Graph

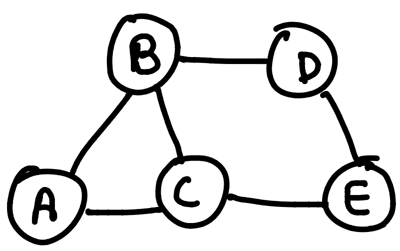
— Karun Karthik

Contents

Graphs

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

Eg



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

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

Applications →

Google maps → To find shortest routes

Social network → user, connection

user ↑ vertex, connection ↑ edge

Representation →

Adj. matrix

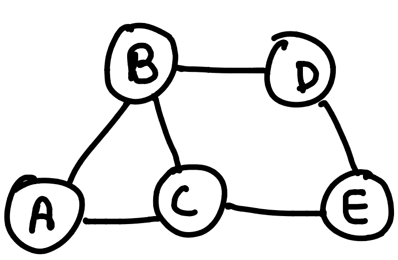
SC → $O(V^2)$

Adj list

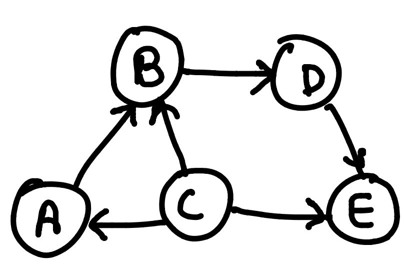
SC → $O(V+E)$

Types →

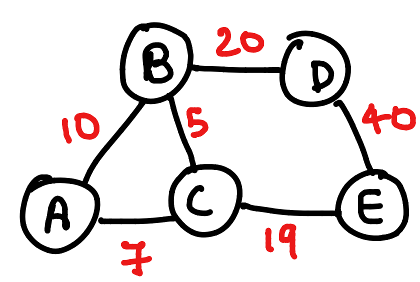
1) Undirected



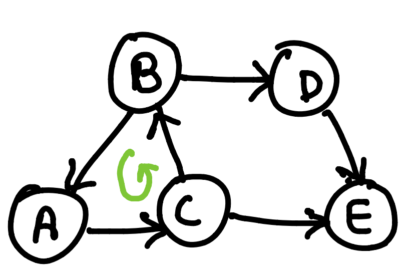
2) Directed



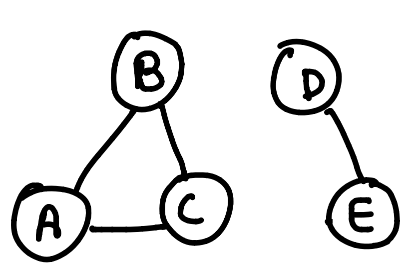
3) weighted



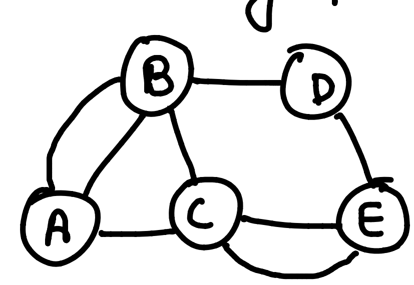
4) Cyclic



5) Disconnected



6) Multigraph

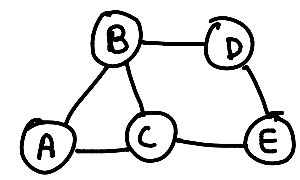


[no self loops]

Graph Traversal

ⓐ **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

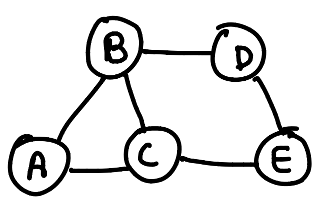


queue.

Visited

A	B	C	D	E

→



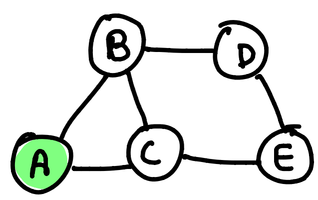
queue.

Visited

A	B	C	D	E

res

→



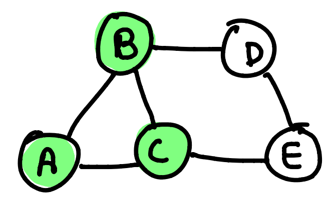
queue. A

Visited

A	B	C	D	E

res A

→



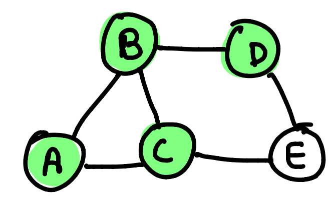
queue. A, B, C

Visited

A	B	C	D	E

res A, B, C

→



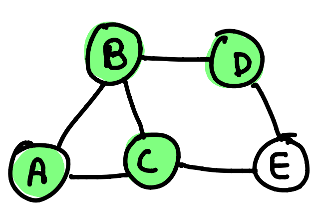
queue. B, C, D

Visited

A	B	C	D	E

res A, B, C, D

→



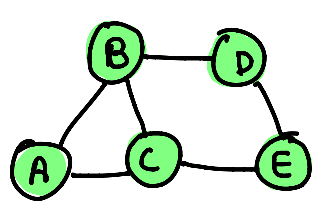
queue. B, C, D

Visited

A	B	C	D	E

res A, B, C, D

→



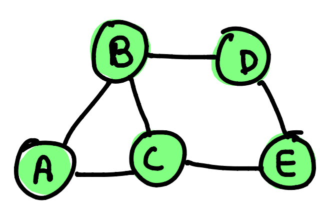
queue. C, D, E

Visited

A	B	C	D	E

res A, B, C, D, E

→



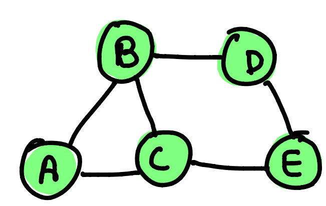
queue. C, D, E

Visited

A	B	C	D	E

res A, B, C, D, E

→



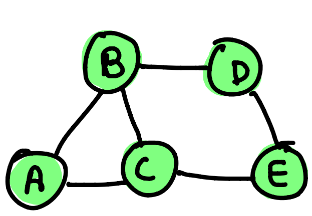
queue. D, E

Visited

A	B	C	D	E

res A, B, C, D, E

→



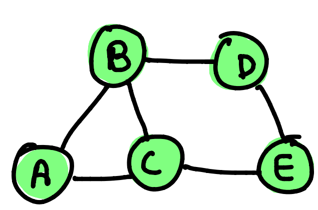
queue. E

Visited

A	B	C	D	E

res A, B, C, D, E

→



queue.

Visited

A	B	C	D	E

res A, B, C, D, E ↳ Return res.

TC → O(V+E)

SC → O(V)

Code

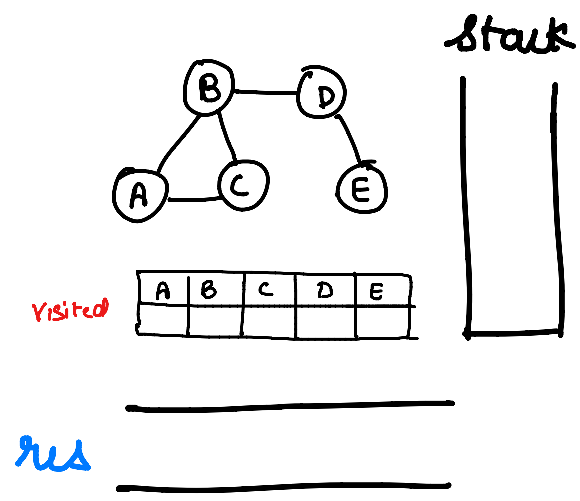
```cpp
class Solution {
  public:

    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;
    }
};
```

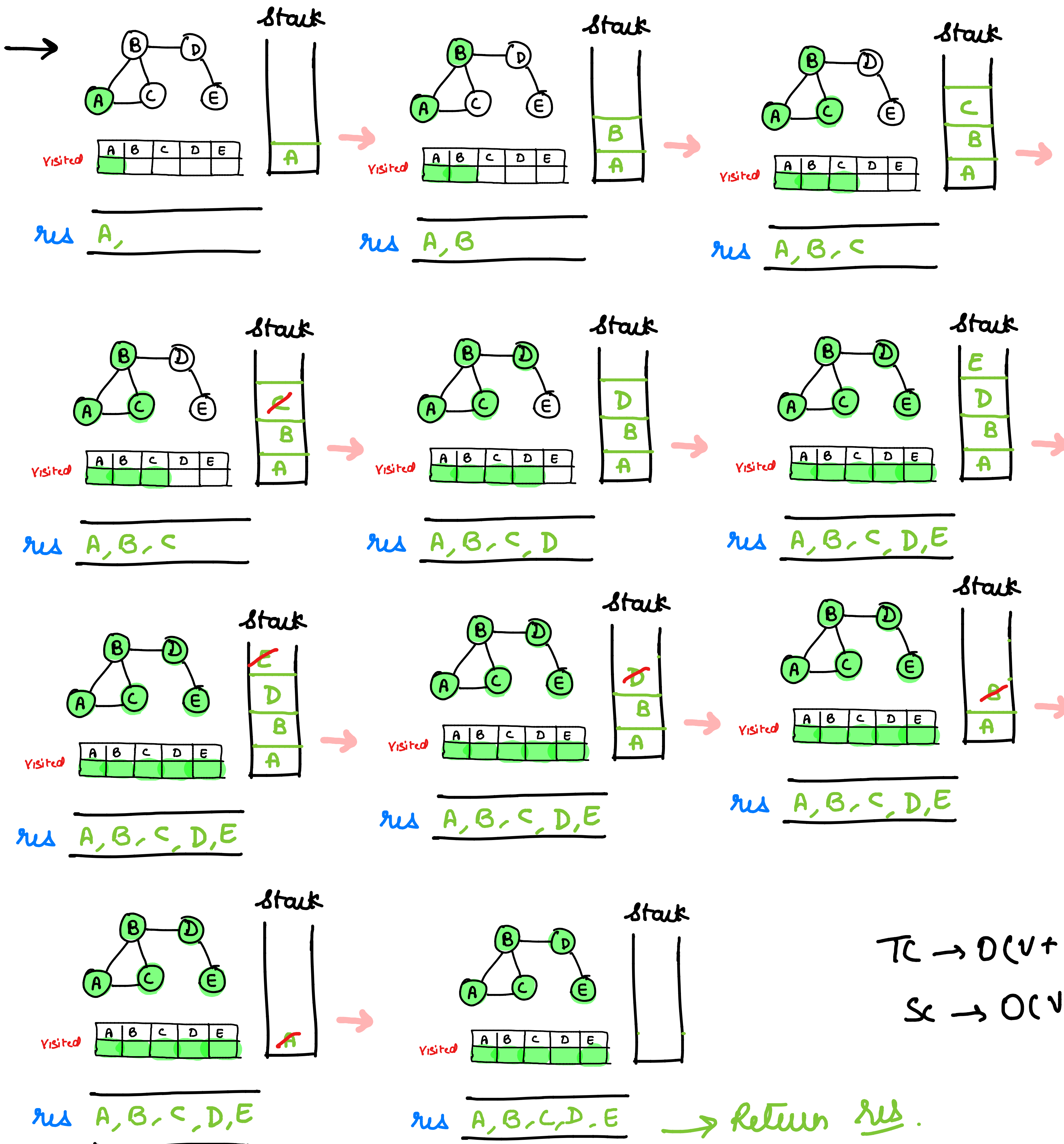
Applications → [BFS]

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 → O(V+E)

SC → O(V)

code

```cpp
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;
    }
};
```

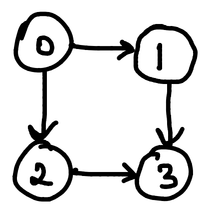
Applications → [DFS]

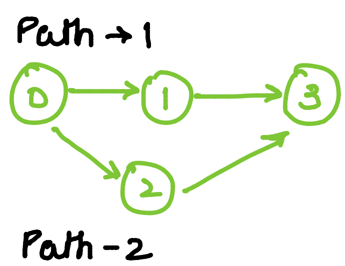
1. Path finding
2. Cycle detection
3. Topological sort
4. Finding strongly connected components.

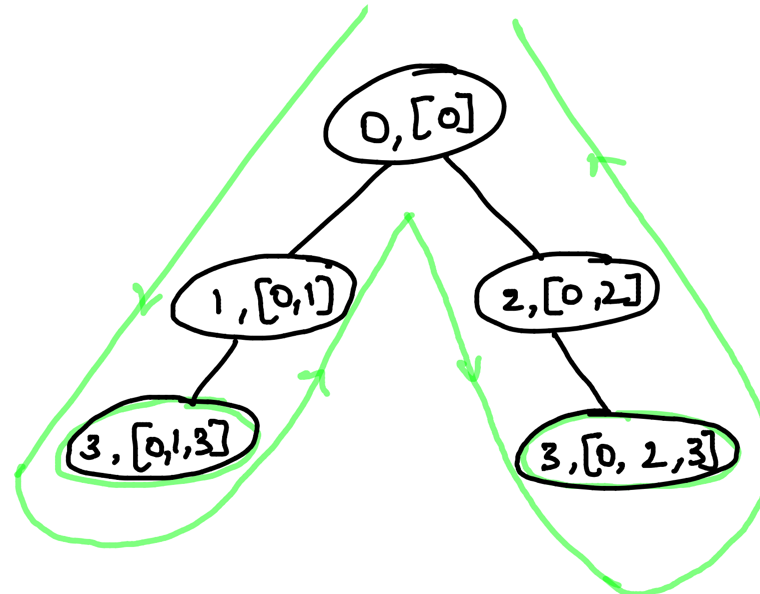
① All paths from src to target

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

Eg







Code →

```cpp
class Solution {
public:
    void findAllPaths(vector<vector<int>>&graph, int currNode, vector<bool>&visited,
                      int n, vector<int> &currPath, vector<vector<int>>&res){

        if(currNode==n-1){
            res.push_back(currPath);
            return;
        }

        if(visited[currNode]==true) return;

        // backtrack for every node
        visited[currNode] = true;

        for(auto neighbour: graph[currNode]){
            currPath.push_back(neighbour);
            findAllPaths(graph, neighbour, visited, n, currPath, res);
            currPath.pop_back();
        }

        visited[currNode] = false;
    }

    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        vector<vector<int>> res;
        vector<int> currPath;
        int n = graph.size();
        vector<bool> visited(n);

        // traversing from 0 node
        currPath.push_back(0);

        findAllPaths(graph, 0, visited, n, currPath, res);
        return res;
    }
};
```

TC → O(V+E)

V → vertices
E → edges

SC →
Recursive stack + Result

② Flood Fill → If 0 then blocker, else fill it with given color.

	0	1	2
0	1	1	1
1	1	1	0
2	1	0	1

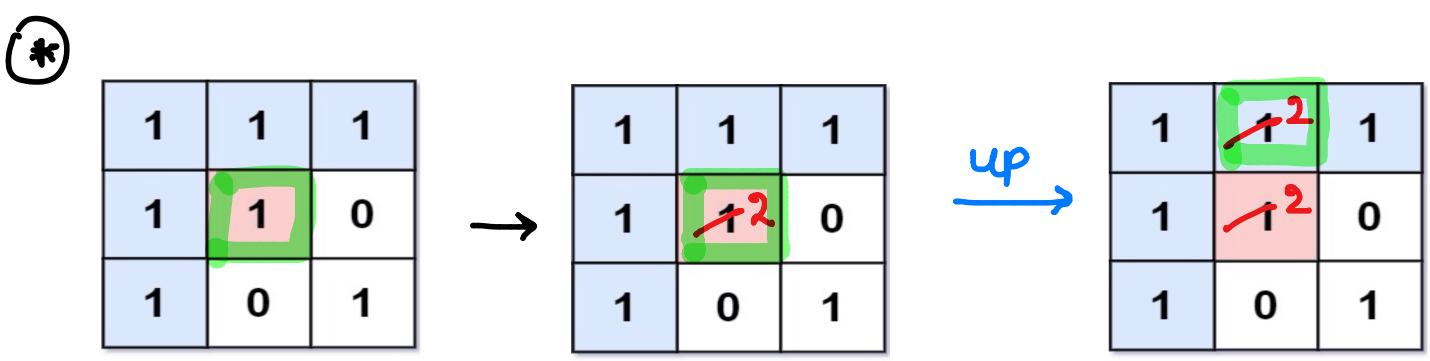
⇒

2	2	2
2	2	0
2	0	1

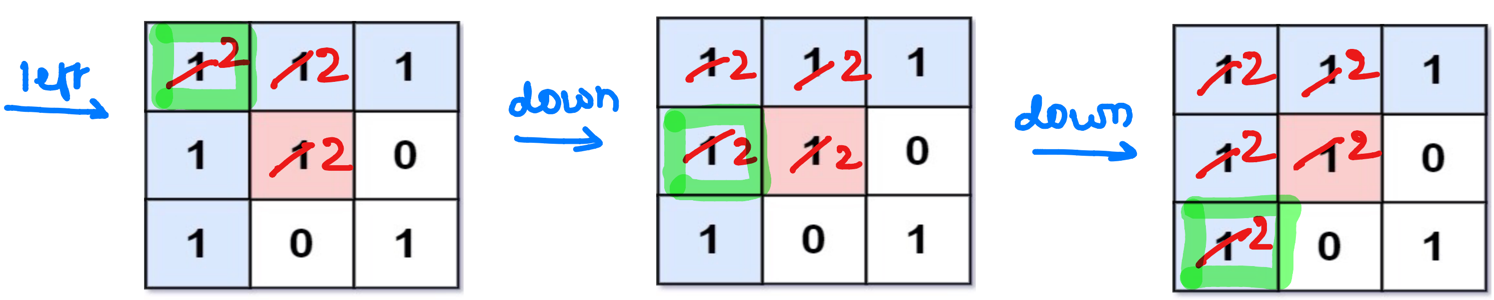
* perform flood fill from given index in all 4-directions & the cell should have same color as src

✓ lets follow the order to fill → UP, DOWN, LEFT, RIGHT

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



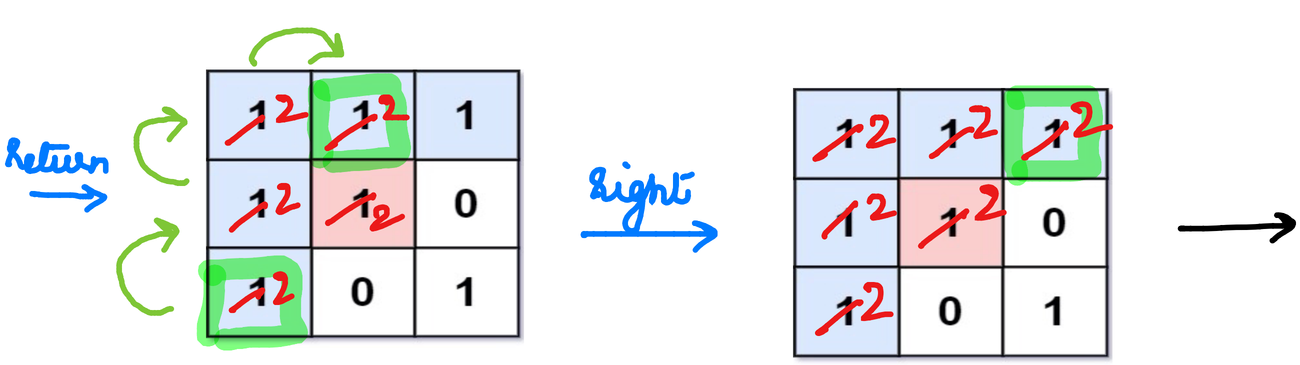
← from here up is not possible & down is filled so left



from here up not possible so down

from here up not possible so down

→ no direction is possible so return



no other way possible

Result

2	2	2
2	2	0
2	0	1

https://www.linkedin.com/in/karun-karthik

Code

```cpp
class Solution {
public:
    void floodFiller(vector<vector<int>>& image, int i, int j,
    int m, int n, int currColor, int newColor)
    {
        if(i<0 || i>=m || j<0 || j>= n || image[i][j] == newColor
            || image[i][j] != currColor)
            return;

        image[i][j] = newColor;
        floodFiller( image, i-1, j, m, n, currColor, newColor);
        floodFiller( image, i+1, j, m, n, currColor, newColor);
        floodFiller( image, i, j-1, m, n, currColor, newColor);
        floodFiller( image, i, j+1, m, n, currColor, newColor);
    }

    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,
    int sc, int newColor)
    {
        int m = image.size();
        int n = image[0].size();
        int currColor = image[sr][sc];
        floodFiller(image, sr, sc, m, n, currColor, newColor);
        return image;
    }
};
```

TC → O(mn)

SC → O(n)

↳ recursive stack

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

Eg

$$\begin{array}{c|c} & 0 \quad 1 \quad 2 \quad 3 \quad 4 \\ 0 & [[1,1,0,0,0], \\ 1 & [1,1,0,0,0], \\ 2 & [0,0,1,0,0], \\ 3 & [0,0,0,1,1]] \end{array}$$

- 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

$$\begin{array}{l} [[1,1,0,0,0], \\ [1,1,0,0,0], \\ [0,0,1,0,0], \\ [0,0,0,1,1]] \end{array}$$

ans = ~~0~~ 1.

- lets start from (0,0) & try moving U, D, L, R

→ the traversal goes in this order

(0,0) → (1,0) → (1,1) → (0,1) ie

& update ans.

→ now grid becomes

$$\begin{array}{c|c} & 0 \quad 1 \quad 2 \quad 3 \quad 4 \\ 0 & [[0,0,0,0,0], \\ 1 & [0,0,0,0,0], \\ 2 & [0,0,1,0,0], \\ 3 & [0,0,0,1,1]] \end{array}$$

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

ans = ~~1~~ 2.

→ now grid becomes

$$\begin{array}{c|c} & 0 \quad 1 \quad 2 \quad 3 \quad 4 \\ 0 & [[0,0,0,0,0], \\ 1 & [0,0,0,0,0], \\ 2 & [0,0,0,0,0], \\ 3 & [0,0,0,1,1]] \end{array}$$

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

ans = ~~2~~ 3. $\boxed{\text{Ans} = 3}$

Code

```cpp
class Solution {
public:
    void countIsland(vector<vector<char>>& grid, int currRow, int currCol, int row, int col){
        if(currRow<0 || currRow>=row || currCol<0 || currCol>=col || grid[currRow][currCol]=='0')
            return;

        grid[currRow][currCol] = '0';
        countIsland(grid, currRow-1, currCol, row, col);
        countIsland(grid, currRow+1, currCol, row, col);
        countIsland(grid, currRow, currCol-1, row, col);
        countIsland(grid, currRow, currCol+1, row, col);
    }

    int numIslands(vector<vector<char>>& grid) {
        int ans = 0;
        int row = grid.size();
        int col = grid[0].size();

        for(int currRow = 0; currRow < row; currRow++)
            for(int currCol = 0; currCol < col; currCol++)
                if(grid[currRow][currCol]=='1'){
                    ans++;
                    countIsland(grid, currRow, currCol, row, col);
                }

        return ans;
    }
};
```

Tc $\rightarrow$ $O(mn)$ 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, compare for max Area of island.

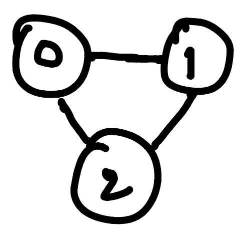
TC → O(mn) Avg case.

code →

```cpp
class Solution {
public:
    int findArea(vector<vector<int>>& grid, int currRow, int currCol, int m, int n){
        if(currRow<0 || currCol<0 || currRow>=m || currCol>=n || grid[currRow][currCol]==0)
            return 0;

        grid[currRow][currCol]=0;

        // this is for single cell where we started traversing
        int count = 1;
        count += findArea(grid, currRow-1, currCol, m, n);
        count += findArea(grid, currRow+1, currCol, m, n);
        count += findArea(grid, currRow, currCol-1, m, n);
        count += findArea(grid, currRow, currCol+1, m, n);
        return count;
    }
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int ans = 0;
        for(int currRow = 0; currRow<m; currRow++)
            for(int currCol = 0; currCol<n; currCol++){
                if(grid[currRow][currCol]==1){
                    ans = max(ans, findArea(grid, currRow, currCol, m, n));
                }
            }
        return ans;
    }
};
```

5 Find if path exist in graph.

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

Eg 

→ [[0,1], [1,2], [2,0]] edges.

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]]

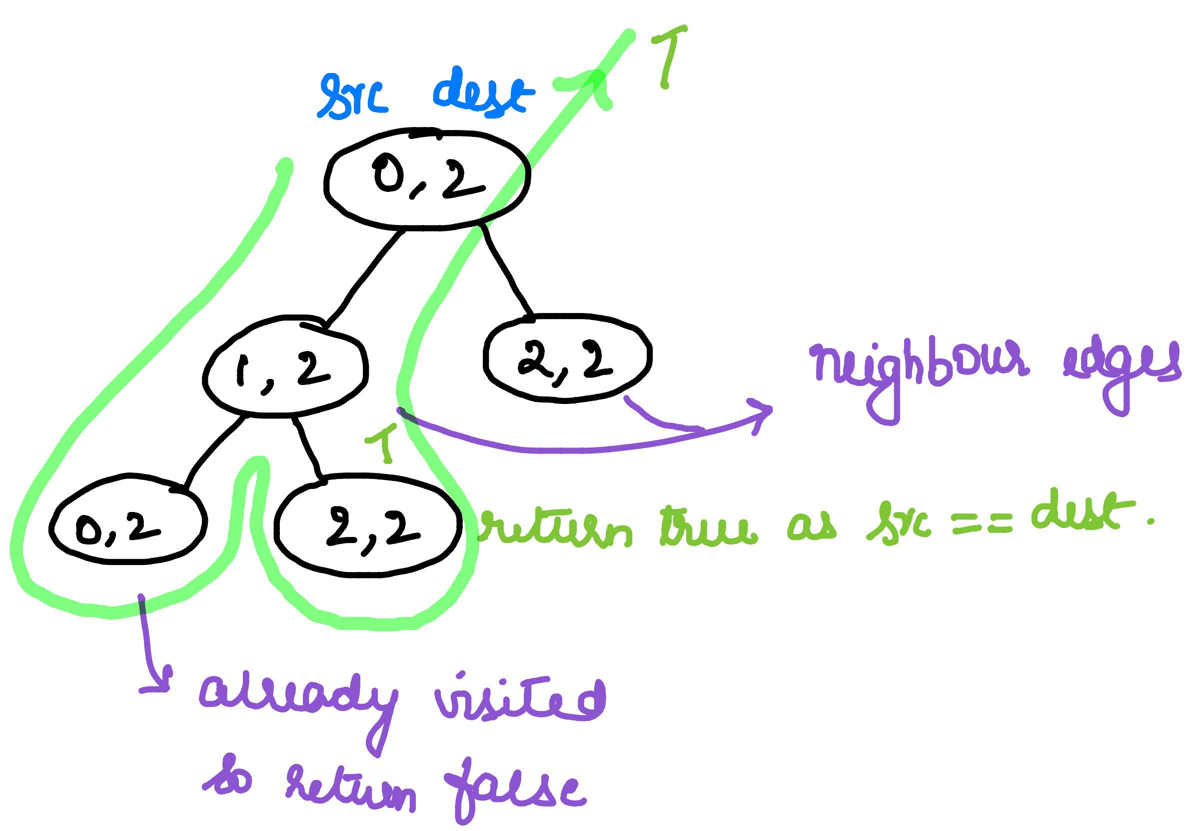
0	1	2

2) Perform dfs

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

0	1	2

T ~~F~~	T ~~F~~	F
0	1	2

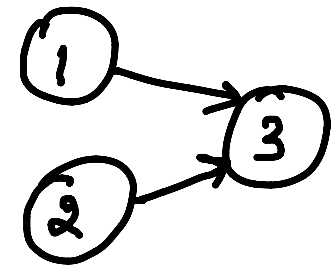


Code →

```cpp
class Solution {
public:
    bool validPath(int n, vector<vector<int>>& edges, int src, int dest) {

        vector<vector<int>>graph(n);
        for(int i=0;i<edges.size();i++)
        {
            int v1 = edges[i][0];
            int v2 = edges[i][1];
            graph[v1].push_back(v2);
            graph[v2].push_back(v1);

        }
        vector<bool>vis(n,false);
        return pathExist(src,  dest, graph, vis);
    }

    bool pathExist(int src , int dest,vector<vector<int>>&graph,vector<bool>&vis){

        if(src==dest)return true;

        vis[src]=true;

        for(int i=0;i<graph[src].size();i++)
            if(vis[graph[src][i]]==false)
                if(pathExist(graph[src][i],dest,graph,vis)==true)
                    return true;

        return false;
    }
};
```

⑥ Find the town judge

n = 3 , trust = [[1, 3], [2, 3]]



* Indegree of town judge = n-1

& outdegree = 0

✓ create 2 arrays

outdegree

0	1	2	3
~~0~~ 1	0	~~0~~ 1	0

indegree

0	1	2	3
0	0	0	~~0~~ ~~1~~ 2

for [1, 3]
indegree of 1 ↑
outdegree of 3 ↑

for [2,3]
indegree of 2 ↑
outdegree of 3 ↑

→ traverse both indegree & outdegree

if indegree == 0 &&
outdegree == n-1
then return that vertex

code

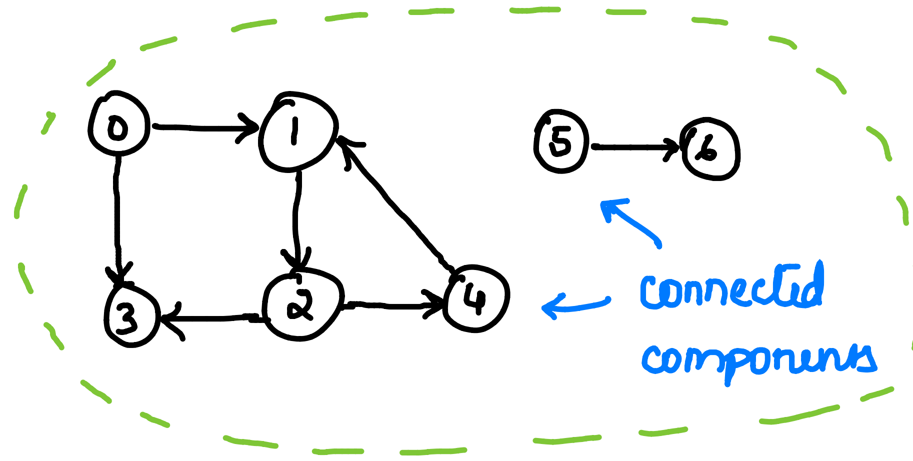
```cpp
class Solution {
public:
    int findJudge(int n, vector<vector<int>>& trust) {
        vector<int>indegree(n+1,0);
        vector<int>outdegree(n+1,0);
        for(int i=0;i<trust.size();i++)
        {
            int v1 = trust[i][0];
            int v2 = trust[i][1];
            outdegree[v1]+=1;
            indegree[v2]+=1;
        }
        for(int i=1;i<=n;i++)
        {
            if(outdegree[i]==0 && indegree[i]==n-1)
                return i;
        }
        return -1;
    }
};
```

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 →



* To detect cycle, check for **backedge**.

Let's start dfs from 0 vertex.

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

If present, then it indicates back edge → Return True

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

Visited → {0, 1, 2, 3, 4

Recursive stack → {0, 1, 2, ~~3~~, 4

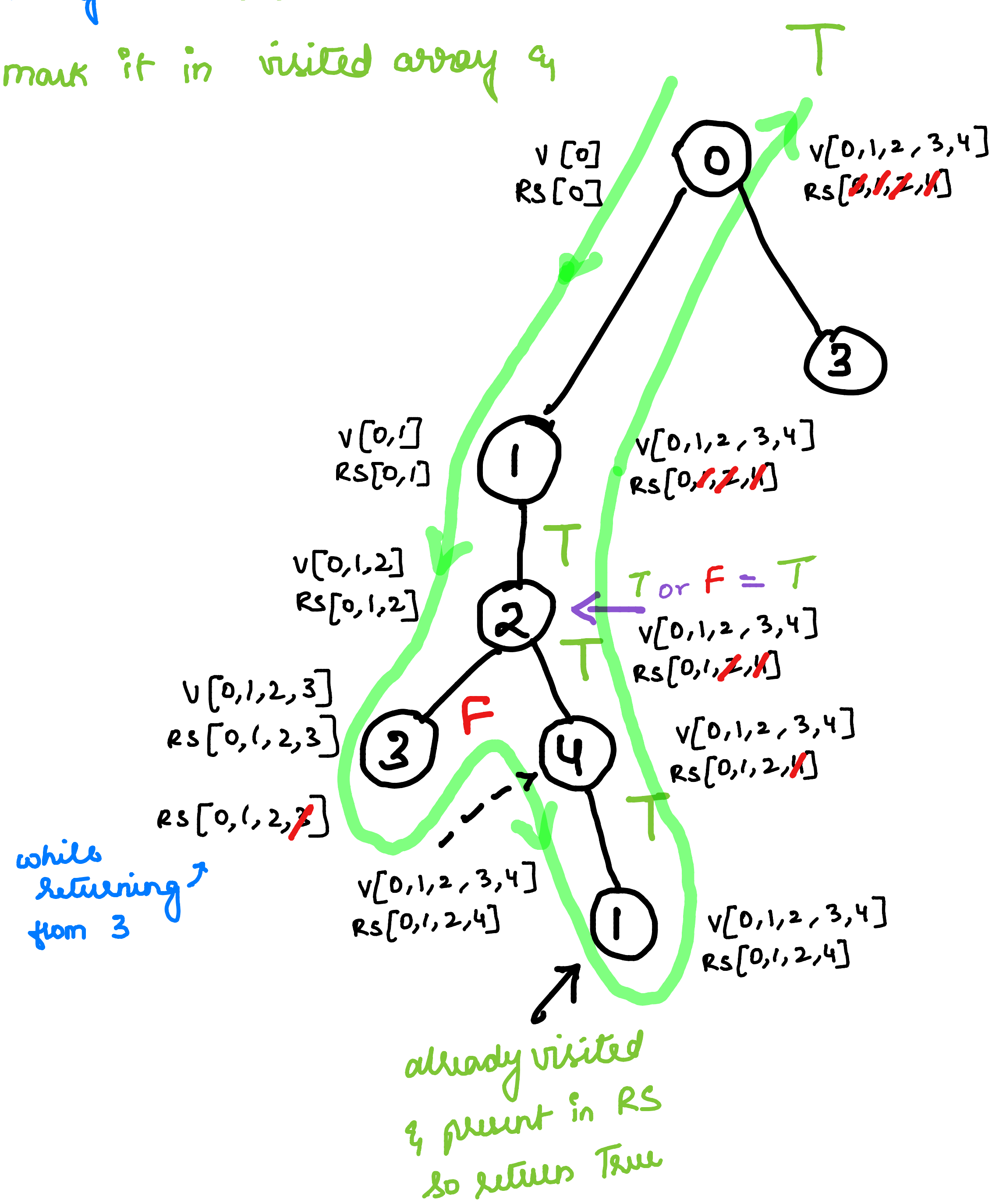
* At 3 vertex, there's no neighbour & no cycle is detected so return F.

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

Visited → {0, 1, 2, 3, 4, 1} &

Recursive stack → {0, 1, 2, ~~3~~, 4 }

1 is already present in recursive stack so return true.



Code

Tc → O(V+E)

Sc → O(V)

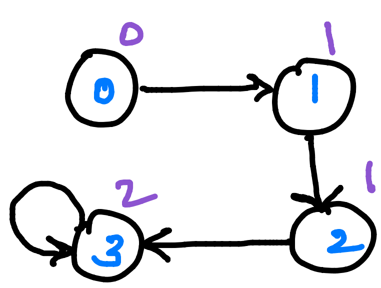
```cpp
class Solution {
  public:
    bool dfs(int node, vector<int>&vis, vector<int>&rs, vector<int> adj[])
    {
        vis[node]=1;
        rs[node]=1;
        for(auto it:adj[node])
        {
            if(vis[it]==0){
                if(dfs(it,vis,rs,adj))
                    return true;
            }
            else if(rs[it]==1)
                return true;
        }
        rs[node]=0;
        return false;
    }
    bool isCyclic(int V, vector<int> adj[]) {

        vector<int>vis(V,0);
        vector<int>rs(V,0);

        for(int i=0;i<V;i++)
        {
            if(vis[i]==0)
            if(dfs(i,vis,rs,adj))
                return true;
        }
        return false;
    }
};
```

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.

Eg



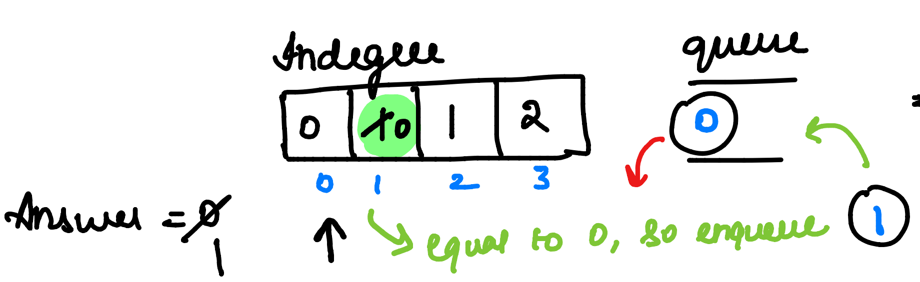
Indegree

0	1	2	3
0	1	1	2

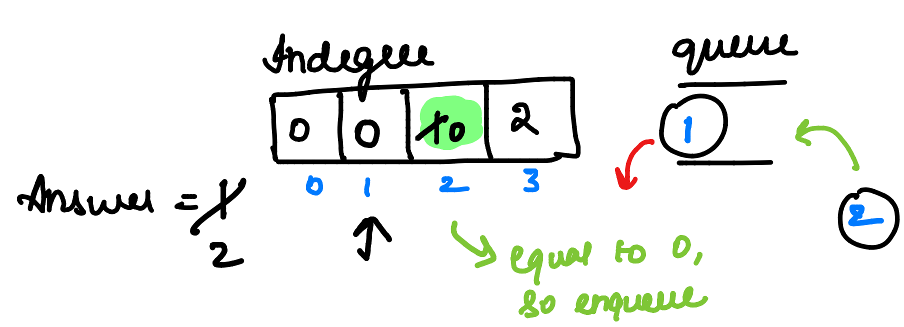
queue

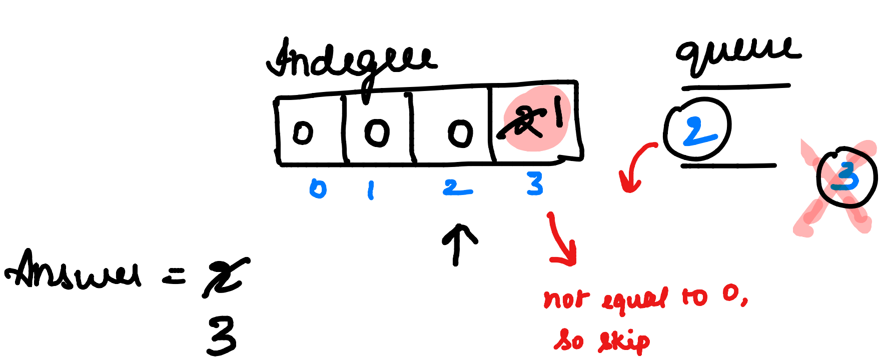
Answer = 0

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



⇒





⇒

Answer = 3

No of vertices = 4

∴ cycle present.

code

```
class Solution{
  public:
    bool isCyclic(int V, vector<int> adj[]) {

        vector<int>indegree(V,0);
        for (int i = 0; i <V; i++)
          for(int it : adj[i])
            indegree[it]++;

        queue<int>q;
        int ans = 0;
        unordered_set<int>vis;

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

        while(!q.empty())
        {
            int currvertex = q.front();
            q.pop();
            if(vis.find(currvertex)!=vis.end())
                continue;
            vis.insert(currvertex);
            for(int neighbour:adj[currvertex])
            {
                indegree[neighbour]-=1;
                if(indegree[neighbour]==0)
                 {
                     q.push(neighbour);
                     ans+=1;
                 }
            }
        }
        if(ans==V)  return false;
        return true;
    }
};
```

8 Topological sort

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

Code →

TC → $O(V + E)$

SC → $O(V)$

```
class Solution
{
    public:
    vector<int> topoSort(int V, vector<int> adj[])
    {
        vector<int> indegree(V,0) ,res;

        for(int i=0; i<V; i++)
            for(auto it:adj[i])
                indegree[it]++;

       queue<int> q;
       int ans = 0;
       unordered_set<int>vis;

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

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

           // add to res
           res.push_back(curr);

           if(vis.find(curr)!=vis.end())
                continue;

           vis.insert(curr);

           for(int neighbour: adj[curr])
           {
               indegree[neighbour]-=1;
               if(indegree[neighbour]==0)
               {
                   q.push(neighbour);
                   ans+=1;
               }
           }
       }

       return res;
    }
};
```

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

TC → O(V + E)

SC → O(V + E)

Code →

```
class Solution {
public:
    vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
        vector<vector<int>> graph(n);
         for(auto it:pre){
             int v = it[1];
             int u = it[0];
             graph[v].push_back(u);
         }
         return graph;
    }

    bool canFinish(int n, vector<vector<int>>& pre) {
        vector<vector<int>> graph = createGraph(n, pre);
        vector<int>indegree(n,0);
        for(int i=0; i<n; i++)
            for(int it: graph[i])
                indegree[it]++;

        queue<int> q;
        int ans = 0;
        unordered_set<int> vis;

        for(int i=0; i<n; i++)
            if(indegree[i]==0){
                q.push(i);
                ans++;
            }

        while(!q.empty()){
            int currvertex = q.front();
            q.pop();
            if(vis.find(currvertex)!=vis.end())
                continue;
            vis.insert(currvertex);
            for(int neighbour: graph[currvertex]){
                indegree[neighbour]--;
                if(indegree[neighbour]==0){
                    q.push(neighbour);
                    ans++;
                }
            }
        }
        if(ans==n) return true;
        return false;
    }
};
```

(10) Course Schedule – II

pre → edges [v, u]

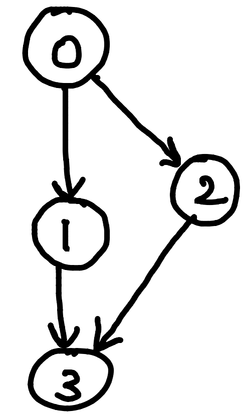
"u should be completed before v"

n → no. of courses [vertices]

Topological sort only for DAG

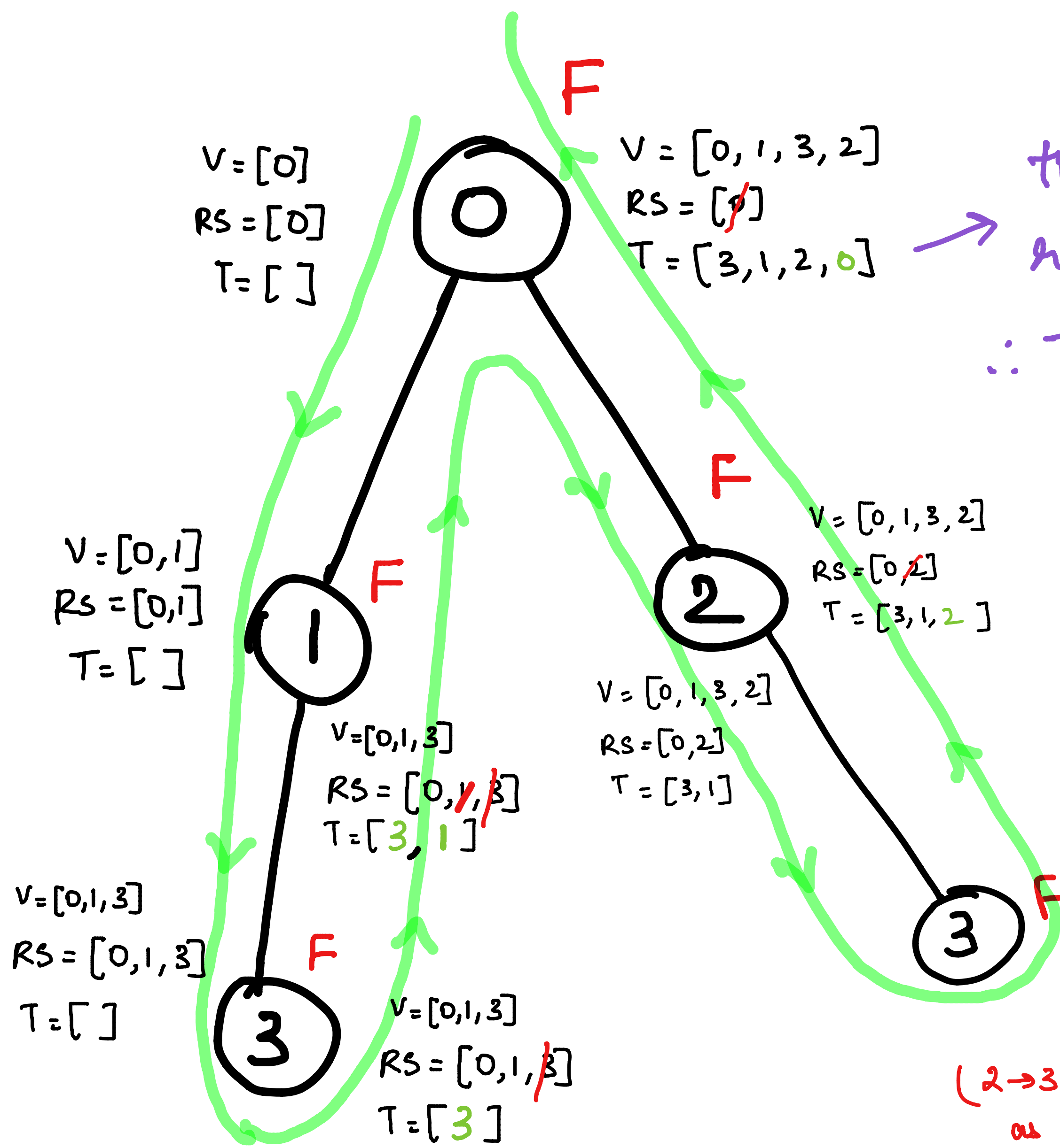
Eg n → 4 (0, 1, 2, 3)

pre → [[1,0], [2,0], [3,1], [3,2]]



Initially

V = [ ], RS = [ ], traversal = [ ]



→ topological order in reverse!

∴ T = [0, 1, 2, 3]

(2→3 is not backedge as it is not in RS)

while returning from 3 ↑

pop 3 & push into traversal array.

return F, as no cycle is found

Code →

TC → $O(V+E)$

SC → $O(V+E)$

```cpp
class Solution {
public:
    bool dfs(vector<vector<int>>&graph, int i, vector<int> &vis,
    vector<int> &rs, vector<int> &traversal){

        vis[i] = 1;
        rs[i] = 1;
        for(int neighbour: graph[i]){
            if(vis[neighbour]==0){
                if(dfs(graph, neighbour, vis, rs, traversal))
                    return true;
            }
            else if(rs[neighbour]==1)   return true;
        }
        traversal.push_back(i);
        rs[i]=0;
        return false;
    }

    vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
        vector<vector<int>> graph(n);
         for(auto it:pre){
             int v = it[1];
             int u = it[0];
             graph[v].push_back(u);
         }
         return graph;
    }

    vector<int> findOrder(int n, vector<vector<int>>& pre) {
        vector<vector<int>> graph = createGraph(n, pre);
        vector<int> vis(n,0), rs(n,0), traversal;
        for(int i=0; i<n; i++){
            if(vis[i]==0)
            if(dfs(graph, i, vis, rs, traversal)) return {};
        }
        reverse(traversal.begin(), traversal.end());
        return traversal;
    }
};
```

Find the rest on

https://linktr.ee/KarunKarthik

Follow Karun Karthik For More Amazing Content !