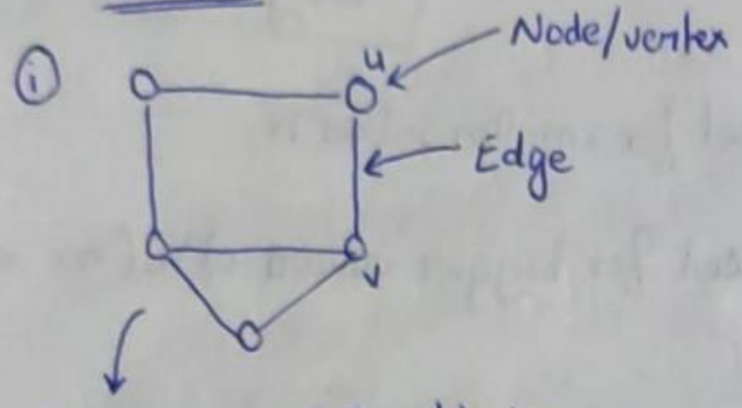
Graph:→

⊙ Graph → ⊡ It is a data structure having two components.

① Node/vertex

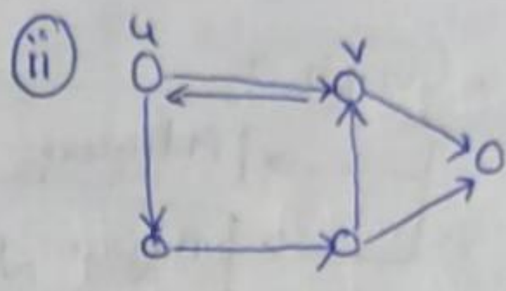
② Edge → It connects vertexes together.

⊡ Examples →



⊡ undirected Graph →

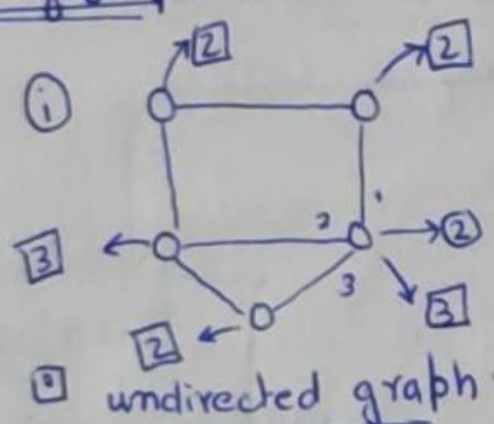
Graph having no direction i.e; there is an edge between u and v, also there is an edge between v and u.



⊡ Directed Graph →

Graphs having direction, i.e there is only one edge from u to v, and other edge from v to u.

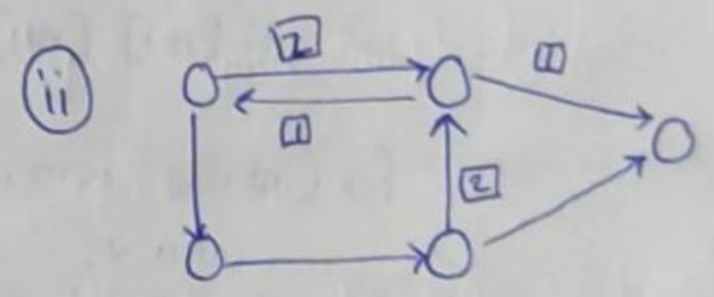
⊙ Degree of Graph →



⊡ undirected graph.

Degree (2) = 3

vertex (2) ∴ Total degree = 2 × Edges = 12 of all vertexes

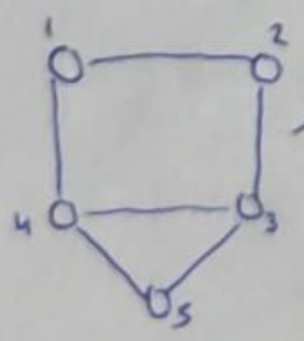


⊡ Directed Graph →

Indegree (2) = 2 (Incoming edges)

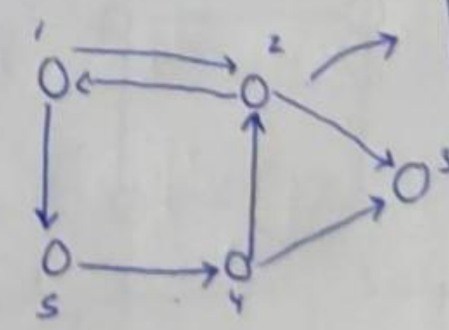
outdegree (2) = 2 (outgoing edges.)

⊙ Path →



If it contains a cycle, then it is known as cyclic undirected graph else Acyclic undirected graph.

⊡ Undirected Graph →



If it contains a cycle then known as cyclic else directed acyclic graph. (DAG)

⊡ Directed Graph :→

Path: Path is sequence of vertex where no vertex is visited twice.

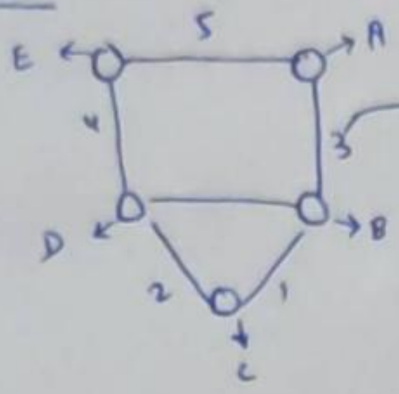
Example → 1232 ✗

123 ✓

Path → Same but path has got direction.

Example → 123.

⊙ Weighted Graph :→



→ weights are given to the edges.

⊡ Note :→

If weights of edges are not given, then assume the weights to 1 unit weight.

⊙ Way to make a Graph :→

- Adjacency matrix → used for smaller values n.
- Adjacency list → used for bigger values of n. (n = vertexes)

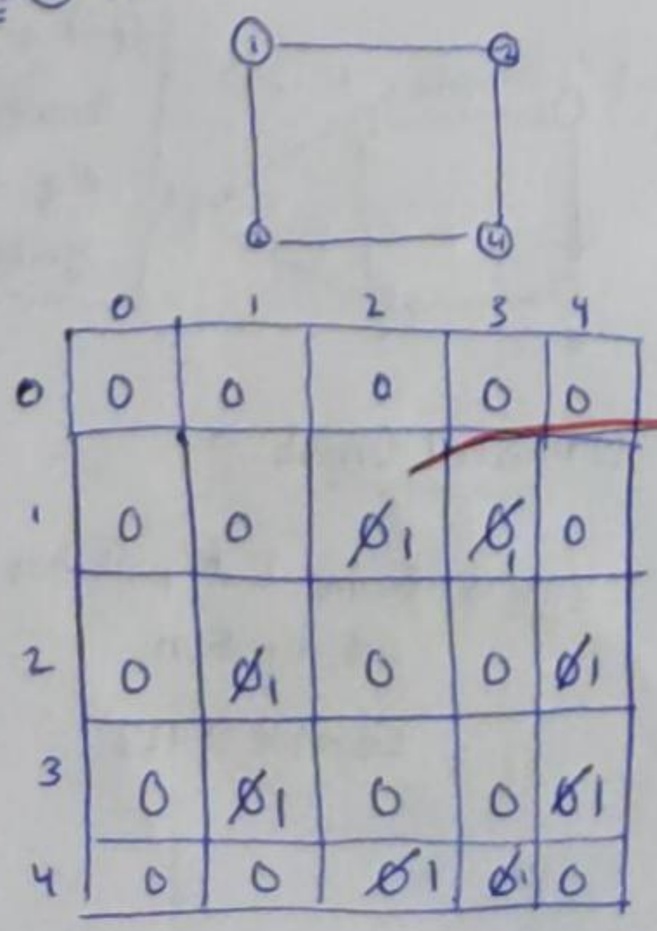
⊙ Approach 1 :→ (Adjacency matrix)

code :→

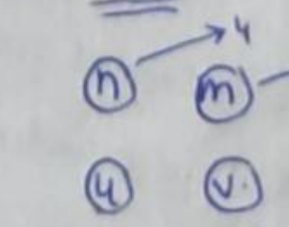
```
int main (){
    int n, m; // n is number of vertex and m is number of edges.
    cin >> n >> m;
    int adj [n+1] [n+1] = {0}; → declare adjacent matrix.

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

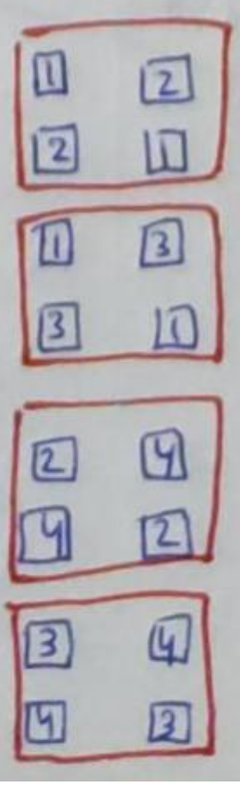
It is a way of representing a graph as a matrix of booleans.

⊙ Example ① :→



⊙ Input :→

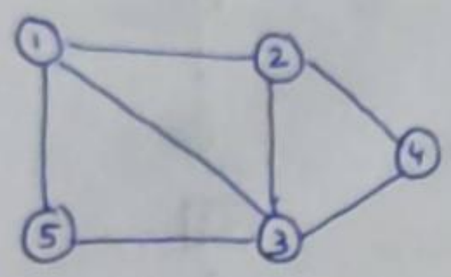




→ All edges between 1 and 2.

① Approach 2 → Adjacency list

⊡ Example ① →



N=5.

vector<int> adj[6];
vector<vector<int>> a;
map<int, list<int>> a;

} 3 ways of creating adjacency list.

⊡ u → v

1→2
1→5
1→3
3→5
2→3
2→4
3→4

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

⊡ Code :→

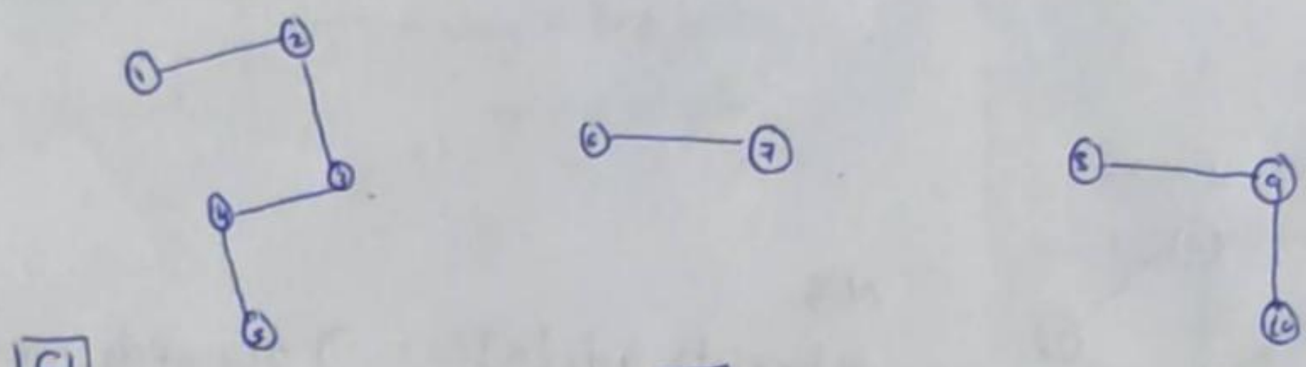
```
#include <iostream>
Using namespace std;
# include <bits/stdc++.h>

int main(){
    int n,m;
    cin>>n>>m;
    vector<int> adj[n+1];
        or,
    unordered_map<int, list<int>> adj;

    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;
}
```

we can skip this step in case of directed graph

⊡ Code :→ (If the graph is a weighted graph)

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

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;
}
```

⊙ Connected component in a Graph →



⊙ Note →

⊡ Every single vertex of a graph can be called as a component.

⊡ For all components of a graph, we need to write the code.

These are not 3 different graphs, rather they are 3 different components of the same graph. A graph can have multiple components.

⊙ Graph Traversal Techniques :→

→ (i) BFS (Breadth First Search)

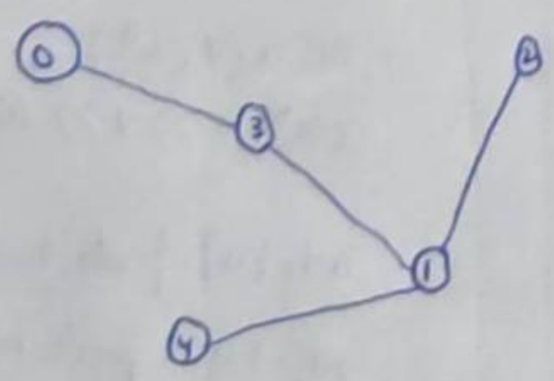
→ (ii) DFS (Depth First Search)

(i) BFS

→ ⊡ traversing the adjacent nodes/vertices first then moving to the next nodes/vertices.

→ ⊡ In BFS, if we start from any node all our nodes would be visited.

⊡ Dry Run:

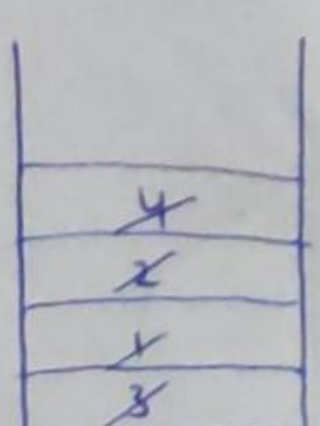


Adjacency list:

0 → 3
1 → 2,3,4
2 → 1
3 → 0,1
4 → 1

visited Map (map<node, bool>)

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



Front node = 0.
Front node = 3
Front node = 1
Front node = 2
Front node = 4

ans array = | 0 | 3 | 1 | 2 | 4 |

⊡ Src code (For multi components)

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

⊡ Code →

```
vector<int> BfsGraph(int V, vector<int> adj[]/unordered_map<int, list<int>> adj){

    vector<int> ans;
    unordered_map<int,bool> visit;

    for(int i=0; i<V; i++){
        if(!visit[i]){
            queue<int> q;
            q.push(i);
            visit[i]=1;
            while(!q.empty()){
                int frontnode = q.front;
                q.pop();
                ans.push_back(frontnode);
                for(auto it: adj[frontnode]){
                    if(!visit[i]){
                        q.push(it);
                        visit[it]=1;
                    }
                }
            }
        }
    }
    return ans;
}
```

if question wants ans in sored format then, we use set instead of list.

⊡ Time Complexity → O(N+E)

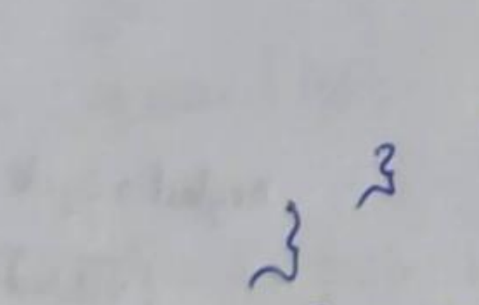
⊡ Space Complexity → O(N+E)

(ii)

DFS (Depth First Search)

└→ ⊡ while traverse we traverse through the depth of particular adjacent node.

Example :→



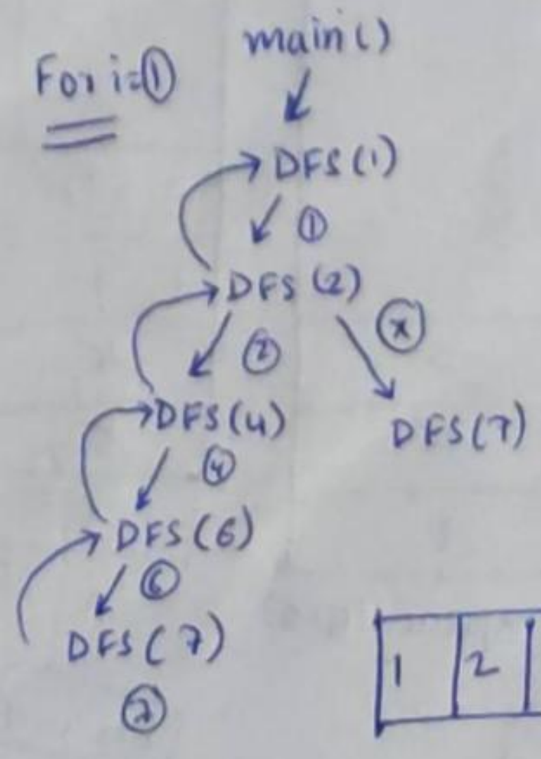
Adjacency list

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

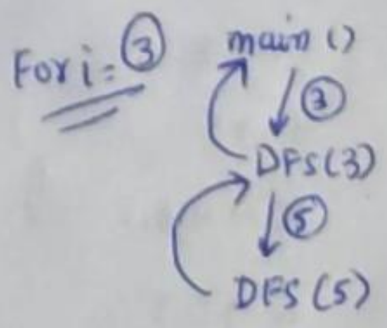
visited (map)

0 → F
1 → ~~F~~ T
2 → ~~F~~ T
3 → ~~F~~ T
4 → ~~F~~ T
5 → ~~F~~ T
6 → ~~F~~ T
7 → ~~F~~ T

For i=1



For i=3



Ans array.

1	2	4	6	7	3	5

▣ Code →

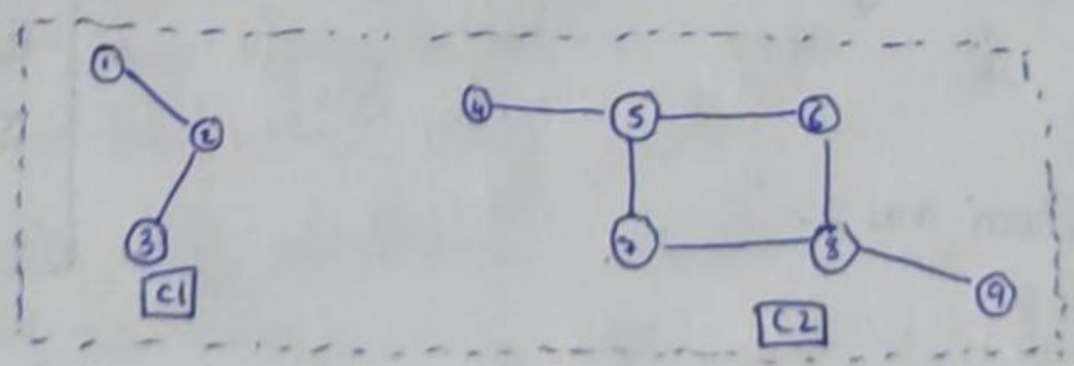
```
void dfs ( int node,   map<int, set<int>> adj,  vector <int> & ans,  map <int, bool> visit) {
        visit [node] = 1;
        ans. push_back (node);
        for ( int it: adj[node]) {
            if (! visit [it]) {
                dfs(it, adj, ans, visit);
            }
        }
    }
```

▣ Time Complexity → O(N+E)

▣ Space Complexity → O(N+E)

```
void DFS  ( int v, map <int, set<int> ) {
        map < int, bool > visit;
        vector <int> ans;
        for (int i=0; i<v; i++) {
            if (! visit[i]) {
                dfs ( i, adj, ans, visit);
            }
        }
    }
```

⊙ Cycle Detection in Undirected Graph (BFS).

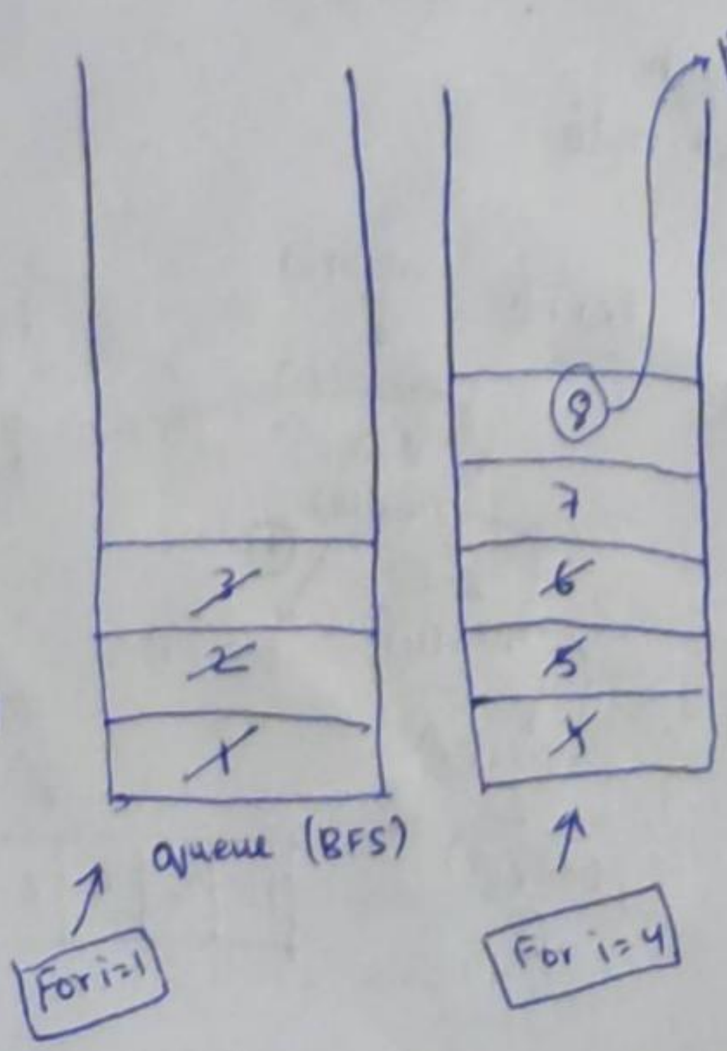


neglect a cycle ...

visited & parent

Adjacency list

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



parent

x → y	
1 → -1	
2 → 1	
3 → 2	
4 → -1	
5 → 4	
6 → 5	
7 → 5	
8 → 6	

Src=1

Src=4

Visited (Map)

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

visit = TRUE and node != parent → cycle.

Code :-

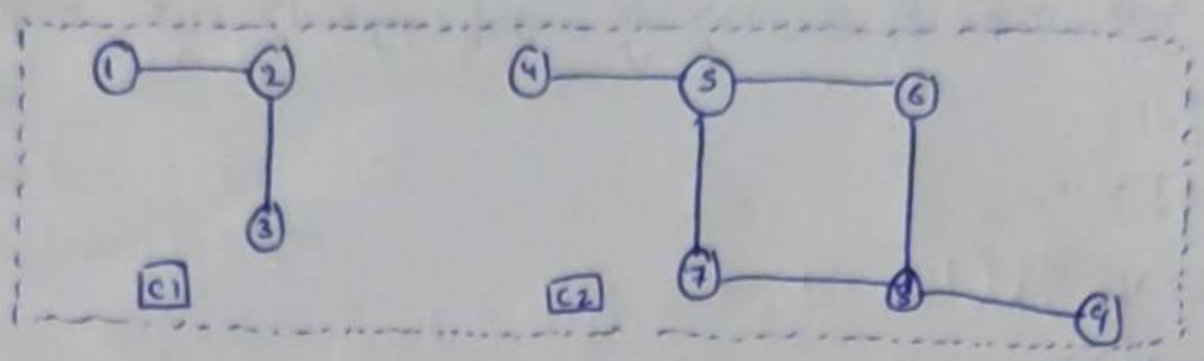
```
bool check_cycle (int node, map<int, set<int>> adj, map<int, bool> visit, map<int, int> parent){
        visit [node] = 1;
        parent [node] = -1;
        queue <int> q;
        q.push (node);
        while (!q.empty()){
                int front node = q.front();
        q.pop();
        for( auto it : adj[front node]){
            if(!visit [it]){
                    q.push (it);
                    visit [it] = 1;
                    parent [it] = frontnode;
            }
            else if (visit [it] == 1 && parent [front node] != it){
                    return true;
            }
        }
        return false;
}

void BFS (int v, map <int, set<int>> adj){
        map <int, bool> visit;
        map <int, int> parent;

        for (int i=0; i<v; i++) {
                if(!visit [i]){
                        if(check_cycle (i, adj, visit, parent)){
                                return true;
        }}}}
        return false;
}
```

main cyclic detection condition

- Time Complexity :- O(N+E)
- Space Complexity :- O(N+E)

⊙ Cycle detection in a undirect Graph → (DFS).

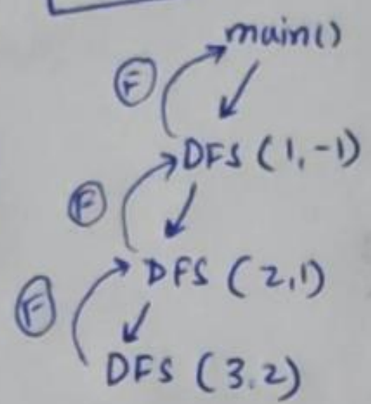
Graph Gr.



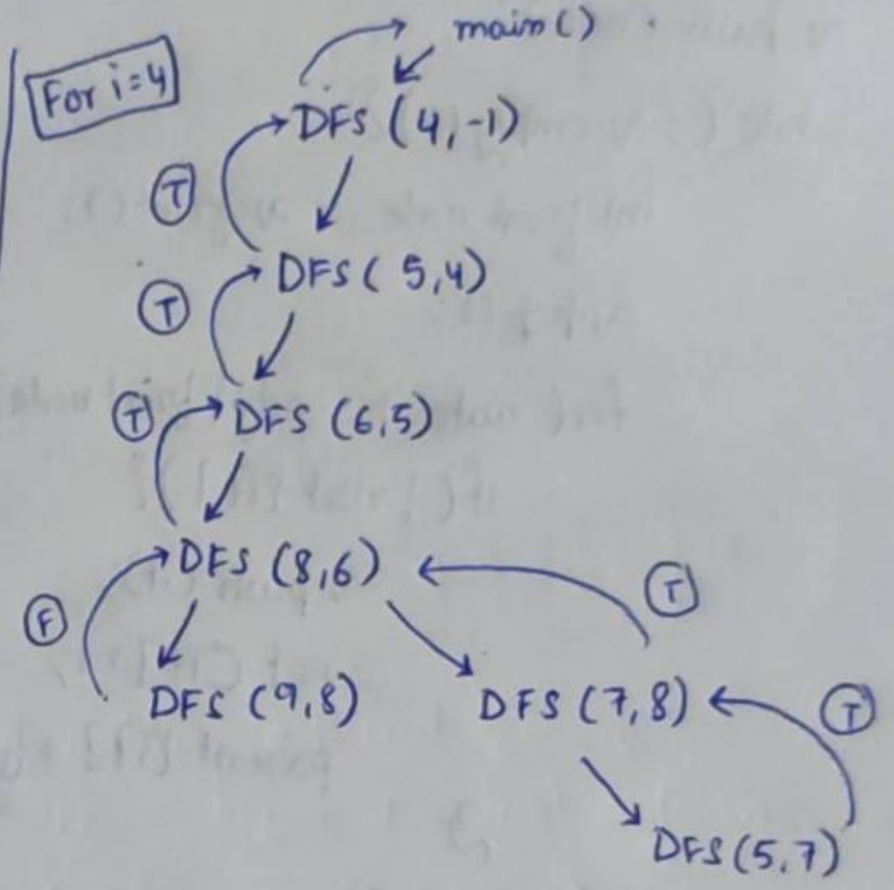
⊡ Dry run →

Parent(variable)	Visited
1 → -1	1 → T
2 → 1	2 → T
3 → 2	3 → T
4 → -1	4 → T
5 → 4	5 → T
6 → 5	6 → T
8 → 6	8 → T
9 → 8	9 → T
7 → 8	7 → T

For i = 1



For i = 4



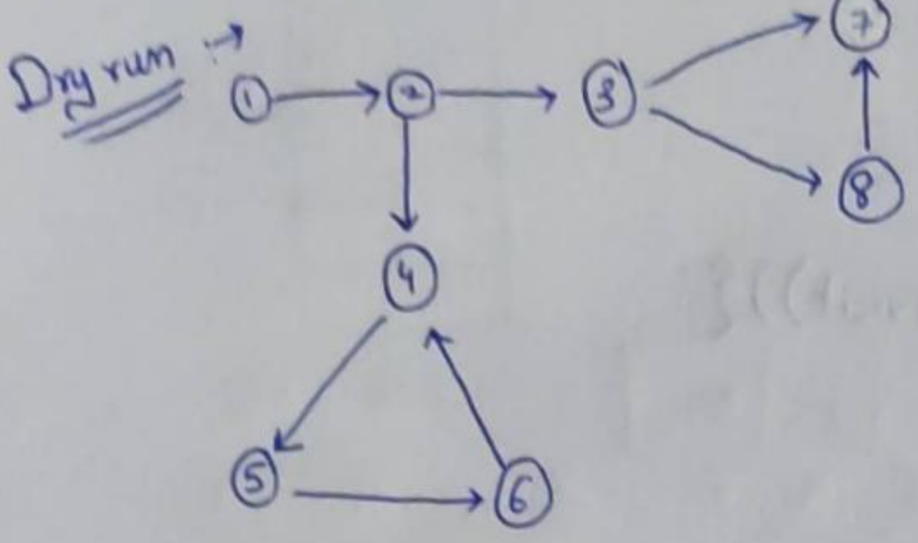
⊙ Code →

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

bool DFS (int v, map <int, set<int> adj) {
        map <int, bool> visit;
        for (int i=0 ; i<v ; i++) {
            if ( check ( i, -1, adj , visit ) {
                return true;
            }
        }
        return false;
    }
```

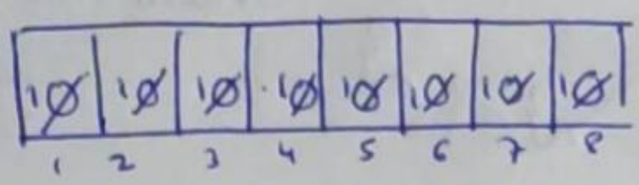
main cyclic detection condition

- ⊡ Time Complexity → O(N+E)
- ⊡ Space Complexity → O(N+E)

⊙ Cycle detection in a (directed Graph) → (DFS)



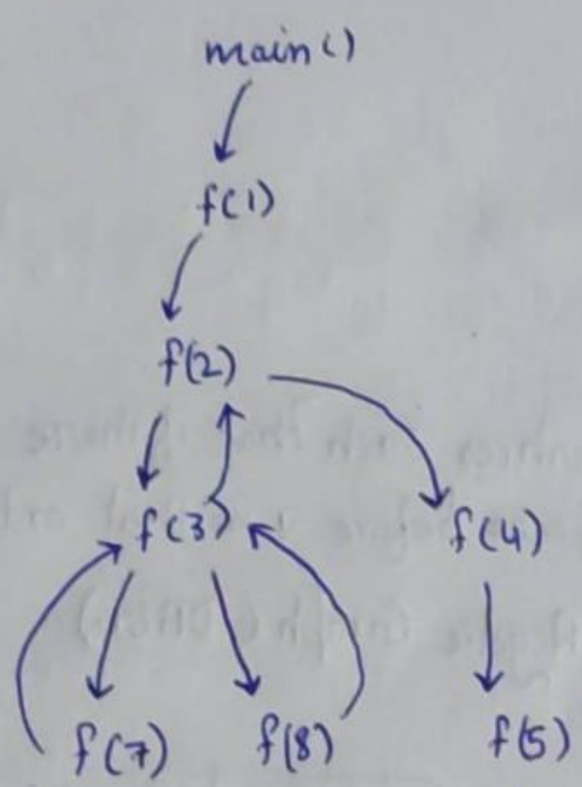
⊙ Time Complexity → O(N+E)
⊙ Space Complexity → O(N+E)

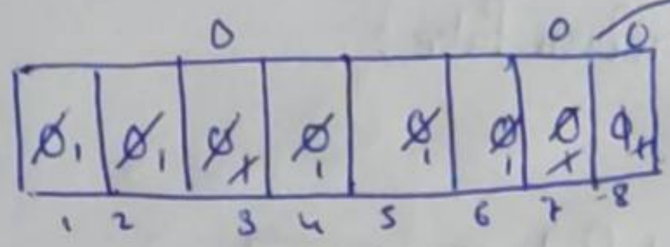


→ visited array/ map

⊡ adjacency List →

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





→ dfs visited/ map.

⊡ visit [node] = True
+ dfs visit [node] = True

⇓

main Cyclic condition.

⊙ Code →

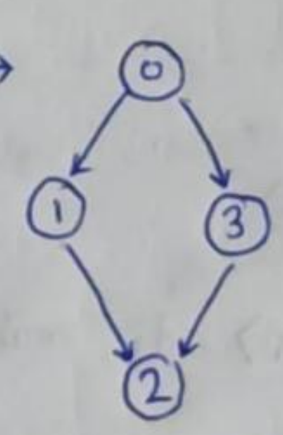
```
bool check_cycle (int node, map<int, set<int>> adj, map<int, bool> visit, map<int,bool> dfsvisit){
    visit [node] = 1;
    dfs visit [node] = 1;
    for (int it : adj [node]){
        if(!visit[it]){
            if(check_cycle(it, adj, visit, dfs visit)) {
                return True;
            }
        }
        else if (dfs visit[it] == 1){
                return true;
        }
    }
    dfs visit [node] = 0;
    return false;
}
```

```
bool DFS( int v, map<int, set<int>> adj){
        map<int, bool> visit;
        map<int, bool> dfsvisit;

        for(int i=0; i<v; i++){
            if(!visit[i]){
                if ( check_cycle( i, adj, visit, dfsvisit)){
                    return true;
                }
            }
        }
    return fake;
}
```

⊙ Topological Sort → DFS

- ⊡ Linearning order of vertices such that if there is an edge u→v, u appears before v in that ordering.
- ⊡ Applied in Directed Acyclic Graph (DAG).

Example →



0 1 3 2 → valid Topological Sort

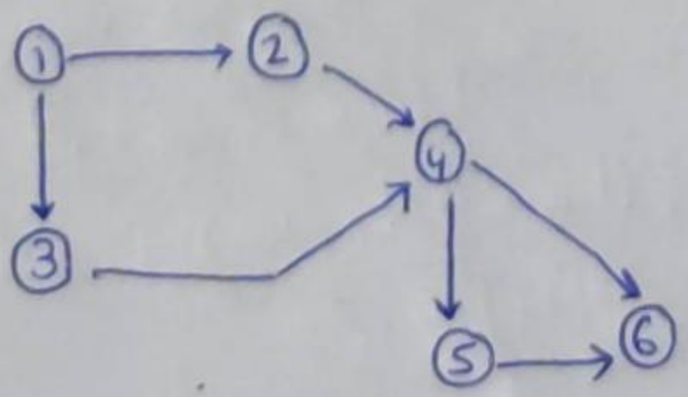
3 2 1 0 → Invalid.

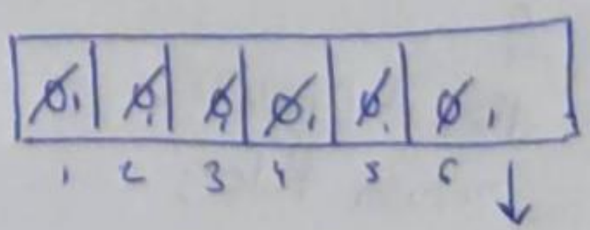
- ⊡ Graph should be directed else there would be 2 directions between the edges
- ⊡ Graph cannot be cyclic as in cyclic graph there is a dependency factor, which means that we cannot be sure that there is linear ordering of vertices ofr not.

⊡ Dry Run →

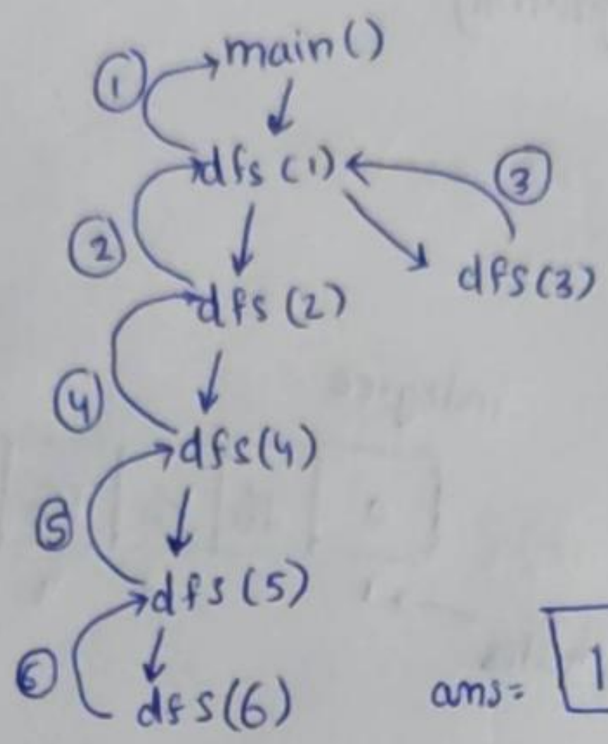
⊡ Adjacency List

1 → 2, 3
2 → 4
3 → 4
4 → 5, 6
5 → 6
6 → –





Stack



Time complexity = O(N+E)
Space Complexity = O(N+E)

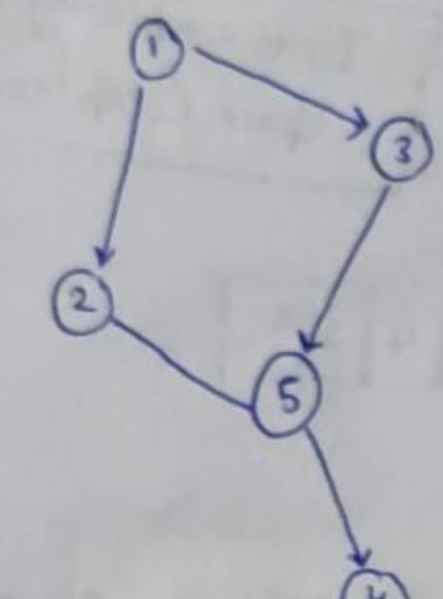
ans = | 1 | 3 | 2 | 4 | 5 | 6 |

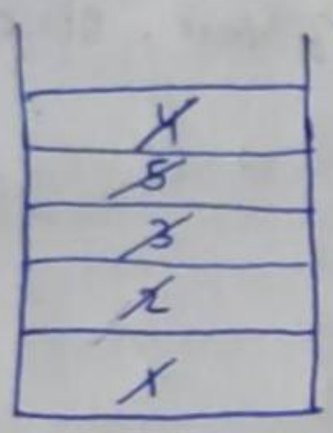
⊙ Code →

```
void topoSort (int node, map <int, set <int>> adj, map <int, bool> & visit, stack <int> s) {
    visit [node] = 1;
    for (int it: adj [node]){
        if( ! visit [it]){
            topo (it, adj, visit, s);
        }
    }
    s. push (node);
}

void DFS (int v, map<int, set<int>> adj){
    map <int, bool> visit;
    stack <int> s;
    for ( int i=0; i<v ; i++) {
        if (! visit [i]){
            topo ( i, adj, visit, s);
        }
    }
    vector <int> ans;
    while (! s.empty()) {
        ans. push_back (s.top());
        s.pop();
    }
}
```

⊙ Topological Sort (BFS) ⇢ (Kahn's Algorithm)

⊡ Example ⇢





indegree

0	✗0	✗0	✗0	✗0
1	2	3	4	5

nodes →

adjacency list

1 ⟶ 2, 3

2 ⟶ 5.

3 ⟶ 5

5 ⟶ 4

4 ⟶

ans =

1	2	3	5	4

⊡ Important Steps ⇢

⊡ find indegree of all nodes and store it in vector.

⊡ Inside queue ⇢ insert the nodes having the indegree as 0.

⊡ do BFS Algo (same approach), then push the poped node in ans vector and then check our adjacent nodes. and then decrement indegree of our adjacent nodes. After decrement we check if the indegree of that node is zero, then we push it to our queue.

⊡ ① Time Complexity ⇢ O(N+E)

⊡ ② Space Complexity ⇢ O(N+E).

⊡ Code ⇢

```
void BFS (int v, map<int, set<int>> adj){

    vector<int> indegree (v);
    for (auto i: adj) {
        for(auto j: i.second) {
            indegree[j]++;
    } }
```

```
queue <int> q;
for (int i=1 ; i<v; i++){
        if (indegree [i] ==0){
                q.push(i);
        }
}

vector <int> ans;
while (! q.empty()) {
        int frontnode = q.front();
        q.pop();
        ans.push_back (frontnode);
        for (int it : adj [frontnode]) {
                indegree [it]--;
                if( indegree [it] == 0) {
                        q.push(it);
                }
        }
}
```

⊙ Detect a cycle in a ~~un~~directed Graph :→ (BFS)

- (i) Time complexity → O(N+E)
- (ii) Space complexity → O(N+E).

Important steps →

(i) Concept is exactly same a Kahn's Algorithm.

(ii) You just need to make changes is here you donot need the ans vector. Instead, use a count variable and increment it value when you writing the BFS algo.

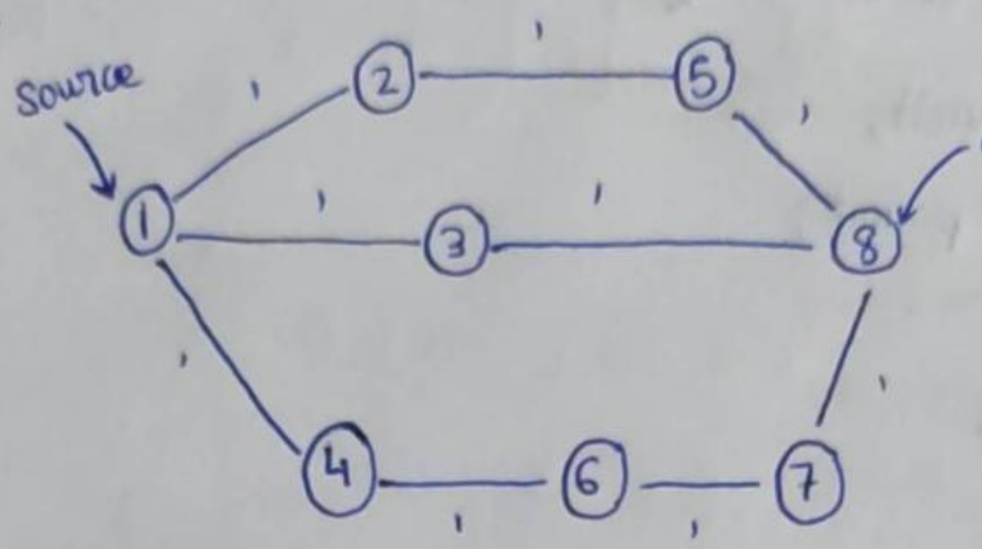
Now, After BFS algo if value of our count == no of vertices, then not a cyclic graph we return false else, we return true.

Code →

```
int vod BFS (int v, map<int,Set<int>> adj) {

        vector <int> indegree (V);
        for (auto i: adj) {
            for (auto j : i.second) {
                indegree [j] ++;
            }
        }

        queue <int> q;
        for (int i=1; i<v ; i++) {
            if (indegree [i] == 0) {
                q.push (i);
            }
        }

        int count = 0;
        while (! q. empty() ) {
            int frontnode = q. front();
            q. pop();
            count++;
            for (int it : adj [frontnode]) {
                indegree [it]--;
                if ( indegree [it] == 0) {
                    q.push (it);
                }
            }
        }
        if ( count == v) {
            cout << "False";
        }
        else {
            cout << "Tome";
        }
}
```

⊙ Shortest path in undirected Graphs →

Example ① →

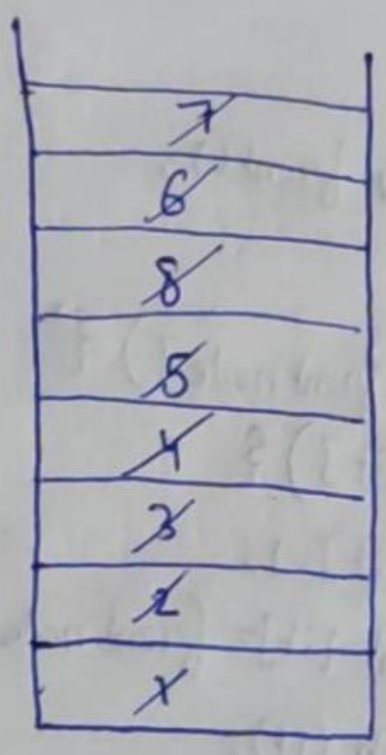


visited

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

parent

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



queue.

front = 7
front = 6
front = 8
front = 5
front = 4
front = 3
front = 2
front = 1

⊡ adjacency list.

1 → 2,3,4
2 → 1,5
3 → 1,8
4 → 1,6
5 → 2,8.
6 → 4,7
7 → 6,8
8 → 3,5,7.

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

= 8 ← 3 ← 1 → Shortest path.

1 → 3 → 8 → Reverse

⊡ Important steps → (i) Create a adjacency list.

(ii) do bfs algo and using visited and parent map. To find parent of all nodes.

(iii) we will backtrack our parent map in such a way, that until our current node == source (currentnode is initialised with destination node), we will update our current node with parent[currentnode] and then push it to our ans array.

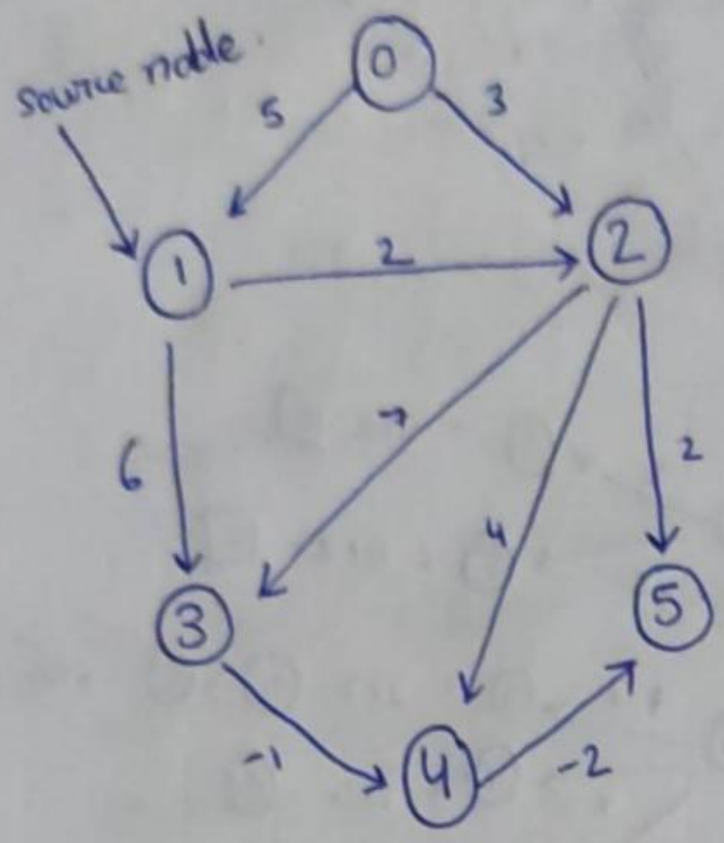
(iv) Lastly reverse the ans array to get the desired shortest path.

⊡ Code →

```
Void BFS (int v, map <int, set <int>> adj, int s, int t){
        map <int, bool> visit;
        map <int, int> parent;

    queue <int> q;
     q.push (s);
     visit [s] = 1;
     parent [s] = -1;
     while (! q.empty ()) {
          int front node = q.front ();
          q.pop ();
          for ( auto it : adj [front node]) {
               if (! visit [it]) {
                    visit [it] = 1;
                    parent [it] = front node;
                    q.push (it);
               }
          }
     }

    vector <int> ans;
     int current node = t;
     ans. push_back (t);
     while ( current node != s) {
          current node = parent [current node];
          ans. push_back ( current node);
     }
     reverse (ans. begin (), ans. end ());
     return ans;
}
```

⊙ Shortest path in DAG →

⊡ Example →



Shortest path from source to other nodes.

1 → 0 → INF

1 → 1 → 0

1 → 2 → 2

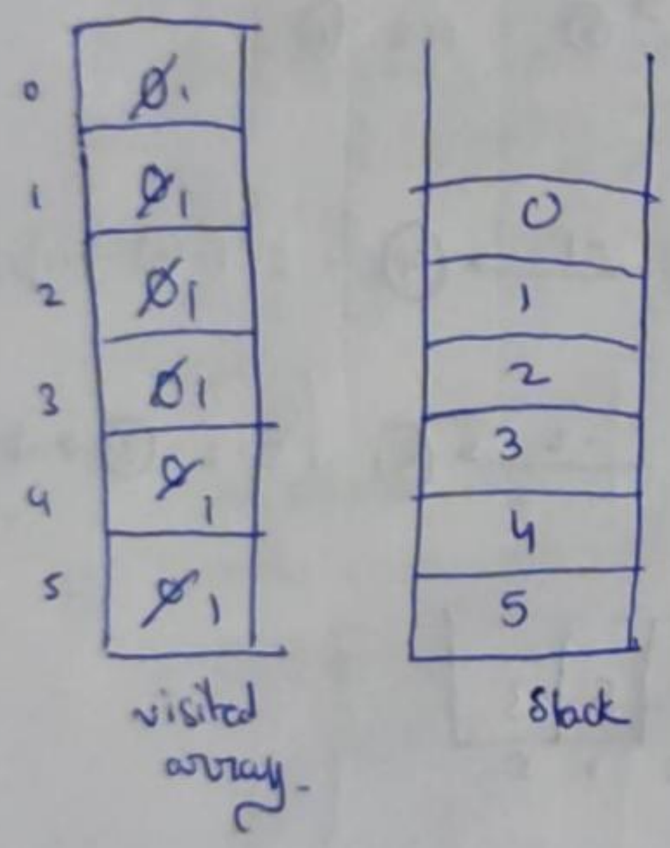
1 → 3 → 6

1 → 4 → 5

1 → 5 → 3

Shortest path =

INF	0	2	6	5	3



Adjacency list

0 → [1,5].

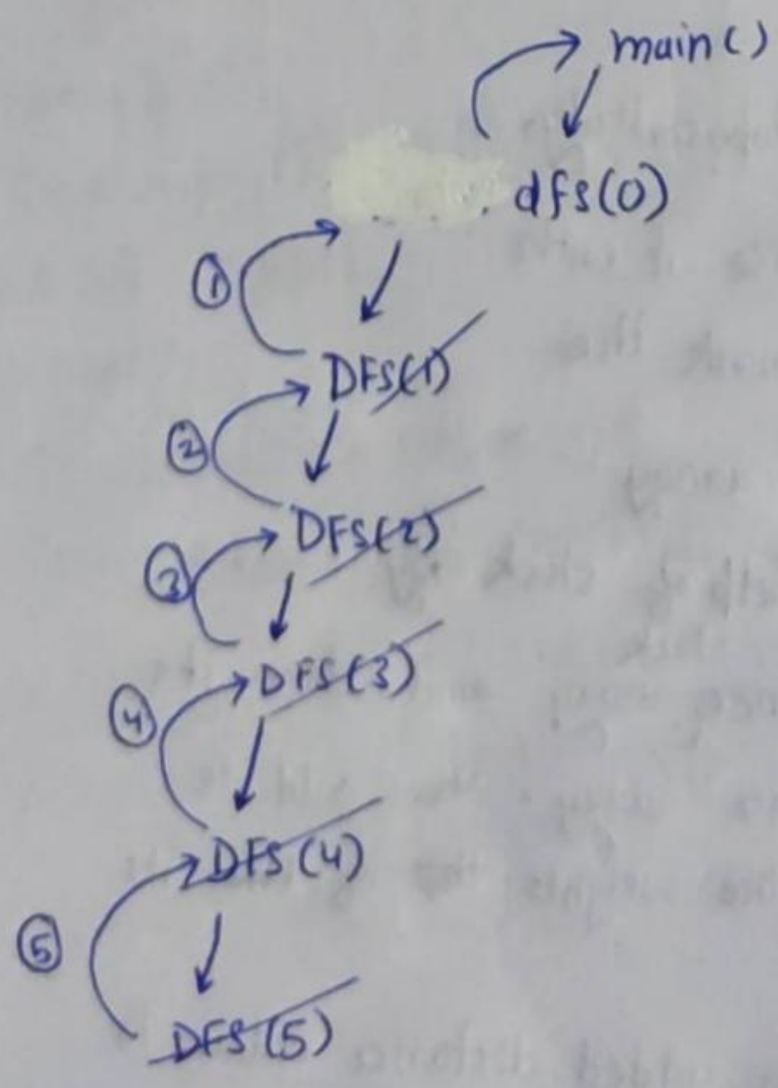
1 → [2,2], [3,6]

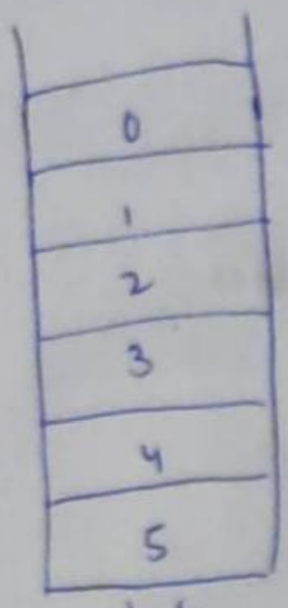
2 → [3,7] [4,4] [5,2]

3 → [4,-1]

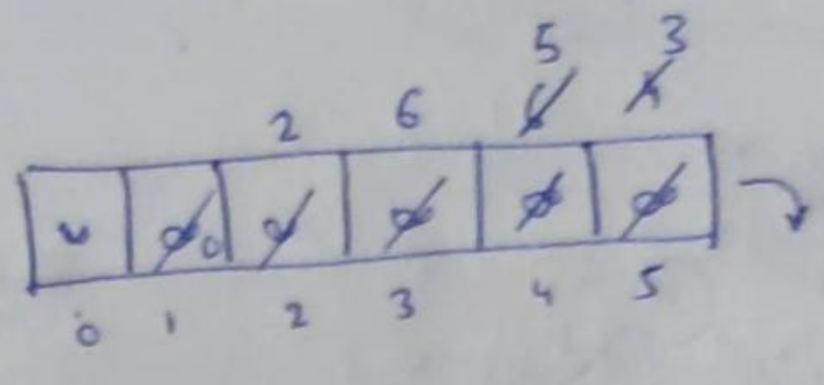
4 → [5,-2]

5 →





stack after Topo sort using DFS.

dist array = 

mark dist [source] = 0

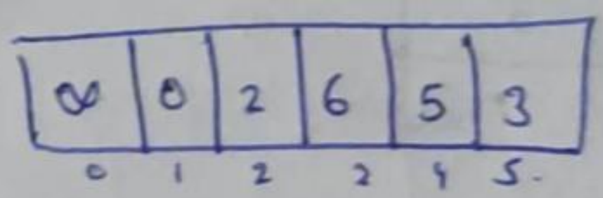
top = 0 →

top = 1 → 0 = ① —2→ ② = 0+2 = 2
① —6→ ③ = 0+6 = 6

top = 2 → 2 = 2 ② —7→ ③ = 2+7 = 9 > 6 → No change
② —4→ ④ = 2+4 = 6
② —2→ ⑤ = 2+2 = 4

top 3 → 6 = 6 ③ —-1→ ④ = 6-1 = 5 → update.

top 4 → 5 = 5 ④ —-2→ ⑤ = 5-2 = 3 → update.

∴ dist array = 

⊙ Important Steps →

① Get a topo sorted stack using toposort algo.

②(i) Create a distance array initialise it with infinity at initial stage and mark the dist [source] = 0 in distance array.

(ii) Update distance array with help of stack by .....

- pop top element from ~~distance array~~ stack and store its distance value from distance array. Now, add its stored distance value with the weights of its adjacent nodes.
- Now, compare the newly added distance value to the distance value of that particular node present in distance array. If the newly added distance value is less found, please update the distance array with it.

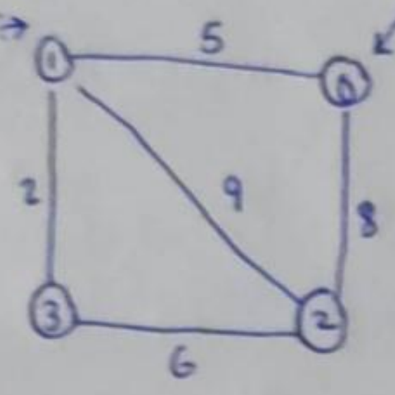
⊡ Code →

```
void topo ( int node, map <int, set < pair <int, int>>> adj , map <int, bool> & visit , stack <int>& s) {
    visit [node] = 1;
    for (auto it: adj [node]){
        if(! visit [it. first]) {
            topo (it. first, adj , visit , s);
        }
    }
    s. push (node);
}

void DFS (int v, map <int , set<pair <int, int>>> adj) {
    map <int, bool> visit;
    stack <int> s;

    for (int i=0 ; i<v ; i++) {
        if (! visit [i]) {
            topo (i, adj , visit , s);
        }
    }

    int source = 1;
    vector <int> dist (v);
    for (int i=0 ; i<v ; i++) {
        dist [i] = INT_ MAX;
    }
    dist [source] = 0;
    while (! s. empty()) {
        int top = s. top();
        s. pop();
        if( dist [top] ! = INT_ MAX) {
            for( auto i : adj [top]) {
                if ( dist [top] + i. second < dist [i. first ]) {
                    dist [i. first ] = dist [top] + i. second;
                }
            }
        }
    }
}
```

vector <int> dist (v, INT_MAX);

⊙ Dijastra Algorithm

Example ① →



adj list

0 → [1,5], [2,8]

1 → [0,5], [2,9], [3,2]

2 → [0,8], [1,9], [3,6]

3 → [1,2], [2,6].

Shortest paths

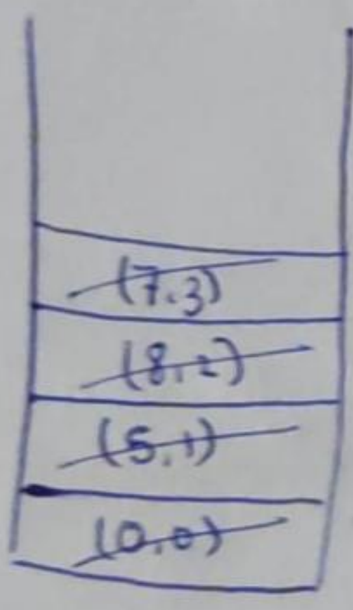
0 → 0 → 0

0 → 1 → ⑤

0 → 2 → ⑧

0 → 3 → ⑦.

{0,5,8,7} := ans.



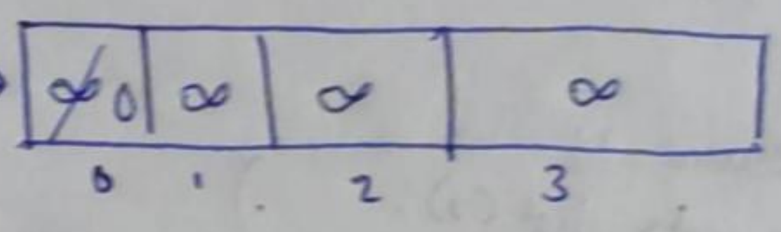
Set.

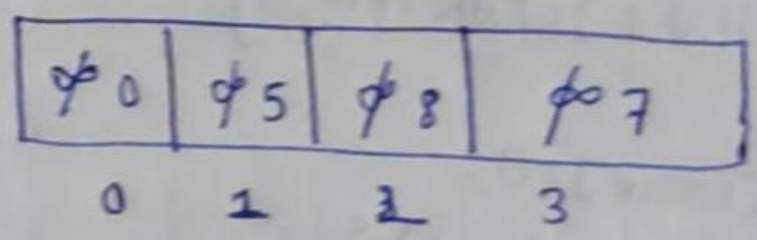
pair<int, int>

distance from dist array

node.

dist [source] = 0





(0,0)

top → 0 → ⓪ —5→ ① = 0+5 = update.

—8→ ② = 0+8 = update.

(5,1)

top → 5 → ① —5→ ① = 5+5 = 10

—9→ ② = 5+9 = 14

—2→ ③ = 5+2 = 7 update.

top (7,3) → 7 → ③ —2→ ① = 2+7 = 9

—6→ ② = 7+6 = 13.

top(8,2) → 8 → (2) —8→ (0) = 8+8 = 16
(2) —0→ (1) = 8+9 = 17
(2) —6→ (3) = 6+8 = 14.

⊙ Code →

```
void Dijkstra's (int v, map<int, set<pair<int,int>>> adj){

    int src = 0;
    priority_queue < pair <int,int>, vector < pair <int,int>>, greater< pair <int,int>> > pq;
    vector <int> distTo (v, INT_MAX);

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

while ( ! pq.empty()) {
    int frontdist = pq.top().first;
    int frontnode = pq.top().second;
    pq.pop();
    vector <pair <int,int>> :: iterator it;
    for ( auto it : adj[frontnode]) {
        int nextnode = it.first;
        int nextdist = it.second;
        if( distTo[nextnode] > frontdist + nextdist) {
            distTo[nextnode] = distTo[frontnode] + nextdist;
            pq.push (make_pair ( distTo[nextnode], nextnode)) ;
}}}
```

due to priority queue.

⊡ Time Complexity → O((N+E) log N)
=> O(E log N)

⊡ Space Complexity → O(N+E)

⊡ Important steps →

① Here we will use 2 datastructures one priority - queue using min heap concept to store values in format of <distance, node> and other a distance array

② Now, intialise the distance[source] → 0 and the entire distance array / vector by INT_MAX;

③ Now start poping out elements from priority - queue and and using it update distance array as previous algo.

⊙ Prim's Algorithm ↦

▣ Minimum Spaming Tree ↦ minimum cost of weights.

(i) Whenever we can convert a graph into a tree such that it contains n nodes and n-1 edges, then is called as a spanning Tree.

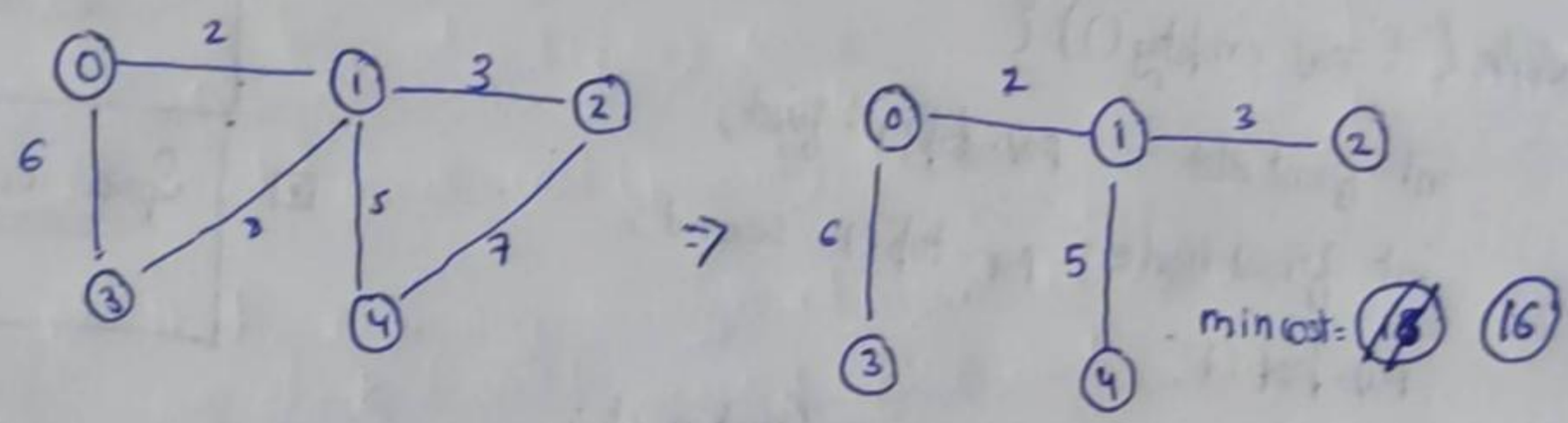
(ii) Every node is reachable from the other node.

(iii) To find spanning tree we can use ... -

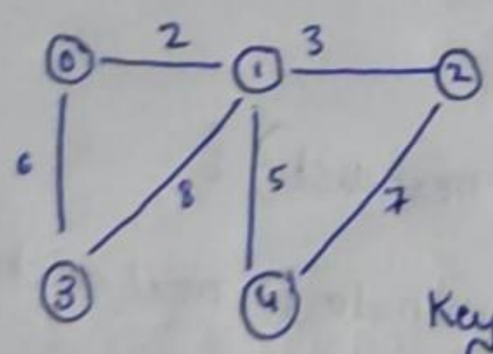
- ▣ Prims Algorithm
- ▣ Kruskals Algorithm.

⊗ ▣

Example ⇢



(1) Prims Algorithm ⇢



Key [0] = 0
parent [0] = -1

Key

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

mst

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

parent

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

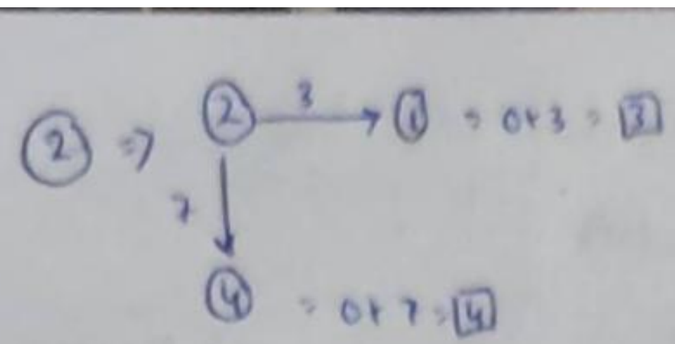
(i) → track the node having min value.

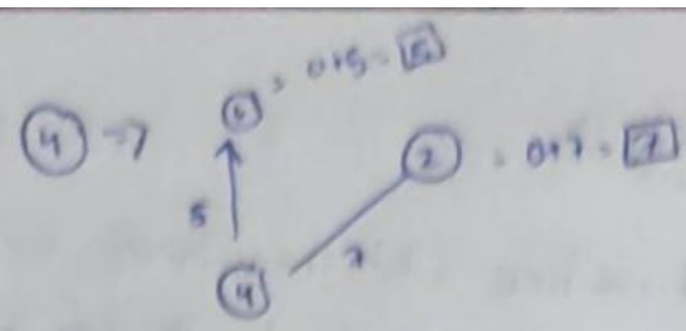
(ii) → mark the mst of min value node as true.

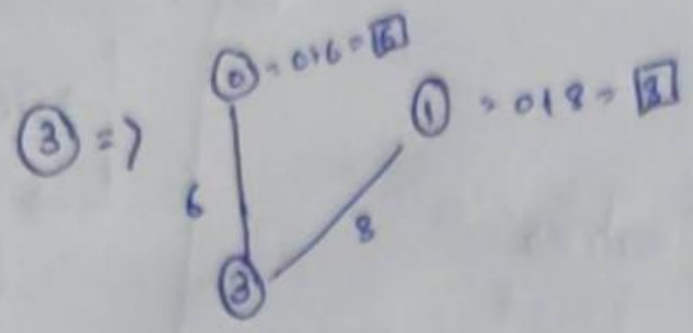
(iii) → adjacent nodes of min value node. Note their weight and if its less than value of adjacent node present in Key array, please update it

0 ⇒ 0 —2→ 1 : 0+2 = 2 update.
0 —6→ 3 : 0+6 = 6 update

1 ⇒ 1 —3→ 2 : 0+3 : 3 update
1 —5→ 4 : 0+5 : 5 update.







1. Code →

```
void Prime (int v, map<int, set<pair<int, int>>> adj)?
```

Replace with priority Queue to optimise solution.

```
{
    int source = 0;
    vector<int> key(v, INT_MAX);
    vector<bool> mst(v, false);
    vector<int> parent(v, -1);

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

    for (int i=0; i<v; i++) {
        int minnode = INT_MAX;
        int parentnode;
```

can be replace with priority Queue concept

```
        for(int j=1; j<v; j++){
            if(mst[j]==false && key[j]<minnode){
                parentnode = j;
                minnode = key[j];
            }
        }

        mst[parentnode] = true;

        for(auto it: adj[parentnode]){
            int nextnode = it.first;
            int nextweight = it.second;
            if(mst[nextnode]==false && nextweight < key[nextnode]){
                parent[nextnode] = parentnode;
                key[nextnode] = nextweight;
            }
        }
    }
    int cost=0;
    for(auto it: key){
        cost += it;
    }
    cout<<cost;
}
```

⊙ Important Steps :→

(i) We will use here 3 data structures, i.e., 3 vectors ...

- ⊡ Key vector, initialised with INT_MAX.
- ⊡ Mst vector, initialised with false
- ⊡ parent vector, initialised with -1.

There size should be equal to the no. of vertices.

(ii) Now, initialise key[source] = 0, and parent[source] = -1.

(iii) Now on traversing each vertices, we will ...

- ⊡ First, calculate the min node + parent node by using key and mst (false) vector.
- ⊡ Second, mark mst[parent node] = true.
- ⊡ Third, traversing through adjacent nodes we check (mst[next node] == false and next < key[next node]), if it holds true then we assign parent node in parent vector and weight of adjacent node to key vector.

⊡ Time complexity :→

- → O(N)²
- → optimise → O(N log N)
- → using priority queue (min heap).

⊡ Space complexity :→

O(N+E).

⊙ Disjoint Set + Kruskal's Algorithm :→

⊡ Disjoint Set :→ ⚠ It is basically a data structure. It has 2 major operations ....

(i) Find Parent() or, findSet() :→ It tells us about nodes parent

(ii) Union() or, UnionSet() :→ It can do the union of two graph components ..

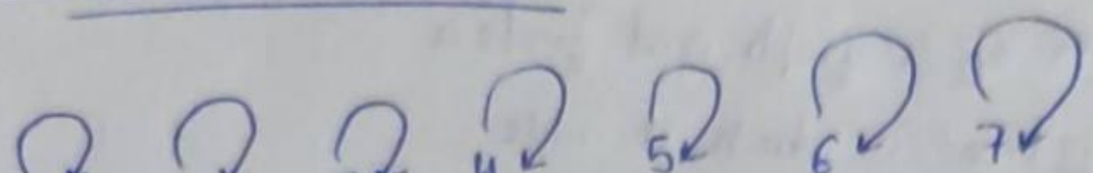
⚠ Disjoint Set use :→ (i) Implement Kruskals Algo.

(ii) Cycle detection.

(iii) If 2 random nodes are present in same component or not (verification).

Union by Rank and Path Compression :→

components of graph :



union (1,2) :

	parent	Rank
1 →	1	0
2 →	2	0

operation
(i) parent[2] = 1;
(ii) rank[1] ++;

union (2,3) :

	Rank
2 → 1	1
3 → 3	0

operation
(i) rank[3] < rank[1];
parent[3] = 1;

union (4,5) :

	Rank
4 → 4	0
5 → 5	0

operation
(i) parent[5] = 4;
(ii) rank[4] ++;

union (6,7) :

	Rank
6 → 6	0
7 → 7	0

operation
(i) parent[7] = 6;
(ii) rank[6] ++;

union (5,6) :

	Rank
5 → 4	1
6 → 6	1

Operation
(i) parent[6] = 4
(ii) rank[4] ++;

union (3,7) :

	Rank
3 → 1	1
7 → 4	2

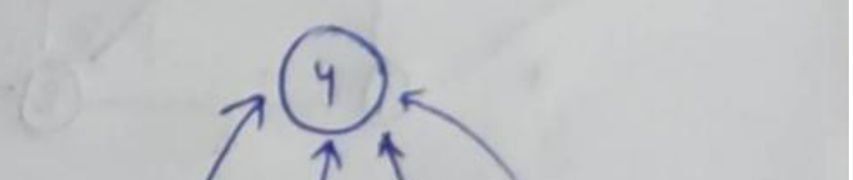
(i) rank[1] < rank[4];
parent[1] = 4;

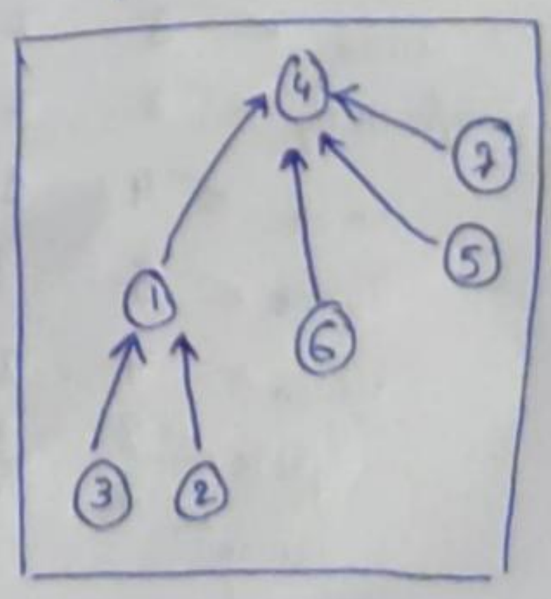
} Apply path compression logic after this operation.

Find parent (1) → 1
Find parent (2) → 2
Find parent (3) → 3
Find parent (4) → 4
Find parent (5) → 5
Find parent (6) → 6
Find parent (7) → 7

rank.

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





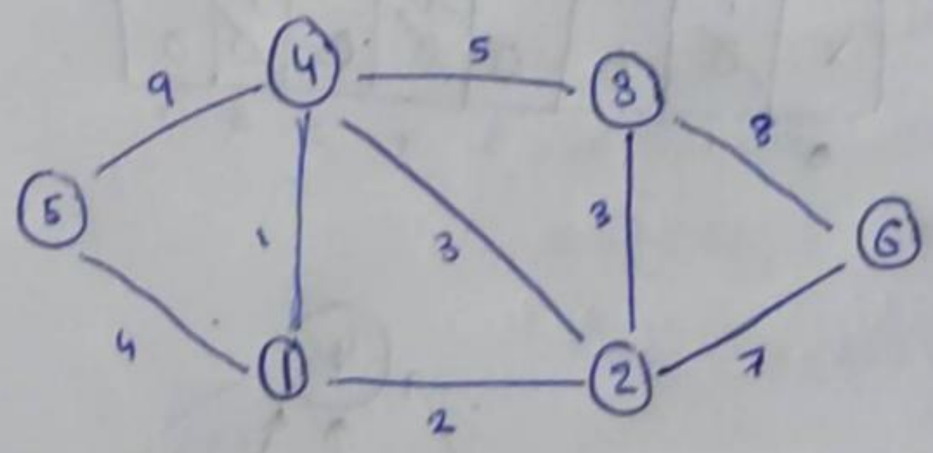
⊙ Note →

Here, the rank of whichever parent is more is considered to be a big tree. And we should place a small tree below a big tree as if we place a short tree below the big one then, our dept decreases but, if we do the opposite then dept increases.

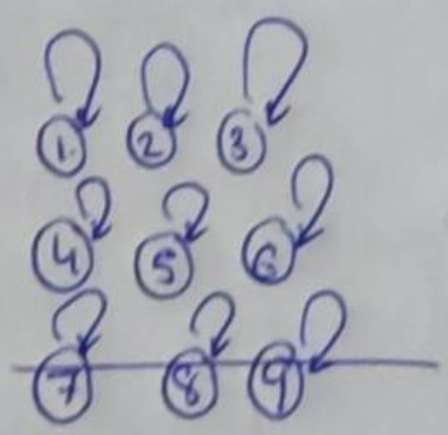
⊙ Note ⇒ ⊡ Union by rank ⇒ It compares different component trees of a graph and finds a big tree in which it adds the short tree.

⊡ Path compression ⇒ It reduces the depth to find the topmost parent node of a particular node by directly adding that node to its topmost parent node.

⊙ Kruskals Algorithm ⇒



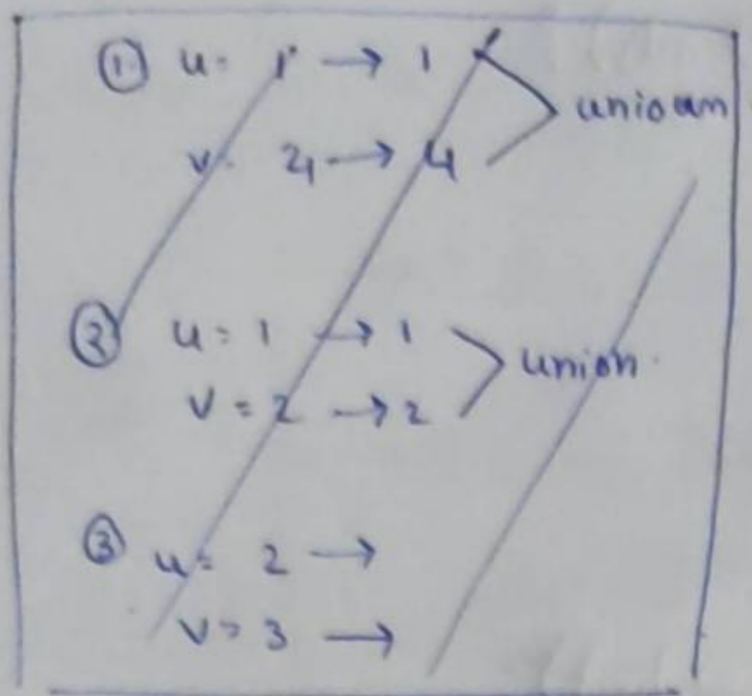
Sorted	wt	u	v
←	1	1	4
union ←	2	1	2
union ←	3	2	3
union ←	3	2	4
ignore/nothing ←	4	1	5
union ←	5	3	4
ignore ←	7	2	6
union ←	8	3	6
ignore ←	9	4	5

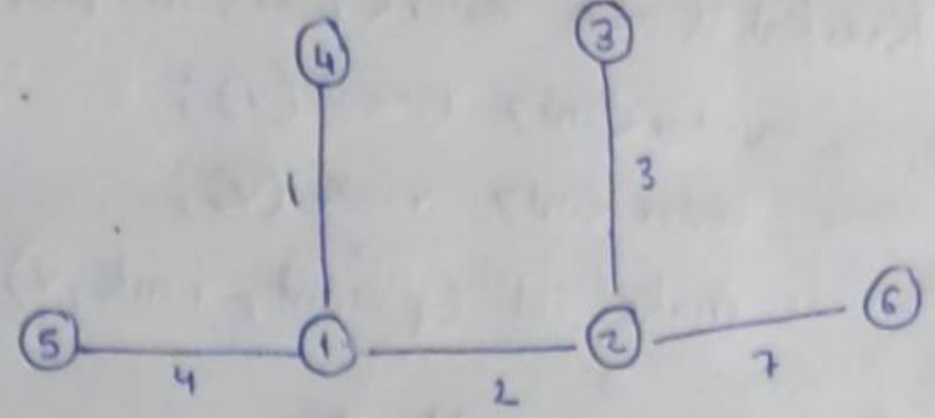


⊡ Import points ⇒

(i) Here, we do not use the adjacency list but, rather we prefer to use a linear data structure (storing wt, u, v), which is sorted in ascending order of wt.

(ii) At beginging every node will be a parent of itself, now we will take parent of our nodes u and v, if the parent of u and v are different we apply union operation, else we do nothing.





↓

Minimum spanning tree :
min cost = (17)

- Time Complexity ⇒ $O(E \log N)$
- Space Complexity ⇒ $O(N)$

Code ⇒

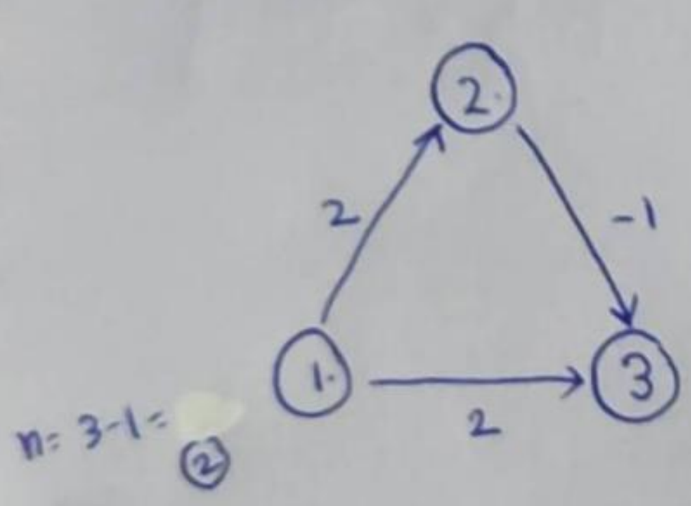
```
void make_set (vector<int>& parent, vector<int>& rank, int v){
    for (int i=0; i<v+1; i++){
        parent[i]=i;
        rank[i]=0;
    }
}

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

void unionSet (int u, int v, vector<int>& parent, vector<int>& rank){
    u=findParent (parent, u);
    v=findParent (parent, 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 Kruskal (int v, vector< pair<int, pair<int,int>>> adj) {
        vector<int> parent(v);
        vector <int> rank (v);
        make_set (parent, rank, v);

        int minweight = 0;

        for (auto i : adj) {
                int u = findParent ( parent, i.second.first);
                int v = findParent ( parent, i.second.second);
                int wt = i.first;

                if(u!=v) {
                        minweight += wt;
                        unionSet (u,v, parent, rank);
                }
        }

        cout << min weight;
}.
```

⊙ Bellman Ford →

- ⊡ It is a shortest path finding algorithm.
- ⊡ As, our Dijkstra's Algorithm cannot be applied in graph having negative weights, so we use Bellman Algorithm.
- ⊡ Moreover, to apply Bellman Ford Algorithm, our graph should not have negative cycle.
  But, we can find negative cycle using Belmann Ford Algorithm.

⊙ Dry Run: →



n = 3-1 = ②

dist → dist[source] = 0

∞	∞	∞
1	2	3

Part 1:

(n-1) times

```
if (dist[source] + weight < dist[dest]) {
    dist[dest] = dist[source] + weight;
}
```

1 → ② update 2

0+2 < ∞ → ② update 2

0+2 < ∞ → ② update 3

2+(-1) < ∞ → ① update 3

Part 2:

1 more time above concept → dist ⇒ update

Negative Cycle present. ← Shortest path (Do not exist.)

□ Code :→

```
void Bellmann ford (int v, vector <pair <int, pair <int ,int > >> adj) {
    int source = 0;
    vector <int> dist (v, INT_MAX);
    dist [source] = 0;
    for (int i = 1 ; i<= v; i++) {
        for ( auto it: adj) {
            int src = it.first;
            int dest = it. second. first;
            int weight = it . second . second;
            if( dist[ src ] != INT_MAX && (dist [src] + weight < dist [dest])){
                dist [dest] = dist [src] + weight;
            }
        }
    }

    bool flag=0;
    for ( auto it: adj){
        int src = it.first ;
        int dest = it . second.first ;
        int weight = it. second . second;
        if ( dist[src] != INT_MAX && ( dist [src] + weight < dist [dest])) {
            flag=1;
```

```
if (flag == 1) {
        cout << "Negative cycle";
}
else {
        cout << "No Negative cycle";
}
```

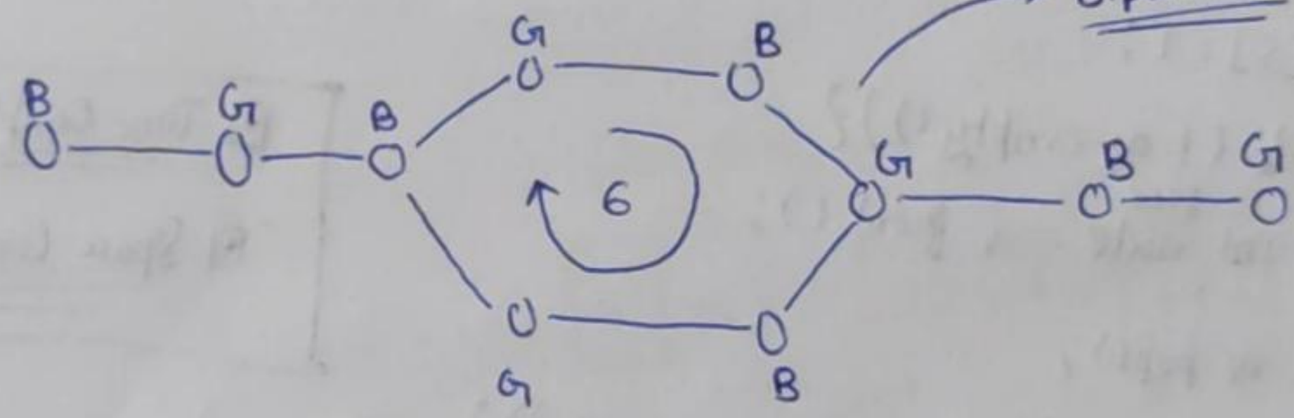
⊙ Time Complexity → O(NE)

⊙ Space Complexity → O(N)

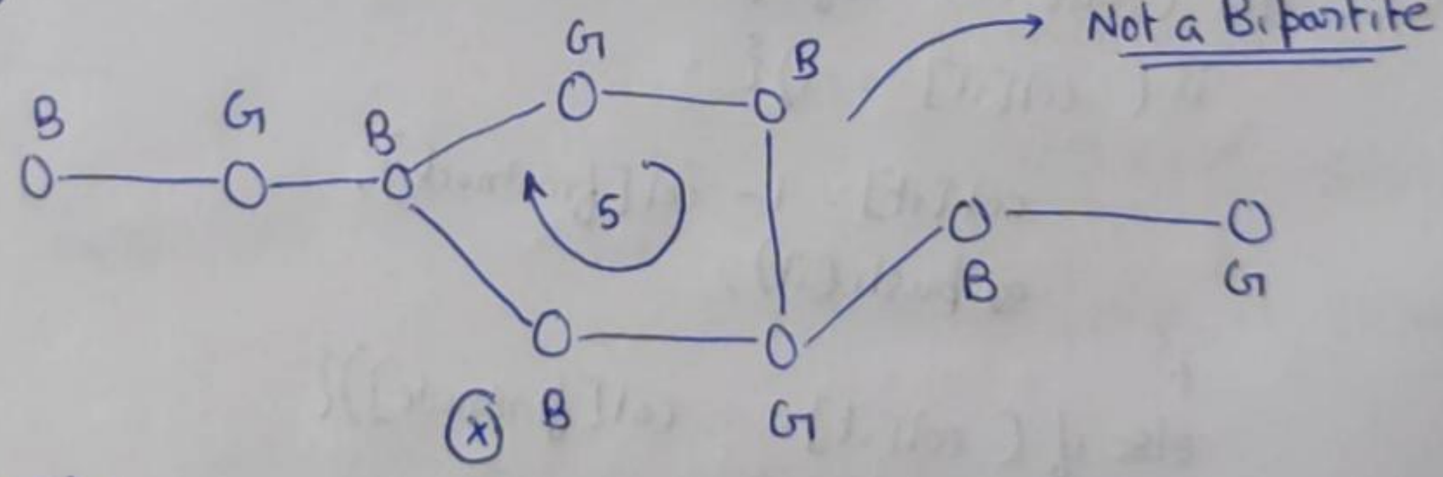
⊙ Bipartite Graph →

→ A graph that can be coloured using 2 colours such that no 2 adjacent nodes have same colour.

⊡ Example →



⊡ Example →



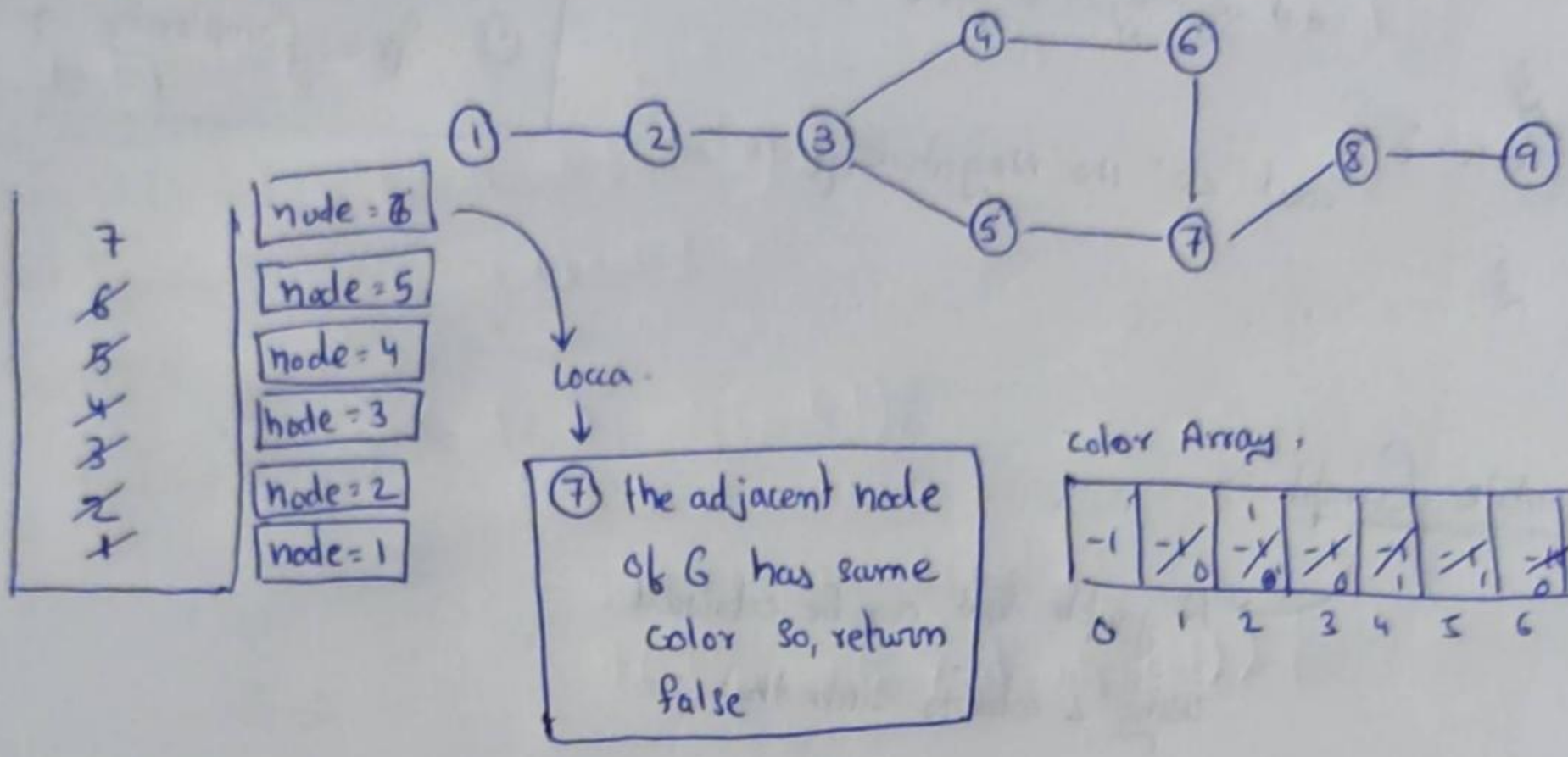
⊙ Important Note →

(i) A Graph having a even length is a Bipartite Graph.

(ii) A Graph having a odd length is not a Bipartite Graph.

→ cycle of edges

⊙ To Check Bipartite Graph (BFS) →



color Array :

0	1	2	3	4	5	6	7	8	9
-1	~~-1~~ 0	~~-1~~ 0	~~-1~~ 0	~~-1~~ 1	~~-1~~ 1	~~-1~~ 0	~~-1~~ 0	-1	-1

⊡ code : →

```
bool Bipartite (int s, vector <int> col, ~~vector~~ map <int, list <int>> adj){

    queue <int> q;
    q.push (s);

    col [s] = 1;

    while (! q.empty ()){
        int front node = q.front ();

        q.pop();

        for ( auto it : adj [frontnode]){
            if ( col[it] == -1){

                col[it] = 1 - col [frontnode];
                q.push (it);
            }
            else if ( col[it] == col [frontnode]){
                return false;
            }
        }
    }
    return true;
}

bool BFS_check ( int v, map <int, list<int>> adj){

    vector <int> col (v, -1);
    for (int i=0 ; i<v ; i++) {
        if ( col[i] == -1) {
            if (! bipartite (i, adj, col)) {
                return false;
            }
        }
    }
}
```

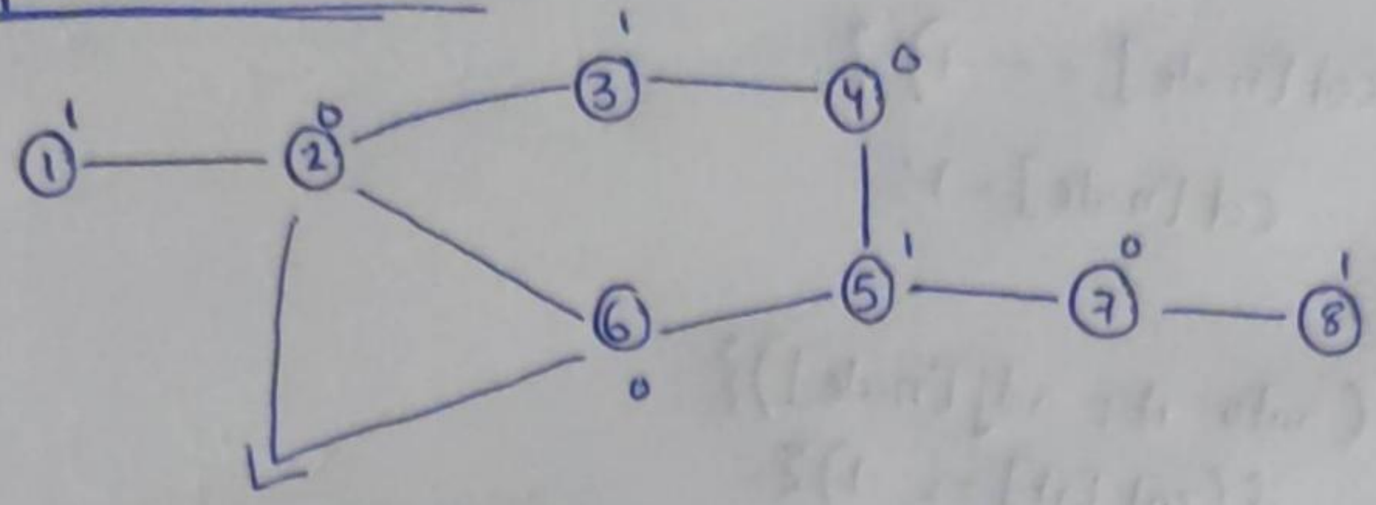
⊡ Time Complexity → O(N+E)

⊡ Space Complexity → O(N+E)

return true;

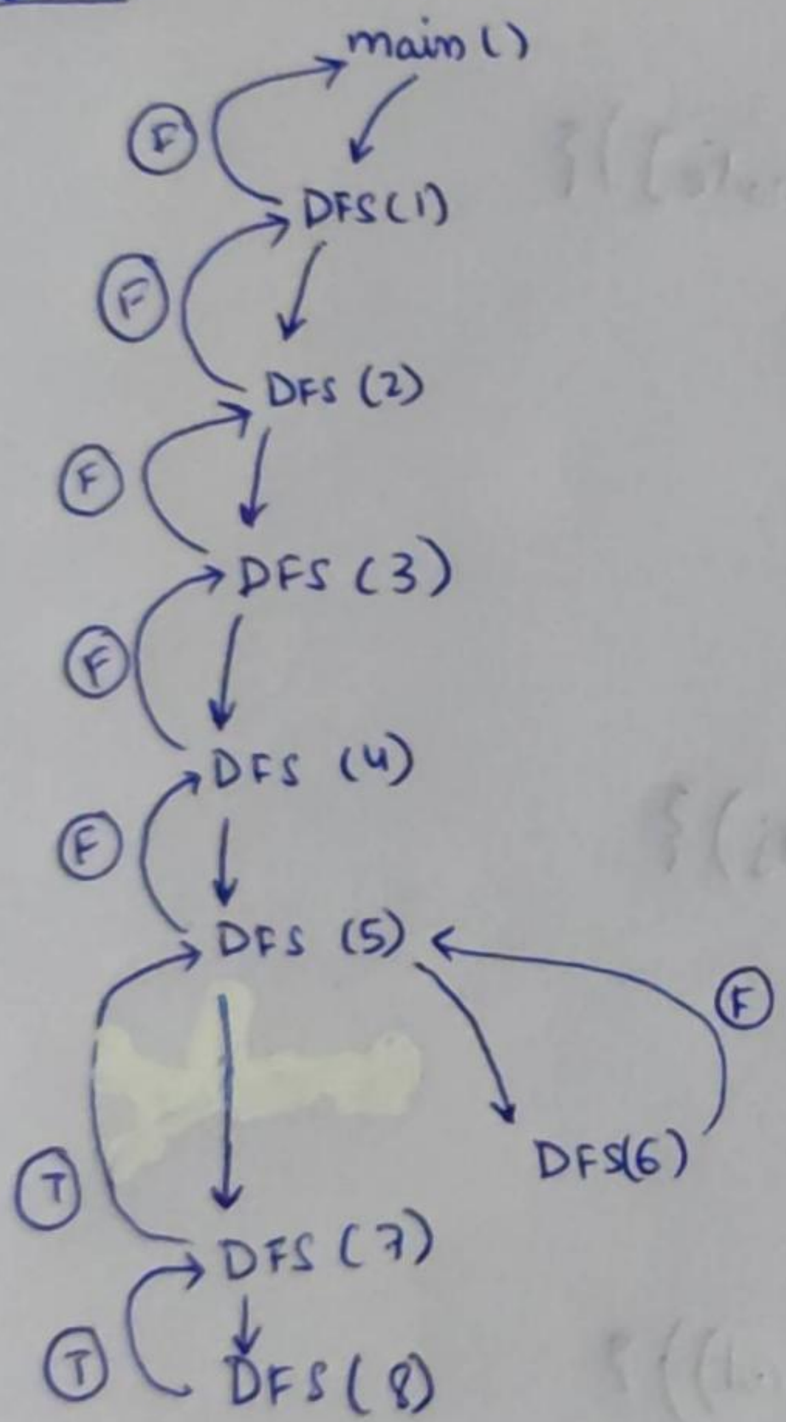
}

⊙ Check Bipartite or Not (DFS) :→

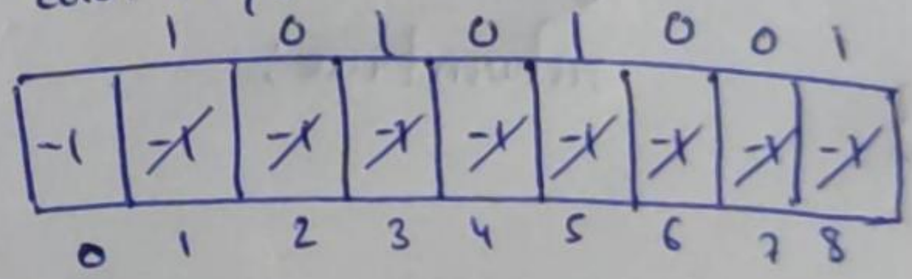


Two adjacent nodes have the same colour, so not a Bipartite.

⊡ Dry Run :→



color Array.



⊡ Time Complexity :→ O(N+E)

⊡ Space Complexity :→ O(N+E)

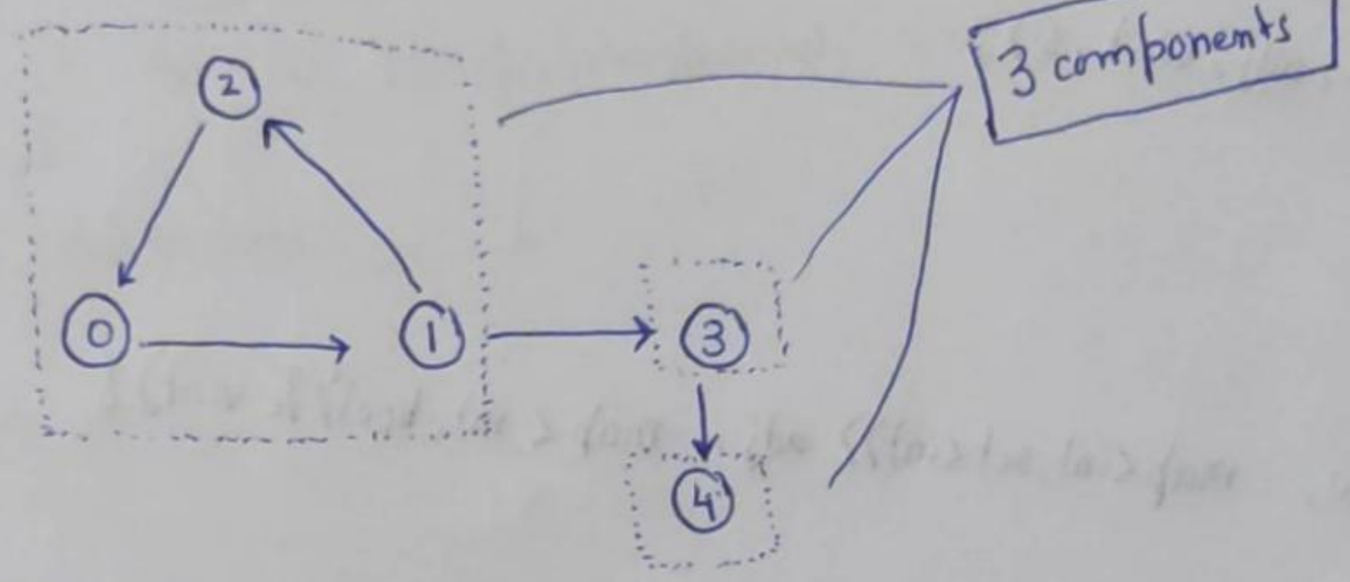
Code →

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

bool DFS ( int v, map<int, list<int>> adj ) {
    vector<int> col (v, -1);
    for (int i=1 ; i< v ; i++) {
        if ( col[i]== -1){
            if ( ! bipartite ( i, adj, col)) {
                return false;
            }
        }
    }
    return true;
}
```

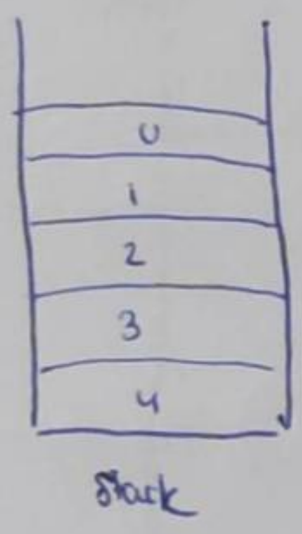
⊙ Kosaraju's Algorithm ⇢

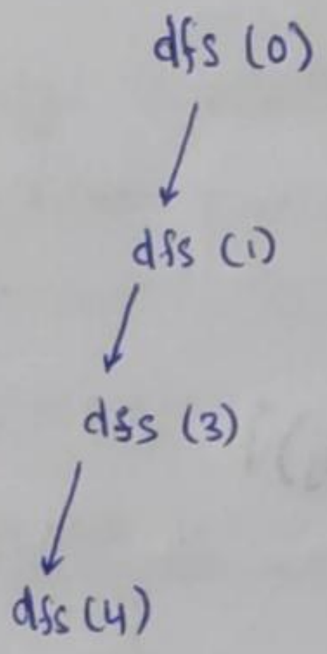
↳ ⊡ It is used to find strongly connected grap ~~algorithms~~ components.

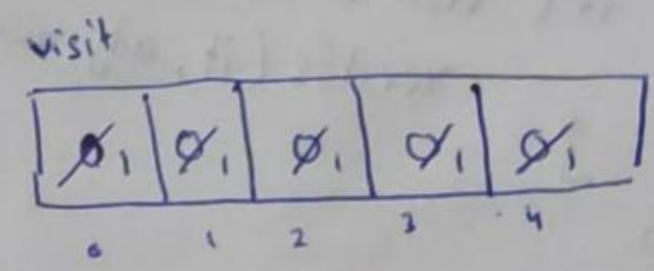
⊡ Dry ⇢ Run



Step 1 ⇢ & Topological sort using DFS.

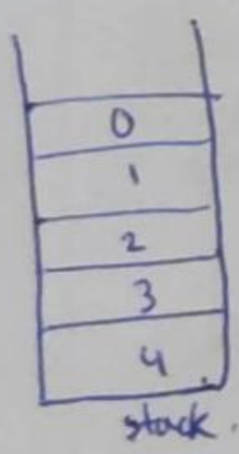


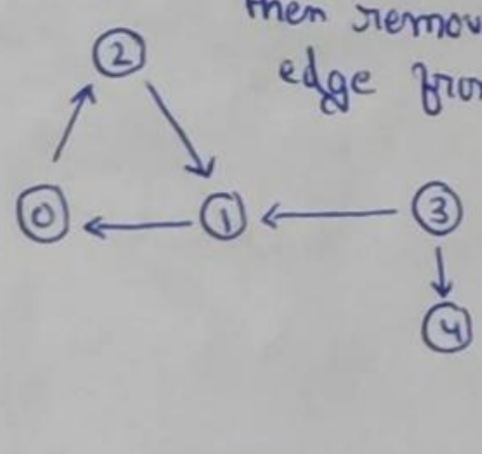




Step 2 ⇢ Do the grap transposal,

if u→v there is an edge, then remove it and make an edge from v→u. Also initialise visit array by 0.





visit.

0	0	0	0	0
0	1	2	3	4

Step 3 ⇢ Lastly, apply ~~&~~ DFS to given orders of stack.

And count number of strongly connected components or display it.

Count = 3

0 2 1

3

4

⊡ Code :→

```
void topo (int node, map<int,set<int>> adj, map<int,bool> & visit, stack<int>& s) {
    visit[node] = 1;
    for(int it: adj[node]) {
        if(!visit[it]) {
            topo(it, adj, visit, s);
        }
    }
    s.push(node);
}

void revdfs (int node, map<int,set<int>> adj, map<int, bool> & visit) {
    visit[node]=1;
    cout << node;
    for(auto it: adj[node]) {
        if(!visit[it]) {
            revdfs(it, adj, visit);
        }
    }
}

void Kosaraju (int v, map<int, set<int>> adj) {
    map<int, bool> visit;
    stack<int> s;
    for(int i=0; i<v; i++) {
        if(!visit[i]) {
            topo(i, adj, visit, s);
        }
    }
    map<int, set<int>> transpose adj;
    for(int i=0; i<v; i++) {
        visit[i]=0;
        for(auto it: adj[i]) {
            transposeadj[it].insert(i);
        }
    }
}
```

⊡ Time Complexity :→ O(N+E)

⊡ Space Complexity :→ O(N+E)

```
int count = 0;
while ( ! s.empty()) {
        int topnode = s.top();
        s.pop();
        if ( ! visit [topnode]) {
            count++;
            revdfs( topnode, transpose adj, visit);
        }
        cout << endl;
    }
    cout << count;
}
```

⊙ Bridge →

→ ⊡ A bridge is actually an edge in a graph which when removed our graph becomes disconnected or, number of ∨ connected component in a graph increases.

→ ⊡ Approach to solve →

→ Brute Approach →

① take every edge
⇓
② remove it
⇓
③ And, check number of connected component using DFS.

Approach 2 →

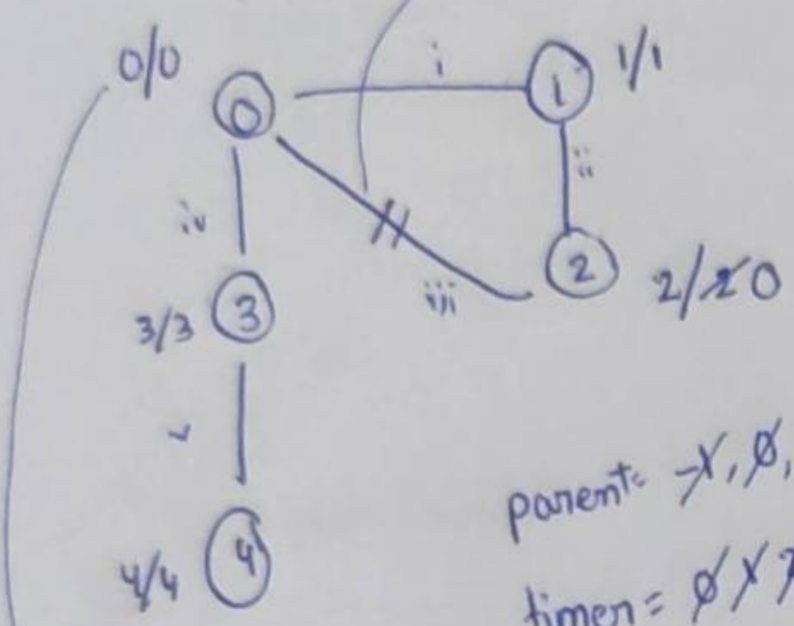
Back Edge =
low[node] = min (low[node], disc[neighbour])

0/0 ⓪ —— ① 1/1

disc

⊡ Approach 2 ⇒

Backedge: When we have 2 paths to reach a particular node from source node, then, that is called as a Backedge.



parent: -1, 0, 1, 0, 3

timer = 0 1 2 3 4

the smallest of all edge

disc ⇒ | -1 | -1 | -1 | -1 | -1 | (0 1 2 3 4) → tracks the time at which we reach this nodes

low ⇒ | 0 | 1 | 2 | 3 | 4 | → Earliest possible time.

visit ⇒ | F | F | F | F | F | (0 1 2 3 4)

⊡ check (low[it] > disc[node])

0>1 → F, 0>0 → F

4>3 → (4,3) ✓ → T

3>0 → (3,0) ✓ → T

These both edges are our bridges.

⊙ important points ⇒

(i) Backedge updation: low[node] = min(low[node], disc[it])

It is done when our adjacent node is already visited.

(ii) If parent and it (adjacent node) are equal then we will skip it.

(iii) if our adjacent node is not visited then we will call dfs recursive function, will update low[node] of parent as due to backedge a new path is discovered that takes less time, so we have to update the low[node] of parent.

Finally, we check for our bridge by condition: (low[it] > disc[node]).

⊙ Code ⇢

```
void dfs (int node, int parent, int& timer, vector<int>& disc, vector<int>& low, map<int, bool> adj,
                                                    map<int, bool>& visit) {

        visit[node] = 1;
        disc[node] = low[node] = timer++;

        for (int it : adj[node]) {
            if (it == parent) {
                continue;
            }
        if (!visit[it]) {
            dfs(it, node, timer, disc, low, adj, visit);
            low[node] = min(low[node], low[it]);

            if (low[it] > disc[node]) {
                cout << node << "," << it;
            }
        }
        else {
            low[node] = min(low[node], disc[it]);
        }
    }
}

void DFS (int v, map<int, set<int>> adj) {
        int timer = 0;
        int parent = -1;
        vector<int> disc(v, -1);
        vector<int> low(v, -1);
        map<int, bool> visit;

        for (int i=0; i<v; i++) {
            if (!visit[i]) {
                dfs(i, parent, disc, low, adj, visit);
            }
        }
}
```

⊡ Time complexity ⇢ O(N+E)

⊡ Space complexity ⇢ O(N)

Graph

⊙ Articulation Point →

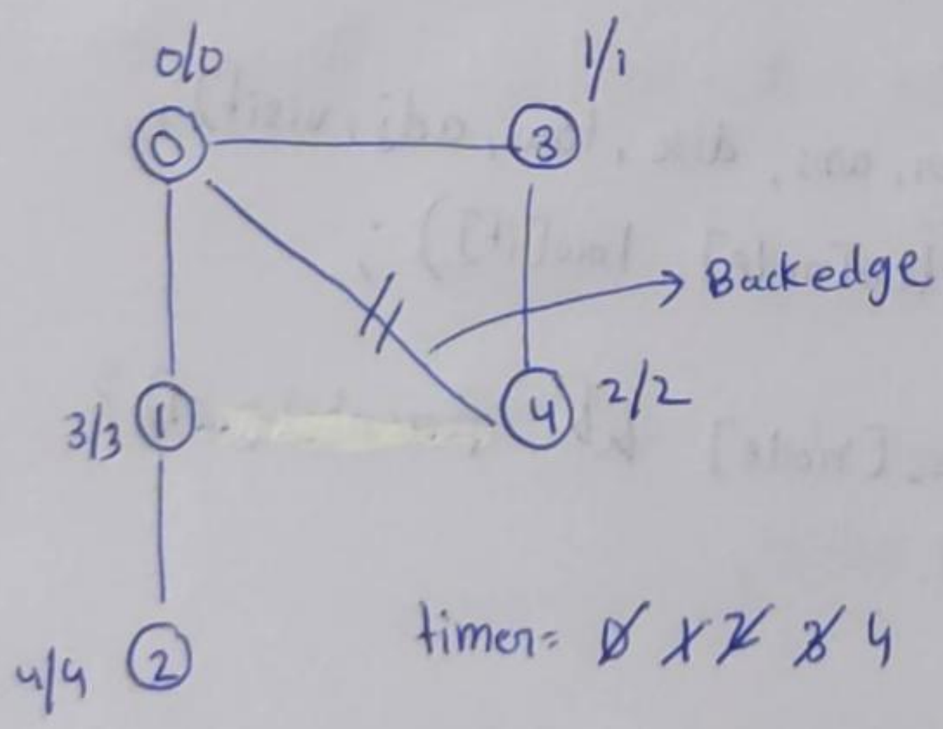
→ ⊡ On removing a particular node/vertices if our graph gets break into 2 or more than 2 components, then that node/vertices is called as a Articulation point..

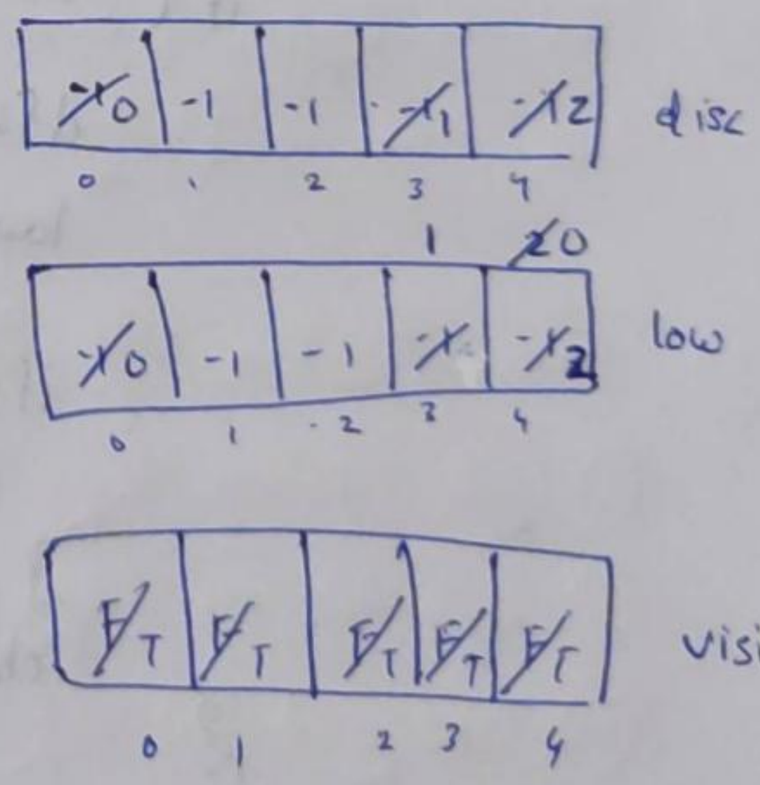
→ ⊡ Brute Force →

every node => remove karke = check number of grab components. (O(N*(N+E)))

→ ⊡ Optimise Appsoach → (Dargen's Theorem)

⊡ Dry Run →





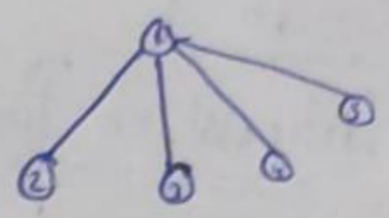
Above algo is almost similar to bridge, except minor changes...

(i) To check articulation Point:

(low[it] >= disc[node] && parent != -1)

(ii) One special condition to handle:

Suppose, our parent = -1 and and child > 1
then it too considered as articulation point. (Basically for starting node)



- Time Complexity → O(N+E)
- Space Complexity → O(N)

Code →

```
void dfs(int node, int parent, int timer, vector<int>& ans, vector<int>& disc, vector<int>low,
                                    map<int, set<int>>adj, map<int, bool>
                                                        & visit ){
    visit[node] = 1;
    disc[node] = low[node] = timer++;

    int child=0;
    for(auto it: adj[node]) {
        if(it == parent) {
            continue;
        }
        if(!visit[it]) {
            dfs(it, node, timer, ans, disc, low, adj, visit);
            low[node] = min(low[node], low[it]);

            if( low[it] >= disc[node] && parent != -1) {
                ans[node] = 1;
            }
            child++;
        }
        else {
            low[node] = min( low[node], disc[it]);
        }
    }
    if(parent == -1 && child > 1) {
        ans[node] = 1;
    }
}
```

```
void DFS (int v, map <int, set<int>> adj){
    int timer =0;
    int parent = -1;
    vector <int> disc (v,-1);
    vector <int> ~~ans~~ low(v,-1);
    vector <int> ans (v,0);
    map <int, bool> visit;

    for ( int i=0; i<v ; i++){
        if( !visit[i]) {
            dfs( i, parent, timer, ans, disc, low, adj, visit);
        }
    }
}
```

⊙ Graph →

① What is a Graph? what are its 2 components. Explain. What is directed and Undirected graph.

② What is a Degree? How to calculate total number of degree of undirected Graph? What is indegree and outdegree.

③ What is path in directed and Undirected Graph? What is cyclic and Acyclic Graph? what is a weight Graph? If a graph does not have a weight what we do?

④ What are 2 ways to implement a graph? what is adjacent list and matrix, Implement. Disscuss 3 ways to create a adjacent list. Implement Adjacent list for a weight Graph. Can we use a set instead of list to create adjacent list, why?

⑤ What is a component of a Graph? Can a single vertices can also be called as a component? Do we need to write code for each and every component?

⑥ What are 2 types of Graph traversal? What is BFS and DFS? Discuss their dry run, code, time complexity and space complexity.

⑦ Give dry Run, code, Time complexity and Special conditions .....

(i) Cycle detection using BFS and DFS (Undirected).
(ii) Cycle detection using DFS (Directed)
(iii) Topological Sort using DFS and BFS (Kahn' Algorithm).
(iv) Cycle detection using BFS (Directed)
(v) Shortest path in Undirected Graph
(vi) Shortest path in DAG.
(vii) Dijastra Algorithm.
(viii) Prim's Algorithm.
(ix) Kruskal's Algorithm.

(x) Bellman Ford.

(xi) Bipartite Graph check using DFS and BFS.

(xii) Kosaraju Algorithm.

(xiii) Bridge Detection.

(xiv) Articulation Point Detection. (Dargen's Theorum.)

(8.) What is topological Sort? In which type of Graph it is used? why it is only used in Directed and Acyclic graphs?

(9.) What is use of Dijastra Algorithm.

(10.) What is minimum Spanning Tree? Can Everynode is reachable to othernode in it? Two algorithms to find it, are what? How can we optimise Prim's algorithm from time complexity $O(n^2)$ to $O(n\log n)$?

(11.) What is a disjoint set? Write its 3 uses? What are it 2 important functions? State uses of ----

(i) UnionSet().

(ii) findParent()

(iii) Union by Rank, why to place a shorter tree under a bigger tree.

(iv) Path compression? State importance of Kruskals Algorithm.

(12.) What is uses of Bellman ford Algo? why to we use it in place of Dijkstra Algorithm? Can we use it to find Negative Cycle.

(13.) What is a Bipartite Graph? Why are graphs having even length and odd length cycles are called?

(14.) What is Kosarajus Algorithm? State its 3 important steps.

(15.) What is a Bridge, Articulation point and a Backedge. what is Brute fore approach to find Bridge or Articulation point. State its time complexity.