

Graph

- Karun Karthik

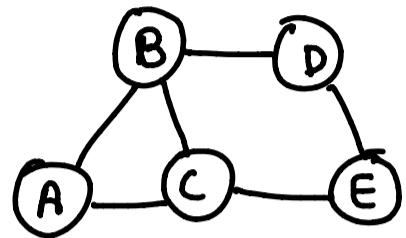
Contents

0. Introduction
1. All paths from source to target
2. Flood Fill
3. Number of Islands
4. Max Area of the Island
5. Find if path exist in Graph
6. Find the town judge
7. Detect cycle in a Directed Graph
8. Topological Sort
9. Course Schedule
10. Course Schedule II

Graphs

graph G is a pair (V, E) where V is set of vertices & E is set of edges. $n = |V|$ & $e = |E|$

Ex



$$V = \{A, B, C, D, E\} \quad n = 5$$

$$E = \{AB, AC, BC, BD, CE, DE\} \quad e = 6$$

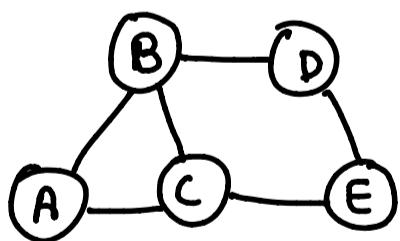
Applications →

Google maps → To find shortest route

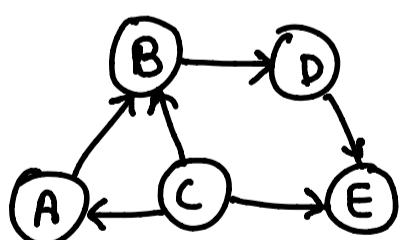
Social network → user, connection
 ↑
 vertex edge

Types →

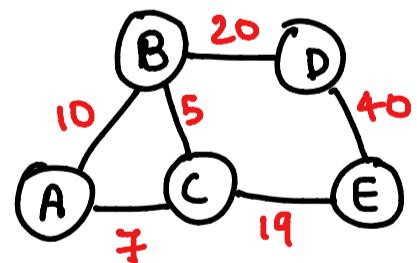
1) Undirected



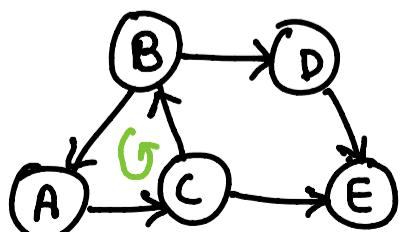
2) Directed



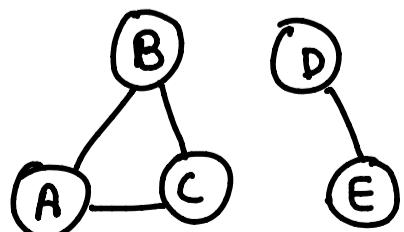
3) Weighted



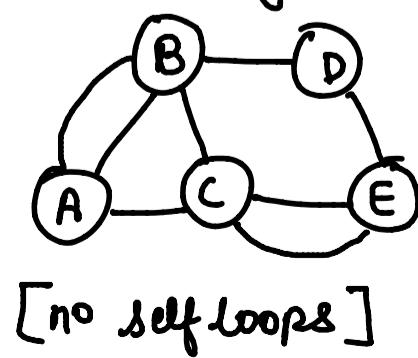
4) Cyclic



5) Disconnected



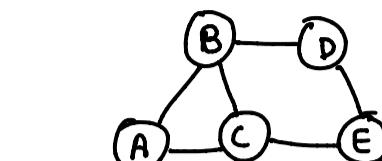
6) Multigraph



Graph Traversal

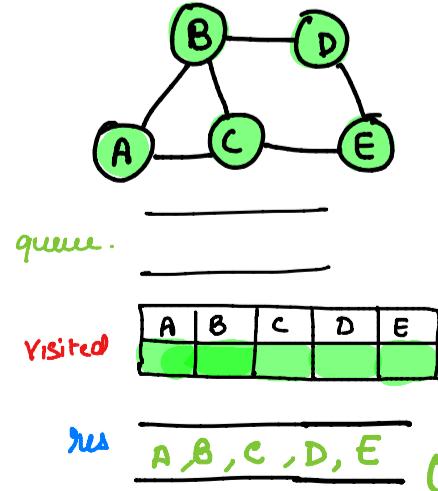
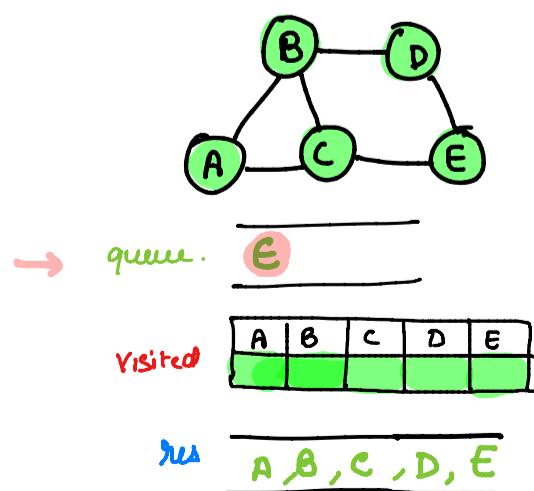
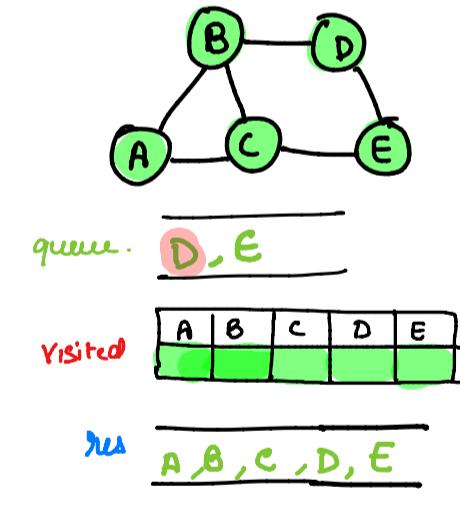
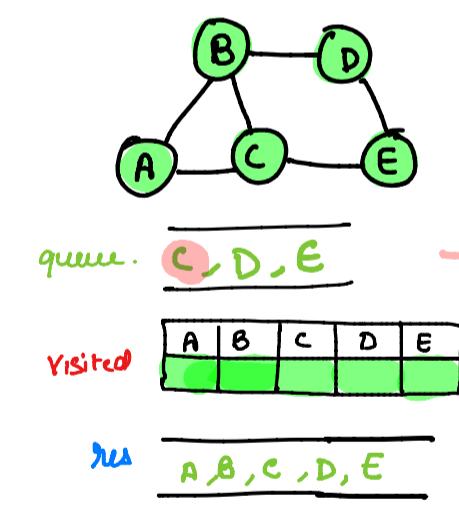
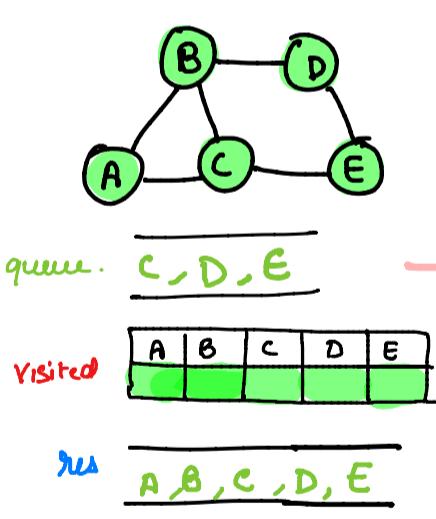
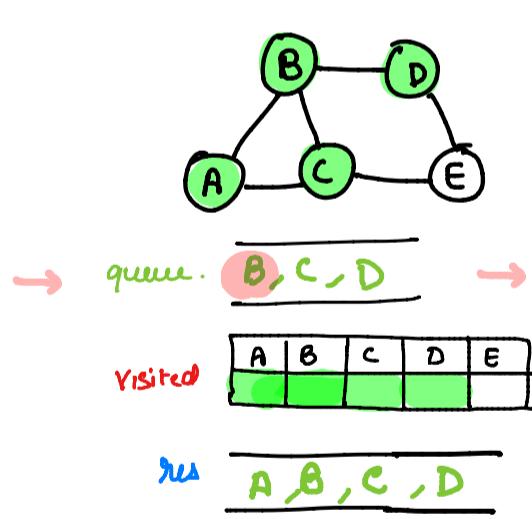
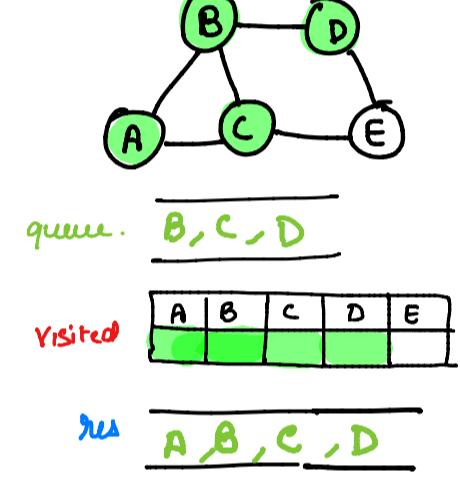
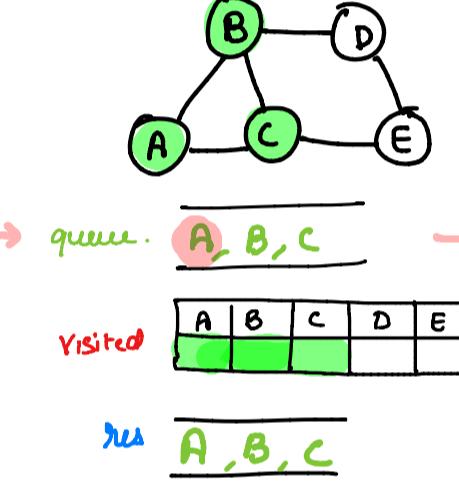
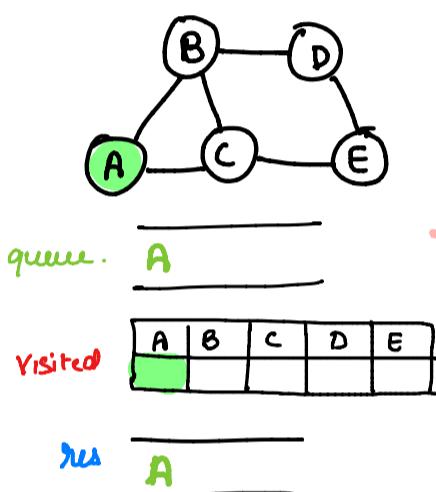
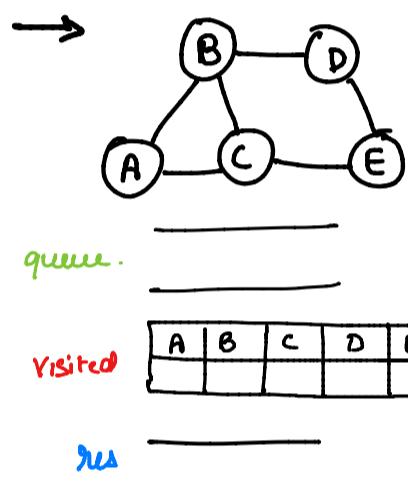
(a) BFS → visit each and every vertex in a defined order.

- select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into queue
- if no neighbours then pop.
- repeat till queue is empty



queue. _____

Visited	A	B	C	D	E
---------	---	---	---	---	---



TC $\rightarrow O(V+E)$

SC $\rightarrow O(V)$

↳ Return res.

Code

```
class Solution {
public:

    vector<int> bfsOfGraph(int v, vector<int> adj[]) {
        vector<int> ans;
        vector<int> vis(v, 0);
        queue<int> q;
        q.push(0);

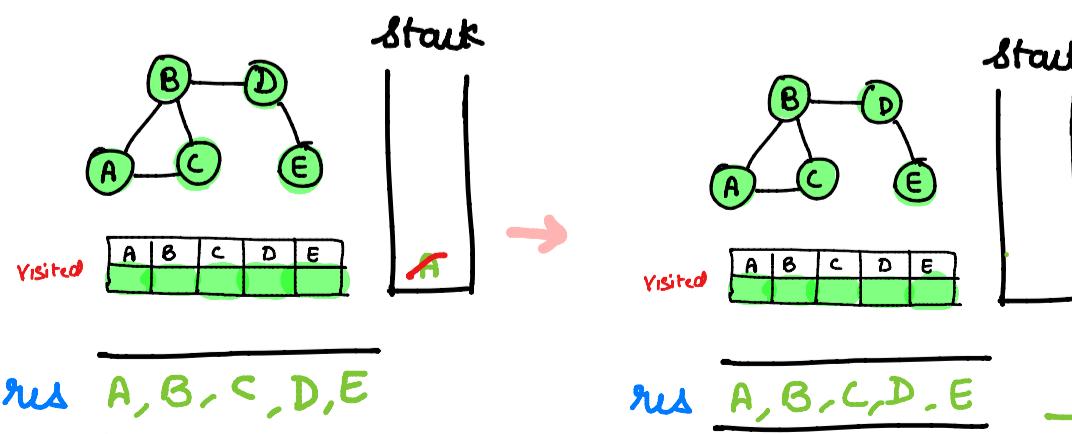
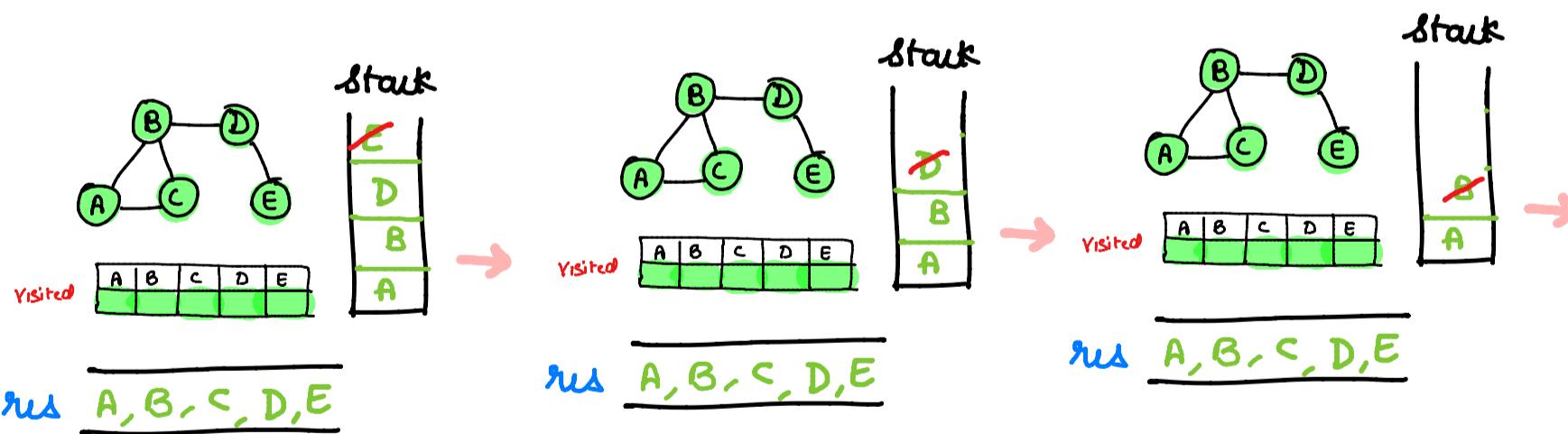
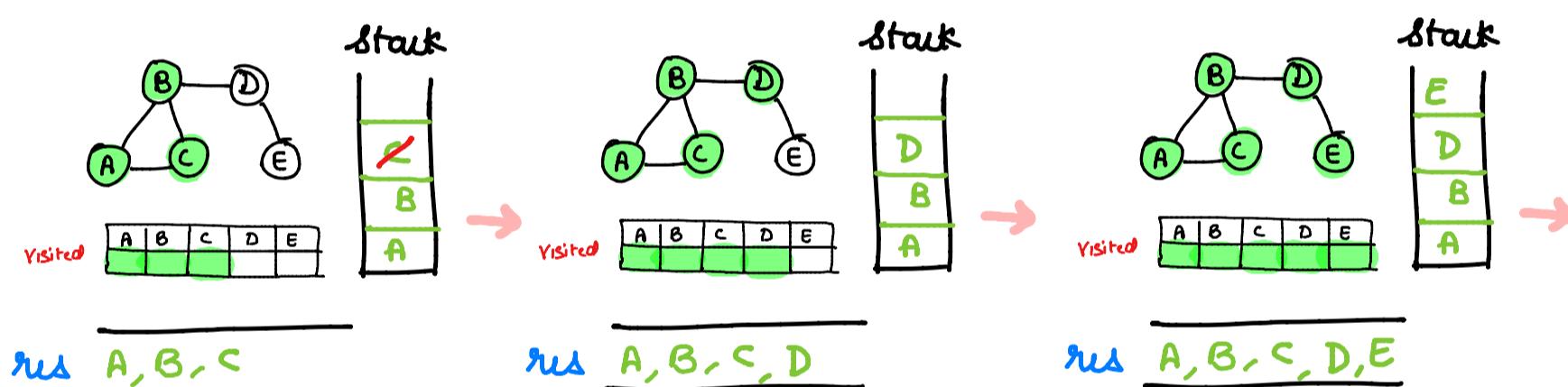
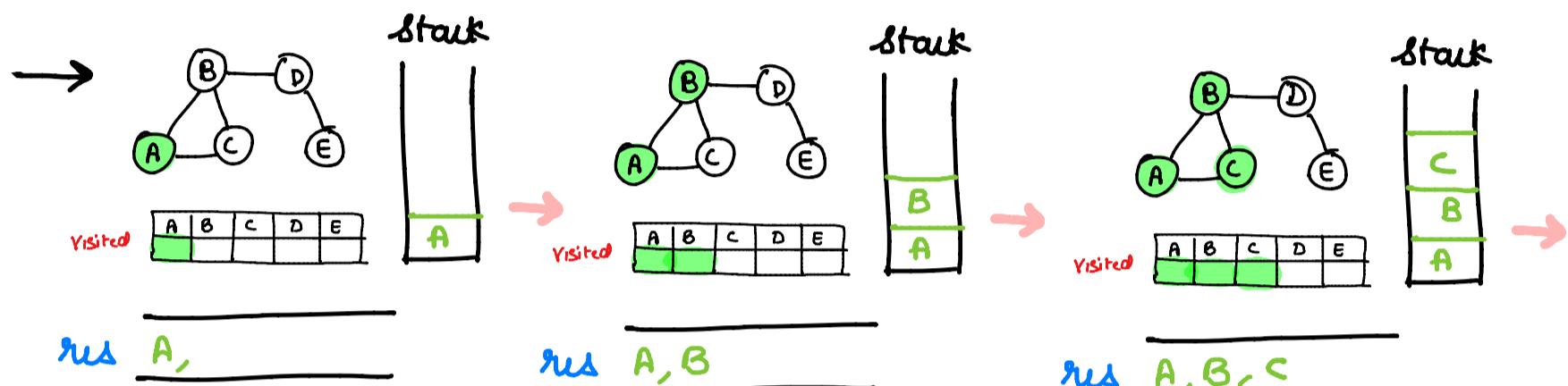
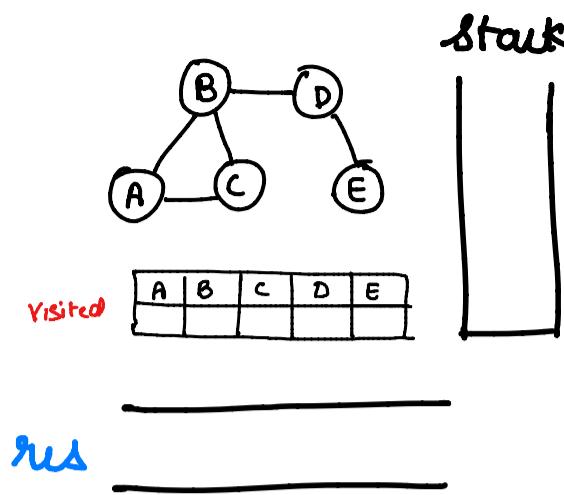
        while(!q.empty()){
            int curr = q.front();
            q.pop();
            vis[curr] = 1;
            ans.push_back(curr);
            for(auto it:adj[curr]){
                if(vis[it]==0){
                    vis[it] = 1;
                    q.push(it);
                }
            }
        }
        return ans;
    }
};
```

Applications → [BFS]

1. shortest path
2. Min. spanning tree for unweighted graph
3. cycle detection
4. GPS
5. social network.

⑥ DFS →

- select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into stack
- if no neighbours then pop.
- repeat till stack is empty



TC $\rightarrow O(V+E)$
SC $\rightarrow O(V)$

→ Return res.

Code

```
class Solution {
public:

    void dfs(vector<int>&ans, vector<int>&vis, int node, vector<int>adj[]){
        vis[node] = 1;
        ans.push_back(node);
        for(auto it:adj[node]){
            if(!vis[it]){
                vis[it] = 1;
                dfs(ans, vis, it, adj);
            }
        }
    }
    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        vector<int> ans;
        vector<int> vis(V, 0);
        for(int i=0; i<V; i++){
            if(vis[i]==0)
                dfs(ans, vis, i, adj);
        }
        return ans;
    }
};
```

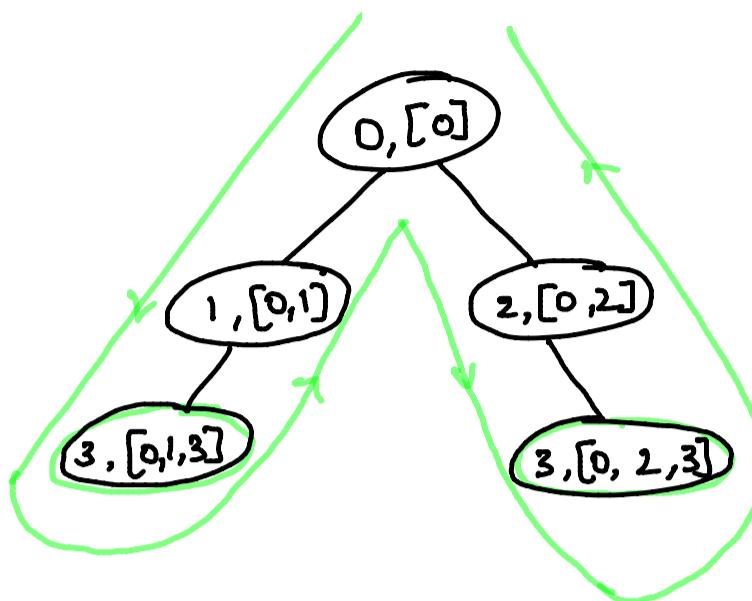
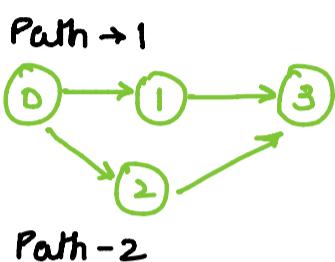
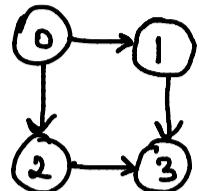
Applications → [DFS]

1. Path finding
2. Cycle detection
3. Topological sort
4. Finding strongly connected components.

① All paths from src to target

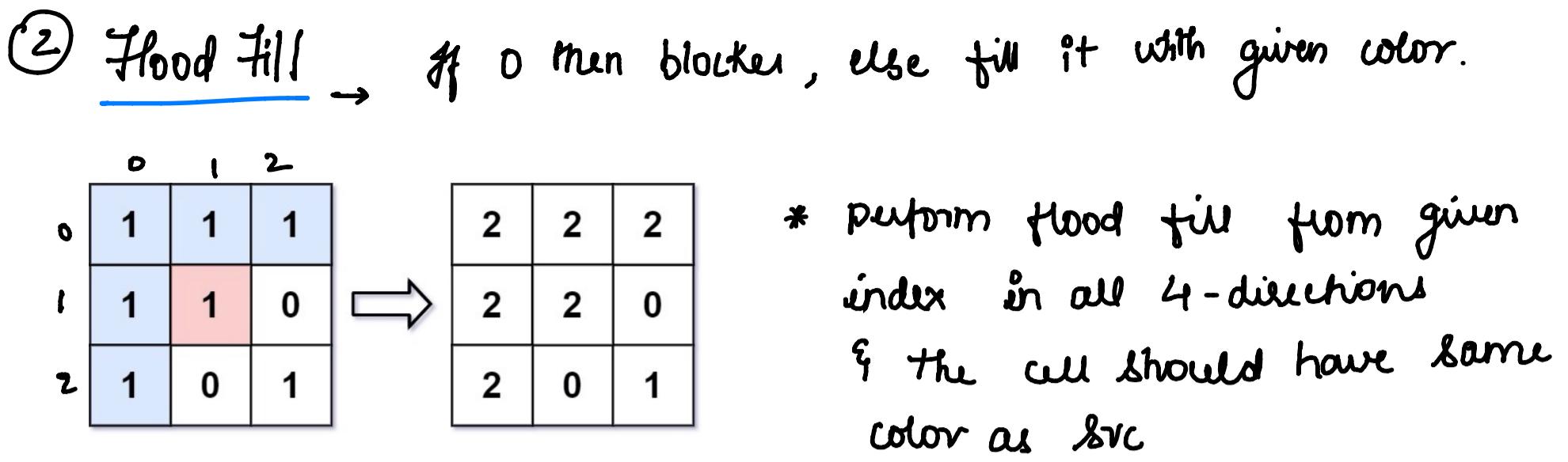
Given a directed acyclic graph, return all paths from node 0 to node n-1.

Eg



Code →

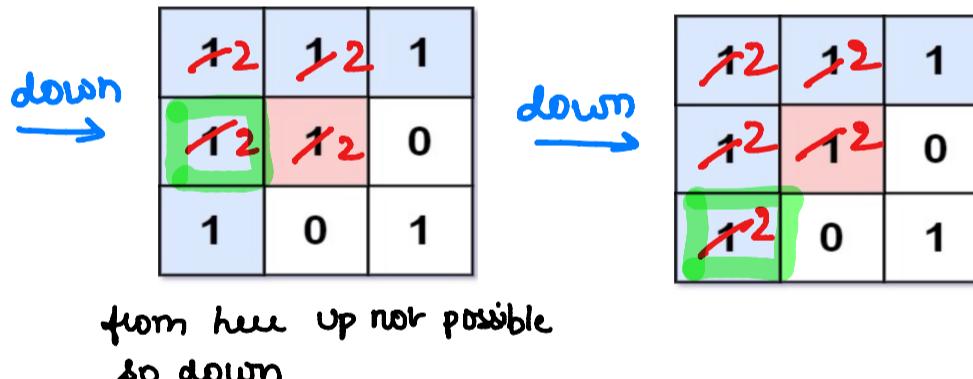
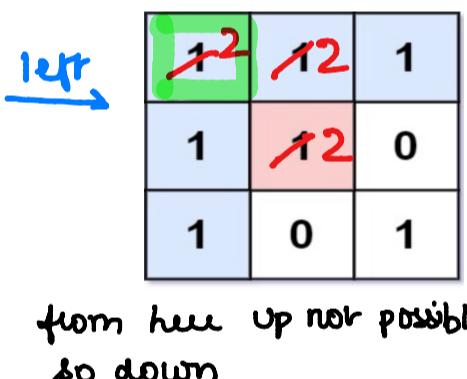
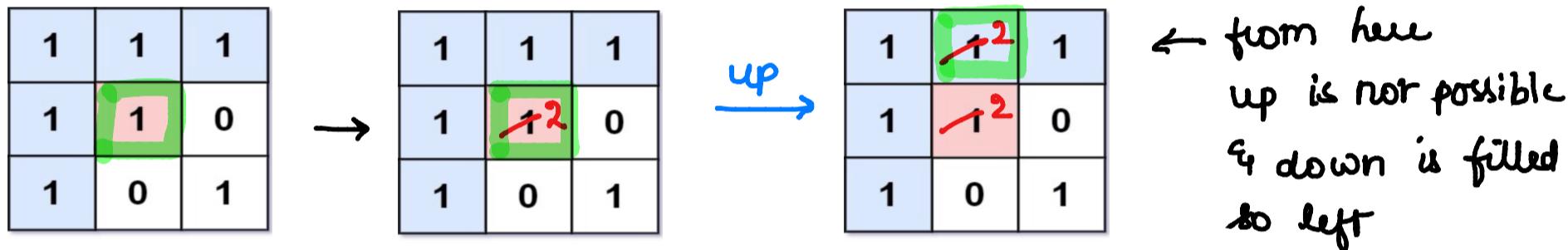
```
1 class Solution {
2 public:
3     void findAllPaths(vector<vector<int>>&graph, int currNode, vector<bool>&visited,
4                        int n, vector<int> &currPath, vector<vector<int>>&res){
5
6         if(currNode==n-1){
7             res.push_back(currPath);
8             return;
9         }
10
11         if(visited[currNode]==true) return;
12
13         // backtrack for every node
14         visited[currNode] = true;
15
16         for(auto neighbour: graph[currNode]){
17             currPath.push_back(neighbour);
18             findAllPaths(graph, neighbour, visited, n, currPath, res);
19             currPath.pop_back();
20         }
21
22         visited[currNode] = false;
23     }
24
25     vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
26         vector<vector<int>> res;
27         vector<int> currPath;
28         int n = graph.size();
29         vector<bool> visited(n);
30
31         // traversing from 0 node
32         currPath.push_back(0);
33
34         findAllPaths(graph, 0, visited, n, currPath, res);
35         return res;
36     }
37 }
```



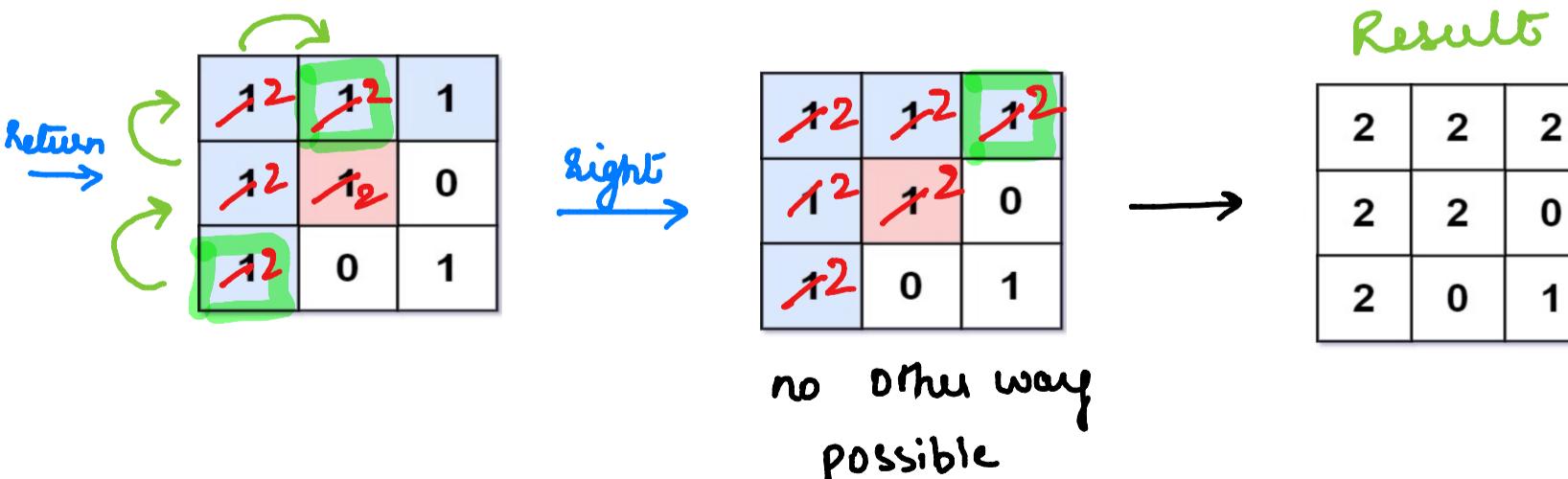
✓ let's follow the order to fill → UP, DOWN, LEFT, RIGHT

Eg In above case starting point is (1,1) & value = 1 so

*



→ no direction is possible so return



Code

```
● ● ●  
1 class Solution {  
2 public:  
3     void floodFiller(vector<vector<int>>& image, int i, int j,  
4     int m, int n, int currColor, int newColor)  
5     {  
6         if(i<0 || i>=m || j<0 || j>= n || image[i][j] == newColor  
7             || image[i][j] != currColor)  
8             return;  
9  
10        image[i][j] = newColor;  
11        floodFiller( image, i-1, j, m, n, currColor, newColor);  
12        floodFiller( image, i+1, j, m, n, currColor, newColor);  
13        floodFiller( image, i, j-1, m, n, currColor, newColor);  
14        floodFiller( image, i, j+1, m, n, currColor, newColor);  
15    }  
16  
17    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,  
18    int sc, int newColor)  
19    {  
20        int m = image.size();  
21        int n = image[0].size();  
22        int currColor = image[sr][sc];  
23        floodFiller(image, sr, sc, m, n, currColor, newColor);  
24        return image;  
25    }  
26};
```

$$Tc \rightarrow O(mn)$$

$$Sc \rightarrow O(h)$$

↳ recursive stack

③ Number of islands → Given grid of 1 (land) & 0 (water), return no. of islands.

Eg
 0 $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$
 1 $\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \end{bmatrix}$
 2 $\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \end{bmatrix}$
 3 $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}$
 4 $\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

- Always start dfs only if value = 1 & change its value to 0, so it cannot be visited again
- if initial value = 0 then skip
- initially ans = 0

• let start from (0,0) & try moving U,D,L,R

→ the traversal goes in this order

(0,0) → (1,0) → (1,1) → (0,1) i.e

& update ans.

$\begin{bmatrix} [1,1,0,0,0] \\ [1,1,0,0,0] \\ [0,0,1,0,0] \\ [0,0,0,1,1] \end{bmatrix}$
 ans = 1.

→ now grid becomes

0 $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$
 1 $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
 2 $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
 3 $\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

- now, we can skip every entry from (1,0) to (2,1) as they are 0s
 - now start from (2,2), as U,D,L,R is not possible, set its value = 0 & update ans.
- ans = 1.

→ now grid becomes

0 $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$
 1 $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
 2 $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
 3 $\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

- now, we can skip every entry from (2,3) to (3,2) as they are 0s
- now start from (3,3), it goes as follows
 $(3,3) \rightarrow (3,4)$
- further traversal from (3,4) is not possible

ans = 3. ans = 3

Code

```
1 class Solution {
2 public:
3     void countIsland(vector<vector<char>>& grid, int currRow, int currCol, int row, int col){
4         if(currRow<0 || currRow>=row || currCol<0 || currCol>=col || grid[currRow][currCol]=='0')
5             return;
6
7         grid[currRow][currCol] = '0';
8         countIsland(grid, currRow-1, currCol, row, col);
9         countIsland(grid, currRow+1, currCol, row, col);
10        countIsland(grid, currRow, currCol-1, row, col);
11        countIsland(grid, currRow, currCol+1, row, col);
12    }
13
14    int numIslands(vector<vector<char>>& grid) {
15        int ans = 0;
16        int row = grid.size();
17        int col = grid[0].size();
18
19        for(int currRow = 0; currRow < row; currRow++)
20            for(int currCol = 0; currCol < col; currCol++)
21                if(grid[currRow][currCol]=='1'){
22                    ans++;
23                    countIsland(grid, currRow, currCol, row, col);
24                }
25
26        return ans;
27    }
28};
```

$T_c \rightarrow O(mn)$ Avg case
 $O(m^2n^2)$ Worst case

④ Max Area of the Island

- * Intuition is same as previous problem.
- * Minor Tweak to count number of 1s in island.
- * Once entire island traversal is done,
compute for max area of island.

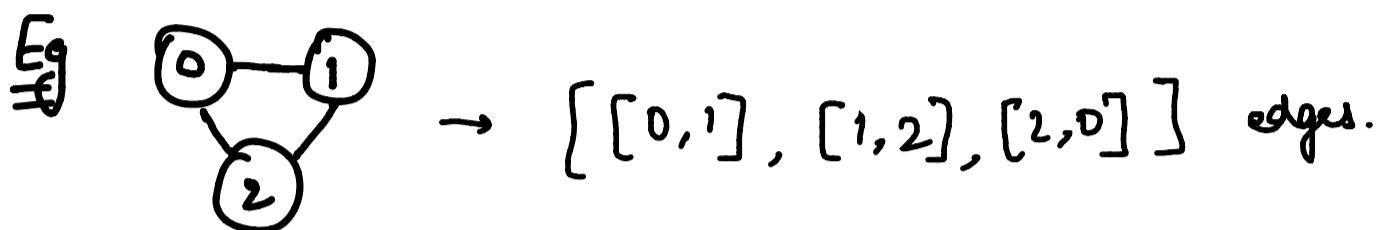
TC $\rightarrow O(mn)$ Avg case.

code →

```
● ● ●  
1 class Solution {  
2 public:  
3     int findArea(vector<vector<int>>& grid, int currRow, int currCol, int m, int n){  
4         if(currRow<0 || currCol<0 || currRow>=m || currCol>=n || grid[currRow][currCol]==0)  
5             return 0;  
6  
7         grid[currRow][currCol]=0;  
8  
9         // this is for single cell where we started traversing  
10        int count = 1;  
11        count += findArea(grid, currRow-1, currCol, m, n);  
12        count += findArea(grid, currRow+1, currCol, m, n);  
13        count += findArea(grid, currRow, currCol-1, m, n);  
14        count += findArea(grid, currRow, currCol+1, m, n);  
15        return count;  
16    }  
17    int maxAreaOfIsland(vector<vector<int>>& grid) {  
18        int m = grid.size();  
19        int n = grid[0].size();  
20        int ans = 0;  
21        for(int currRow = 0; currRow<m; currRow++)  
22            for(int currCol = 0; currCol<n; currCol++){  
23                if(grid[currRow][currCol]==1){  
24                    ans = max(ans, findArea(grid, currRow, currCol, m, n));  
25                }  
26            }  
27        return ans;  
28    }  
29};
```

5) Find if path exist in graph.

Given src, dest, no. of nodes & set of edges, find if path exist b/w src & dest.

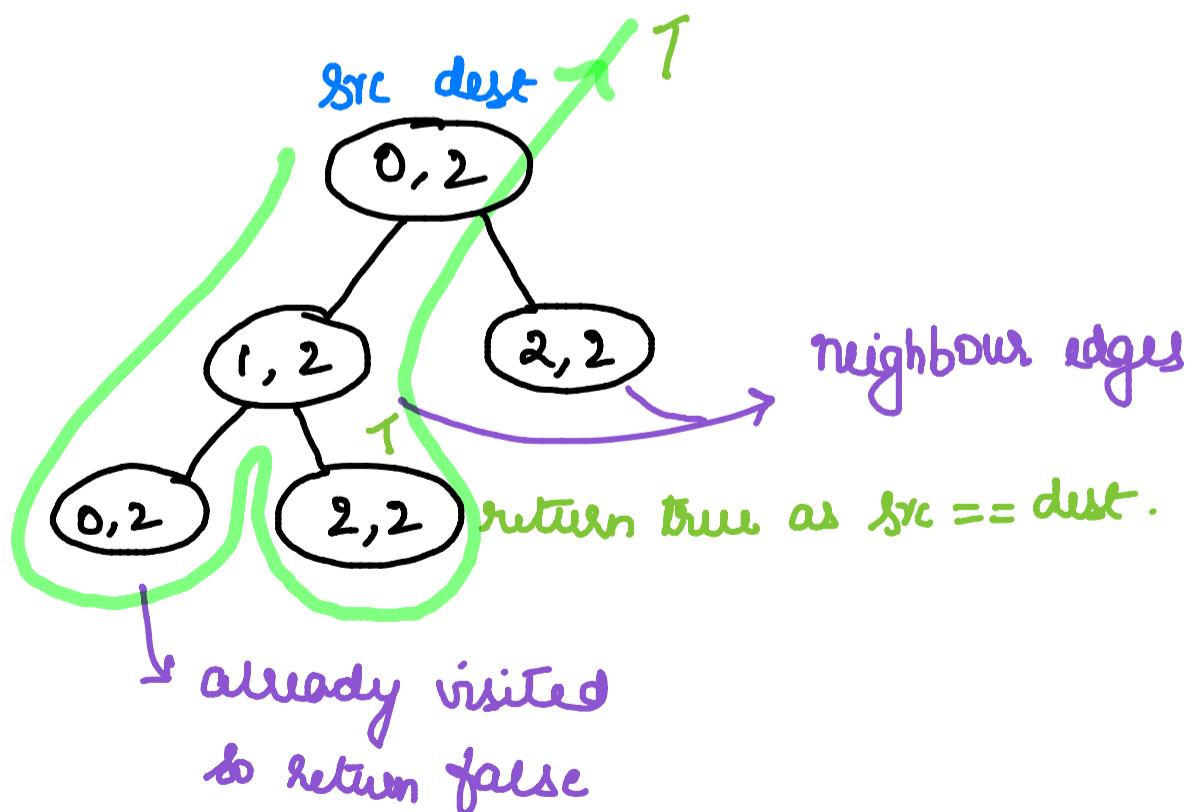


$n = 3$ edges = $[[0,1], [1,2], [2,0]]$ src = 0, dest = 2.

- 1) Create a graph using adj list rep. $[[1,2], [0,2], [1,0]]$
- 2) Perform dfs

$[[1,2], [0,2], [1,0]]$

0	1	2
T	F	I

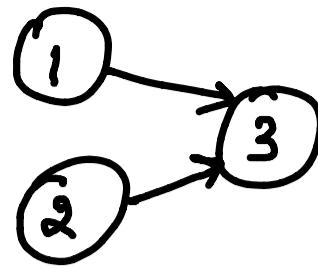


Code →

```
● ● ●  
1 class Solution {  
2 public:  
3     bool validPath(int n, vector<vector<int>>& edges, int src, int dest) {  
4  
5         vector<vector<int>>graph(n);  
6         for(int i=0;i<edges.size();i++)  
7         {  
8             int v1 = edges[i][0];  
9             int v2 = edges[i][1];  
10            graph[v1].push_back(v2);  
11            graph[v2].push_back(v1);  
12        }  
13        vector<bool>vis(n,false);  
14        return pathExist(src, dest, graph, vis);  
15    }  
16  
17    bool pathExist(int src , int dest,vector<vector<int>>&graph,vector<bool>&vis){  
18  
19        if(src==dest) return true;  
20  
21        vis[src]=true;  
22  
23        for(int i=0;i<graph[src].size();i++)  
24            if(vis[graph[src][i]]==false)  
25                if(pathExist(graph[src][i],dest,graph,vis)==true)  
26                    return true;  
27  
28        return false;  
29    }  
30 }  
31 };
```

⑥ Find the town judge

$n = 3$, trust = $\begin{bmatrix} [1, 3], [2, 3] \end{bmatrix}$



* In degree of town judge = $n - 1$

& Outdegree = 0

✓ Create 2 arrays

outdegree	<table border="1"> <tr> <td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>2</td><td>3</td></tr> </table>	0	0	1	0	0	1	2	3
0	0	1	0						
0	1	2	3						
indegree	<table border="1"> <tr> <td>0</td><td>0</td><td>0</td><td>2</td></tr> <tr> <td>0</td><td>1</td><td>2</td><td>3</td></tr> </table>	0	0	0	2	0	1	2	3
0	0	0	2						
0	1	2	3						

for $[1, 3]$

indegree of 1 ↑
Outdegree of 3 ↑

for $[2, 3]$

indegree of 2 ↑
Outdegree of 3 ↑

→ traverse both indegree & outdegree

if $\text{indegree} == 0 \ \&\ \&$

$\text{outdegree} == n - 1$

then return that vertex

code

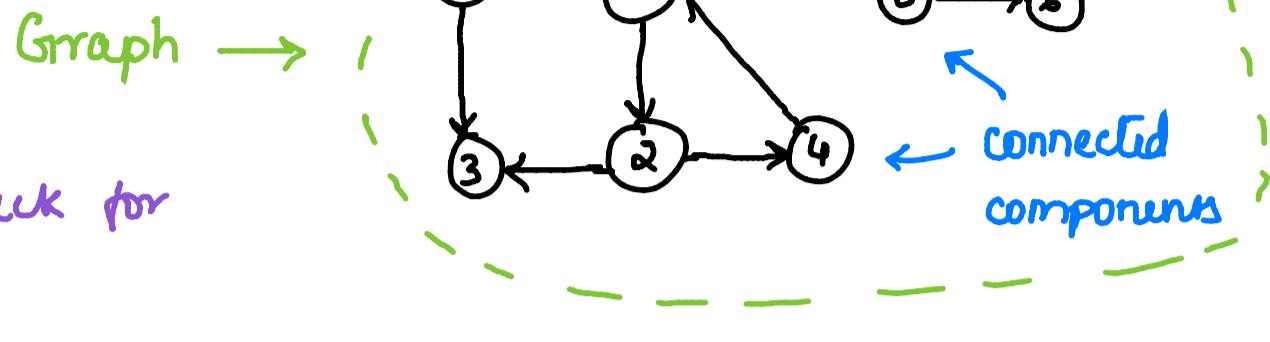
```
● ● ●  
1 class Solution {  
2 public:  
3     int findJudge(int n, vector<vector<int>>& trust) {  
4         vector<int>indegree(n+1,0);  
5         vector<int>outdegree(n+1,0);  
6         for(int i=0;i<trust.size();i++)  
7         {  
8             int v1 = trust[i][0];  
9             int v2 = trust[i][1];  
10            outdegree[v1]+=1;  
11            indegree[v2]+=1;  
12        }  
13        for(int i=1;i<=n;i++)  
14        {  
15            if(outdegree[i]==0 && indegree[i]==n-1)  
16                return i;  
17        }  
18        return -1;  
19    }  
20};
```

7 Detect cycle in a directed graph

Consider a graph with 'n' vertices labelled as $[0..n-1]$

Eg $n=7 [0, 1, 2, 3, 4, 5, 6]$

Graph \rightarrow



* To detect cycle, check for backedge.

Let's start dfs from 0 vertex.

* At every vertex, check if it's already visited, if already visited then check if it is present in recursive stack.

If present, then it indicates back edge \rightarrow Returns True

* If vertex is not visited then mark it in visited array & recursive stack

Visited $\rightarrow \{0, 1, 2, 3, 4\}$

Recursive stack $\rightarrow \{0, 1, 2, 3, 4\}$

* At 3 vertex, there's no neighbour

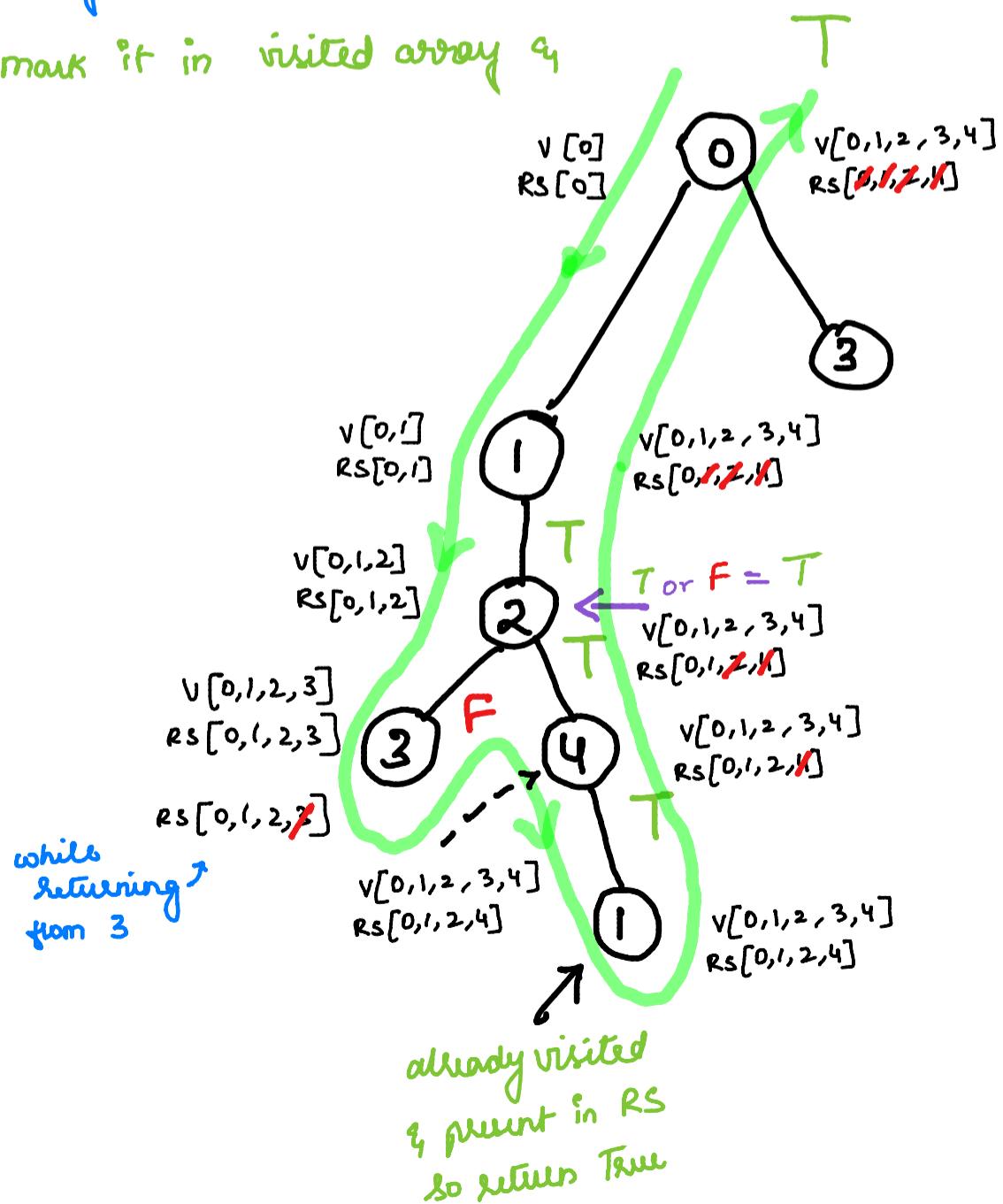
& no cycle is detected so return F.

Before returning, undo change made in Recursive stack by popping it.

Visited $\rightarrow \{0, 1, 2, 3, 4, 1\} \&$

Recursive stack $\rightarrow \{0, 1, 2, 3, 4\}$

1 is already present in recursive stack so return true.



Code

$T_C \rightarrow O(V+E)$

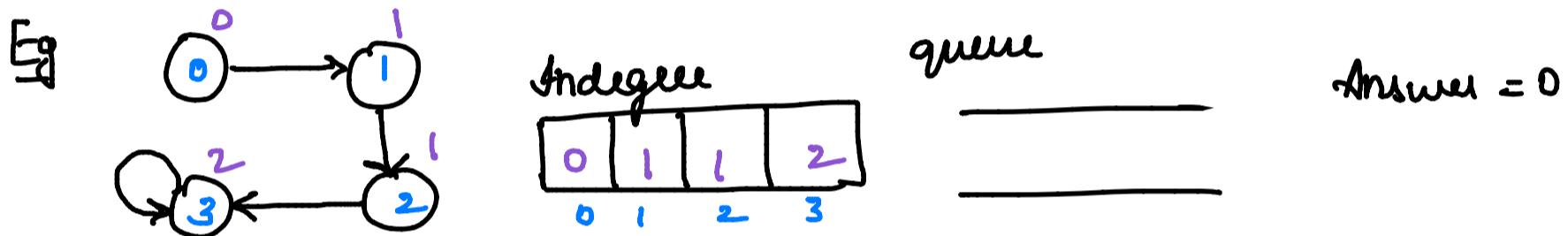
$S_C \rightarrow O(V)$

```
1 class Solution {
2     public:
3         bool dfs(int node, vector<int>&vis, vector<int>&rs, vector<int> adj[])
4         {
5             vis[node]=1;
6             rs[node]=1;
7             for(auto it:adj[node])
8             {
9                 if(vis[it]==0){
10                     if(dfs(it,vis,rs,adj))
11                         return true;
12                 }
13                 else if(rs[it]==1)
14                     return true;
15             }
16             rs[node]=0;
17             return false;
18         }
19         bool isCyclic(int V, vector<int> adj[]) {
20
21             vector<int>vis(V,0);
22             vector<int>rs(V,0);
23
24             for(int i=0;i<V;i++)
25             {
26                 if(vis[i]==0)
27                     if(dfs(i,vis,rs,adj))
28                         return true;
29             }
30             return false;
31         }
32     };
}
```

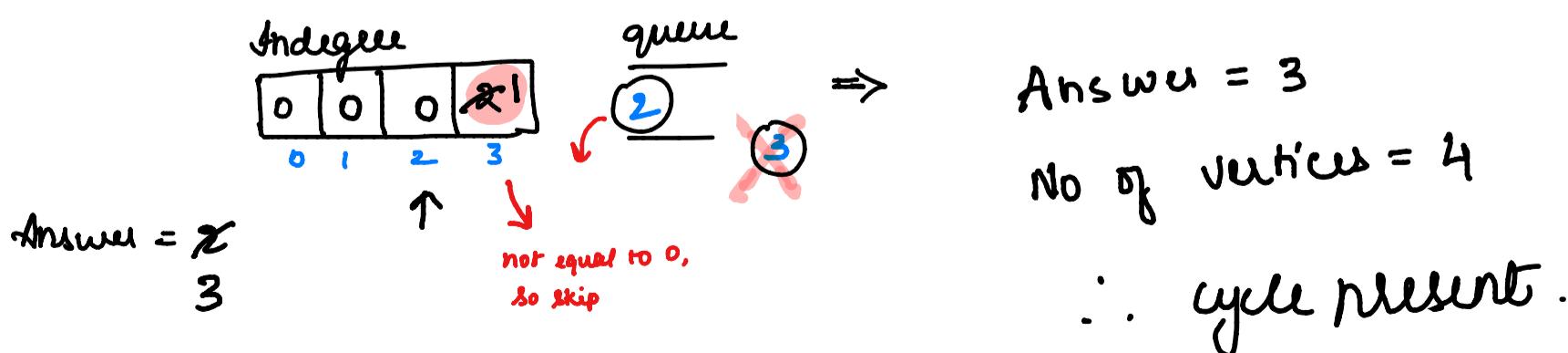
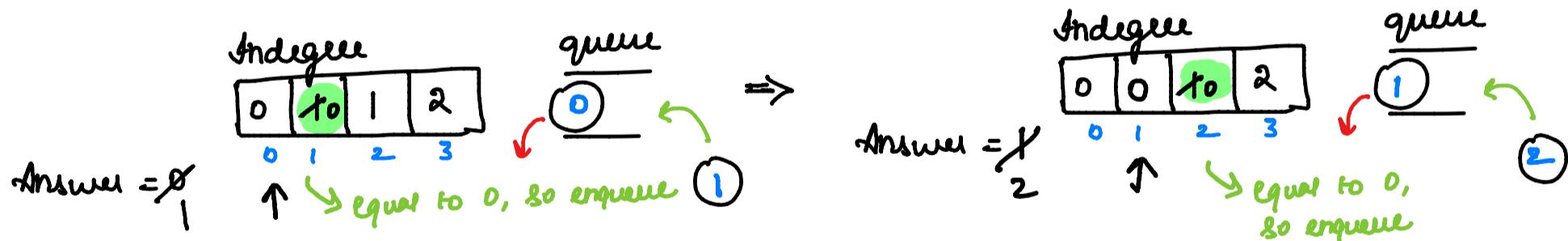
* Kahn's Algorithm → To find topological Ordering

↓
can be used to find cycle using BFS.

- ① Find indegree of every vertex in graph & answer = 0
- ② If indegree of vertex is 0, then push into queue & do bfs till queue is not empty & while doing bfs decrease the indegree of neighbour by 1.
if indegree of neighbour = 0, then enqueue & increment answer by 1
- ③ If answer != no. of vertices then cycle is present.



→ As indegree of 0 is 0, we push into queue & do bfs till queue is not empty.



code

```
1 class Solution{
2     public:
3         bool isCyclic(int V, vector<int> adj[]) {
4
5             vector<int>indegree(V,0);
6             for (int i = 0; i <V; i++)
7                 for(int it : adj[i])
8                     indegree[it]++;
9
10            queue<int>q;
11            int ans = 0;
12            unordered_set<int>vis;
13
14            for (int i=0;i<V;i++)
15            {
16                if(indegree[i]==0){
17                    q.push(i);
18                    ans+=1;
19                }
20            }
21
22            while(!q.empty())
23            {
24                int currvertex = q.front();
25                q.pop();
26                if(vis.find(currvertex)!=vis.end())
27                    continue;
28                vis.insert(currvertex);
29                for(int neighbour:adj[currvertex])
30                {
31                    indegree[neighbour]-=1;
32                    if(indegree[neighbour]==0)
33                    {
34                        q.push(neighbour);
35                        ans+=1;
36                    }
37                }
38            }
39            if(ans==V)  return false;
40            return true;
41        }
42    };
```

⑧ Topological sort

→ use Kahn's algorithm. & add node to result while performing dfs.

Code →

TC → O(V + E)

SC → O(V)

```
● ● ●  
1 class Solution  
2 {  
3     public:  
4     vector<int> topoSort(int V, vector<int> adj[]){  
5         vector<int> indegree(V, 0), res;  
6  
7         for(int i=0; i<V; i++)  
8             for(auto it:adj[i])  
9                 indegree[it]++;  
10  
11         queue<int> q;  
12         int ans = 0;  
13         unordered_set<int> vis;  
14  
15         for(int i=0; i<V; i++)  
16         {  
17             if(indegree[i]==0){  
18                 q.push(i);  
19                 ans+=1;  
20             }  
21         }  
22  
23         while(!q.empty())  
24         {  
25             int curr = q.front();  
26             q.pop();  
27  
28             // add to res  
29             res.push_back(curr);  
30  
31             if(vis.find(curr)!=vis.end())  
32                 continue;  
33  
34             vis.insert(curr);  
35  
36             for(int neighbour: adj[curr])  
37             {  
38                 indegree[neighbour]-=1;  
39                 if(indegree[neighbour]==0)  
40                 {  
41                     q.push(neighbour);  
42                     ans+=1;  
43                 }  
44             }  
45         }  
46     }  
47  
48     return res;  
49 }  
50 };
```

⑨ Course Schedule → can be solved using Kahn's algo.

$$Tc \rightarrow O(V + E)$$

$$Sc \rightarrow O(V + E)$$

Code →

```
● ● ●

1 class Solution {
2 public:
3     vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
4         vector<vector<int>> graph(n);
5         for(auto it:pre){
6             int v = it[1];
7             int u = it[0];
8             graph[v].push_back(u);
9         }
10        return graph;
11    }
12
13    bool canFinish(int n, vector<vector<int>>& pre) {
14        vector<vector<int>> graph = createGraph(n, pre);
15        vector<int> indegree(n, 0);
16        for(int i=0; i<n; i++)
17            for(int it: graph[i])
18                indegree[it]++;
19
20        queue<int> q;
21        int ans = 0;
22        unordered_set<int> vis;
23
24        for(int i=0; i<n; i++)
25            if(indegree[i]==0){
26                q.push(i);
27                ans++;
28            }
29
30        while(!q.empty()){
31            int currvertex = q.front();
32            q.pop();
33            if(vis.find(currvertex)!=vis.end())
34                continue;
35            vis.insert(currvertex);
36            for(int neighbour: graph[currvertex]){
37                indegree[neighbour]--;
38                if(indegree[neighbour]==0){
39                    q.push(neighbour);
40                    ans++;
41                }
42            }
43        }
44        if(ans==n) return true;
45        return false;
46    }
47};
```

10 Course Schedule - II

$\text{pre} \rightarrow \text{edge } [v, u]$

" u should be completed before v "

$n \rightarrow$ no. of courses [vertices]

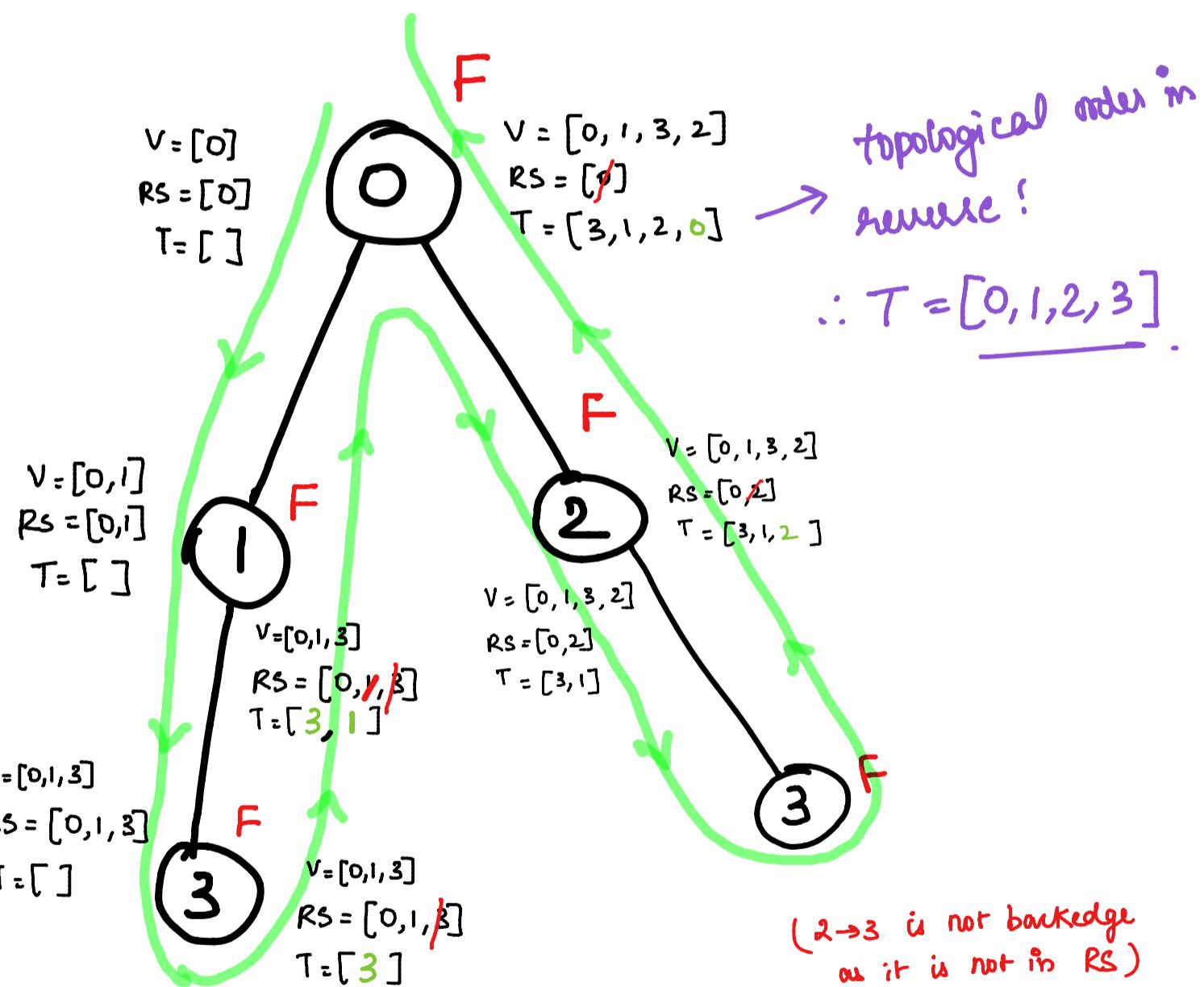
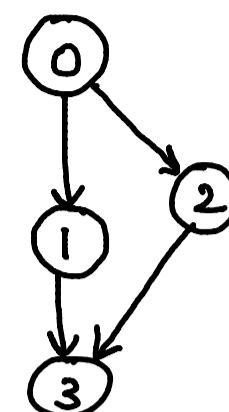
Topological sort only for DAG

Eg $n \rightarrow 4 (0, 1, 2, 3)$

$\text{pre} \rightarrow [[1, 0], [2, 0], [3, 1], [3, 2]]$

Initially

$V = []$, $RS = []$, $\text{traversal} = []$



while returning from $3 \uparrow$
pop 3 & push into traversal array.
return F, as no cycle is found

Code →

$$Tc \rightarrow O(v + E)$$

$$Sc \rightarrow O(v + E)$$

```
1 class Solution {
2 public:
3     bool dfs(vector<vector<int>>&graph, int i, vector<int> &vis,
4             vector<int> &rs, vector<int> &traversal){
5
6         vis[i] = 1;
7         rs[i] = 1;
8         for(int neighbour: graph[i]){
9             if(vis[neighbour]==0){
10                 if(dfs(graph, neighbour, vis, rs, traversal))
11                     return true;
12             }
13             else if(rs[neighbour]==1)    return true;
14         }
15         traversal.push_back(i);
16         rs[i]=0;
17         return false;
18     }
19
20     vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
21         vector<vector<int>> graph(n);
22         for(auto it:pre){
23             int v = it[1];
24             int u = it[0];
25             graph[v].push_back(u);
26         }
27         return graph;
28     }
29
30     vector<int> findOrder(int n, vector<vector<int>>& pre) {
31         vector<vector<int>> graph = createGraph(n, pre);
32         vector<int> vis(n,0), rs(n,0), traversal;
33         for(int i=0; i<n; i++){
34             if(vis[i]==0)
35                 if(dfs(graph, i, vis, rs, traversal)) return {};
36         }
37         reverse(traversal.begin(), traversal.end());
38         return traversal;
39     }
40 }
```

Graph - 2

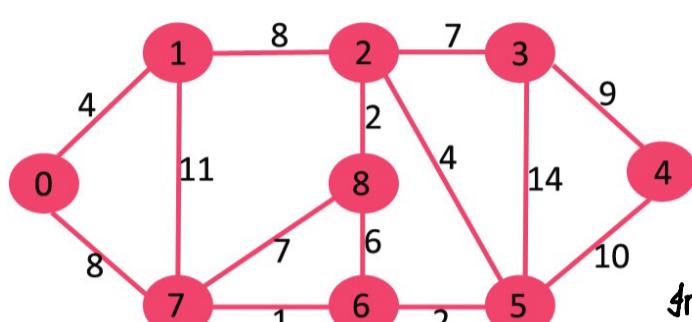
- Karun Karthik

Content

11. Dijkstra Algorithm
12. Network Delay Time
13. Bellman Ford Algorithm
14. Negative Weight Cycle
15. Floyd Warshall Algorithm
16. Prim's Algorithm
17. Min Cost to Connect All Points
18. Is Graph Bipartite ?
19. Possible Bipartition
20. Disjoint Set
21. Kruskal's Algorithm
22. Critical Connection in a Network

11) Dijkstra Algorithm → single source shortest path (only +ve weights)

→ Helps in finding the shortest path to every node from src node.



$n = 9$ (nodes from 0 to 8)

$src = 1$

dist away = min cost from src to every other vertex

initially cost =

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

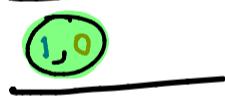
 vis = { } 3

→ As it is weighted graph, we'll use priority queue (PQ) instead of normal queue. An element pushed into it will be of form curr node, curr cost

→ PQ always pops element with least curr cost, always calculated from src to curr node.



⇒ now neighbours of 1 = 0, 4 7, 11 2, 8 ∴ push



vis = {1}

cost[1] = 0

→ lowest cost among 4, 11, 8
is 4 ∴ pop it & push its neighbours.

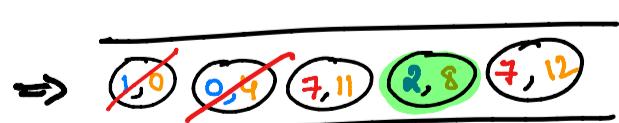
⇒
⇒ now neighbours of 0 = 1 (visited), 7, 12 ∴ push

vis = {1, 0}

cost[0] = 4



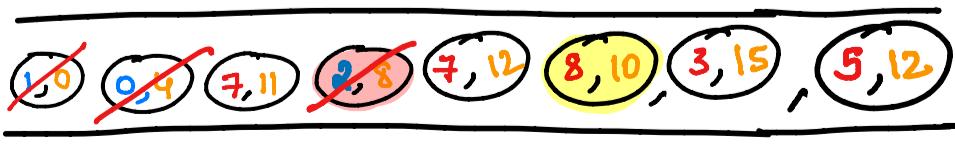
→ lowest cost is 8 ∴
pop & push its neighbours



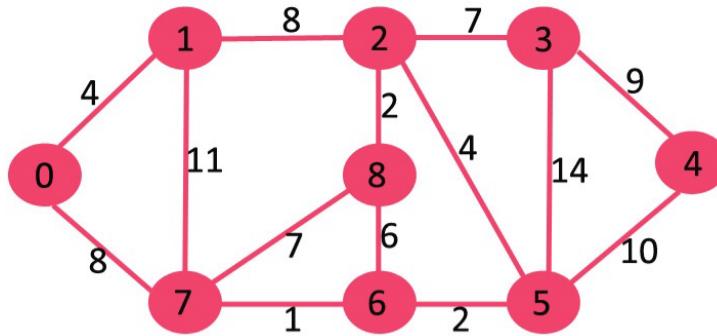
⇒ neighbours of 2 = 1 (visited), 8, 10, 3, 15, 5, 12 ∴ push

vis = {1, 0, 2}

cost[2] = 8



→ lowest cost is 10 ∴
pop & push its neighbours



⇒



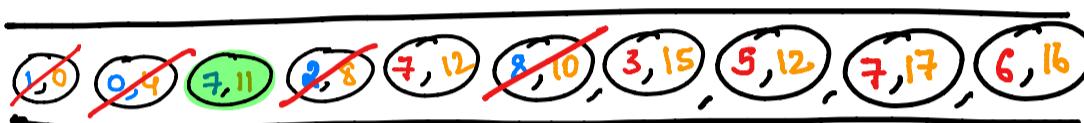
$$\text{vis} = \{1, 0, 2, 8\}$$

$$\text{cost}[8] = 10$$



↳ lowest cost = 11 ∴ pop & push its neighbours.

⇒



⇒ neighbours of 7 = 0, 1, 8 are visited.

& ~~6,12~~ ∴ push

$$\text{vis} = \{1, 0, 2, 8, 7\}$$

$$\text{cost}[7] = 11$$



↳ lowest cost = 12

∴ Anything among 5, 6 can be selected & pop & push its neighbours
Not 7, because it is already visited & cost is < 12.

⇒



⇒ neighbours of 5 = ~~4,22~~, ~~3,26~~, ~~6,14~~ ∴ push

$$\text{vis} = \{1, 0, 2, 8, 7, 5\}$$

$$\text{cost}[5] = 12$$



→ lowest cost = 12
∴ pop & push its neighbours

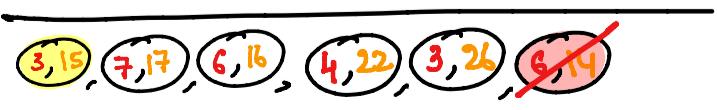
⇒



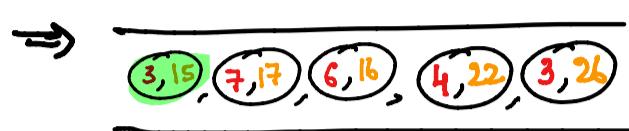
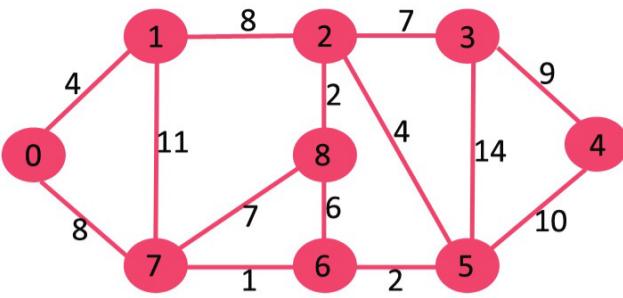
⇒ neighbours of 6 = 5, 7, 8 are visited .

∴ no push

→ next lowest is 14, but 6 is already visited .



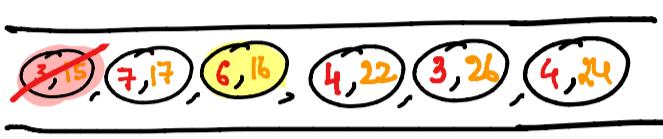
∴ Next lowest is 15, ∴ pop & push its neighbours



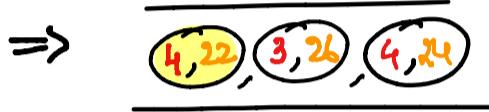
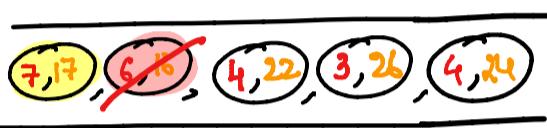
\Rightarrow neighbours of 3 = 2, 5 (visited) $(4, 24)$ \therefore push

$$vis = \{1, 0, 2, 8, 7, 5, 6, 3\}$$

$$cost[3] = 15$$



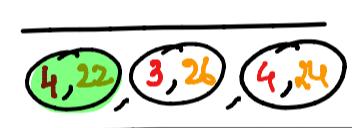
\rightarrow next lowest cost = 16
but 6 is already visited \therefore pop



\rightarrow next lowest cost = 22

\therefore pop q & push its neighbour.

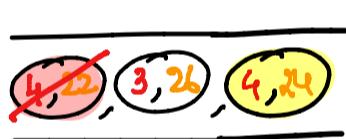
\rightarrow next lowest cost = 17
but 7 is already visited \therefore pop



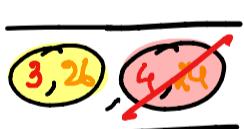
\Rightarrow neighbours of 4 = 3, 5 (visited) \therefore no push

$$vis = \{1, 0, 2, 8, 7, 5, 6, 3, 4\}$$

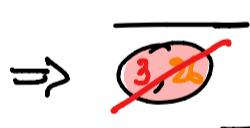
$$cost[4] = 22$$



\rightarrow next lowest cost = 24
but 4 is already visited
 \therefore pop



\rightarrow next lowest cost = 26
but 3 is already visited
 \therefore pop



\therefore empty PQ.

Answer \Rightarrow

4	0	8	15	22	12	12	11	10
0	1	2	3	4	5	6	7	8

Dijkshaus = BFS + PQ

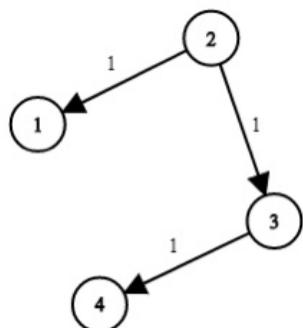
$T_C \rightarrow O(V + E \log V)$

$S_C \rightarrow O(V)$

Code →

```
1 class Solution
2 {
3     public:
4     vector <int> dijkstra(int V, vector<vector<int>> adj[], int src)
5     {
6         vector<int>cost(V,0);
7         cost[src]=0;
8
9         vector<bool>vis(V, false);
10        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;
11
12        pq.push({0,src}); // {cost, node}
13
14        while(!pq.empty())
15        {
16            pair<int,int>p = pq.top();
17            int currCost = p.first;
18            int currNode = p.second;
19            pq.pop();
20
21            if(vis[currNode]) continue;
22
23            vis[currNode] = true;
24            cost[currNode] = currCost;
25
26            for(int i=0;i<adj[currNode].size();i++)
27            {
28                int neighbourNode = adj[currNode][i][0];
29                int weight = adj[currNode][i][1];
30                // if already visited then skip
31                if(vis[neighbourNode]) continue;
32                // else push
33                pq.push({currCost + weight, neighbourNode});
34            }
35        }
36        return cost;
37    }
38 };
39
```

12 Network Delay Time



$\text{src} = 2$.

You are given a network of n nodes, labeled from 1 to n . You are also given times , a list of travel times as directed edges $\text{times}[i] = (u_i, v_i, w_i)$, where u_i is the source node, v_i is the target node, and w_i is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k . Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1 .

✓ Similar to Dijkstra's algo. $\text{cost} = \boxed{0 \ 0 \ 0 \ 0 \ 0}$ $\text{vis} = \{2\}$ $\text{pq} = \underline{\quad \quad \quad}$

\Rightarrow push $(2, 0)$ to pq . $\Rightarrow \underline{(2, 0) \quad \quad \quad}$

$\Rightarrow \underline{(2, 0)}$ $\text{neighbours} = \underline{(1, 1), (3, 1)}$ \therefore push
 $\text{vis} = \{2\}$ $\text{cost}[2] = 0$ \rightarrow next lowest cost = 1 \therefore choose 1 or 3
 \therefore pop & push their neighbour.

$\Rightarrow \underline{(1, 1), (3, 1)}$ no new neighbours \therefore pop
 $\text{vis} = \{2, 1\}$ $\text{cost}[1] = 1$ \rightarrow next lowest cost = 1
 \therefore pop & push their neighbour.

$\Rightarrow \underline{(3, 1)}$ neighbour = $(4, 2)$ \therefore push
 $\text{vis} = \{2, 1, 3\}$ $\text{cost}[3] = 1$ \rightarrow next lowest cost = 2
 \therefore pop & push neighbours.

$\Rightarrow \underline{(4, 2)}$ no new neighbours \therefore pop
 $\text{vis} = \{2, 1, 3, 4\}$ $\text{cost}[4] = 2$ \rightarrow pq is empty.

$\therefore \text{cost} = \boxed{0 \ 1 \ 0 \ 1 \ 2}$

$T_c \rightarrow O(V + E \log V)$
 $S_c \rightarrow O(V)$

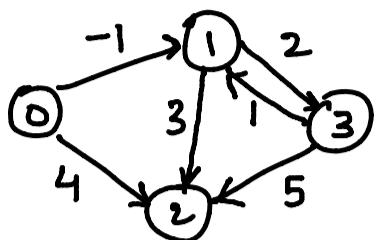
\rightarrow check if all nodes are in visited,
else return -1.
 \rightarrow Return max value in cost as

Code →

```
1  class Solution {
2  public:
3
4      int networkDelayTime(vector<vector<int>>& times, int n, int k) {
5          vector<vector<vector<int>>> graph = createGraph(times,n);
6          return minTime(graph,n,k);
7      }
8
9      vector<vector<vector<int>>> createGraph(vector<vector<int>>& edges,int n) {
10
11         vector<vector<vector<int>>> graph(n+1);
12
13         for(int i=0;i<=n;i++) {
14             graph.push_back({{}});
15         }
16         // add every edge to the graph
17         for(vector<int> edge:edges) {
18             int source = edge[0];
19             int dest = edge[1];
20             int cost = edge[2];
21             graph[source].push_back({dest,cost});
22         }
23         return graph;
24     }
25
26     int minTime(vector<vector<vector<int>>> &graph,int n,int src) {
27
28         vector<int> cost(n+1,0);
29         cost[src] = 0;
30         vector<bool>vis(n+1, false);
31
32         priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
33         pq.push({0,src}); // {cost, node}
34
35         while(!pq.empty()) {
36             pair<int,int>p = pq.top();
37             int currNode = p.second;
38             int currCost = p.first;
39             pq.pop();
40             // if already visited then skip
41             if(vis[currNode])    continue;
42
43             vis[currNode] = true;
44             cost[currNode] = currCost;
45
46             for(int i=0;i<graph[currNode].size();i++)
47             {
48                 int neighbourNode = graph[currNode][i][0];
49                 int weight = graph[currNode][i][1];
50                 // if already visited then skip
51                 if(vis[neighbourNode])  continue;
52                 // else push into pq
53                 pq.push({currCost + weight, neighbourNode});
54             }
55         }
56
57         for(int i=1; i<=n; i++)
58             if(vis[i]==0)    return -1;
59
60         int ans = 0;
61         for(int x:cost)    ans = max(ans,x);
62         return ans;
63     }
64 }
```

⑬ Bellman Ford Algorithm → useful when weights < 0 (Dijkstra fails)
 ↳ dp algo → useful when finding negative weight cycle.
 [src, dest, wt]

Eg $n = 4$ edges = $\{[0, 1, -1], [0, 2, 4], [1, 2, 3], [1, 3, 2], [3, 1, 1], [3, 2, 5]\}$



initially dist

inf	inf	inf	inf
0	1	2	3

$\Rightarrow \text{dist}[0] = 0$ &

\Rightarrow relax every edge $n-1$ time is run for loop & perform the following operation

$$\text{dist}[dest] = \min(\text{dist}[src] + \text{weight}, \text{dist}[dest])$$

\Rightarrow finally relax one more time &

if $\text{dist}[dest] > \text{dist}[src] + \text{wt} \Rightarrow$ -ve weight cycle present

\Rightarrow we should relax 3 times & $src=0 \Rightarrow \text{dist}[0] = 0$ $\text{dist} \begin{array}{|c|c|c|c|}\hline 0 & \text{inf} & \text{inf} & \text{inf} \\ \hline 0 & 1 & 2 & 3 \\ \hline \end{array}$

\rightarrow for edge $[0, 1, -1]$, $\text{dist}[1] = \min(0 + (-1), \text{inf}) = -1$

$[0, 2, 4]$, $\text{dist}[2] = \min(0 + 4, \text{inf}) = 4$

$[1, 2, 3]$, $\text{dist}[2] = \min(-1 + 3, 4) = 2$

$[1, 3, 2]$, $\text{dist}[3] = \min(-1 + 2, \text{inf}) = 1$

$[3, 1, 1]$, $\text{dist}[1] = \min(1 + 1, -1) = -1$

$[3, 2, 5]$, $\text{dist}[2] = \min(1 + 5, 2) = 2$.

$$\therefore \text{dist} = \begin{array}{|c|c|c|c|}\hline 0 & -1 & 2 & 1 \\ \hline 0 & 1 & 2 & 3 \\ \hline \end{array}$$

\rightarrow now use the above dist & perform same operation twice, in this case dist remains same.

\rightarrow during final relaxation, -ve weight cycle condition is not met.

Answer $\Rightarrow \text{dist} = \begin{array}{|c|c|c|c|}\hline 0 & -1 & 2 & 1 \\ \hline 0 & 1 & 2 & 3 \\ \hline \end{array}$

$$\begin{aligned} TC &\rightarrow O(V * E) \\ SC &\rightarrow O(V) \end{aligned}$$

⑯ Negative weight cycle → Bellman Ford Algorithm.

→ To check the presence of negative weight cycle using Bellman Ford Algorithm.

$$TC \rightarrow O(V * E)$$
$$SC \rightarrow O(V)$$

Code →

```
● ● ●  
1 class Solution {  
2 public:  
3     int isNegativeWeightCycle(int n, vector<vector<int>>edges){  
4         vector<int>dis(n, INT_MAX);  
5         // initially, dist to src is 0  
6         dis[0] = 0;  
7         // relax n-1 times  
8         for(int i=0;i<n-1;i++)  
9         {  
10             for(auto edge:edges)  
11             {  
12                 int src = edge[0];  
13                 int dest = edge[1];  
14                 int wt = edge[2];  
15                 if(dis[src]!=INT_MAX) // to avoid integer overflow  
16                     dis[dest] = min(dis[dest],dis[src]+wt);  
17             }  
18         }  
19         // final relaxation  
20         for(auto edge:edges)  
21         {  
22             int src = edge[0];  
23             int dest = edge[1];  
24             int wt = edge[2];  
25             if(dis[src]!=INT_MAX && dis[dest]>dis[src]+wt)  
26                 return 1;  
27         }  
28         return 0;  
29     }  
30 };
```

15) Floyd Warshall Algorithm

- All source shortest path & -ve edges allowed.
- Since its all source shortest path we need to run the loop for all nodes, considering it as intermediary vertex.
- $\text{cost}[i][j] = \min(\text{cost}[i][j], \text{cost}[i][k] + \text{cost}[k][j])$

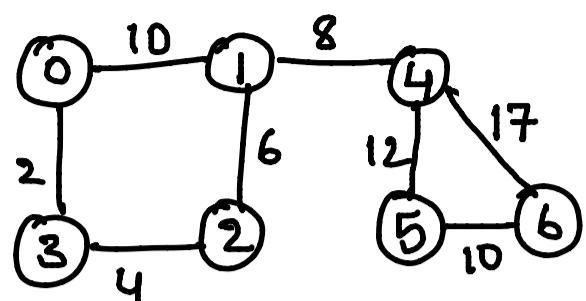
$$TC \rightarrow O(N^3) \quad SC \rightarrow O(N^2)$$

Code →

```
● ● ●  
1 class Solution {  
2     public:  
3         void shortest_distance(vector<vector<int>>&matrix){  
4             int V = matrix.size();  
5             vector<vector<int>> costs(matrix.size(), vector<int>(matrix.size()));  
6  
7             for(int i=0;i<V;i++)  
8                 for(int j=0;j<V;j++)  
9                     costs[i][j] = matrix[i][j];  
10  
11            for(int k=0;k<V;k++)  
12                for(int i=0;i<V;i++)  
13                    for(int j=0;j<V;j++){  
14                        // if intermediate is not -1 then  
15                        if(costs[i][k]!=-1 && costs[k][j]!=-1){  
16                            if(costs[i][j]==-1)  
17                                costs[i][j] = costs[i][k]+costs[k][j];  
18                            else  
19                                costs[i][j] = min(costs[i][j], costs[i][k]+costs[k][j]);  
20                        }  
21                    }  
22  
23            for(int i=0;i<V;i++)  
24                for(int j=0;j<V;j++)  
25                    matrix[i][j] = costs[i][j];  
26  
27        }  
28    };
```

16 Prim's Algorithm → Minimum Spanning Tree (MST)

Eg

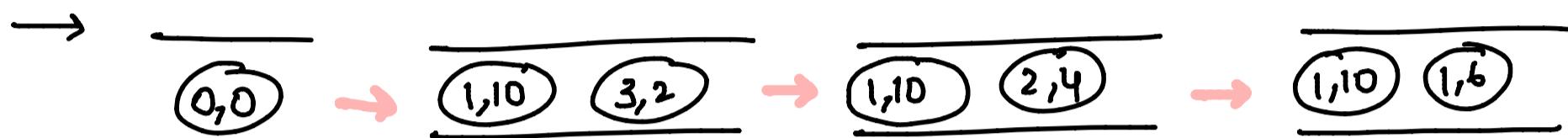


$$Vis = \{ \}$$

PQ (node, weight)

↑ returns node with lowest cost/weight.

* To find MST, just push node along with its weight.

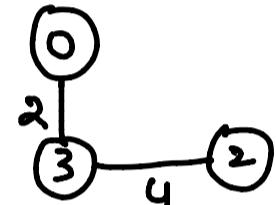


$$Vis = \{ \}$$

$$Vis = \{ 0 \}$$

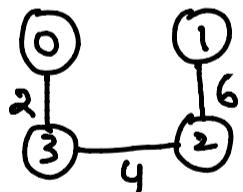
$$Vis = \{ 0, 3 \}$$

$$Vis = \{ 0, 3, 2 \}$$



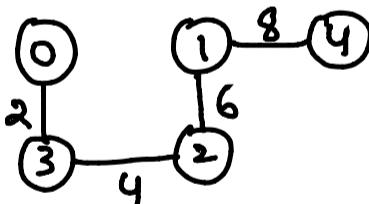
$$\begin{array}{c} 1,10 \\ 4,8 \end{array}$$

$$Vis = \{ 0, 3, 2, 1 \}$$



$$\begin{array}{c} 1,10 \\ 5,12 \\ 6,17 \end{array}$$

$$Vis = \{ 0, 3, 2, 1, 4 \}$$



$$\begin{array}{c} 5,12 \\ 6,17 \end{array}$$

$$Vis = \{ 0, 3, 2, 1, 4 \}$$

$$\begin{array}{c} 6,17 \\ 6,10 \end{array}$$

$$Vis = \{ 0, 3, 2, 1, 4, 5 \}$$

$$\begin{array}{c} 6,17 \\ 6,10 \\ 8,4 \end{array}$$

$$\begin{array}{c} 6,17 \\ 6,10 \\ 8,4 \end{array}$$

$$\begin{array}{c} 6,17 \\ 6,10 \\ 8,4 \\ 12,5 \end{array}$$

$$\begin{array}{c} 6,17 \\ 6,10 \\ 8,4 \\ 12,5 \end{array}$$

$$\begin{array}{c} 6,17 \\ 6,10 \\ 8,4 \\ 12,5 \end{array}$$

$$TC \rightarrow O(V + E \log V)$$

$$SC \rightarrow O(V)$$

Code →

```
1 class Solution
2 {
3     public:
4     //Function to find sum of weights of edges of the Minimum Spanning Tree.
5     int spanningTree(int V, vector<vector<int>> adj[])
6     {
7         int minCost = 0;
8         vector<int> costs(V, INT_MAX);
9         costs[0] = 0;
10        vector<bool> vis(V, false);
11        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;
12        pq.push({0,0}); // {cost, Node}
13
14        while(!pq.empty())
15        {
16            pair<int,int> p = pq.top();
17            int currNode = p.second;
18            int currCost = p.first;
19            pq.pop();
20
21            if(vis[currNode])    continue;
22
23            minCost += currCost;
24
25            vis[currNode] = true;
26            costs[currNode] = currCost;
27
28            for(int i=0;i<adj[currNode].size();i++)
29            {
30                int neighbourNode = adj[currNode][i][0];
31                int neighbourNodeCost = adj[currNode][i][1];
32                if(vis[neighbourNode])  continue;
33                pq.push({neighbourNodeCost, neighbourNode});
34            }
35        }
36        return minCost;
37    }
38 };
39
```

17 Min Cost to Connect all points

→ Create graph with each node containing $Wt \triangleq$ Node value

$$Wt = \text{abs}(X_i - X) + \text{abs}(Y_i - Y)$$

→ Perform Prims algo.

$$Tc \rightarrow O(V + E \log V)$$

$$Sc \rightarrow O(V)$$

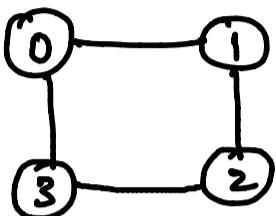
Code →

```
1
2 class Solution {
3 public:
4     int minCostConnectPoints(vector<vector<int>>& points) {
5
6         int n = points.size();
7         vector<vector<pair<int, int>>> graph(n);
8
9         for (int i = 0; i < n; i++) {
10            for (int j = 0; j < n; j++) {
11                if (i == j) continue;
12                graph[i].push_back({abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1]), j});
13            }
14        }
15
16        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
17        vector<bool> vis(n, false);
18        pq.push({0, 0}); // {cost, Node}
19
20        int ans = 0;
21        while (!pq.empty())
22        {
23            pair<int,int> p = pq.top();
24            int currNode = p.second;
25            int currCost = p.first;
26            pq.pop();
27
28            if (vis[currNode]) continue;
29            ans += currCost;
30            vis[currNode] = true;
31
32            for(int i=0;i<graph[currNode].size();i++)
33            {
34                int neighbourNode = graph[currNode][i].second;
35                int neighbourNodeCost = graph[currNode][i].first;
36                if(vis[neighbourNode]) continue;
37                pq.push({neighbourNodeCost, neighbourNode});
38            }
39        }
40        return ans;
41    }
42 }
43 };
44 }
```

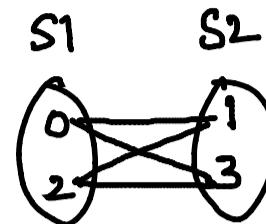
(18) Is graph Bipartite

Bipartite graph is undirected graph, such that all vertices can be divided into 2 sets, $S_1 \& S_2$ and no two vertices present in same set share an edge.

Eg $n = 4$



thus



\therefore the graph is bipartite.

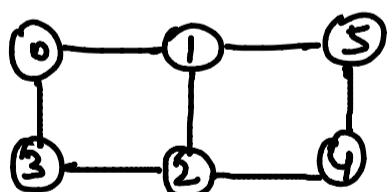
\Rightarrow for graph to be bipartite,

- it needs to be undirected acyclic graph (or)
- it needs to be even length cyclic graph

\rightarrow we generally denote sets by coloring it, color = 0, 1.

$$\begin{matrix} \downarrow & \downarrow \\ S_1 & S_2 \end{matrix}$$

Eg $n = 6$



$$\text{vis} = \{3\} \quad S_1 = \{3\} \quad S_2 = \{3\}$$

initially color

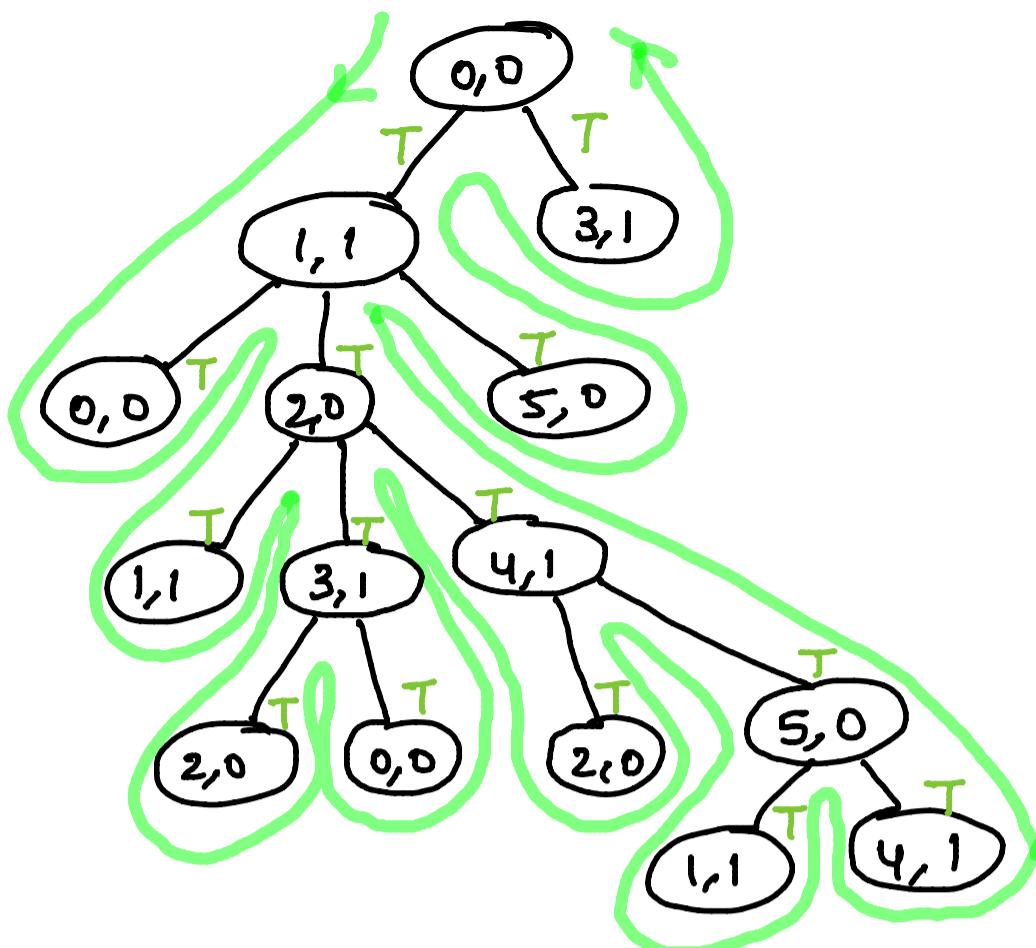
-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

\rightarrow at each vertex, check if it's visited or not.

\rightarrow if visited then check if it's present in the intended set or not.

\rightarrow if yes then return true, else false

\rightarrow return AND of all the boolean values.



Code →

```
● ● ●  
1 class Solution {  
2 public:  
3  
4     bool isBipartite(vector<vector<int>>& graph) {  
5  
6         int n= graph.size();  
7         vector<int>colors(n,-1);  
8  
9         for(int curr=0; curr<n ; curr++){  
10             // if already colored then skip  
11             if(colors[curr]!=-1)  continue;  
12             // check for even length cycle  
13             if(hasEvenLengthCycle(graph, curr, 0, colors)==false)  return false;  
14         }  
15         return true;  
16     }  
17  
18     bool hasEvenLengthCycle(vector<vector<int>>& graph,int curr,int color,vector<int>&colors)  
19     {  
20         if(colors[curr]!=-1)  
21             return colors[curr]==color;  
22  
23         // if not colored then color it  
24         colors[curr] = color;  
25  
26         // check for neighbours  
27         for(int neigh: graph[curr])  
28         {  
29             if(hasEvenLengthCycle(graph, neigh, 1-color, colors)==false)  
30                 // 1- color will handle both changing colors 0 to 1 and 1 to 0  
31                 return false;  
32         }  
33         return true;  
34     }  
35  
36 };
```

19 Possible Bipartition →

- Create a graph using dislikes array.
- use previous problem's approach to solve it.

Code →

TC → O(V+E) SC → O(V+E)

```
● ○ ●  
1 class Solution {  
2 public:  
3  
4     bool dfs(vector<int> graph[], int curr, vector<int>& color){  
5  
6         // if not colored then color  
7         if(color[curr] == -1)  
8             color[curr] = 1;  
9  
10        // process the neighbours and check their colors  
11        for(auto neigh : graph[curr])  
12        {  
13            if(color[neigh] == -1)  
14            {  
15                color[neigh] = 1 - color[curr];  
16                if(dfs(graph, neigh, color)==false) return false;  
17            }  
18            else if(color[neigh] == color[curr]) return false;  
19        }  
20        return true;  
21    }  
22  
23    bool possibleBipartition(int n, vector<vector<int>>& dislikes) {  
24        vector<int> color(n+1, -1);  
25        vector<int> graph[n+1];  
26  
27        // populating the graph  
28        for(auto edge : dislikes){  
29            graph[edge[0]].push_back(edge[1]);  
30            graph[edge[1]].push_back(edge[0]);  
31        }  
32  
33        for(int i=1; i<=n; i++){  
34            if(color[i] == -1)  
35                if(!dfs(graph, i, color)) return false;  
36        }  
37  
38        return true;  
39    }  
40};
```

20 Disjoint Set \rightarrow UNION & FIND./getParent

\hookdownarrow helps in finding parent of component
helps in UNION of components/vertices.

Eg $0 \ 1 \Rightarrow \text{UNION}(0,1) \rightarrow$ 

Eg $n=7$ initially every component is parent of itself



parent =	<table border="1" data-bbox="696 819 1466 983"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	0	1	2	3	4	5	6	0	1	2	3	4	5	6
0	1	2	3	4	5	6									
0	1	2	3	4	5	6									

now $\text{getParent}(2) = 2$, $\text{getParent}(3) = 3$.

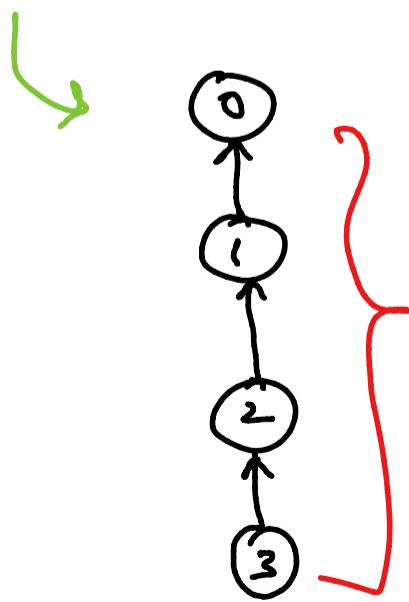
& if $\text{UNION}(0,1) \Rightarrow$  & $\text{parent}[1] = 0$

now $\text{getParent}(1) = 0$

& $\text{UNION}(1,2) \Rightarrow$ 

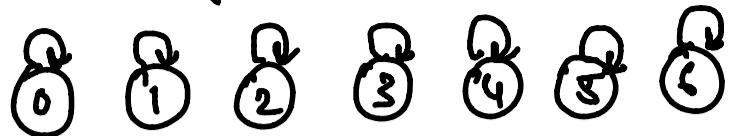
$\text{UNION}(2,3) \Rightarrow$ 

& $\text{getParent}(3) = 0$



This increases the recursive calls
and the tree is unbalanced
so we'll use rank array to
store min. height tree for node.

$n=7$ initially every component is parent of itself



parent =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	0	1	2	3	4	5	6	0	1	2	3	4	5	6
0	1	2	3	4	5	6									
0	1	2	3	4	5	6									

rank =	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	0	0	0	0	0	0	0	0	1	2	3	4	5	6
0	0	0	0	0	0	0									
0	1	2	3	4	5	6									

$\Rightarrow \text{UNION}(0,1) \Rightarrow$ then $\text{find}(0) \neq \text{find}(1) \neq 0 \neq 1 \therefore$ diff components.
as they are diff components find rank & $\text{rank}[0] = \text{rank}[1] = 0$

\therefore select either 0 or 1 & make it as root & inc the rank by 1



parent =	<table border="1"><tr><td>0</td><td>0</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	0	0	2	3	4	5	6	0	1	2	3	4	5	6
0	0	2	3	4	5	6									
0	1	2	3	4	5	6									

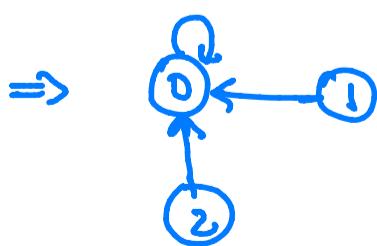
rank =	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	1	0	0	0	0	0	0	0	1	2	3	4	5	6
1	0	0	0	0	0	0									
0	1	2	3	4	5	6									

$\Rightarrow \text{UNION}(1,2) \Rightarrow \text{parent}(1)=0 \neq \text{parent}(2)=2$
now $\text{rank}[0]=1 \neq \text{rank}[2]=0$

as $\text{rank}[0] > \text{rank}[2]$,

vertex 0 should be the parent

& do not update rank if they are unequal.



parent =	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	0	0	0	3	4	5	6	0	1	2	3	4	5	6
0	0	0	3	4	5	6									
0	1	2	3	4	5	6									

rank =	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	1	0	0	0	0	0	0	0	1	2	3	4	5	6
1	0	0	0	0	0	0									
0	1	2	3	4	5	6									

Code

```
1  class DisjSet {
2      int *rank, *parent, n;
3
4      public:
5      DisjSet(int n)
6      {
7          rank = new int[n];
8          parent = new int[n];
9          this->n = n;
10         makeSet();
11     }
12
13     void makeSet()
14     {
15         for (int i = 0; i < n; i++) {
16             parent[i] = i;
17         }
18     }
19
20     int find(int x)
21     {
22         // if x is not parent of itself then
23         // find parent recursively
24         if (parent[x] != x) {
25             parent[x] = find(parent[x]);
26         }
27         return parent[x];
28     }
29
30     void Union(int x, int y)
31     {
32         int xset = find(x);
33         int yset = find(y);
34
35         // if set of x and y are same then return
36         if (xset == yset)    return;
37
38         // place the elements in small rank
39         if (rank[xset] < rank[yset]) {
40             parent[xset] = yset;
41         }
42         else if (rank[xset] > rank[yset]) {
43             parent[yset] = xset;
44         }
45         // if same rank then increment it
46         else {
47             parent[yset] = xset;
48             rank[xset] = rank[xset] + 1;
49         }
50     }
51 };
52 }
```

(21)

Kruskal's Algorithm →

- This is used to find minimum spanning tree.
- can be implemented using Disjoint set.
- sort all the edges in ↑ order of weight.
- pick smallest edge & check if it contributes to cycle in graph
- if yes then discard else include.

Code →

```

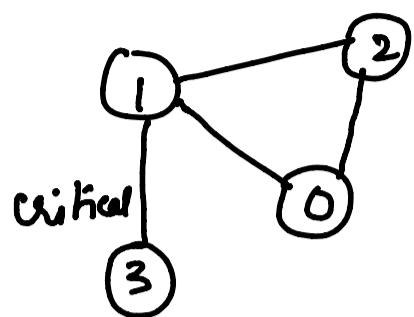
● ● ●

1 class Graph {
2     vector<vector<int>> edgelist;
3     int V;
4
5 public:
6     Graph(int V) { this->V = V; }
7
8     void addEdge(int x, int y, int w)
9     {
10         edgelist.push_back({ w, x, y });
11     }
12
13     void kruskals_mst()
14     {
15         // 1. Sort all edges
16         sort(edgelist.begin(), edgelist.end());
17
18         // Initialize the DSU - DisjointSet
19         DSU s(V);
20         int ans = 0;
21         for (auto edge : edgelist) {
22             int w = edge[0];
23             int x = edge[1];
24             int y = edge[2];
25             // take that edge in MST if it does form a cycle
26             if (s.find(x) != s.find(y)) {
27                 s.union(x, y);
28                 ans += w;
29                 cout << x << " -- " << y << " == " << w
30                             << endl;
31             }
32         }
33         cout << "Minimum Cost Spanning Tree: " << ans;
34     }
35 };

```

22 Critical Connection in a Network

Eg $n=4$ edges = $[[0, 1], [1, 2], [2, 0], [1, 3]]$



→ Critical connection is a connection, when removed from graph, would result in breaking graph into different components.

Here if $[1, 3]$ is removed then graph becomes disconnected.

Approach 1

- Remove one edge each time
- Perform dfs
- If all vertices are not visited then
- Removed edge is a critical connection.

Approach 2

- Initialise distime array & mintime array with -1.
- discovery time for vertex → min time for vertex to be discovered.
- perform dfs from one node
 - if $\text{neighbours} == \text{parent}$ then continue
 - else if neighbour is already visited then
 $\text{mintime}[\text{curr}] = \min(\text{mintime}[\text{curr}], \text{distime}[\text{neigh}])$
 - while returning $\text{mintime}[\text{curr}] = \min(\text{mintime}[\text{curr}], \text{mintime}[\text{neigh}])$
if at any point if $\text{distime}[\text{curr}] < \text{mintime}[\text{neigh}]$
This indicates critical connection

Code →

```
● ● ●
1 class Solution {
2 public:
3
4     vector<vector<int>> criticalConnections(int n, vector<vector<int>& connections) {
5         vector<int> graph[n];
6         for(vector<int> edge: connections){
7             int u = edge[0];
8             int v = edge[1];
9             graph[u].push_back(v);
10            graph[v].push_back(u);
11        }
12        return findCriticalConnections(n, graph);
13    }
14
15    vector<vector<int>> findCriticalConnections(int n, vector<int> graph[]){
16        vector<int> disTime(n,-1);
17        vector<int> lowTime(n,-1);
18        int time = 0;
19        vector<vector<int>> answer;
20        tarjansDFS(graph, 0, -1, disTime, lowTime, time, answer);
21        return answer;
22    }
23
24    void tarjansDFS(vector<int> graph[], int curr, int parent, vector<int>&disTime,
25    vector<int> &lowTime, int &time, vector<vector<int>> &answer){
26
27        disTime[curr] = time;
28        lowTime[curr] = time;
29        time += 1;
30
31        for(int neigh: graph[curr]){
32            if(neigh == parent) continue;
33
34            if(disTime[neigh]!=-1){
35                lowTime[curr] = min(lowTime[curr], disTime[neigh]);
36                continue;
37            }
38
39            tarjansDFS(graph, neigh, curr, disTime, lowTime, time, answer);
40            lowTime[curr] = min(lowTime[curr], lowTime[neigh]);
41
42            if(disTime[curr] < lowTime[neigh]){
43                vector<int> temp;
44                temp.push_back(curr);
45                temp.push_back(neigh);
46                answer.push_back(temp);
47            }
48        }
49        return;
50    }
51
52};
```