

Graph

- Karun Karthik

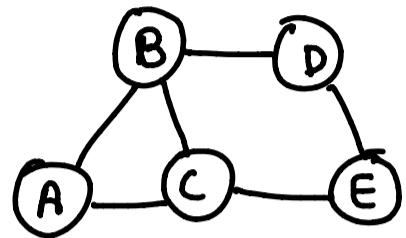
Contents

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



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

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

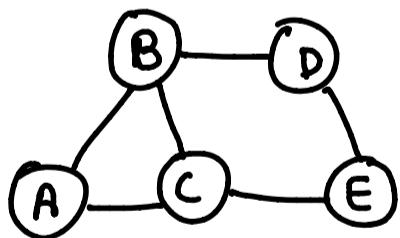
Applications →

Google maps → To find shortest route

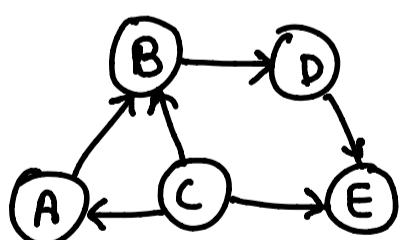
Social network → user, connection
 ↑
 vertex edge

Types →

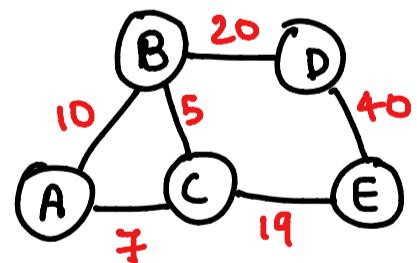
1) Undirected



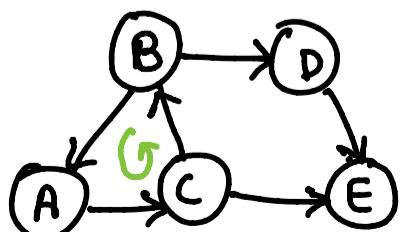
2) Directed



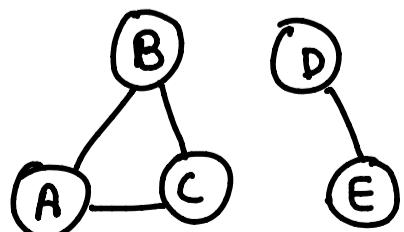
3) Weighted



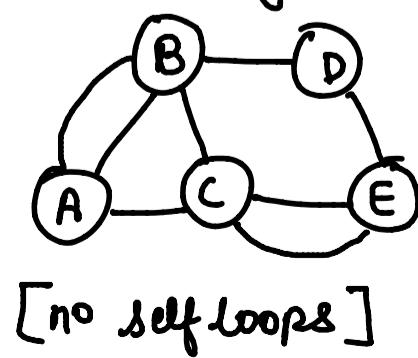
4) Cyclic



5) Disconnected



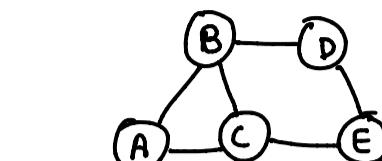
6) Multigraph



Graph Traversal

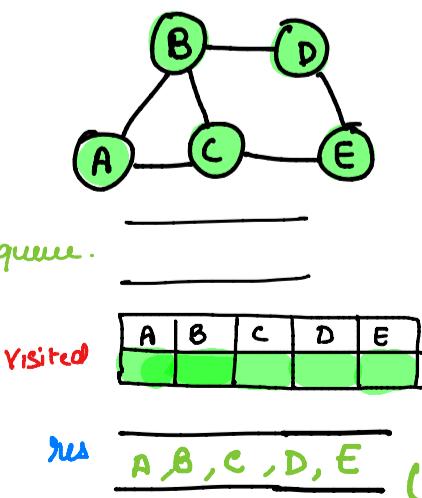
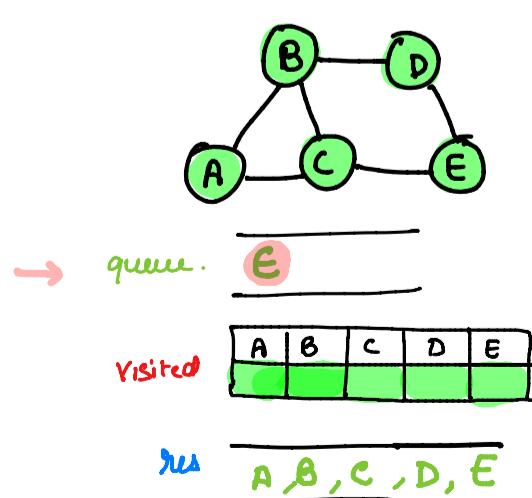
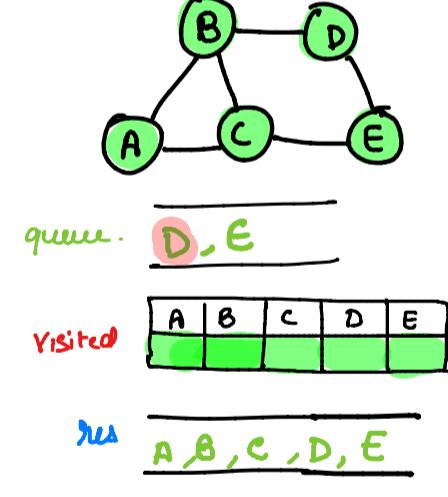
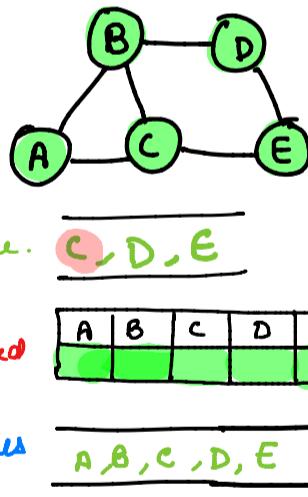
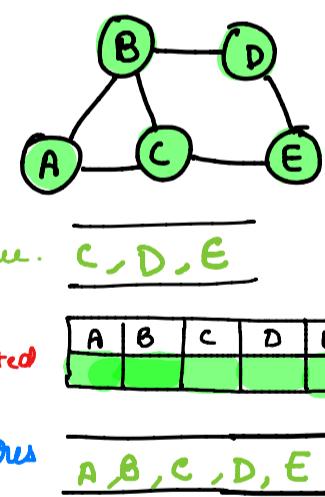
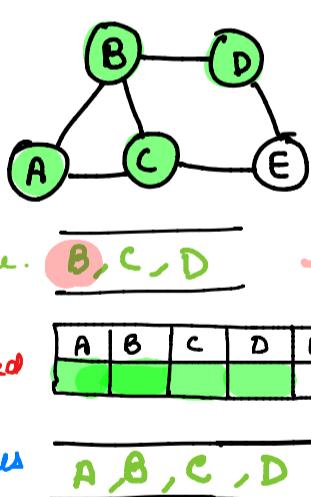
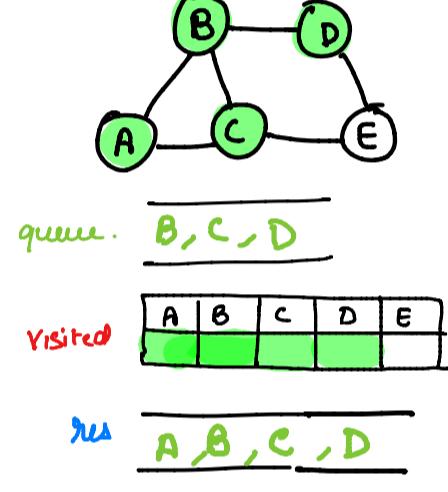
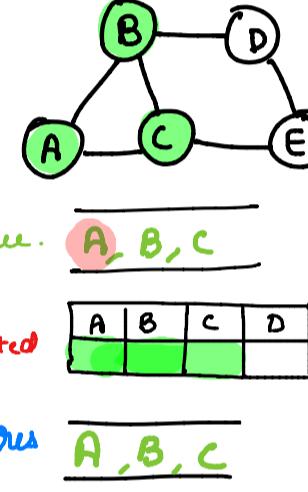
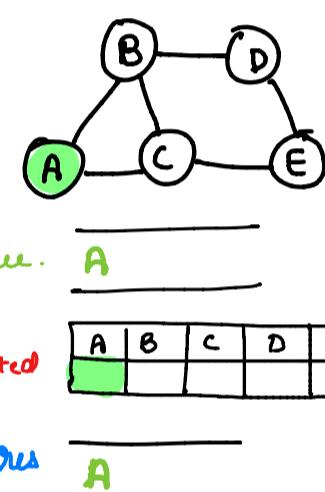
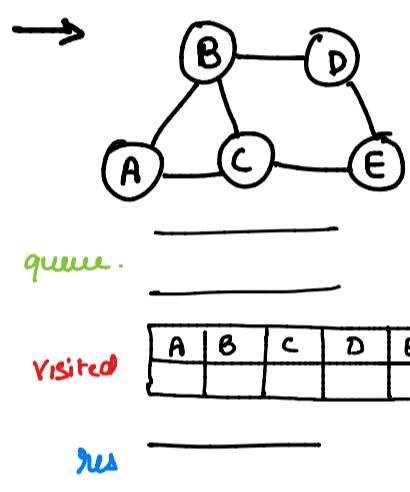
(a) BFS → visit each and every vertex in a defined order.

- 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);

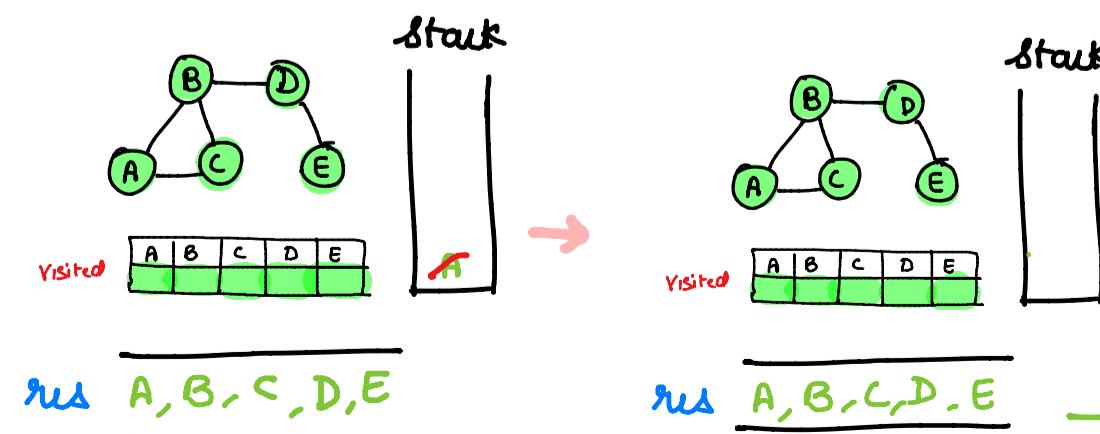
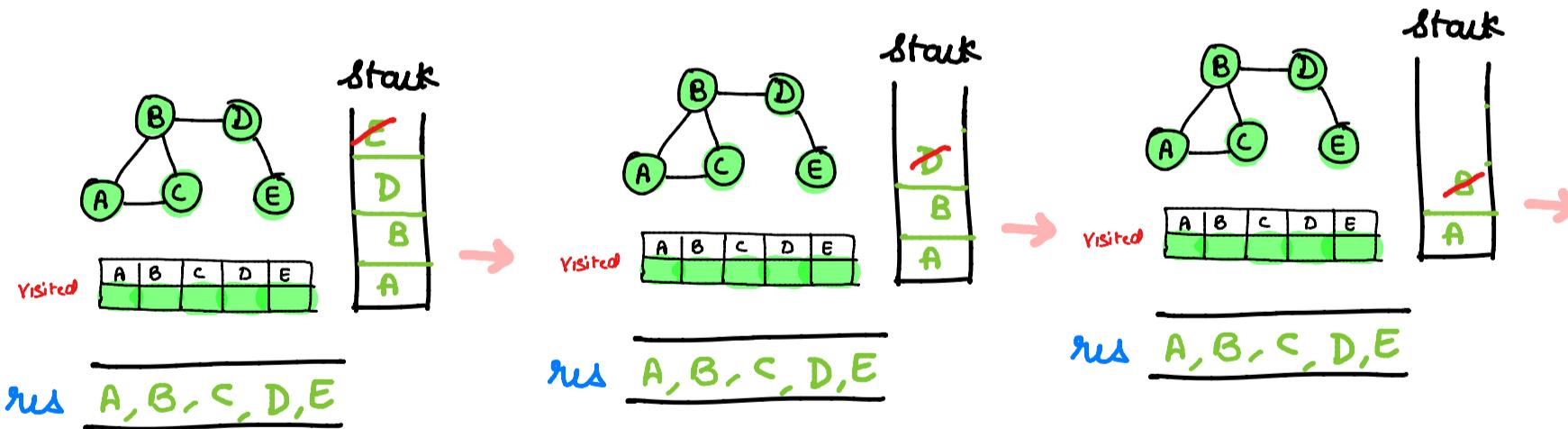
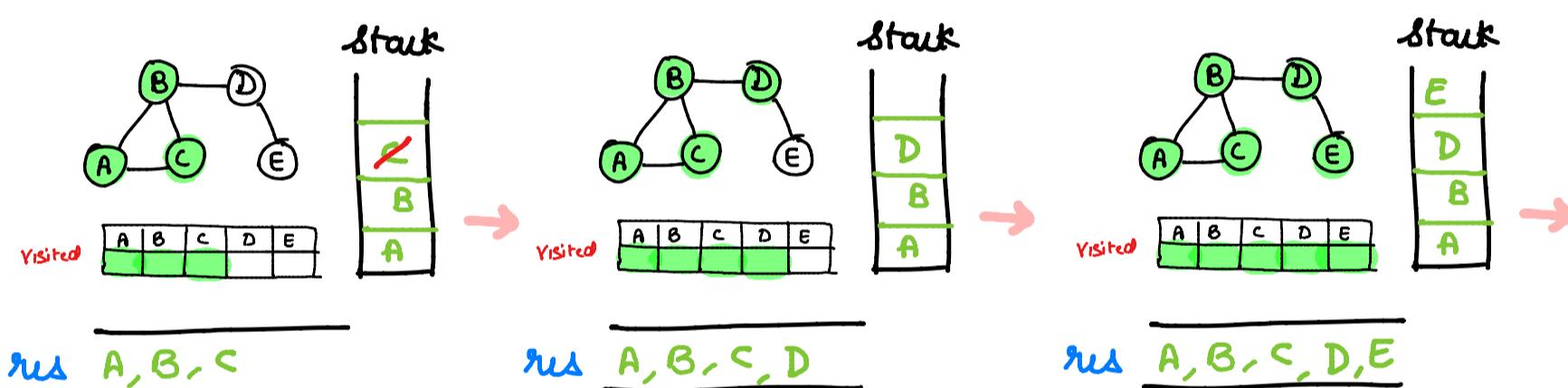
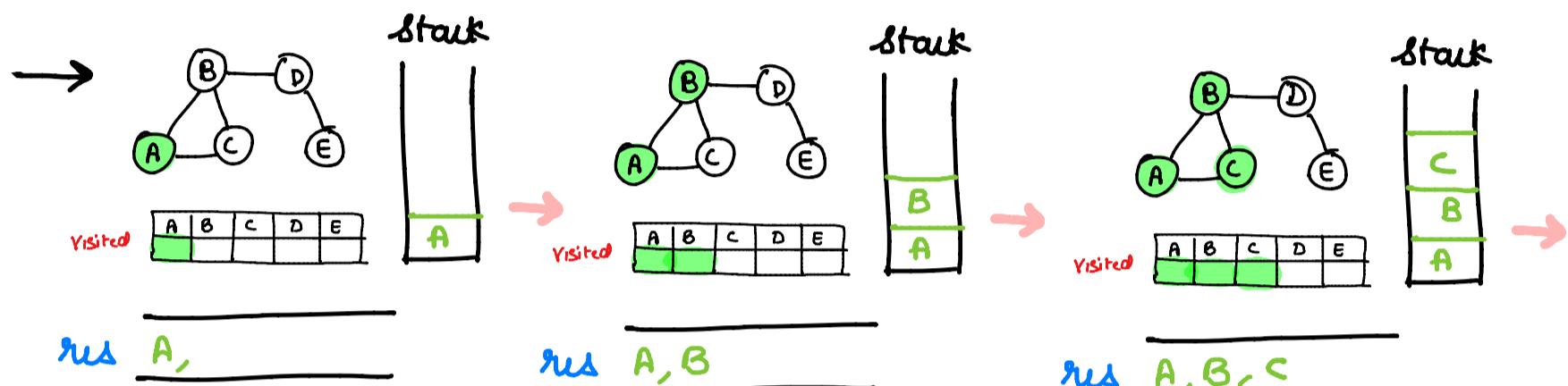
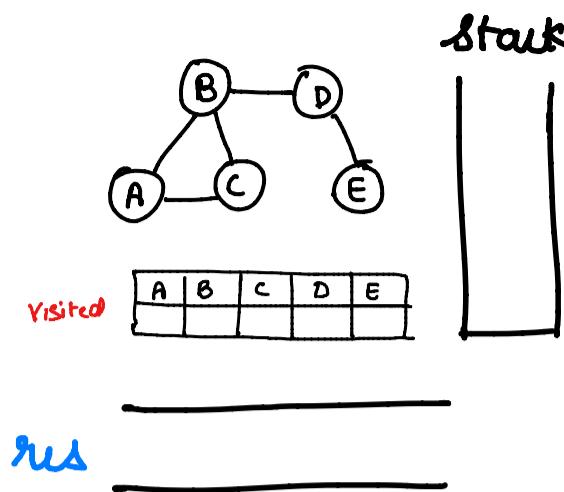
        while(!q.empty()){
            int curr = q.front();
            q.pop();
            vis[curr] = 1;
            ans.push_back(curr);
            for(auto it:adj[curr]){
                if(vis[it]==0){
                    vis[it] = 1;
                    q.push(it);
                }
            }
 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

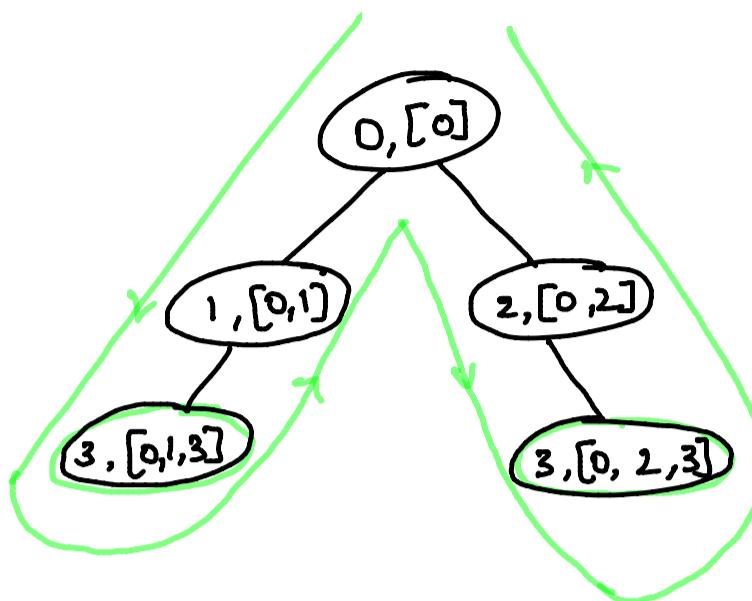
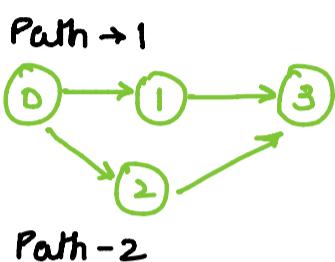
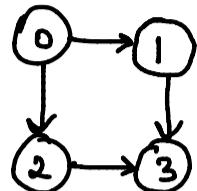
Applications → [DFS]

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

① All paths from src to target

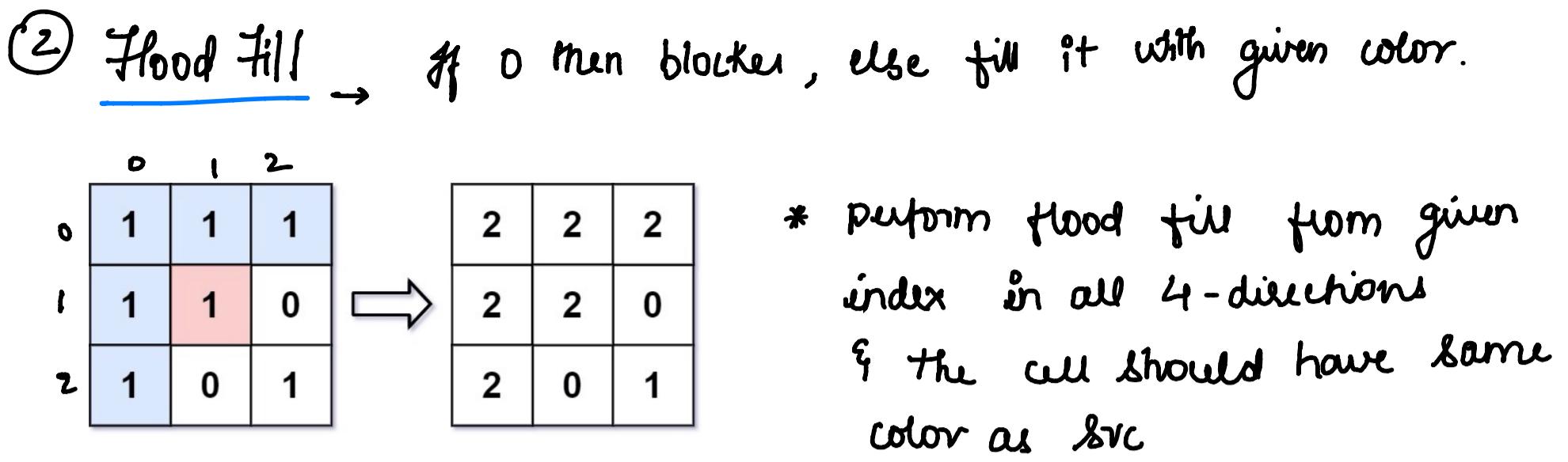
Given a directed acyclic graph, return all paths from node 0 to node n-1.

Eg



Code →

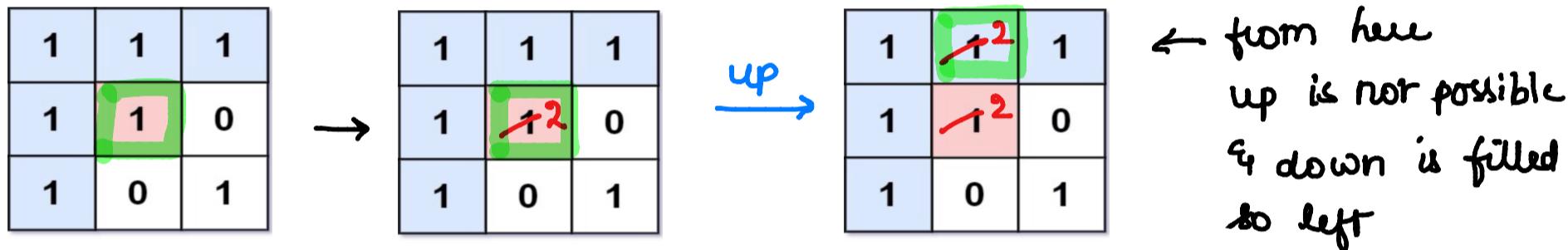
```
1 class Solution {
2 public:
3     void findAllPaths(vector<vector<int>>&graph, int currNode, vector<bool>&visited,
4                        int n, vector<int> &currPath, vector<vector<int>>&res){
5
6         if(currNode==n-1){
7             res.push_back(currPath);
8             return;
9         }
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

*



left

1	1	1
1	1	0
1	0	1

from here up not possible
so down

down

1	1	1
1	1	0
1	0	1

from here up not possible
so down

down

1	1	1
1	1	0
1	0	1

→ no direction is possible
so return

return

1	1	1
1	1	0
1	0	1

right

1	1	1
1	1	0
1	0	1

no other way possible

Result

2	2	2
2	2	0
2	0	1

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	1	2	3	4
0	[[1,1,0,0,0],			
1	[1,1,0,0,0]			
2	[0,0,1,0,0]			
3	[0,0,0,1,1]]			

- 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.

[[1,↑,0,0,0],		
[1,1,0,0,0]		
[0,0,1,0,0]		
[0,0,0,1,1]]		

ans = ⚡ 1.

→ now grid becomes

0	1	2	3	4
0	[[0,0,0,0,0],			
1	[0,0,0,0,0]			
2	[0,0,1,0,0]			
3	[0,0,0,1,1]]			

- 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 = ⚡ 2.

→ now grid becomes

0	1	2	3	4
0	[[0,0,0,0,0],			
1	[0,0,0,0,0]			
2	[0,0,0,0,0]			
3	[0,0,0,1,1]]			

- 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) → (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.

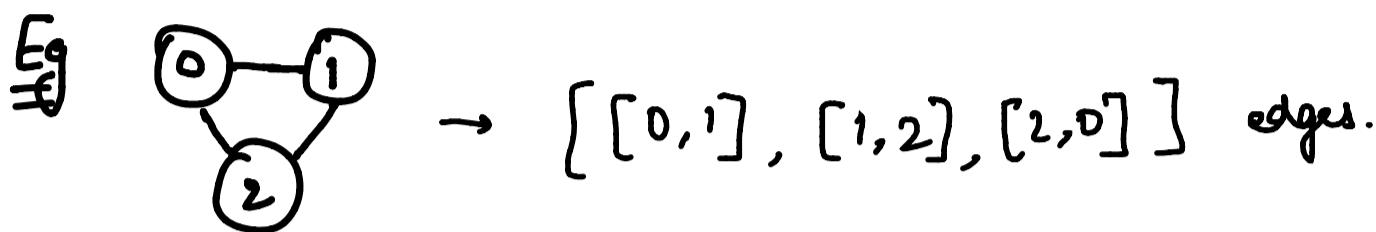
T_C → O(mn) Avg case.

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.

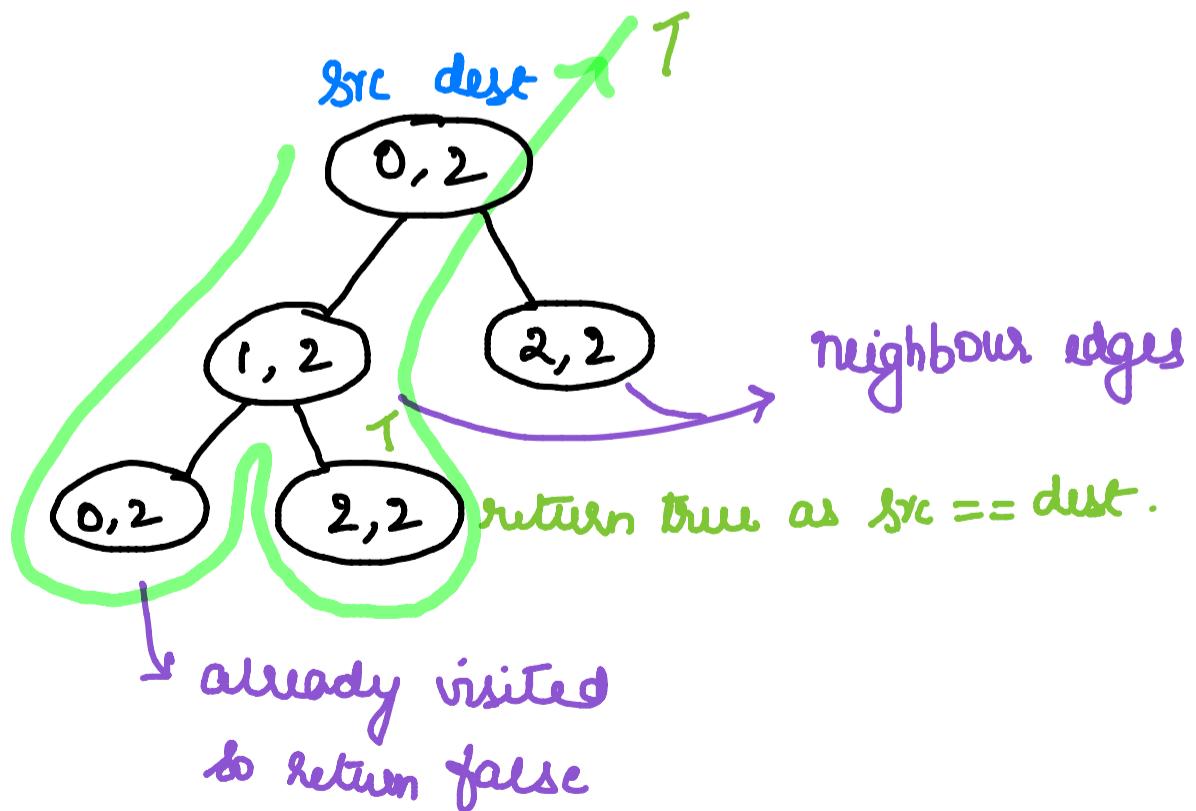


$n = 3$ edges = $[[0,1], [1,2], [2,0]]$ src = 0, dest = 2.

- 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

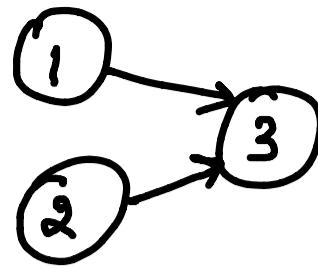


Code →

```
● ● ●  
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

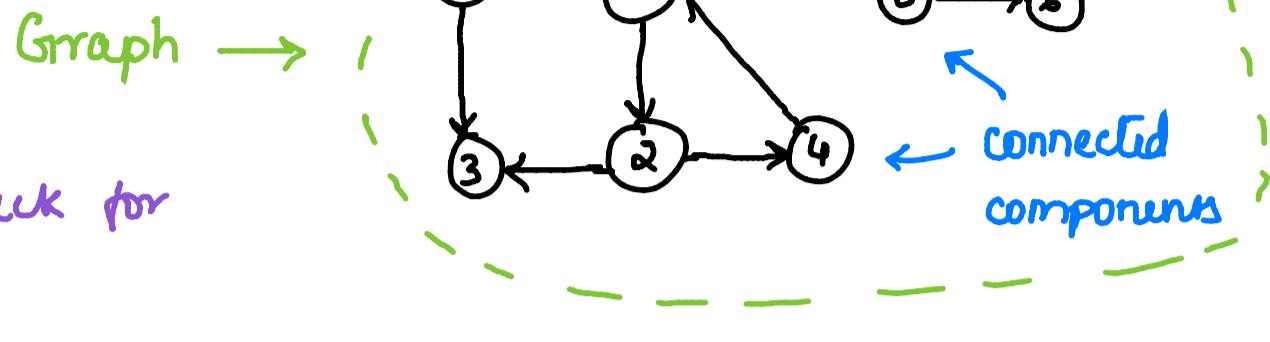
```
● ● ●  
1 class Solution {  
2 public:  
3     int findJudge(int n, vector<vector<int>>& trust) {  
4         vector<int>indegree(n+1,0);  
5         vector<int>outdegree(n+1,0);  
6         for(int i=0;i<trust.size();i++)  
7         {  
8             int v1 = trust[i][0];  
9             int v2 = 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

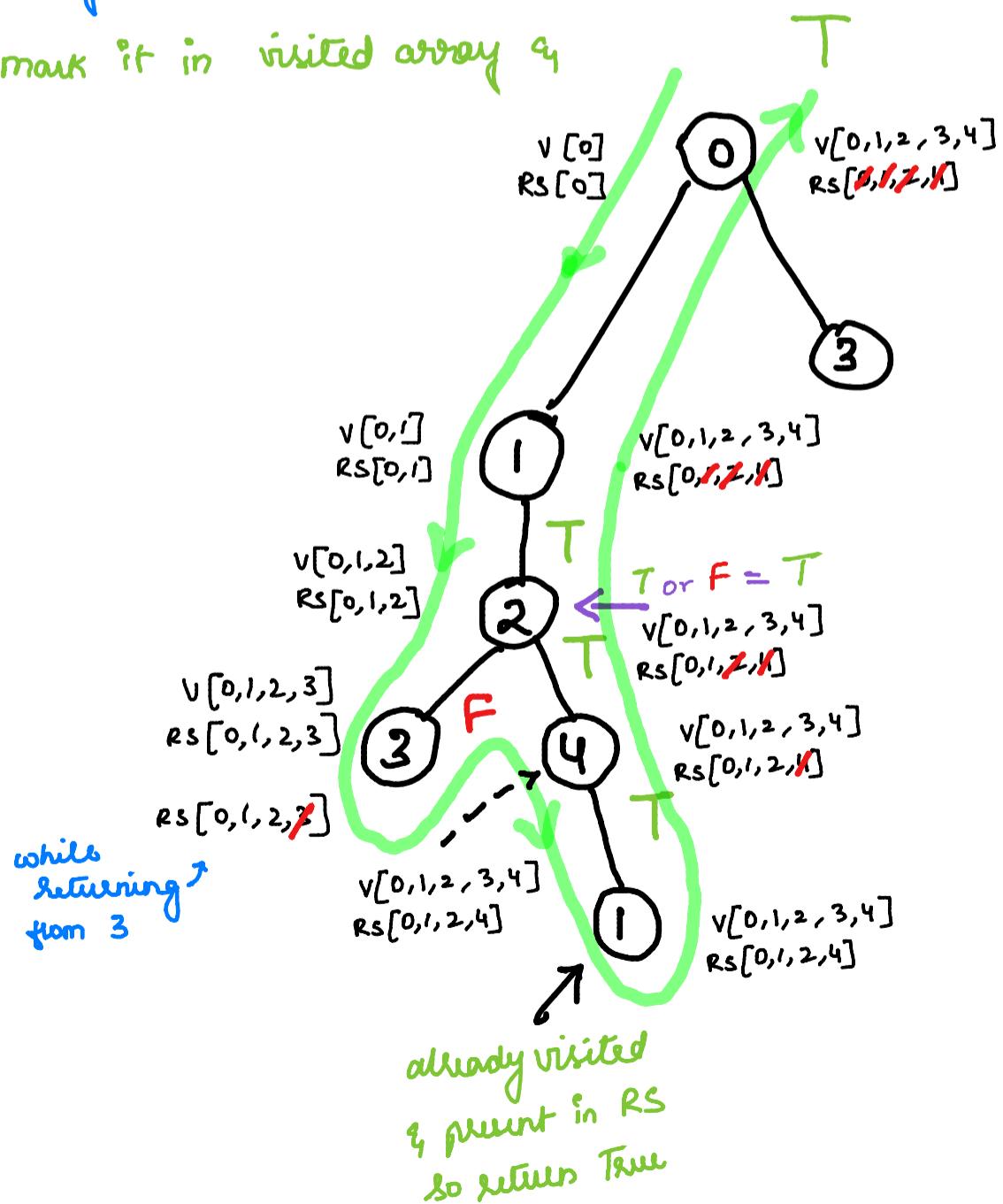
& no cycle is detected so return F.

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

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

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

1 is already present in recursive stack so return true.



Code

$T_C \rightarrow O(V+E)$

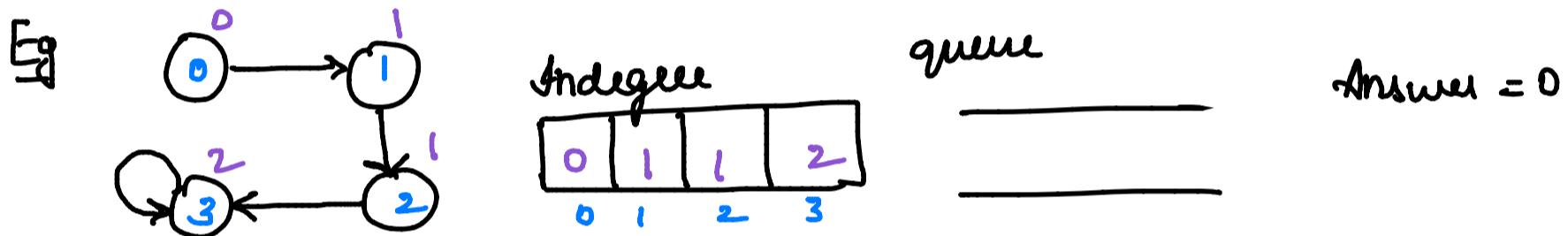
$S_C \rightarrow O(V)$

```
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     };
}
```

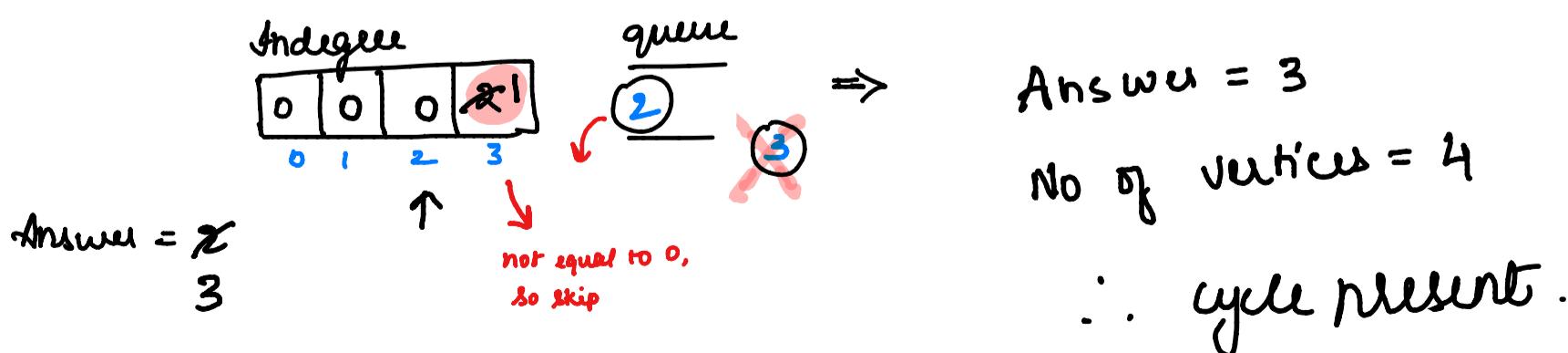
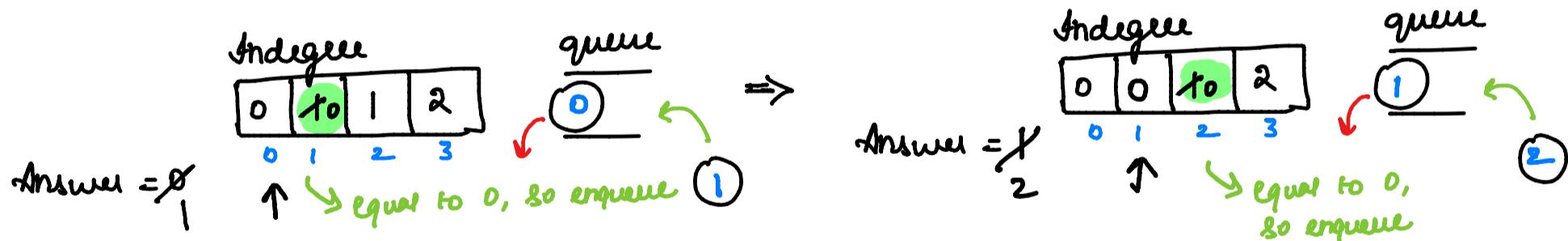
* Kahn's Algorithm → To find topological Ordering

↓
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 →

$T\mathcal{C} \rightarrow O(V + E)$

$S\mathcal{C} \rightarrow O(V)$

```
● ● ●  
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]

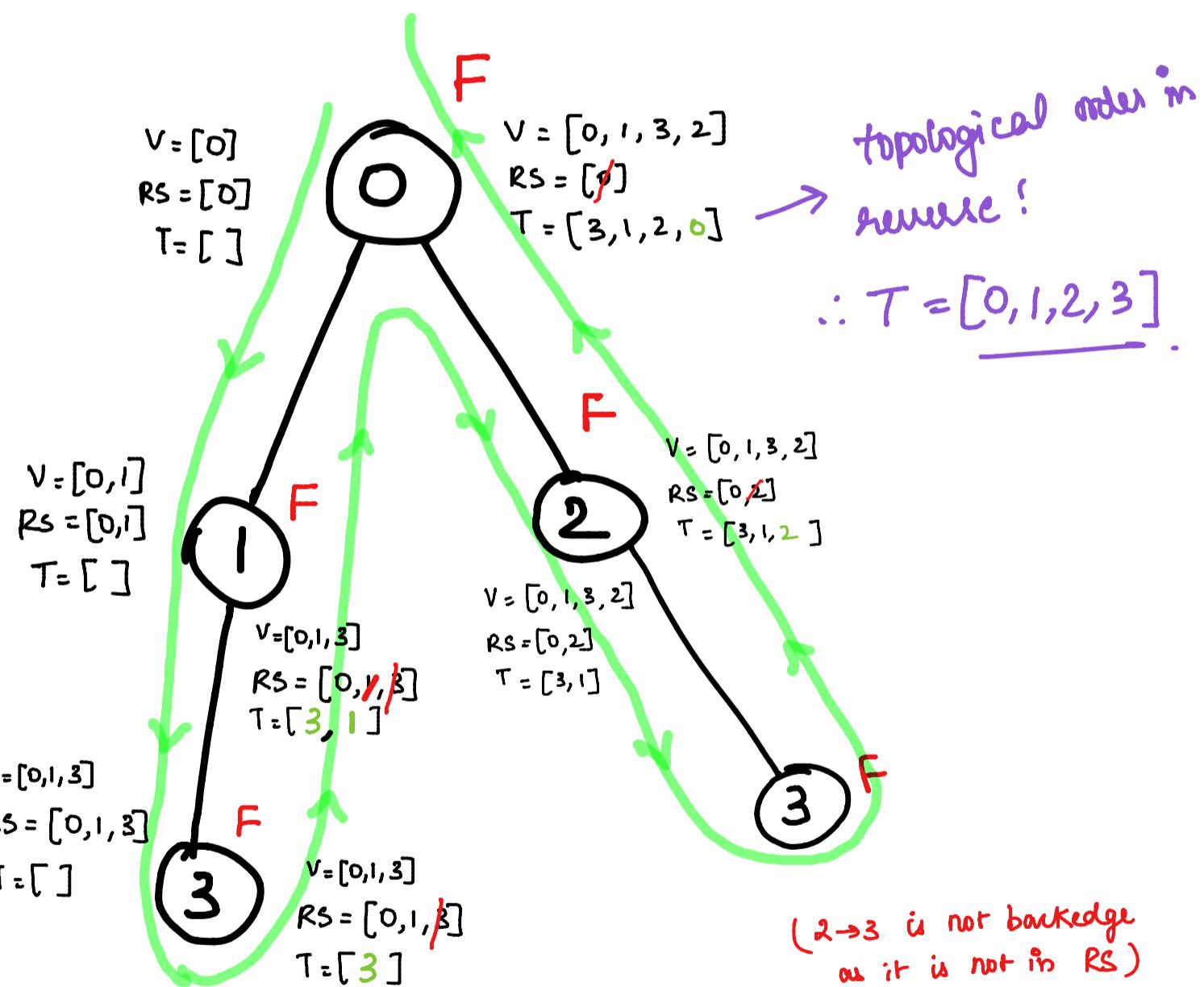
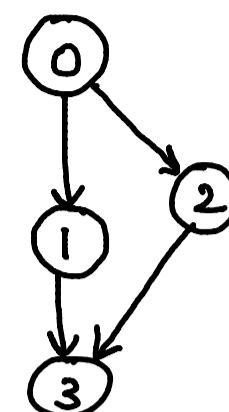
Topological sort only for DAG

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 };
```

Find the rest on

<https://linktr.ee/KarunKarthik>

Follow **Karun Karthik** For More Amazing Content !