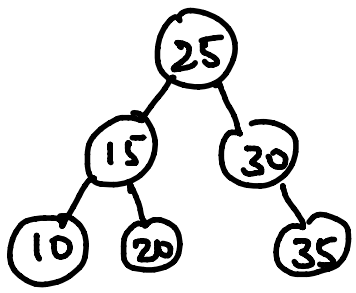
(34) Inorder Successor of BST

Given root, find inorder successor of given node

↳ the element just after the node in inorder traversal.

Eg



n = 15 o/p → 20

n = 35 o/p → null.

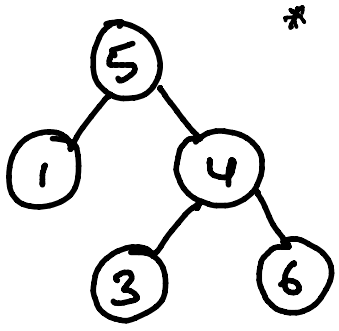
Code →

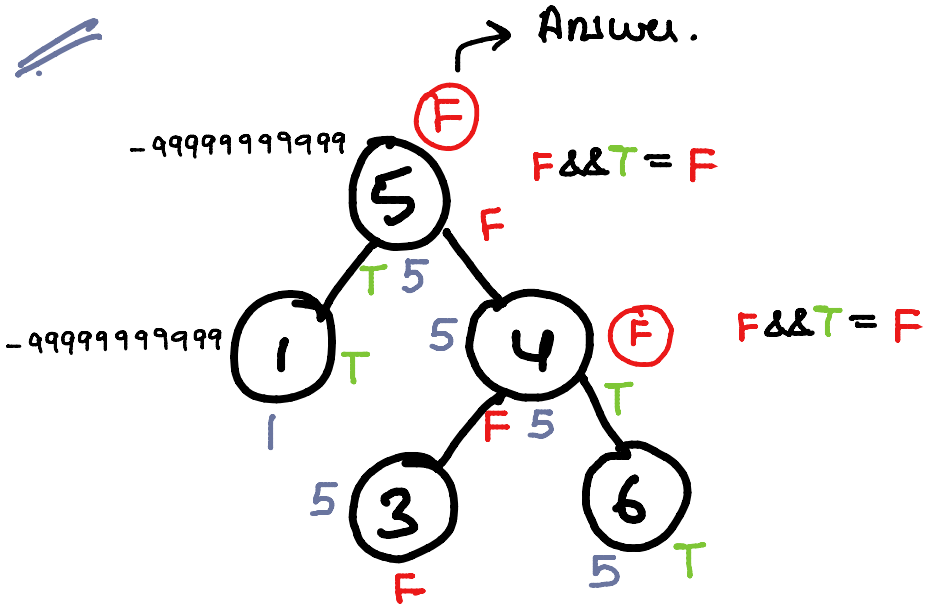
```
class Solution{
  public:

    void inorder(Node *root, vector<Node*> &res){
        if(root==NULL)  return;
        inorder(root->left, res);
        res.push_back(root);
        inorder(root->right, res);
    }

    Node * inOrderSuccessor(Node *root, Node *x)
    {
        vector<Node*> res;
        inorder(root, res);
        for(int i=0; i<res.size(); i++){
            if(res[i]==x && i<res.size()-1){
                return res[i+1];
            }
        }
        return NULL;
    }
};
```

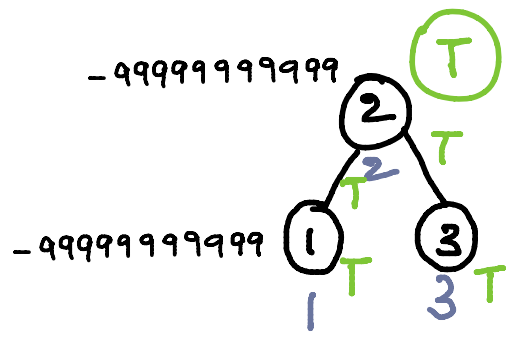
D12 (35) **Validate BST** Given a root node, return true if it is valid BST

* Every value should be less than previous one in Inorder Traversal

Eg







→ Return True on NULL nodes

→ Check for left subtree

→ previous value gets updated before checking Right subtree & after checking left subtree

→ if curVal <= Previous then return false

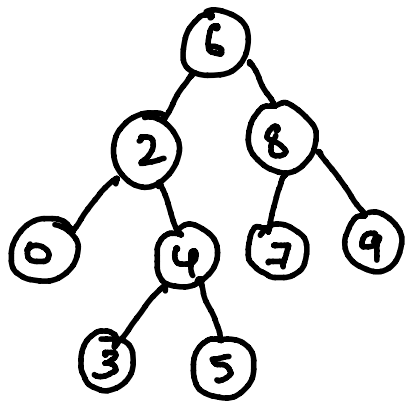
→ return true if both LST & RST are BST

code

```
class Solution {
public:
    bool isBST(TreeNode* root, long int &prev){
        if(root==NULL)  return true;
        bool isLeftBalanced = isBST(root->left, prev);
        if(root->val <= prev)   return false;
        prev = root->val;
        bool isRightBalanced = isBST(root->right, prev);
        return isLeftBalanced && isRightBalanced;
    }

    bool isValidBST(TreeNode* root) {
        long int prev = -99999999999;
        return isBST(root, prev);
    }
};
```

(36) LCA of BST →

Eg.



P = 2, q = 8

if currNode > both p & q
then LCA lies in LST

if currNode < both p & q
then LCA lies in RST

in every other case the currNode is LCA as p & q will be on

	worst	Avg
TC →	O(n)	O(logn)
SC →	O(n)	

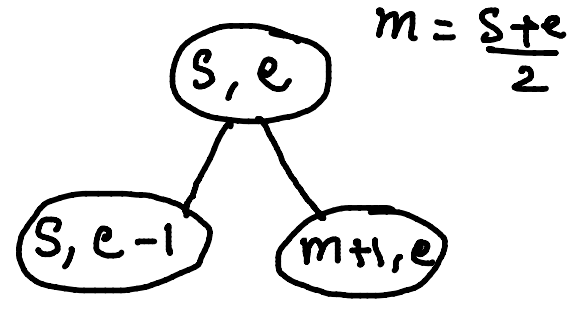
code

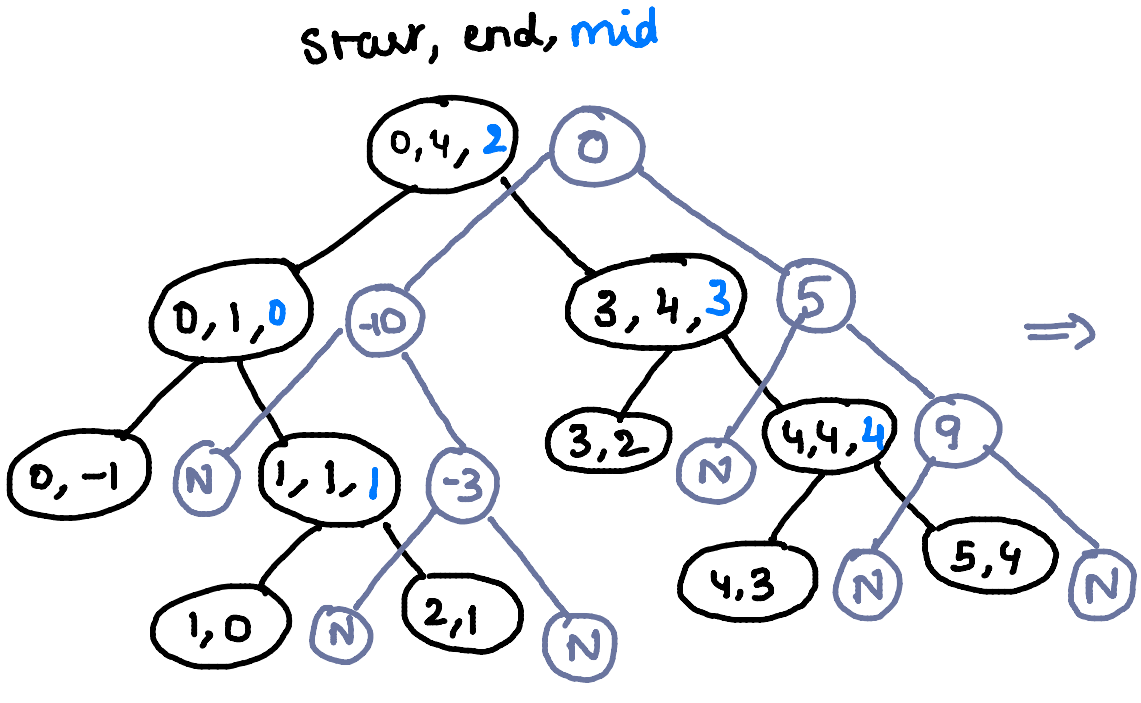
```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL)  return NULL;

        if(root->val < p->val && root->val < q->val){
            return lowestCommonAncestor(root->right, p, q);
        }
        else if(root->val > p->val && root->val > q->val){
            return lowestCommonAncestor(root->left, p, q);
        }
        else {
            return root;
        }
    }
};
```

(37) Convert Sorted Array to BST

Given sorted array, create a BST

Eg [-10, -3, 0, 5, 9]
0 1 2 3 4





Code →

```
class Solution {
public:
    TreeNode* createBST(vector<int>&nums, int start, int end){
        if(start>end)   return NULL;

        int mid = (start + end)/2;
        TreeNode* root = new TreeNode(nums[mid]);

        root->left = createBST(nums, start, mid-1);
        root->right = createBST(nums, mid+1, end);
        return root;
    }

    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return createBST(nums, 0, nums.size()-1);
    }
};
```

D13 (38) Construct BinaryTree from Pre & Inorder traversal

Eg

	0	1	2	3	4
pre =	3	9	20	15	7
in =	9	3	15	20	7

Tc → O(n²)

Sc → O(n)

* for every node in Pre, the corresponding LST & RST are in In

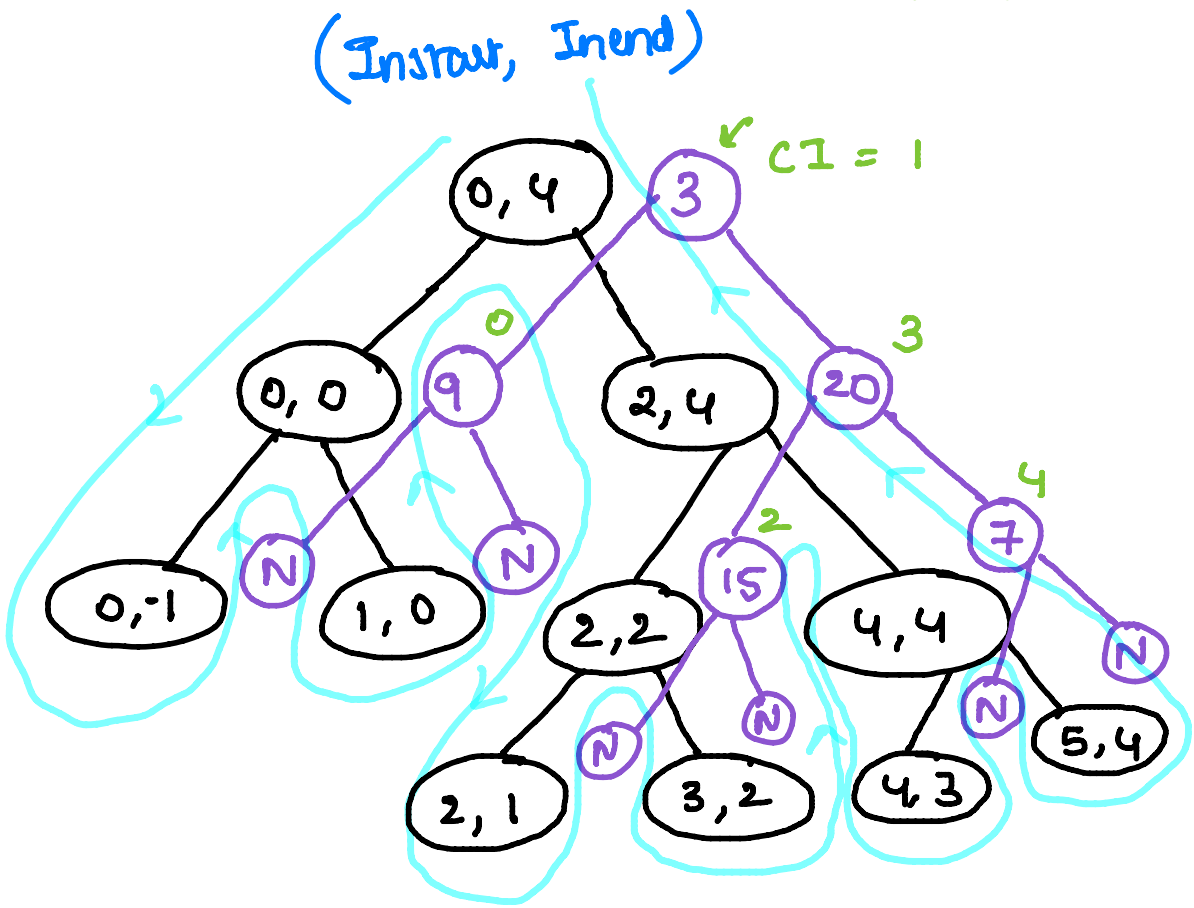
ie 3 → [LST: 9 (idx 0) | CI: 3 (idx 1) | RST: 15, 20, 7 (idx 2, 3, 4)]

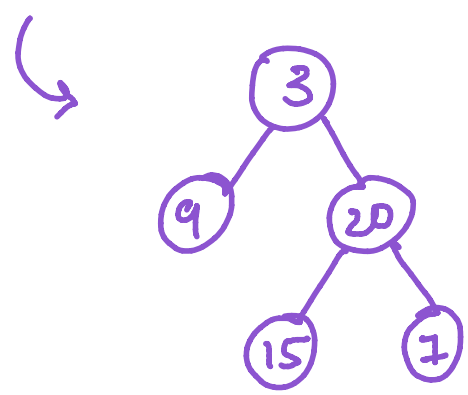
LST = (instart, CI-1)

RST = (CI+1, inend)

CI = index of pre[0] in In

(Instart, Inend)





	0	1	2	3	4
pre =	3	9	20	15	7
in =	9	3	15	20	7

↑ (pre) ↑ (in)

1

① for preorder index = 0, inorder boundry = [0, 4]

② find root value in Inorder array & it's index is CurrIndex

③ if instart > CI-1 or CI+1 < inend return NULL

To reduce Tc we can use hashtable to find indexing

Tc → O(n)

Sc → O(n)+O(n)

code →

```cpp
class Solution {
public:
    TreeNode* constructTree(vector<int>& preorder, unordered_map<int, int> &mp,
    int start, int end, int &preIdx ){

        if(start>end)   return NULL;
        TreeNode* root = new TreeNode(preorder[preIdx]);

        // find currIndex as per inorder array
        int currIdx = mp[preorder[preIdx]];
        // increment preIdx to find next root
        preIdx++;

        // recursively call LST & RST
        root->left = constructTree(preorder, mp, start, currIdx-1, preIdx);
        root->right = constructTree(preorder, mp, currIdx+1, end, preIdx);
        return root;
    }

    unordered_map<int,int> populate(vector<int>&inorder){
        unordered_map<int,int> mp;
        for(int i=0; i<inorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return mp;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int,int> mp = populate(inorder);
        int preIdx = 0;
        return constructTree(preorder, mp, 0, inorder.size()-1, preIdx);
    }
};

```

(39) Construct BinaryTree from In & Postorder traversal

Intuition is same as previous program, only changes are

→ traverse from last element in postorder array

→ process RST & then go for LST

Code →

```cpp
class Solution {
public:

    TreeNode* constructTree(vector<int>& postorder, unordered_map<int, int> &mp,
    int start, int end, int &postIdx ){

        if(start>end)   return NULL;
        TreeNode* root = new TreeNode(postorder[postIdx]);

        // find currIndex as per inorder array
        int currIdx = mp[postorder[postIdx]];
        postIdx--;

        // recursively call RST & LST
        root->right = constructTree(postorder, mp, currIdx+1, end, postIdx);
        root->left = constructTree(postorder, mp, start, currIdx-1, postIdx);
        return root;
    }

    unordered_map<int,int> populate(vector<int>&inorder){
        unordered_map<int,int> mp;
        for(int i=0; i<inorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return mp;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int,int> mp = populate(inorder);
        int postIdx = postorder.size()-1;
        return constructTree(postorder, mp, 0, inorder.size()-1, postIdx);
    }
};
```

(40) Construct BST from Preorder traversal

[8, 5, 1, 7, 10, 12]

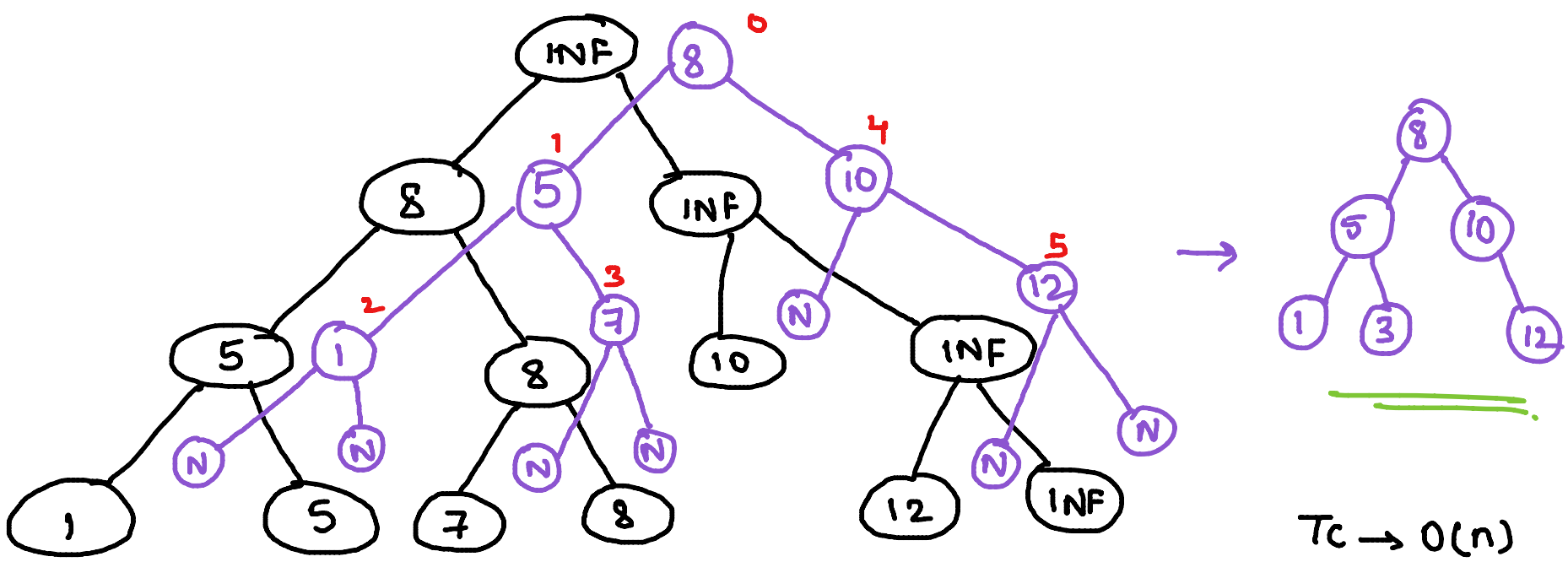
Tc → O(nlogn) (due to sorting)

Approach 1 → Sort given Preorder to get Inorder, now similar to problem (38).

Approach 2 →

[8, 5, 1, 7, 10, 12]
0 1 2 3 4 5

Boundary of LST → Val
RST → boundVal → initially (INF)



Code →

```cpp
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, int &preIdx, int boundary){
        if(preIdx >= preorder.size() || preorder[preIdx] >= boundary)
            return NULL;

        // create root using preIdx
        TreeNode* root = new TreeNode(preorder[preIdx]);
        preIdx++;

        // recursively call LST & RST
        root->left = buildTree(preorder, preIdx, root->val);
        root->right = buildTree(preorder, preIdx, boundary);
        return root;
    }

    TreeNode* bstFromPreorder(vector<int>& preorder) {
        int preIdx = 0;
        return buildTree(preorder, preIdx, 1001);
    }
};
```

Find the rest on
https://linktr.ee/KarunKarthik

Follow Karun Karthik for more!

https://www.linkedin.com/in/karun-karthik-5a0794187/

Graph

- Karun Karthik

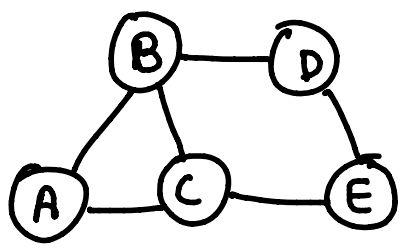
Contents

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. $n = |V|$ & $e = |E|$

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
↑ vertex ↑ edge

Representation →

Adj. matrix

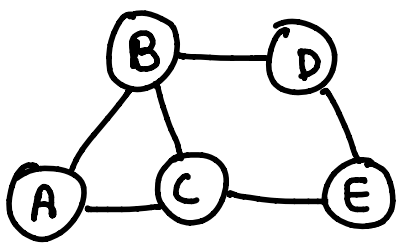
SC → $O(V^2)$

Adj list

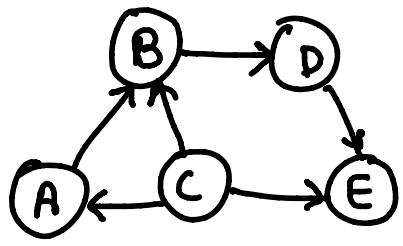
SC → $O(V+E)$

Types →

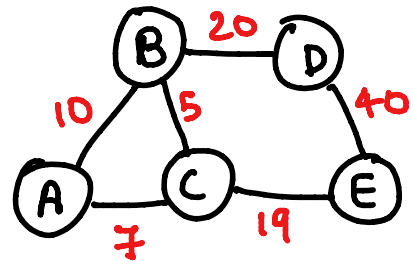
1) Undirected



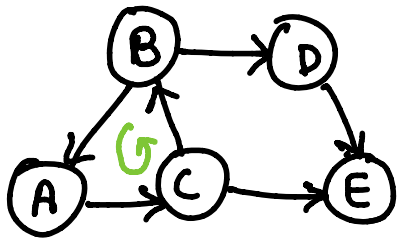
2) Directed



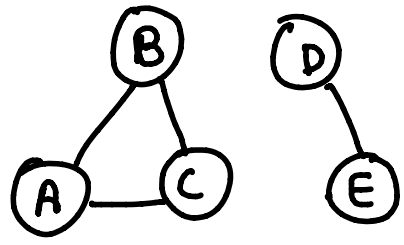
3) Weighted



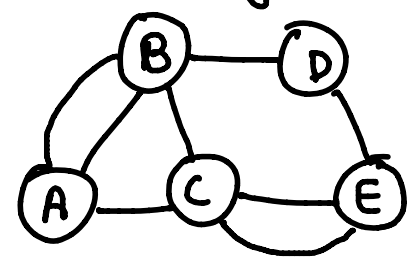
4) Cyclic



5) Disconnected



6) Multigraph

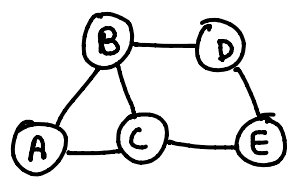


[no self loops]

Graph Traversal

(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

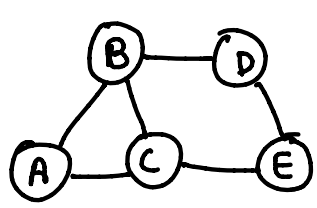


queue. _____

Visited

A	B	C	D	E

→



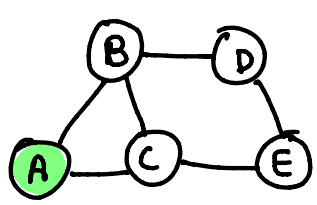
queue. _____

Visited

A	B	C	D	E

res _____

→



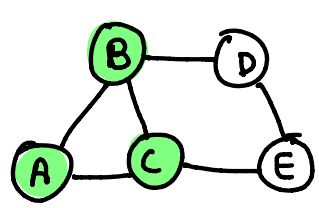
queue. A

Visited

A	B	C	D	E
■				

res A

→



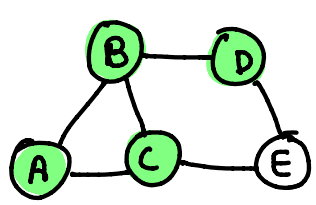
queue. A, B, C

Visited

A	B	C	D	E
■	■	■		

res A, B, C

→



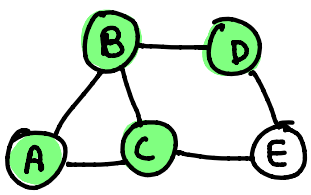
queue. B, C, D

Visited

A	B	C	D	E
■	■	■	■	

res A, B, C, D

→



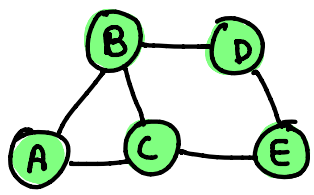
queue. B, C, D

Visited

A	B	C	D	E
■	■			

res A, B, C, D

→



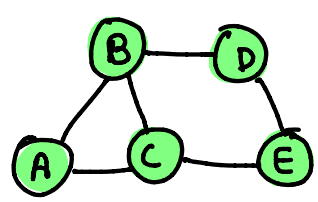
queue. C, D, E

Visited

A	B	C	D	E
■	■	■	■	

res A, B, C, D, E

→



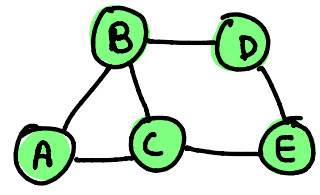
queue. C, D, E

Visited

A	B	C	D	E
■	■	■	■	

res A, B, C, D, E

→



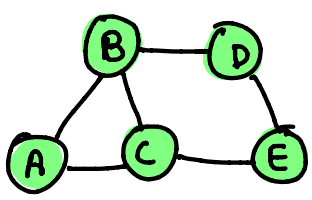
queue. D, E

Visited

A	B	C	D	E
■	■	■	■	

res A, B, C, D, E

→



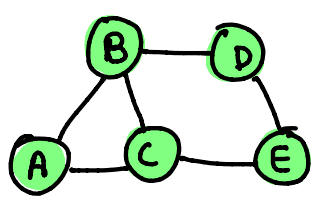
queue. E

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

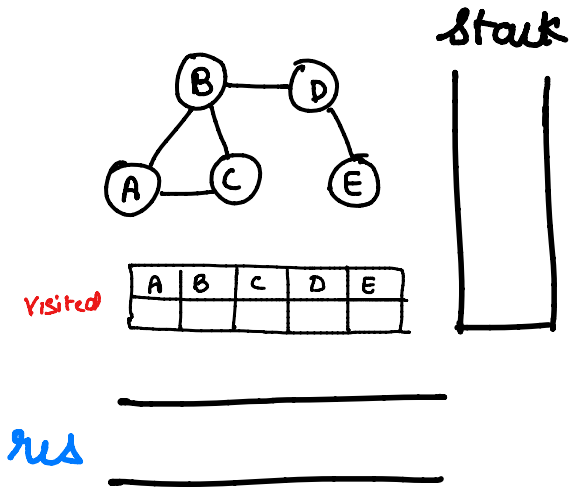
```
class Solution {
  public:

    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;
    }
};
```

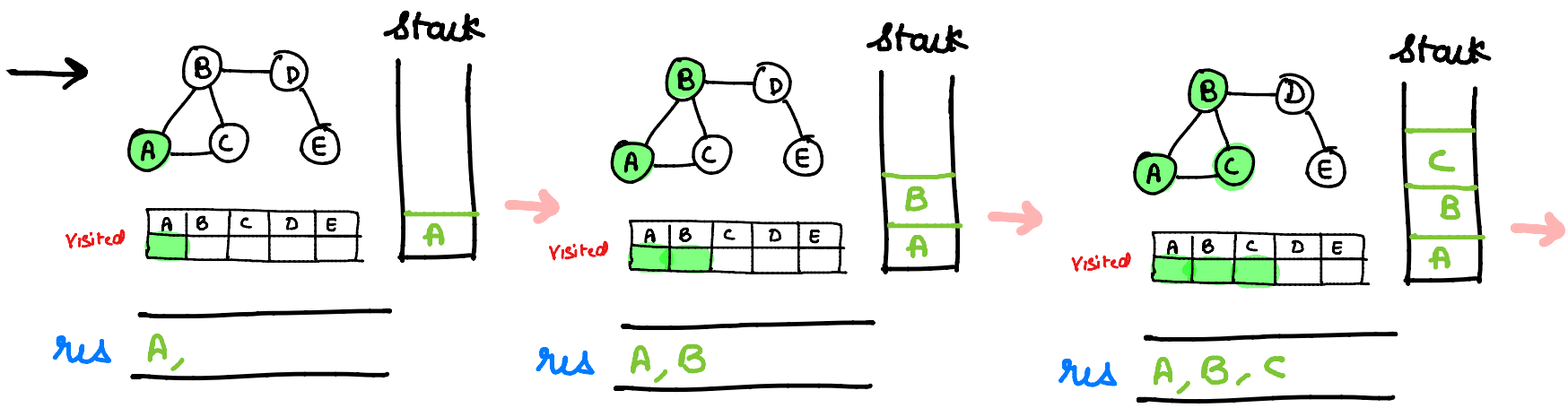
Applications → [BFS]

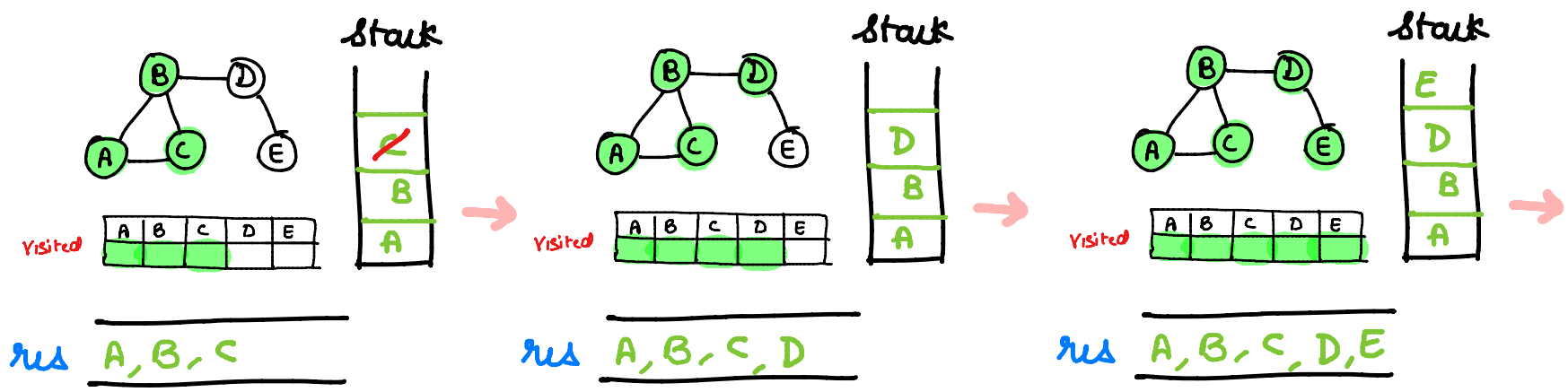
1. Shortest path
2. Min. spanning tree for unweighted graph
3. Cycle detection
4. GPS
5. Social network.

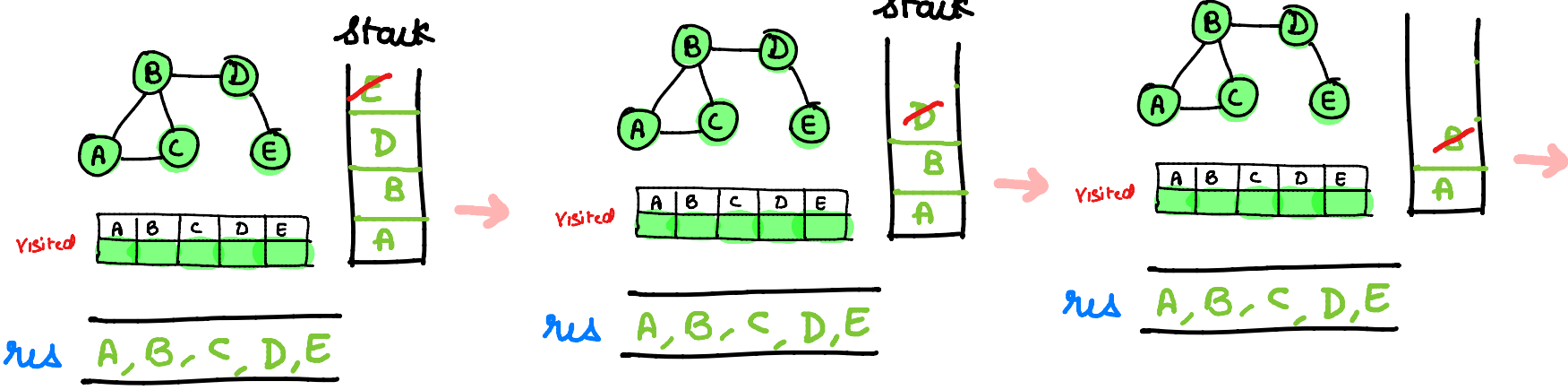
6 DFS →

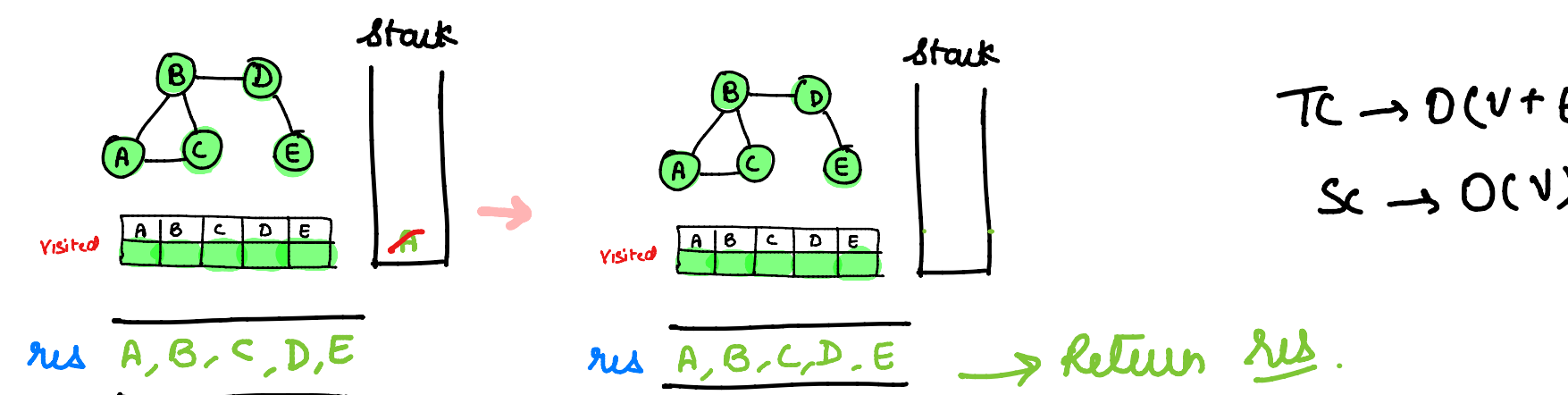
- Select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into Stack
- if no neighbours then pop.
- Repeat till Stack is empty











TC → O(V+E)

SC → O(V)

code

```
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;
    }
};
```

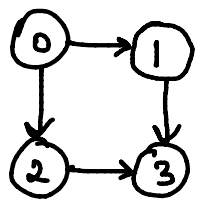
Applications → [DFS]

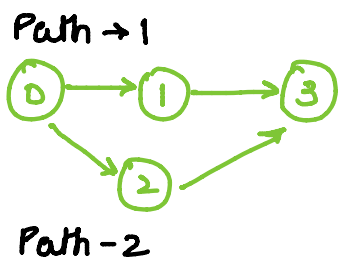
1. Path finding
2. Cycle detection
3. Topological sort
4. Finding strongly connected components.

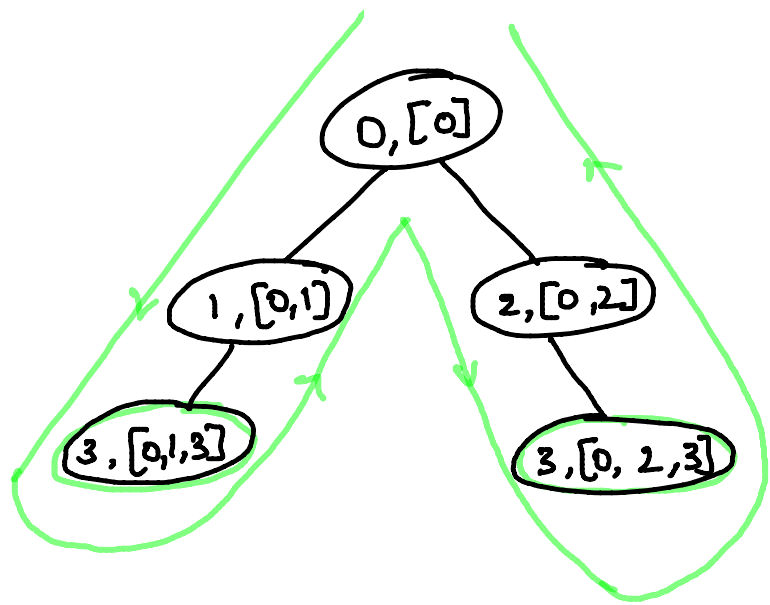
① All paths from src to taiger

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)

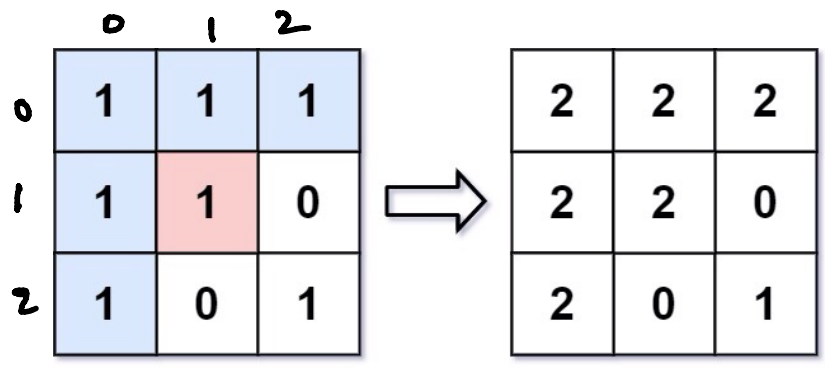
V → Vertices

E → edges

SC →

Recursive Stack + Result

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

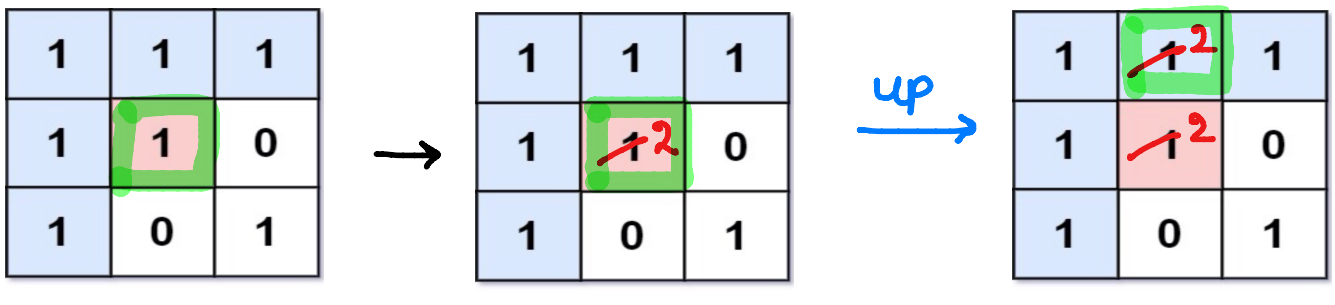


* perform flood fill from given index in all 4-directions & the cell should have same color as src

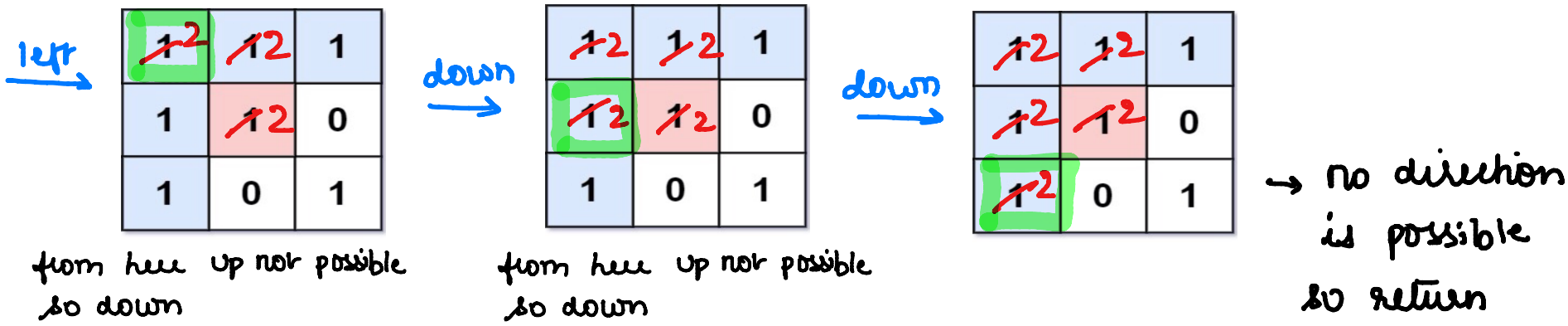
✓ Lets follow the order to fill → UP, DOWN, LEFT, RIGHT

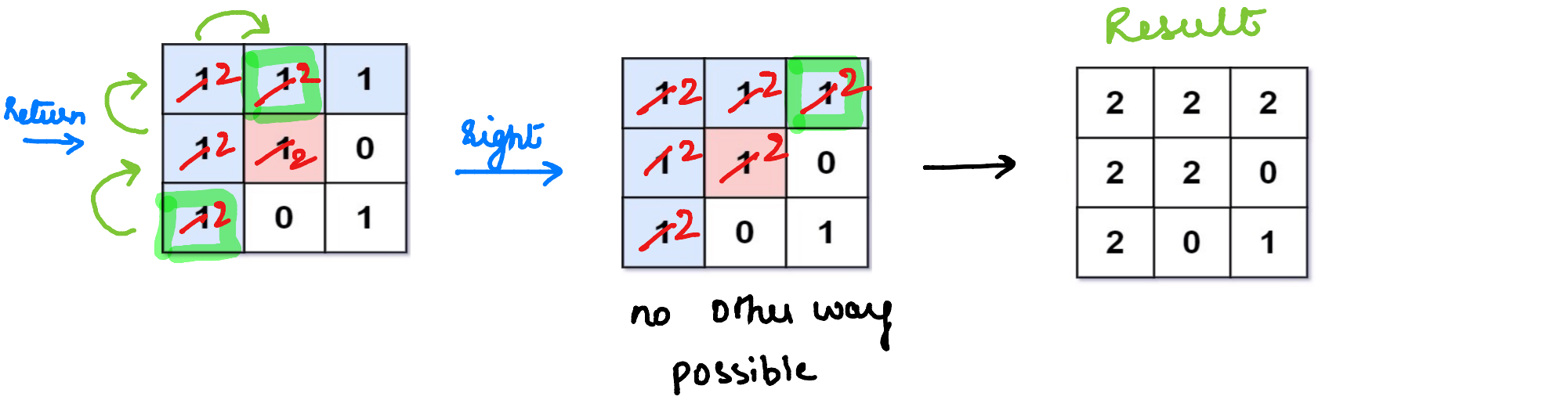
Eg In above case starting point is (1,1) & value = 1 so

(*)



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





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(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's 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, 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

```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 → $O(mn)$ 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.

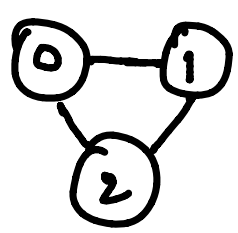
TC → O(mn) Avg case.

code →

```java
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 

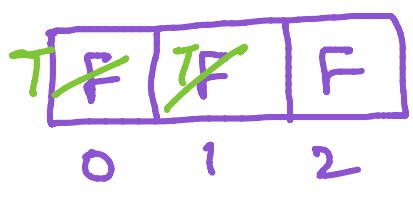
→ [[0,1], [1,2], [2,0]] edges.

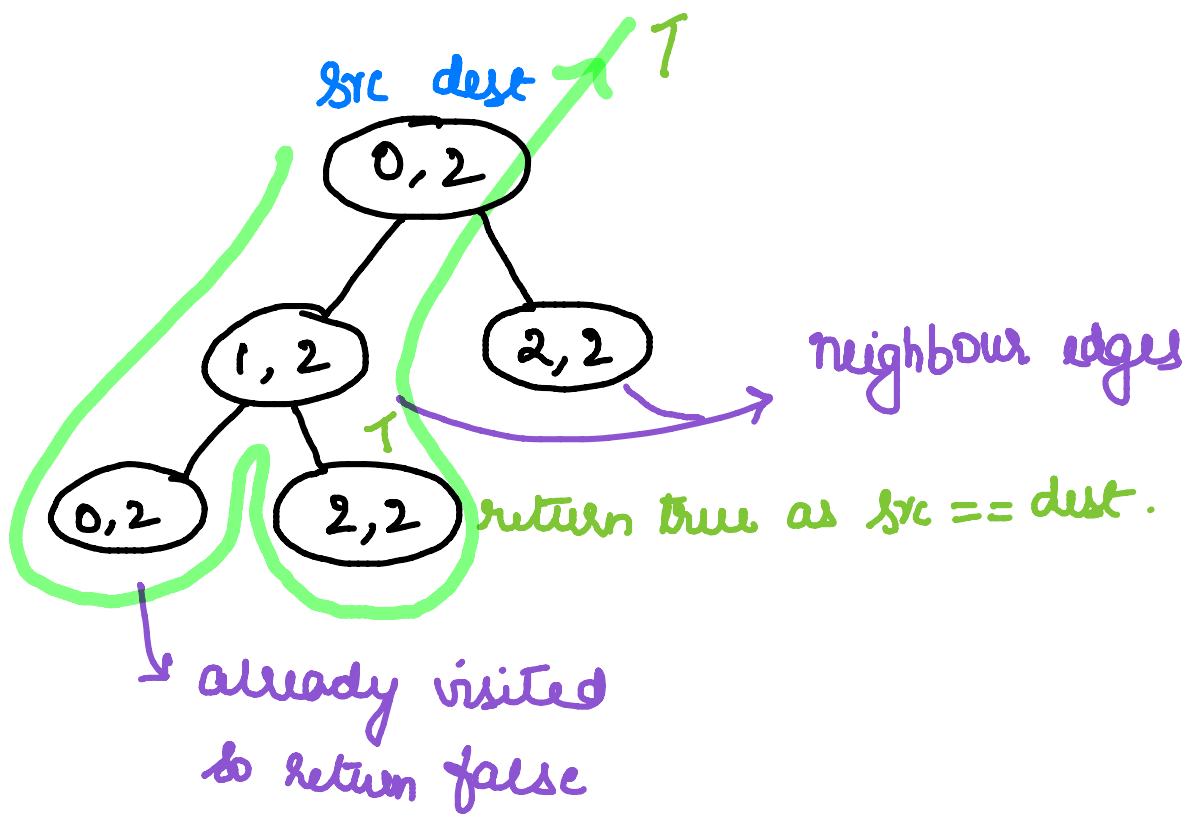
n = 3 edges = [[0,1],[1,2],[2,0]] src = 0, dest = 2.

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



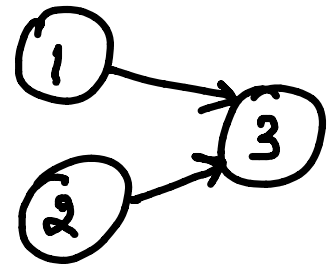


Code →

```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

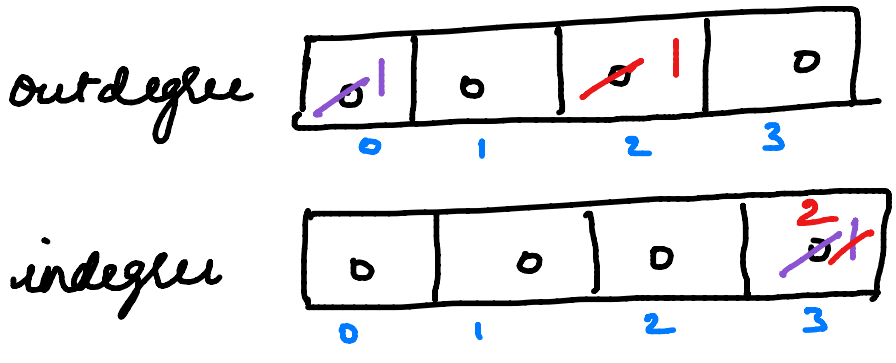
$n = 3$, trust = $[[1,3],[2,3]]$ →



* Indegree of town judge = n-1

& Outdegree = 0

✓ Create 2 arrays



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

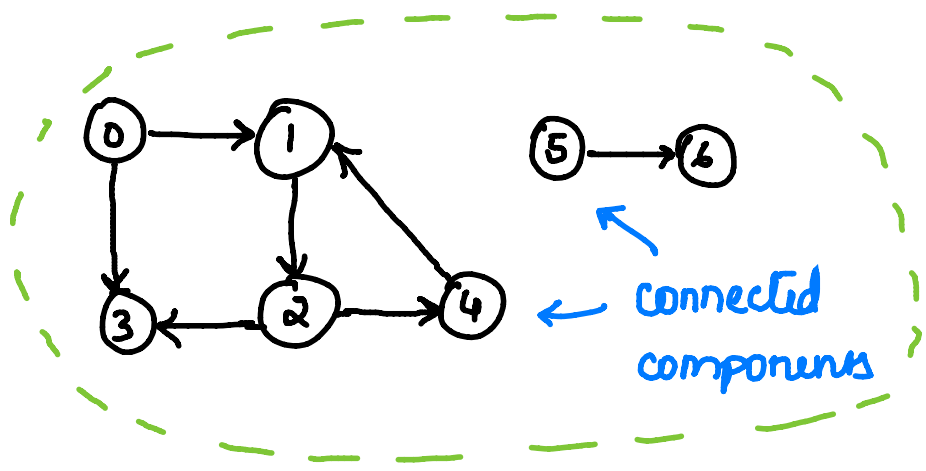
```cpp
class Solution {
public:
    int findJudge(int n, vector<vector<int>>& trust) {
        vector<int>indegree(n+1,0);
        vector<int>outdegree(n+1,0);
        for(int i=0;i<trust.size();i++)
        {
            int v1 = trust[i][0];
            int v2 = trust[i][1];
            outdegree[v1]+=1;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

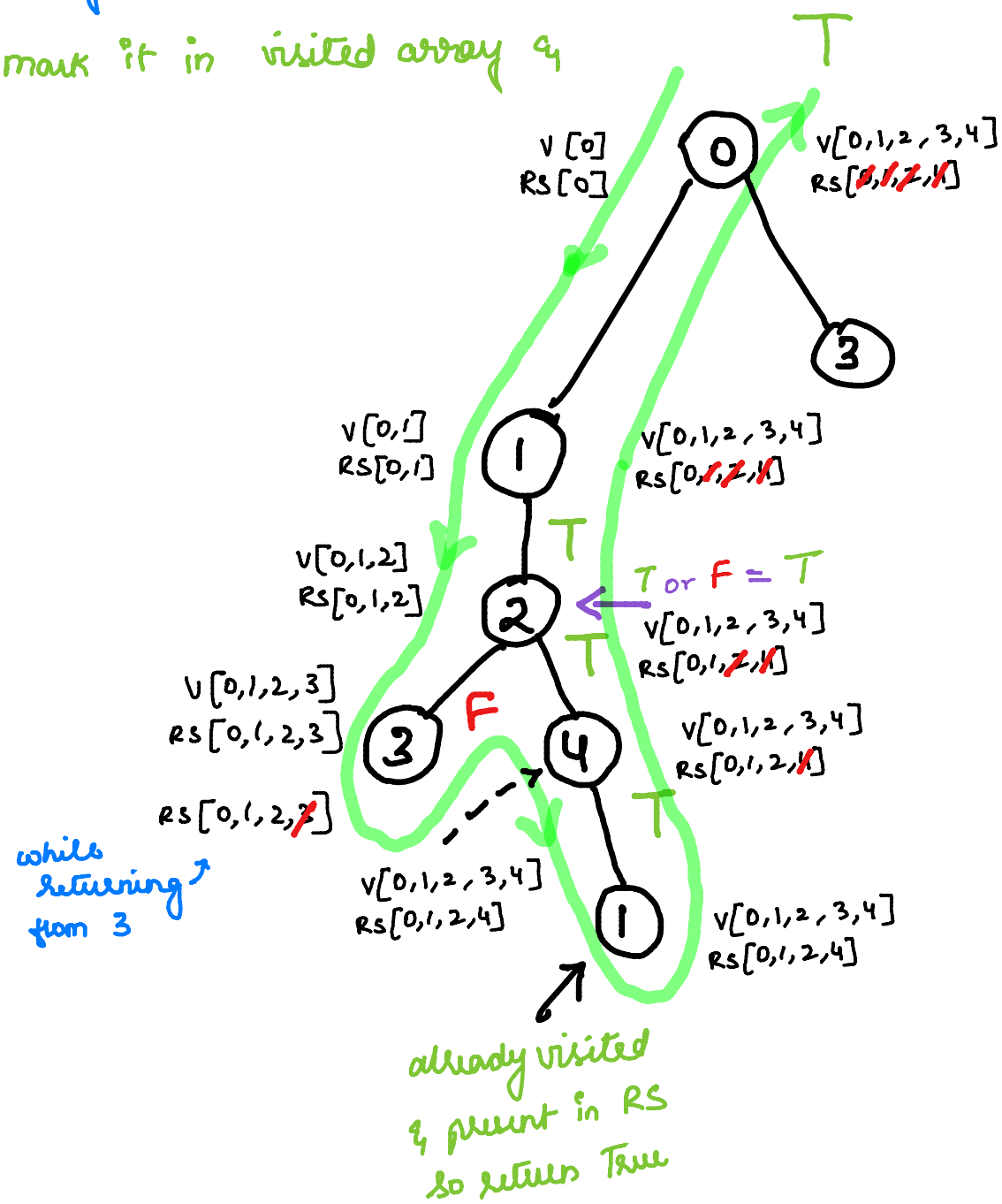
* At 3 vertex, there's no neighbour & no cycle is detected so return F.

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

$Tc \rightarrow O(V+E)$

$Sc \rightarrow O(V)$

```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

↓

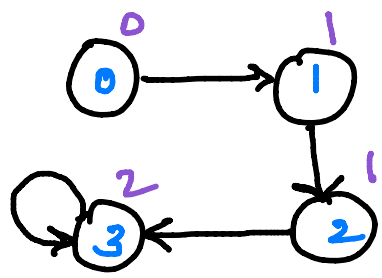
can be used to find cycle using BFS.

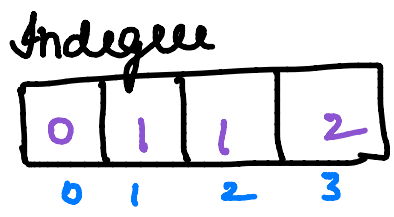
① find indegree of every vertex in graph & answer = 0

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

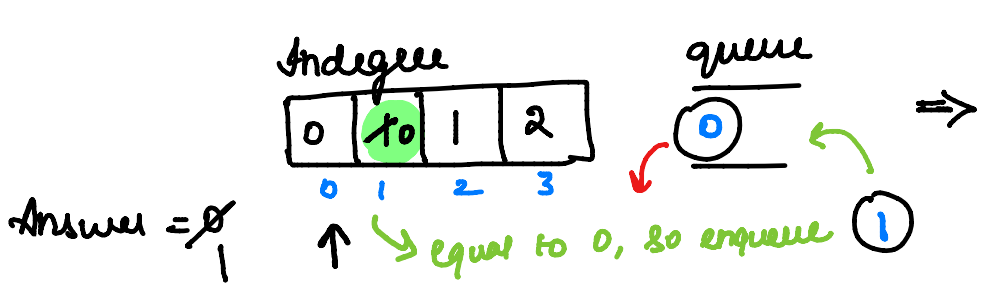


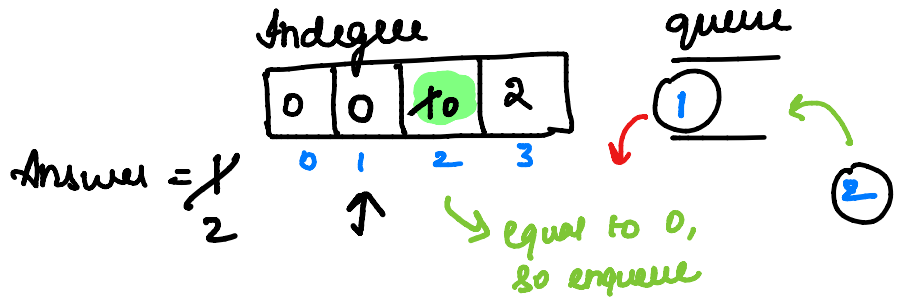


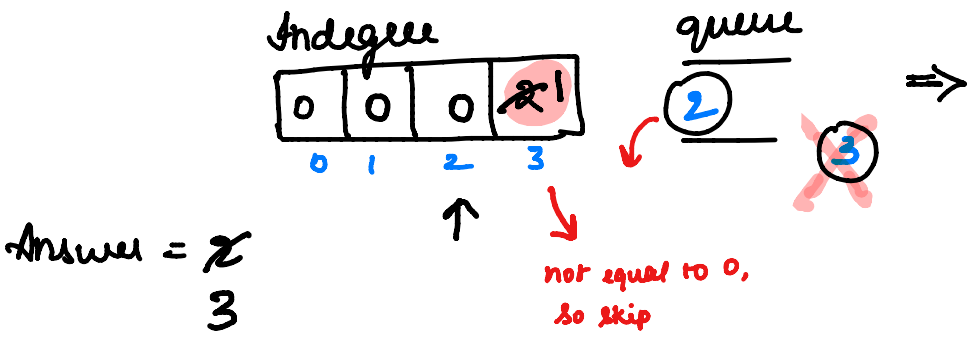
queue

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

```cpp
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 results while performing dfs.

Code →

TC → O(V + E)

SC → O(V)

```cpp
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.

$Tc \rightarrow O(V+E)$

$Sc \rightarrow O(V+E)$

Code →

```cpp
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;
    }
};
```

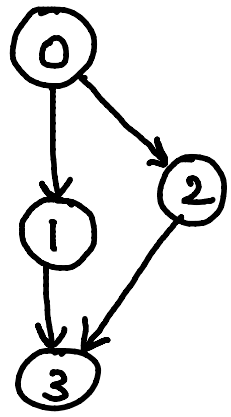
(10) Course Schedule – II

pre → edge [v, u]

"u should be completed before v"

n → no. of courses [vertices]

Topological sort only for DAG

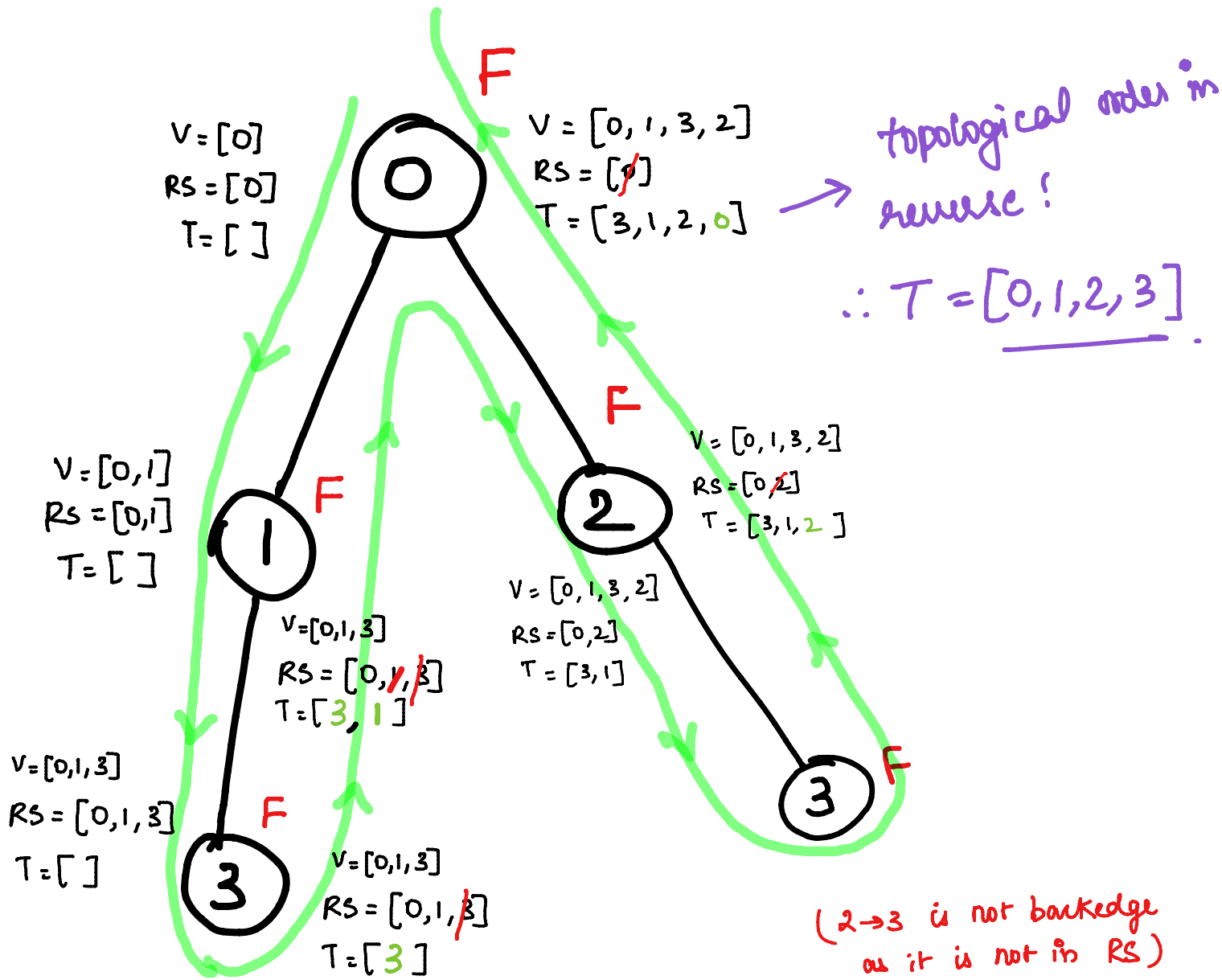


Eg n → 4 (0, 1, 2, 3)

pre → [[1,0], [2,0], [3,1], [3,2]]

Initially

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



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 !

Graph - 2

- Karun Karthik

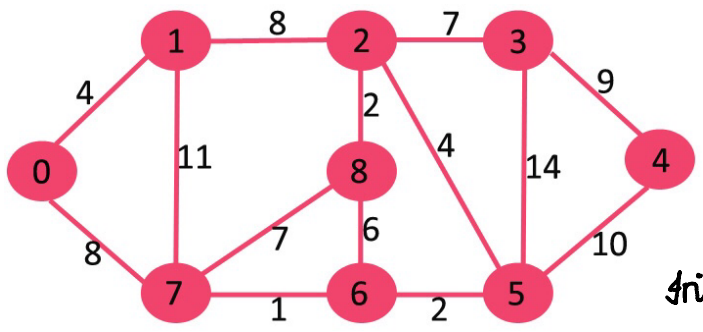
Content

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)

cost array = 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 = { }

→ As it is weighted graph, we'll use priority queue (pq) instead of normal queue. & 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.

✓

vis = {1}

cost[1] = 0

(1,0)

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

(1,0) (0,4) (7,11) (2,8)

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

⇒ (1,0) (0,4) (7,11) (2,8)

vis = {1, 0}

cost[0] = 4

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

(1,0) (0,4) (7,11) (2,8) (7,12)

→ lowest cost is 8 ∴ pop & push its neighbours

⇒ (1,0) (0,4) (7,11) (2,8) (7,12)

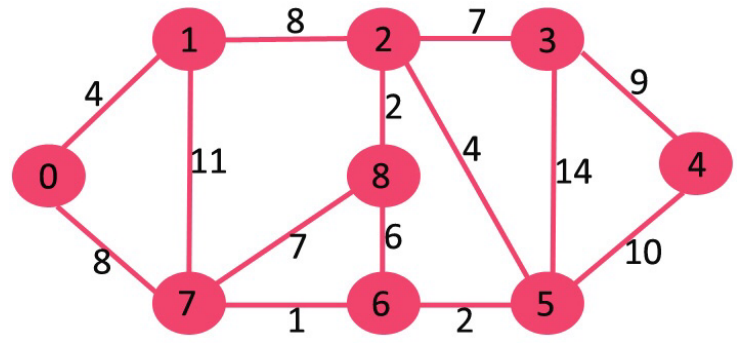
vis = {1, 0, 2}

cost[2] = 8

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

(1,0) (0,4) (7,11) (2,8) (7,12) (8,10) (3,15) (5,12)

→ lowest cost is 10 ∴ pop & push its neighbours



⇒

(1,0) (0,4) (7,11) (2,8) (7,12) (8,10) (3,15) (5,12) ⇒ neighbours of 8 = 2 (visited), (7,17), (6,16) ∴ push

vis = {1,0,2,8}

cost[8] = 10

(1,0) (0,4) (7,11) (2,8) (7,12) (8,10) (3,15) (5,12) (7,17) (6,16)

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

⇒

(1,0) (0,4) (7,11) (2,8) (7,12) (8,10) (3,15) (5,12) (7,17) (6,16) ⇒ neighbours of 7 = 0, 1, 8 are visited. & (6,12) ∴ push

vis = {1,0,2,8,7}

cost[7] = 11

(1,0) (0,4) (7,11) (2,8) (7,12) (8,10) (3,15) (5,12) (7,17) (6,16) (6,12) ≃ (7,12) (3,15) (5,12) (7,17) (6,16) (6,12)

↳ lowest cost = 12

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

⇒

(3,15) (5,12) (7,17) (6,16) (6,12) ⇒ neighbours of 5 = (4,22) (3,26) (6,14) ∴ push

vis = {1,0,2,8,7,5}

cost[5] = 12

(3,15) (5,12) (7,17) (6,16) (6,12) (4,22) (3,26) (6,14) → lowest cost = 12 ∴ pop & push its neighbours

⇒ (3,15) (7,17) (6,16) (6,12) (4,22) (3,26) (6,14) ⇒ neighbours of 6 = 5, 7, 8 are visited. ∴ no push

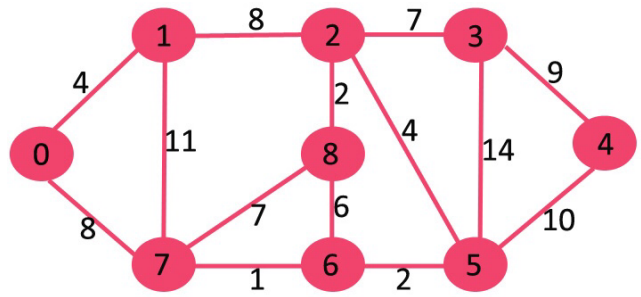
vis = {1,0,2,8,7,5,6}

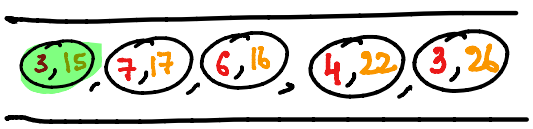
cost[6] = 12

(3,15) (7,17) (6,16) (6,12) (4,22) (3,26) (6,14) → next lowest is 14, but 6 is already visited.

(3,15) (7,17) (6,16) (4,22) (3,26) (6,14)

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



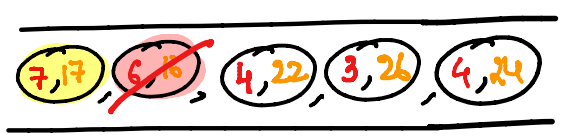
⇒ 

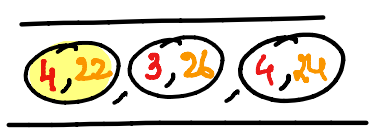
⇒ neighbours of 3 = 2,5 (visited) 4,24 ∴ push

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

cost[3] = 15

→ next lowest cost = 16
but 6 is already visited ∴ pop

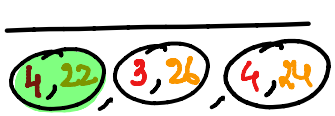


⇒ 

→ next lowest cost = 22

∴ pop & push its neighbours.

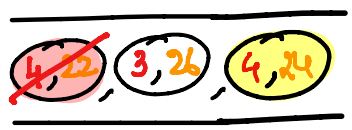
→ next lowest cost = 17
but 7 is already visited ∴ pop

⇒ 

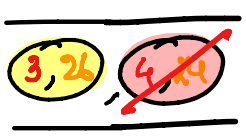
⇒ neighbours of 4 = 3,5 (visited) ∴ no push

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

cost [4] = 22



→ next lowest cost = 24
but 4 is already visited
∴ pop



→ next lowest cost = 26
but 3 is already visited
∴ pop

⇒ 

⇒ ∴ empty PQ.

Answer ⇒ cost =

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

Dijkstras = BFS + PQ

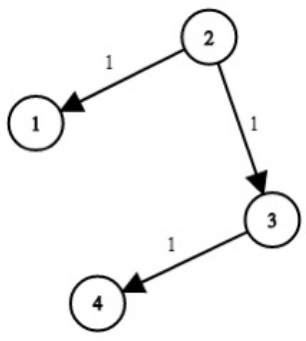
Tc → O(V + E log V)

Sc → O(V)

Code →

```cpp
class Solution
{
    public:
    vector <int> dijkstra(int V, vector<vector<int>> adj[], int src)
    {
        vector<int>cost(V,0);
        cost[src]=0;

        vector<bool>vis(V, false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;

        pq.push({0,src}); // {cost, node}

        while(!pq.empty())
        {
            pair<int,int>p = pq.top();
            int currCost = p.first;
            int currNode = p.second;
            pq.pop();

            if(vis[currNode])   continue;

            vis[currNode] = true;
            cost[currNode] = currCost;

            for(int i=0;i<adj[currNode].size();i++)
            {
                int neighbourNode = adj[currNode][i][0];
                int weight = adj[currNode][i][1];
                // if already visited then skip
                if(vis[neighbourNode])  continue;
                // else push
                pq.push({currCost + weight, neighbourNode});
            }
        }
        return cost;
    }
};
```

(12) Network Delay Time



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 `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`.

src = 2.

✓. Similar to dijkstra's algo. cost = [0 0 0 0 0] (0 1 2 3 4) vis = {} pq = ____

⇒ push (2, 0) to pq. ⇒ pq: (2, 0)

⇒ pq: (2, 0) neighbours = (1, 1), (3, 1) ∴ push

vis = {2}

cost[2] = 0

pq: ~~(2, 0)~~, (1, 1), (3, 1) → next lowest cost = 1 ∴ choose 1 of 3

pop & push their neighbours.

⇒ pq: (1, 1), (3, 1) no new neighbours ∴ pop

vis = {2, 1}

cost[1] = 1

pq: ~~(1, 1)~~, (3, 1) → next lowest cost = 1

∴ pop & push their neighbours.

⇒ pq: (3, 1) neighbour = (4, 2) ∴ push

vis = {2, 1, 3}

cost[3] = 1

pq: ~~(3, 1)~~, (4, 2) → next lowest cost = 2

∴ pop & push neighbours.

⇒ pq: (4, 2) no new neighbours ∴ pop

vis = {2, 1, 3, 4}

cost[4] = 2

pq: ~~(4, 2)~~ → pq is empty.

Tc → O(V + E log V)

Sc → O(V)

∴ cost = [0 1 0 1 2] (0 1 2 3 4)

→ check if all nodes are in visited, else return -1.

→ Return max value in cost as

Code →

```cpp
class Solution {
public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<vector<int>>> graph = createGraph(times,n);
        return minTime(graph,n,k);
    }

    vector<vector<vector<int>>> createGraph(vector<vector<int>>& edges,int n) {

        vector<vector<vector<int>>> graph(n+1);

        for(int i=0;i<=n;i++) {
            graph.push_back({{}});
        }
        // add every edge to the graph
        for(vector<int> edge:edges) {
            int source = edge[0];
            int dest = edge[1];
            int cost = edge[2];
            graph[source].push_back({dest,cost});
        }
        return graph;
    }

    int minTime(vector<vector<vector<int>>> &graph,int n,int src) {

        vector<int> cost(n+1,0);
        cost[src] = 0;
        vector<bool>vis(n+1, false);

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,src}); // {cost, node}

        while(!pq.empty()) {
            pair<int,int>p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();
            // if already visited then skip
            if(vis[currNode])   continue;

            vis[currNode] = true;
            cost[currNode] = currCost;

            for(int i=0;i<graph[currNode].size();i++)
            {
                int neighbourNode = graph[currNode][i][0];
                int weight = graph[currNode][i][1];
                // if already visited then skip
                if(vis[neighbourNode])  continue;
                // else push into pq
                pq.push({currCost + weight, neighbourNode});
            }
        }

        for(int i=1; i<=n; i++)
            if(vis[i]==0)   return -1;

        int ans = 0;
        for(int x:cost)    ans = max(ans,x);
        return ans;
    }
};
```

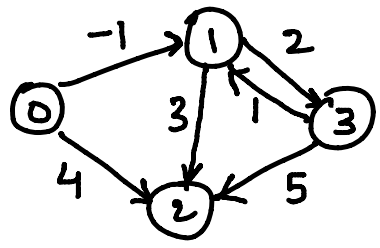
(13) Bellman Ford Algorithm → useful when weights < 0 (Dijkstra fails)

↳ 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

⇒ dist[src] = 0 &

⇒ relax every edge n-1 time ie run for loop & perform the following operation

dist[dest] = min(dist[src] + weight, dist[dest])

⇒ finally relax one more time &

if dist[dest] > dist[src] + wt ⇒ -ve weight cycle present

⇒ we should relax 3 times & src = 0 ⇒ dist[src] = 0 dist

0	inf	inf	inf
0	1	2	3

→ for edge [0,1,-1], dist[1] = min(0 + (-1), inf) = -1

[0,2,4], dist[2] = min(0+4, inf) = 4

[1,2,3], dist[2] = min(-1+3, 4) = 2

[1,3,2], dist[3] = min(-1+2, inf) = 1

[3,1,1], dist[1] = min(1+1, -1) = -1

[3,2,5], dist[2] = min(1+5, 2) = 2.

∴ dist =

0	-1	2	1
0	1	2	3

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

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

Answer ⇒ dist =

0	-1	2	1
0	1	2	3

TC → O(V * E)

SC → O(V)