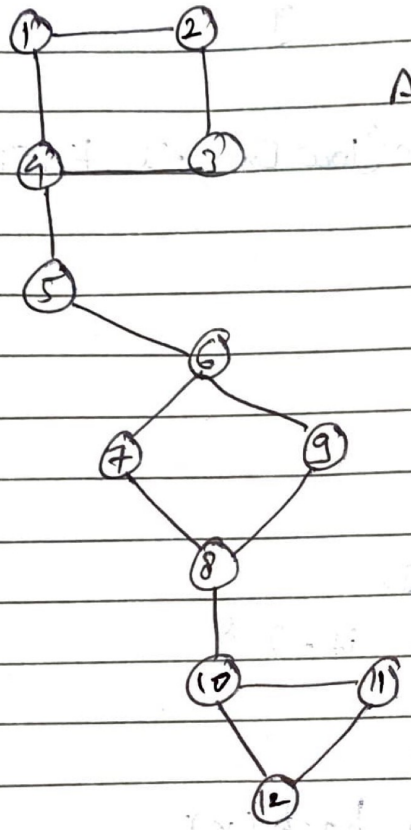
```
        low[node] = min(low[node], low[it]);
        if (low[it] > tin[node]) {
            cout << node << " " << it << endl;
        }
    }
    else {
        low[node] = min(low[node], tin[it]);
    }
}
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];
    for(int i=0; i<m; it++) {
        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; it++) {
        if(!vis[i]) {
            dfs(i, -1, vis, tin, low, timer, adj);
        }
    }
}
```

Nand
Date
Page

☆ Articulation Point / Cut vertex :-

→ its a point/node/vertex whose removal increases the no of components in graph.

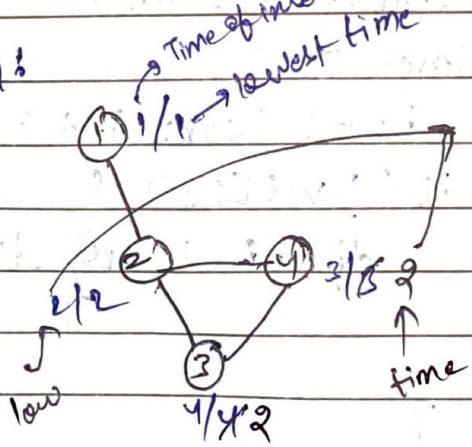
eg.



Articulation Points: 4, 5, 6, 8, 10

Ideq : lowest time of insertion & time of insertion of each node will be maintained.
If : low[adjacent] >= time[node] && parent != -1, then the node is articulation point.

Intuition :

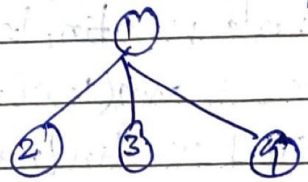


2>=2 ⇒ so 2 is cut vertex (because we are reaching at 2 before other half that is 3 & 4) (we cant reach 3 & 4 before 2)

→ if parent = -1 then that means there is no upper half, so even if you remove that node, no. of components are not going to increase.

→ starting node can be articulation point (even if it dont have parent) if it has individual childs

eg.

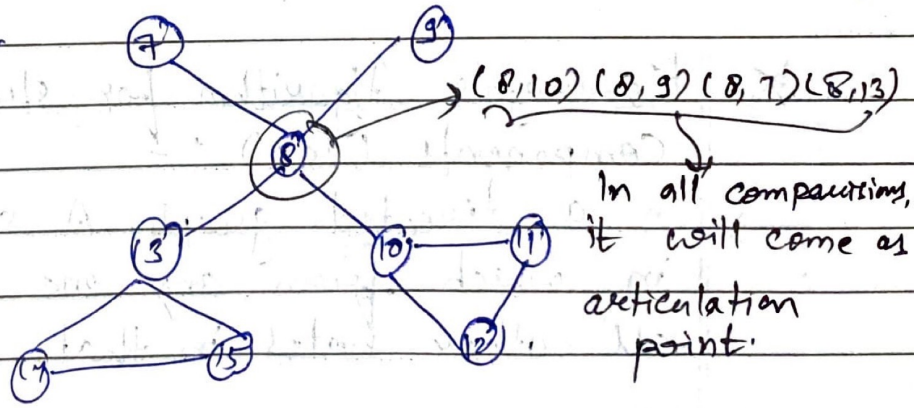


if (#childs > 1 && parent == -1)
then also the node is cut vertex

TC ≈ SC ≈ O(N)

NOTE: While running algo, one node can come multiple times due to multiple childs, so its better to mark the node in hash map.

eg.



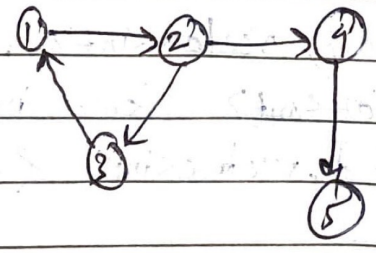
code: void dfs(int node, int parent, vector<int> &vis, vector<int> &tin, vector<int> &low, int &timer, vector<int> adj[], vector<int> &isArticulation){
vis[node] = 1;

```
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]);
        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;
}
```

★ Kosaraju's Algorithm for strongly Connected Components (scc) :-

→ In a directed graph, A scc is a component in which from any one node, we can reach all other nodes in that component.

eg.



scc: 1 2 3
4
5

3 sccs

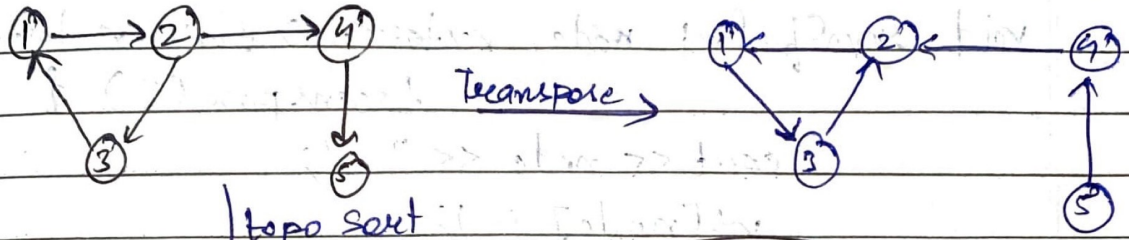
Nand
Date
Page

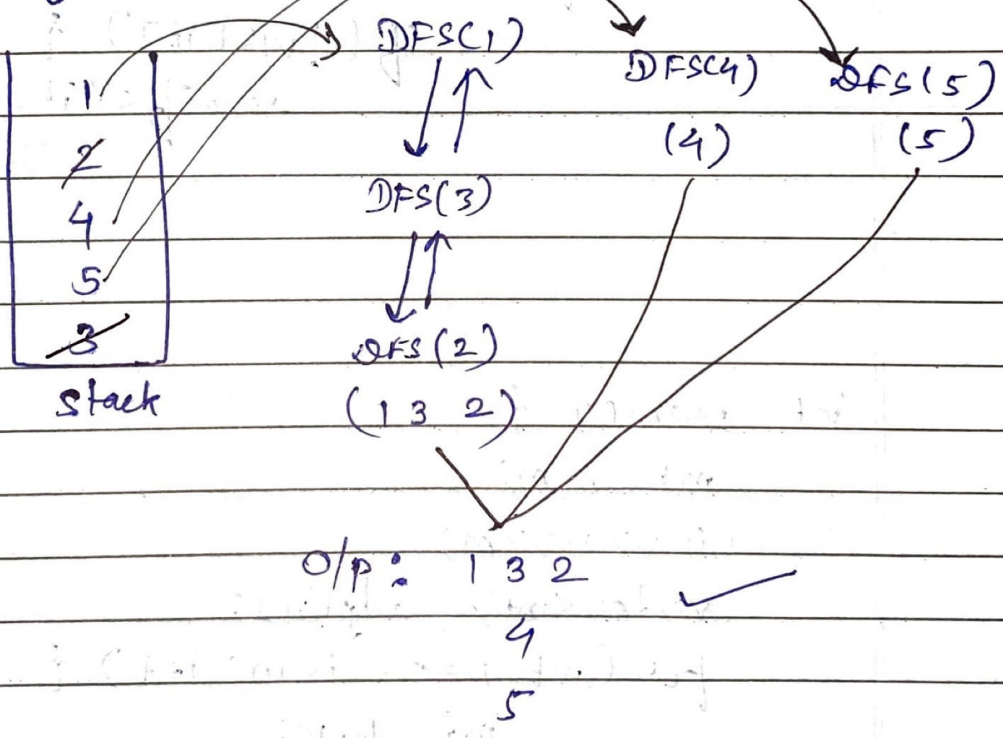
steps ① sort all nodes in order of finishing time. (i.e. topological sort).

② Transpose the graph (i.e. reverse the edges) so that one DFS call in a component will not enter in other component.

③ perform DFS according to topological sort

eg.





O/p: 1 3 2
4
5

TC ≈ SC ≈ O(N)

Code:

```
void topoSort(int node, stack<int> &st, vector<int> &vis,
              vector<int> adj[]) {
    vis[node] = 1;
```

```
    for(auto it : adj[node]) {
        if(!vis[it])
            topoSort(it, st, vis, adj);
    }
    st.push(node);
}

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

```
vector<int> vis(n,0);
for(int i=0; i<n; i++) {
    if(!vis[i])
        topoSort(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]) {
        cout << "SCC: ";
        revDfs(node, vis, transpose);
        cout << endl;
    }
}
return 0;
}
```

Nand
Date
Page

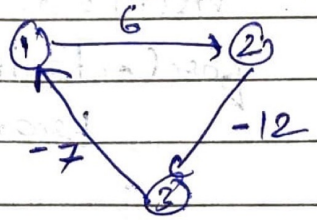
Bellman Ford Algorithm :-

→ used to find shortest path from a given source node to all other nodes.

→ Dijkstra's Algo will fail if there are edges with negative weight in the graph.

→ Bellman ford algo can work even if negative weight edges are there but it will fail if there is a negative weight cycle (i.e. a cycle with sum of edge weights being negative)

eg



→ in this, we want shortest path between 1 & 3 then it will be:

1 → 2 → 3 → 1 → 2 → 3 → 1 .....
6 -12 -7 6 -12
(-6) (-19) ----

distance will keep on reducing

→ Howevere, Bellman ford algo can detect the negative edge weight cycle

Idea: Relax all the edges n-1 times.

→ i.e. if (dist[u] + wt < dist[v])
dist[v] = dist[u] + wt

→ if after relaxing n-1 times, if you relax again & if distance reduces then it indicates that

negative edge weight cycle is there.

→ we are relaxing n-1 times because in a graph of n nodes because the longest path from source to any node will contain n-1 edges atmost.

TC ≈ O(NE)
SC ≈ O(N)

code:
```
struct node {
    int u,v, 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 fl = 0;
for(auto it: edges) {
    if(dist[it.u] + it.wt < 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;
}
```