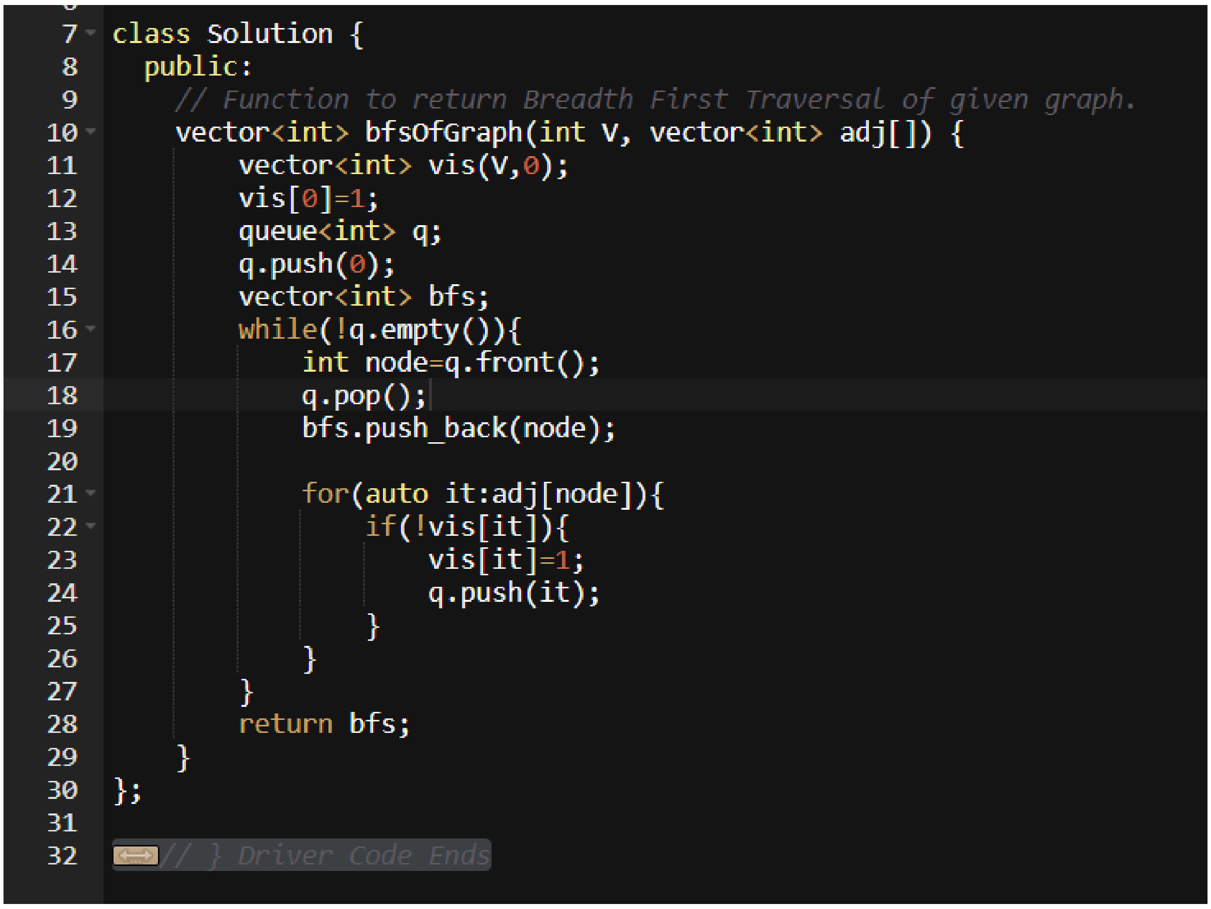
1.) Adjacency matrix to Adjacency List code

```
vector<vector<int>> convert( vector<vector<int>> a){
    vector<vector<int>> adjList(a.size());
    for (int i = 0; i < a.size(); i++){
        for (int j = 0; j < a[i].size(); j++){
            if (a[i][j] != 0){
                adjList[i].push_back(j);
            }
        }
    }
    return adjList;
}
```

2.) BFS TRAVERSAL

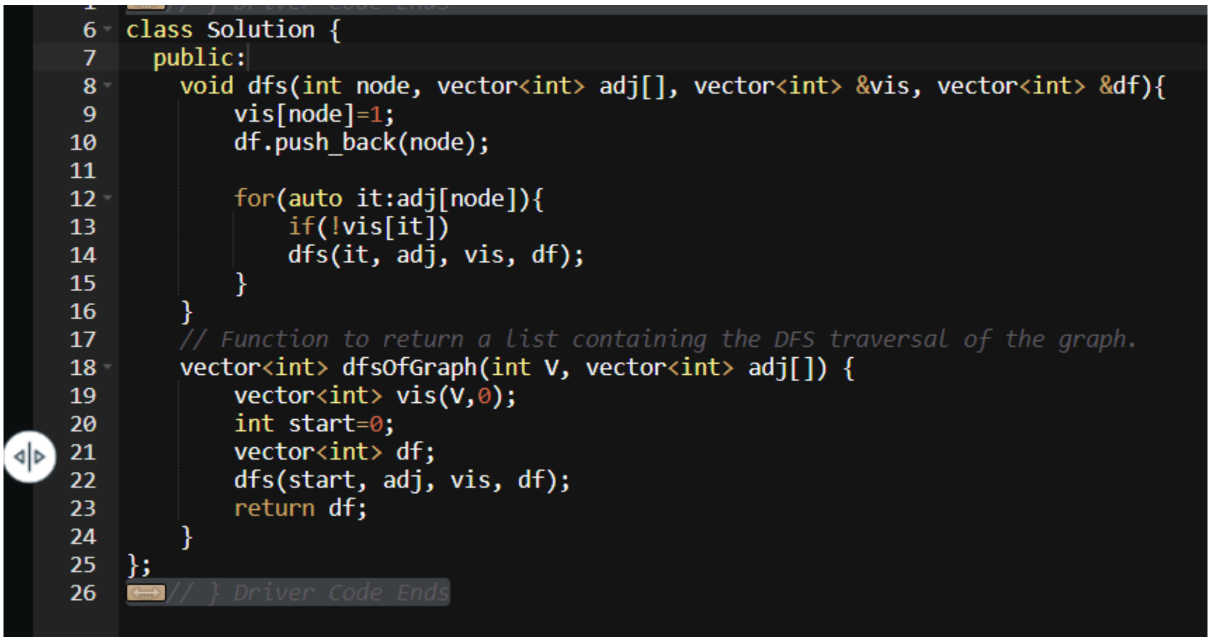


```cpp
class Solution {
  public:
    // Function to return Breadth First Traversal of given graph.
    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        vector<int> vis(V,0);
        vis[0]=1;
        queue<int> q;
        q.push(0);
        vector<int> bfs;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            bfs.push_back(node);

            for(auto it:adj[node]){
                if(!vis[it]){
                    vis[it]=1;
                    q.push(it);
                }
            }
        }
        return bfs;
    }
};

// } Driver Code Ends
```

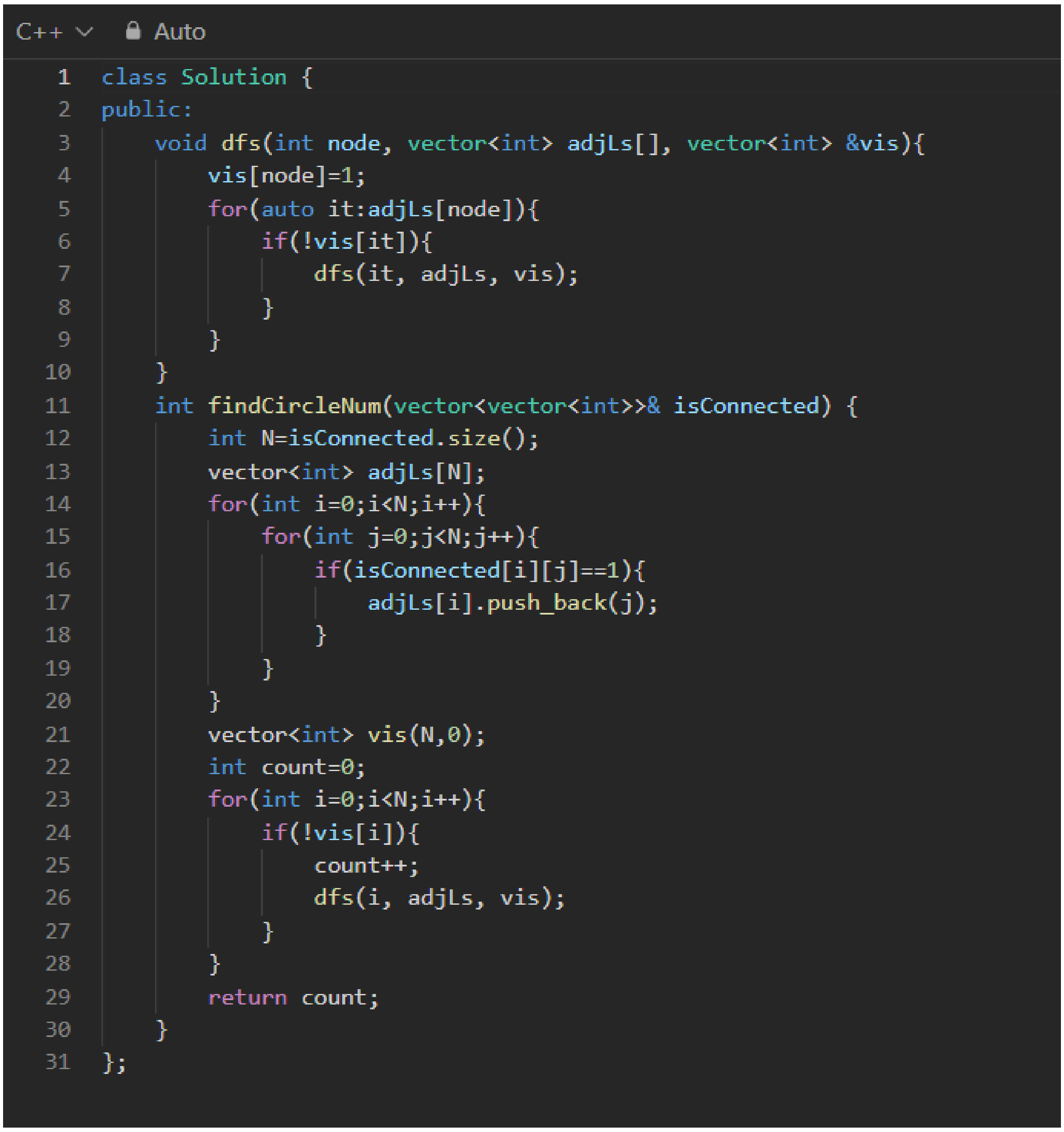
3.) DFS TRAVERSAL



```cpp
class Solution {
  public:
    void dfs(int node, vector<int> adj[], vector<int> &vis, vector<int> &df){
        vis[node]=1;
        df.push_back(node);

        for(auto it:adj[node]){
            if(!vis[it])
            dfs(it, adj, vis, df);
        }
    }
    // Function to return a list containing the DFS traversal of the graph.
    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        vector<int> vis(V,0);
        int start=0;
        vector<int> df;
        dfs(start, adj, vis, df);
        return df;
    }
};
// } Driver Code Ends
```

4.) NUMBER OF PROVINCES



```cpp
class Solution {
public:
    void dfs(int node, vector<int> adjLs[], vector<int> &vis){
        vis[node]=1;
        for(auto it:adjLs[node]){
            if(!vis[it]){
                dfs(it, adjLs, vis);
            }
        }
    }
    int findCircleNum(vector<vector<int>>& isConnected) {
        int N=isConnected.size();
        vector<int> adjLs[N];
        for(int i=0;i<N;i++){
            for(int j=0;j<N;j++){
                if(isConnected[i][j]==1){
                    adjLs[i].push_back(j);
                }
            }
        }
        vector<int> vis(N,0);
        int count=0;
        for(int i=0;i<N;i++){
            if(!vis[i]){
                count++;
                dfs(i, adjLs, vis);
            }
        }
        return count;
    }
};
```

5.) NUMBER OF ISLANDS

```
class Solution {
public:
    void bfs(int row, int col, vector<vector<int>> &vis, vector<vector<char>>
&grid){
        vis[row][col]=1;
        queue<pair<int,int>> q;
        q.push({row,col});
        int N=grid.size();
        int M=grid[0].size();
        while(!q.empty()){
            int row=q.front().first;
            int col=q.front().second;
            q.pop();
            for(int delrow=-1;delrow<=1;delrow++){
                for(int delcol=-1;delcol<=1;delcol++){
                    int neighrow=row+delrow;
                    int neighcol=col+delcol;
                    if((abs(delrow-delcol)==1) &&
                        (neighrow>=0 && neighrow<N &&
                         neighcol>=0 && neighcol<M &&
                         grid[neighrow][neighcol]=='1'
                         && !vis[neighrow][neighcol])){
                        vis[neighrow][neighcol]=1;
                        q.push({neighrow,neighcol});
                    }
                }
            }
        }
    }

    int numIslands(vector<vector<char>>& grid) {
        int N=grid.size();
        int M=grid[0].size();
        vector<vector<int>> vis(N, vector<int>(M,0));
        int count=0;
        for(int row=0;row<N;row++){
            for(int col=0;col<M;col++){
                if(!vis[row][col] && grid[row][col]=='1'){
                    bfs(row, col, vis, grid);
                    count++;
                }
            }
        }
        return count;
    }
};
```

6.) FLOOD FILL ALGORITHM

```
class Solution {
public:
    void dfs(int row, int col, vector<vector<int>> &ans,vector<vector<int>>
&image, int color, vector<int> &delrow, vector<int> &delcol, int inicolor){
        ans[row][col]=color;
        int N=image.size();
        int M=image[0].size();
        for(int i=0;i<4;i++){
            int nrow=row+delrow[i];
            int ncol=col+delcol[i];
            if(nrow>=0 && nrow<N && ncol>=0
               && ncol<M &&  image[nrow][ncol]==inicolor
               && ans[nrow][ncol]!=color){
                dfs(nrow, ncol, ans, image, color, delrow, delcol, inicolor);
            }
        }
    }

    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc,
int color) {
        int inicolor=image[sr][sc];
        vector<vector<int>> ans=image;
        vector<int> delrow={-1,0,+1,0};
        vector<int> delcol={0,+1,0,-1};
        dfs(sr, sc, ans, image, color, delrow, delcol, inicolor);
        return ans;
    }
};
```

7.) NUMBER OF ROTTEN ORANGES

```cpp
class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {
        int N=grid.size();
        int M=grid[0].size();
        queue<pair<pair<int,int>,int>> q;
        vector<vector<int>> vis(N, vector<int>(M));
        int count_fresh=0;
        for(int i=0;i<N;i++){
            for(int j=0;j<M;j++){
                if(grid[i][j]==2){
                    q.push({{i,j},0});
                    vis[i][j]=2;
                }
                else{
                    vis[i][j]=0;
                }
                if(grid[i][j]==1)
                count_fresh++;
            }
        }
        int time=0;
        //BFS TRAVERSAL
        vector<int> delrow={-1,0,+1,0};
        vector<int> delcol={0,1,0,-1};
        int count=0;
        while(!q.empty()){
            int row=q.front().first.first;
            int col=q.front().first.second;
            int t=q.front().second;
            time=max(time,t);
            q.pop();
            for(int i=0;i<4;i++){
                int neighrow=row+delrow[i];
                int neighcol=col+delcol[i];
                if(neighrow>=0 && neighrow<N && neighcol>=0
                   && neighcol<M && vis[neighrow][neighcol]==0
                   && grid[neighrow][neighcol]==1){
                    q.push({{neighrow,neighcol},t+1});
                    vis[neighrow][neighcol]=2;
                    count++;
                }
            }
        }
        if(count!=count_fresh)
        return -1;
        return time;
    }
};

//Time Complexity: O(NxM + NxMx4) ~ O(N x M)
// the BFS function will be called for (N x M) nodes and for every node, we
are traversing for 4 neighbours, it will take O(N x M x 4) time.

// Space Complexity ~ O(N x M), O(N x M)
```

8.) CYCLE DETECTION IN UNDIRECTED GRAPH (BFS)

```
class Solution {
  public:
    bool detect(int src, vector<int> adj[], vector<int> &vis){
        vis[src]=1;
        queue<pair<int,int>> q;
        q.push({src,-1});
        while(!q.empty()){
            int node=q.front().first;
            int parent=q.front().second;
            q.pop();

            for(auto adjNode:adj[node]){
                if(!vis[adjNode]){
                    vis[adjNode]=1;
                    q.push({adjNode, node});
                }
                else if(parent!=adjNode){
                    return true;
                }
            }
        }
        return false;
    }
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
        vector<int> vis(V,0);
        for(int i=0;i<V;i++){
            if(!vis[i]){
                if(detect(i, adj, vis))
                return true;
            }
        }
        return false;
    }
};
```

9.) CYCLE DETECTION IN UNDIRECTED GRAPH (DFS)

```cpp
class Solution {
  public:
    bool dfs(int node, int parent, vector<int> &vis, vector<int> adj[]){
        vis[node]=1;
        for(auto adjNode:adj[node]){
            if(!vis[adjNode]){
                if(dfs(adjNode, node, vis, adj)==true)
                return true;
            }
            else if(adjNode!=parent)
            return true;
        }
        return false;
    }
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
        vector<int> vis(V,0);
        for(int i=0;i<V;i++){
            if(!vis[i]){
                if(dfs(i,-1,vis,adj)==true)
                return true;
            }
        }
        return false;
    }
};
```

10.) DISTANCE OF NEAREST CELL HAVING 1

```
class Solution
{
    public:
    //Function to find distance of nearest 1 in the grid for each cell.
    vector<vector<int>>nearest(vector<vector<int>>grid){
        int n=grid.size();
        int m=grid[0].size();
        vector<vector<int>> vis(n, vector<int> (m,0));
        vector<vector<int>> dist(n, vector<int> (m,0));
        queue<pair<pair<int,int>,int>> q;

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(grid[i][j]==1){
                    q.push({{i,j},0});
                    vis[i][j]=1;
                }
                else{
                    vis[i][j]=0;
                }
            }
        }
        vector<int> drow={-1,0,1,0};
        vector<int> dcol={0,1,0,-1};

        while(!q.empty()){
            int row=q.front().first.first;
            int col=q.front().first.second;
            int steps=q.front().second;
            q.pop();
            dist[row][col]=steps;

            for(int i=0;i<4;i++){
                int nrow=row+drow[i];
                int ncol=col+dcol[i];

                if((nrow>=0 && nrow<n && ncol>=0
                    && ncol<m && vis[nrow][ncol]==0)){
                    vis[nrow][ncol]=1;
                    q.push({{nrow,ncol},steps+1});
                }
            }
        }
        return dist;
    }
};
```

11.) SURROUNDED REGIONS

```
class Solution {
public:
    void dfs(int row, int col, vector<vector<int>> &vis, vector<vector<char>>
&mat, vector<int> &delrow, vector<int> &delcol){
        vis[row][col]=1;

        int N=mat.size();
        int M=mat[0].size();

        for(int i=0;i<4;i++){
            int neighrow=row+delrow[i];
            int neighcol=col+delcol[i];
            if(neighrow>=0 && neighrow<N && neighcol>=0
               && neighcol<M && !vis[neighrow][neighcol]
               && mat[neighrow][neighcol]=='O'){
                dfs(neighrow, neighcol, vis, mat, delrow, delcol);
            }
        }
    }

    void solve(vector<vector<char>>& mat) {
        int N=mat.size();
        int M=mat[0].size();

        vector<int> delrow={-1,0,1,0};
        vector<int> delcol={0,1,0,-1};

        vector<vector<int>> vis(N,vector<int>(M,0));

        for(int j=0;j<M;j++){
            if(!vis[0][j] && mat[0][j]=='O'){
                dfs(0, j, vis, mat, delrow, delcol);
            }
            if(!vis[N-1][j] && mat[N-1][j]=='O'){
                dfs(N-1, j, vis, mat, delrow, delcol);
            }
        }

        for(int i=0;i<N;i++){
            if(!vis[i][0] && mat[i][0]=='O'){
                dfs(i, 0, vis, mat, delrow, delcol);
            }
            if(!vis[i][M-1] && mat[i][M-1]=='O'){
                dfs(i, M-1, vis, mat, delrow, delcol);
            }
        }
        for(int i=0;i<N;i++){
            for(int j=0;j<M;j++){
                if(!vis[i][j] && mat[i][j]=='O')
                mat[i][j]='X';
            }
        }
    }
};

// TC - O(N) + O(M) + O(NxMx4) ~ O(N x M)
// SC - O(N x M), O(N x M)
```

12.) NUMBER OF ENCLAVES

```cpp
class Solution {
public:
    int numEnclaves(vector<vector<int>>& grid) {
        queue<pair<int,int>> q;

        int N=grid.size();
        int M=grid[0].size();

        vector<vector<int>> vis(N,vector<int>(M,0));
        for(int i=0;i<N;i++){
            for(int j=0;j<M;j++){
                if(i==0 || j==0 || i==N-1 || j==M-1){
                    if(grid[i][j]==1){
                        q.push({i,j});
                        vis[i][j]=1;
                    }
                }
            }
        }

        vector<int> delrow={-1,0,+1,0};
        vector<int> delcol={0,+1,0,-1};

        while(!q.empty()){
            int row=q.front().first;
            int col=q.front().second;
            q.pop();

            for(int i=0;i<4;i++){
                int neighrow=row+delrow[i];
                int neighcol=col+delcol[i];
                if(neighrow>=0 && neighrow<N && neighcol>=0 && neighcol<M &&
vis[neighrow][neighcol]==0 && grid[neighrow][neighcol]==1){
                    q.push({neighrow,neighcol});
                    vis[neighrow][neighcol]=1;
                }
            }
        }

        int count=0;
        for(int i=0;i<N;i++){
            for(int j=0;j<M;j++){
                if(grid[i][j]==1 && vis[i][j]==0)
                count++;
            }
        }
        return count;
    }
};

// TC - O(N*M*4) ~ O(N*M);
// SC - O(N*M);
```

13.) NUMBER OF DISTINCT ISLANDS

```
class Solution {
  public:
    void dfs(int row, int col, vector<vector<int>>&grid,
             vector<vector<int>> &vis, vector<pair<int,int>> &vec,
             int row0, int col0){
        vis[row][col]=1;
        vec.push_back({row-row0, col-col0});

        int n=grid.size();
        int m=grid[0].size();

        vector<int> drow={-1,0,1,0};
        vector<int> dcol={0,-1,0,1};

        for(int i=0;i<4;i++){
            int nrow=row+drow[i];
            int ncol=col+dcol[i];

            if(nrow>=0 && nrow<n && ncol>=0 && ncol<m
               && grid[nrow][ncol]==1 && !vis[nrow][ncol]){
                dfs(nrow,ncol, grid, vis, vec, row0, col0);
            }
        }
    }

    int countDistinctIslands(vector<vector<int>>& grid) {
        int n=grid.size();
        int m=grid[0].size();

        vector<vector<int>> vis(n, vector<int> (m,0));
        set<vector<pair<int,int>>> st;

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(!vis[i][j] && grid[i][j]==1){
                    vector<pair<int,int>> vec;
                    dfs(i, j, grid, vis, vec, i, j);
                    st.insert(vec);
                }
            }
        }
        return st.size();
    }
};
```

14.) BIPARTITE GRAPH (BFS)

```
class Solution {
public:
    bool check(int start, int N, vector<vector<int>> &graph,
               vector<int> &color){
        queue<int> q;
        q.push(start);

        color[start]=0;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            for(auto it:graph[node]){
                if(color[it]==-1){
                    color[it]=!color[node];
                    q.push(it);
                }
                else if(color[it]==color[node]){
                    return false;
                }
            }
        }
        return true;
    }

    bool isBipartite(vector<vector<int>>& graph) {
        int N=graph.size();
        vector<int> color(N,-1);
        for(int i=0;i<N;i++){
            if(color[i]==-1){
                if(check(i, N, graph, color)==false)
                return false;
            }
        }
        return true;
    }
};

// BFS APPROACH
// TIME COMPLEXITY - O(V+2E)
// SPACE COMPLEXITY - O(3V)~O(V)
```

15.) BIPARTITE GRAPH (DFS)

```
class Solution {
public:
    bool dfs(int node, int col, vector<int> &color,
             vector<vector<int>> &graph){
        color[node]=col;
        for(auto it:graph[node]){
            if(color[it]==-1){
                if(dfs(it, !col, color, graph)==false)
                return false;
            }
            else if(color[it]==col)
            return false;
        }
        return true;
    }

    bool isBipartite(vector<vector<int>>& graph) {
        int N=graph.size();
        vector<int> color(N,-1);
        for(int i=0;i<N;i++){
            if(color[i]==-1){
                if(dfs(i, 0, color, graph)==false)
                return false;
            }
        }
        return true;
    }
};

// Time Complexity: O(V + E)
// Space Complexity: O(V)
```

16.) DETECT CYCLE IN A DIRECTED GRAPH

```cpp
class Solution {
  public:
    bool dfs(int node, vector<int> adj[], vector<int> &vis,
              vector<int> &pathVis){
        vis[node]=1;
        pathVis[node]=1;

        for(auto it:adj[node]){
            if(!vis[it]){
                if(dfs(it, adj, vis, pathVis)==true)
                return true;
            }
            else if(pathVis[it]){
                return true;
            }
        }
        pathVis[node]=0;
        return false;
    }
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
        vector<int> vis(V,0);
        vector<int> pathVis(V,0);

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

17.) FIND EVENTUAL SAFE NODES (DFS)

```cpp
class Solution {
public:
    bool dfs(int node, vector<vector<int>> &graph,
             vector<int> &vis, vector<int> &pathvis,
             vector<int> &check){
        vis[node]=1;
        pathvis[node]=1;
        check[node]=1;
        for(auto it:graph[node]){
            if(!vis[it]){
                if(dfs(it, graph, vis, pathvis, check)==true){
                    check[node]=0;
                    return true;
                }
            }
            else if(pathvis[it]){
                check[node]=0;
                return true;
            }
        }
        check[node]=1;
        pathvis[node]=0;
        return false;
    }
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        int N=graph.size();
        vector<int> vis(N,0);
        vector<int> pathvis(N,0);
        vector<int> check(N,0);
        vector<int> ans_safenodes;
        for(int i=0;i<N;i++){
            if(!vis[i]){
                dfs(i, graph, vis, pathvis, check);
            }
        }
        for(int i=0;i<N;i++){
            if(check[i]==1)
            ans_safenodes.push_back(i);
        }

        return ans_safenodes;
    }
};

// SC - O(3N)~O(N)
// TC - O(V+E)
```

18.) TOPOLOGICAL SORTING (DFS)

```cpp
class Solution
{
    public:
    void dfs(int node, vector<int> &vis, stack<int> &st, vector<int> adj[]){
        vis[node]=1;
        for(auto it:adj[node]){
            if(!vis[it]){
                dfs(it, vis, st, adj);
            }
        }
        st.push(node);
    }
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        vector<int> vis(V,0);
        stack<int> st;

        for(int i=0;i<V;i++){
            if(!vis[i]){
                dfs(i, vis, st, adj);
            }
        }

        vector<int> ans;
        while(!st.empty()){
            ans.push_back(st.top());
            st.pop();
        }
        return ans;
    }
};
```

19.) TOPOLOGICAL SORTING (BFS)

[KAHN’S ALGORITHM]

```
class Solution
{
    public:
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        vector<int> indegree(V,0);
        for(int i=0;i<V;i++){
            for(auto it:adj[i]){
                indegree[it]++;
            }
        }

        queue<int> q;
        for(int i=0;i<V;i++){
            if(indegree[i]==0){
                q.push(i);
            }
        }
        vector<int> topo;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            topo.push_back(node);

            for(auto it:adj[node]){
                indegree[it]--;
                if(indegree[it]==0)
                q.push(it);
            }
        }
        return topo;
    }
};
```

20.) FIND EVENTUAL SAFE NODES (TOPO SORT)

```
class Solution {
public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        int N=graph.size();

        vector<int> adjrev[N];
        vector<int> indegree(N,0);

        for(int i=0;i<N;i++){
            for(auto it:graph[i]){
                adjrev[it].push_back(i);
                indegree[i]++;
            }
        }

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

        while(!q.empty()){
            int node=q.front();
            q.pop();
            safenodes.push_back(node);
            for(auto it:adjrev[node]){
                indegree[it]--;
                if(indegree[it]==0)
                q.push(it);
            }
        }

        sort(safenodes.begin(),safenodes.end());
        return safenodes;
    }
};

// TC ~ similar to topo sort + sorting time
// SC ~ similar to topo sort + for storing reverse graph
```

21.) COURSE SCHEDULE I

```cpp
class Solution {
public:
    bool canFinish(int numcourses, vector<vector<int>>& prerequisites) {
        vector<int> adj[numcourses];
        for(auto it:prerequisites){
            adj[it[1]].push_back(it[0]);
        }

        vector<int> indegree(numcourses,0);
        for(int i=0;i<numcourses;i++){
            for(auto it:adj[i]){
                indegree[it]++;
            }
        }

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

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

        if(topo.size()==numcourses)
        return true;
        return false;
    }
};
```

22.) COURSE SCHEDULE II

```cpp
class Solution {
public:
    vector<int> findOrder(int numcourses, vector<vector<int>>& prerequisites)
{
        vector<int> adj[numcourses];
        for(auto it:prerequisites){
            adj[it[1]].push_back(it[0]);
        }

        vector<int> indegree(numcourses,0);
        for(int i=0;i<numcourses;i++){
            for(auto it:adj[i]){
                indegree[it]++;
            }
        }

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

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

        if(topo.size()==numcourses)
        return topo;
        return {};
    }
};
```

23.) ALIEN DICTIONARY

```
class Solution{
    public:
    vector<int> topoSort(int V, vector<int> adj[]) {
        vector<int> indegree(V,0);
        for(int i=0;i<V;i++){
            for(auto it:adj[i]){
                indegree[it]++;
            }
        }

        queue<int> q;
        for(int i=0;i<V;i++){
            if(indegree[i]==0){
                q.push(i);
            }
        }
        vector<int> topo;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            topo.push_back(node);

            for(auto it:adj[node]){
                indegree[it]--;
                if(indegree[it]==0)
                q.push(it);
            }
        }
        return topo;
    }
    string findOrder(string dict[], int N, int K) {
        vector<int> adj[K];
        for(int i=0;i<N-1;i++){
            string s1=dict[i];
            string s2=dict[i+1];
            int len=min(s1.size(),s2.size());

            for(int ptr=0;ptr<len;ptr++){
                if(s1[ptr]!=s2[ptr]){
                    adj[s1[ptr]-'a'].push_back(s2[ptr]-'a');
                    break;
                }
            }
        }
        vector<int> topo=topoSort(K, adj);
        string ans="";
        for(auto it:topo){
            ans=ans+char(it+'a');
        }
        return ans;
    }
};
```

24.) SHORTEST PATH IN DIRECTED ACYCLIC GRAPH (TOPO)

```
class Solution {
  public:
    void topoSort(int node, vector<pair<int,int>> adj[],
                  vector<int> &vis, stack<int> &st){
        vis[node]=1;
        for(auto it:adj[node]){
            int v=it.first;
            if(!vis[v]){
                topoSort(v, adj, vis, st);
            }
        }
        st.push(node);
    }
    vector<int> shortestPath(int N,int M, vector<vector<int>>& edges){
        vector<pair<int,int>> adj[N];
        for(int i=0;i<M;i++){
            int u=edges[i][0];
            int v=edges[i][1];
            int wt=edges[i][2];

            adj[u].push_back({v,wt});
        }

        vector<int> vis(N,0);
        stack<int> st;

        for(int i=0;i<N;i++){
            if(!vis[i]){
                topoSort(i, adj, vis, st);
            }
        }

        vector<int> dist(N);
        for(int i=0;i<N;i++){
            dist[i]=1e9;
        }

        dist[0]=0;

        while(!st.empty()){
            int node=st.top();
            st.pop();

            for(auto it:adj[node]){
                int v=it.first;
                int wt=it.second;
                if(dist[node]+wt<dist[v]){
                    dist[v]=dist[node]+wt;
                }
            }
        }

        for (int i = 0; i < N; i++) {
            if (dist[i] == 1e9) dist[i] = -1;
        }
        return dist;
    }
};
```

25.) SHORTEST PATH IN UNDIRECTED GRAPH HAVING UNIT DISTANCES

```
class Solution {
  public:
    vector<int> shortestPath(vector<vector<int>>& edges, int N,int M, int
src){
        vector<int> adj[N];
        for(auto it:edges){
            adj[it[0]].push_back(it[1]);
            adj[it[1]].push_back(it[0]);
        }

        vector<int> dist(N);
        for(int i=0;i<N;i++){
            dist[i]=1e9;
        }

        dist[src]=0;
        queue<int> q;
        q.push(src);
        while(!q.empty()){
            int node=q.front();
            q.pop();
            for(auto it:adj[node]){
                if(dist[node]+1<dist[it]){
                    dist[it]=1+dist[node];
                    q.push(it);
                }
            }
        }
        vector<int> ans(N, -1);
        for(int i=0;i<N;i++){
            if(dist[i]!=1e9){
                ans[i]=dist[i];
            }
        }
        return ans;
    }
};
```

26.) WORD LADDER I

```
class Solution {
public:
    int ladderLength(string beginword, string endword, vector<string>&
wordlist) {
        queue<pair<string,int>> q;
        q.push({beginword,1});
        unordered_set<string> st(wordlist.begin(),wordlist.end());
        st.erase(beginword);
        while(!q.empty()){
            string word=q.front().first;
            int steps=q.front().second;
            q.pop();

            if(word==endword)
            return steps;

            for(int i=0;i<word.size();i++){
                char original=word[i];
                for(char ch='a';ch<='z';ch++){
                    word[i]=ch;
                    // it exists in a set
                    if(st.find(word)!=st.end()){
                        st.erase(word);
                        q.push({word,steps+1});
                    }
                }
                word[i]=original;
            }
        }
        return 0;
    }
};

// TC ~ O(wordlength * 26 * beginword.size()); to be more specific extra
log(N)
// SC ~ O(N)
```

27.) WORD LADDER II

(MEMORY LIMIT EXCEEDED)

(BEST FOR INTERVIEW PURPOSE)

```
class Solution {
public:
    vector<vector<string>> findLadders(string beginWord, string endWord,
vector<string>& wordList) {
        unordered_set<string> s(wordList.begin(),wordList.end());
        queue<vector<string>> q;
        vector<vector<string>> ans;
        q.push({beginWord});
        vector<string> used;
        used.push_back(beginWord);
        int level=0;
        while(!q.empty())
        {
            vector<string> v=q.front();
            q.pop();
            if(v.size()>level){
                level++;
                for(auto it:used)
                {
                    s.erase(it);
                }
            }
            string word = v.back();
            if(word==endWord)
            {
                if(ans.size()==0)
                ans.push_back(v);
                else
                 if(ans[0].size()==v.size())
                    ans.push_back(v);

            }
            for(int i=0;i<word.length();i++)
            {
                char temp=word[i];
                for(char p='a';p<='z';p++)
                {
                    word[i]=p;
                    if(s.find(word)!=s.end())
                    {
                        v.push_back(word);
                        q.push(v);
                        used.push_back(word);
                        v.pop_back();
                    }
                }
                word[i]=temp;
            }
        }
        return ans;
    }
};
```

28.) WORD LADDER II

(BEST OPTIMISED WAY TO DO SO)

(NOT GOOD FOR INTERVIEW PURPOSE)

```
class Solution {
    unordered_map<string,int> mpp;
    vector<vector<string>> ans;
    string b;
private:
    void dfs(string word, vector<string> &seq){
        if(word==b){
            reverse(seq.begin(),seq.end());
            ans.push_back(seq);
            reverse(seq.begin(),seq.end());
            return;
        }
        int steps=mpp[word];
        int sz=word.size();
        for(int i=0;i<sz;i++){
            char original=word[i];
            for(char ch='a';ch<='z';ch++){
                word[i]=ch;
                if(mpp.find(word)!=mpp.end() && mpp[word]+1==steps){
                    seq.push_back(word);
                    dfs(word, seq);
                    seq.pop_back();
                }
            }
            word[i]=original;
        }
    }
public:
    vector<vector<string>> findLadders(string beginword,
                   string endword, vector<string>& wordlist) {
        unordered_set<string> st(wordlist.begin(),wordlist.end());
        queue<string> q;
        b=beginword;
        q.push({beginword});
        mpp[beginword]=1;
        int sz=beginword.size();
        st.erase(beginword);
        while(!q.empty()){
            string word=q.front();
            int steps=mpp[word];
            q.pop();
            if(word==endword)
            break;
            for(int i=0;i<sz;i++){
                char original=word[i];
                for(char ch='a';ch<='z';ch++){
                    word[i]=ch;
                    if(st.count(word)){
                        q.push(word);
                        st.erase(word);
                        mpp[word]=steps+1;
                    }
                }
                word[i]=original;
            }
        }
        if(mpp.find(endword)!=mpp.end()){
            vector<string> seq;
            seq.push_back(endword);
            dfs(endword, seq);
        }
        return ans;
    }
};
```

29.) DIJKSTRA'S ALGORITHM

(BY USING PRIORITY QUEUE)

```
class Solution
{
    public:
    //Function to find the shortest distance of all the vertices
    //from the source vertex S.
    vector <int> dijkstra(int V, vector<vector<int>> adj[], int S)
    {
        priority_queue<pair<int,int>, vector<pair<int,int>>,
greater<pair<int,int>>> pq;
        vector<int> dist(V);
        for(int i=0;i<V;i++){
            dist[i]=1e9;
        }

        dist[S]=0;
        pq.push({0,S});
        while(!pq.empty()){
            int dis=pq.top().first;
            int node=pq.top().second;
            pq.pop();

            for(auto it:adj[node]){
                int edgeWt=it[1];
                int adjNode=it[0];

                if(dis+edgeWt<dist[adjNode]){
                    dist[adjNode]=dis+edgeWt;
                    pq.push({dist[adjNode],adjNode});
                }
            }
        }
        return dist;
    }
};
```

30.) DIJKSTRA'S ALGORITHM

(BY USING SET)

```
class Solution
{
    public:
    //Function to find the shortest distance of all the vertices
    //from the source vertex S.
    vector <int> dijkstra(int V, vector<vector<int>> adj[], int S)
    {
        set<pair<int,int>> st;
        vector<int> dist(V, 1e9);
        st.insert({0, S});
        dist[S]=0;

        while(!st.empty()){
            auto it=*(st.begin());
            int node=it.second;
            int dis=it.first;
            st.erase(it);

            for(auto it:adj[node]){
                int adjNode=it[0];
                int edgeWt=it[1];

                if(dis+edgeWt<dist[adjNode]){
                    if(dist[adjNode]!=1e9){
                        st.erase({dist[adjNode], adjNode});
                    }
                    dist[adjNode]=dis+edgeWt;
                    st.insert({dist[adjNode],adjNode});
                }
            }
        }
        return dist;
    }
};
```

31..) PRINT SHORTEST PATH (DIJKSTRA'S ALGORITHM)

```
class Solution
{
public:
    vector<int> shortestPath(int n, int m, vector<vector<int>> &edges)
    {
        // Create an adjacency list of pairs of the form node1 -> {node2, edge
weight}
        // where the edge weight is the weight of the edge from node1 to
node2.
        vector<pair<int, int>> adj[n + 1];
        for (auto it : edges)
        {
            adj[it[0]].push_back({it[1], it[2]});
            adj[it[1]].push_back({it[0], it[2]});
        }
        // Create a priority queue for storing the nodes along with distances
        // in the form of a pair { dist, node }.
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int,int>>> pq;

        // Create a dist array for storing the updated distances and a parent
array
        //for storing the nodes from where the current nodes represented by
indices of
        // the parent array came from.
        vector<int> dist(n + 1, 1e9), parent(n + 1);
        for (int i = 1; i <= n; i++)
            parent[i] = i;

        dist[1] = 0;

        // Push the source node to the queue.
        pq.push({0, 1});
        while (!pq.empty())
        {
            // Topmost element of the priority queue is with minimum distance
value.
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int dis = it.first;

            // Iterate through the adjacent nodes of the current popped node.
            for (auto it : adj[node])
            {
                int adjNode = it.first;
                int edW = it.second;

                // Check if the previously stored distance value is
                // greater than the current computed value or not,
                // if yes then update the distance value.
                if (dis + edW < dist[adjNode])
                {
                    dist[adjNode] = dis + edW;
                    pq.push({dis + edW, adjNode});

                    // Update the parent of the adjNode to the recent
                    // node where it came from.
                    parent[adjNode] = node;
                }
            }
        }

        // If distance to a node could not be found, return an array
containing -1.
        if (dist[n] == 1e9)
            return {-1};

        // Store the final path in the ‘path’ array.
        vector<int> path;
        int node = n;

        // Iterate backwards from destination to source through the parent
array.
        while (parent[node] != node)
        {
            path.push_back(node);
            node = parent[node];
        }
        path.push_back(1);

        // Since the path stored is in a reverse order, we reverse the array
        // to get the final answer and then return the array.
        reverse(path.begin(), path.end());
        return path;
    }
};
```

32.) SHORTEST DISTANCE IN A BINARY MAZE

```cpp
class Solution {
  public:
    int shortestPath(vector<vector<int>> &grid,
                     pair<int, int> source, pair<int, int> destination) {
        if (source.first == destination.first
            && source.second == destination.second)
            return 0;
        queue<pair<int,pair<int,int>>> q;
        int n=grid.size();
        int m=grid[0].size();

        vector<vector<int>> dist(n, vector<int> (m, 1e9));
        dist[source.first][source.second]=0;
        q.push({0, {source.first, source.second}});

        vector<int> drow={-1,0,1,0};
        vector<int> dcol={0,1,0,-1};

        while(!q.empty()){
            auto it=q.front();
            q.pop();
            int dis=it.first;
            int r=it.second.first;
            int c=it.second.second;

            for(int i=0;i<4;i++){
                int nrow=r+drow[i];
                int ncol=c+dcol[i];

                if(nrow>=0 && nrow<n && ncol>=0 && ncol<m
                   && grid[nrow][ncol]==1 && dis+1<dist[nrow][ncol]){
                    dist[nrow][ncol]=1+dis;
                    if(nrow==destination.first && ncol==destination.second){
                        return dis+1;
                    }
                    q.push({1+dis, {nrow,ncol}});
                }
            }
        }
        return -1;
    }
};
```

33.) PATH WITH MINIMUM EFFORTS

```
class Solution {
public:
    int minimumEffortPath(vector<vector<int>>& heights) {
        priority_queue<pair<int,pair<int,int>>,vector<pair<int,pair<int,int>>>
,greater<pair<int,pair<int,int>>>> pq;

        int n=heights.size();
        int m=heights[0].size();

        vector<vector<int>> dist(n, vector<int> (m,1e9));
        dist[0][0]=0;
        pq.push({0,{0,0}});

        vector<int> dr={-1,0,1,0};
        vector<int> dc={0,1,0,-1};

        while(!pq.empty()){
            auto it=pq.top();
            pq.pop();

            int diff=it.first;
            int row=it.second.first;
            int col=it.second.second;

            if(row==n-1 && col==m-1)
            return diff;

            for(int i=0;i<4;i++){
                int newr=row+dr[i];
                int newc=col+dc[i];
                if(newr>=0 && newr<n && newc>=0 && newc<m){
                    int neweffort=max(abs(heights[row][col]-
heights[newr][newc]),diff);
                    if(neweffort<dist[newr][newc]){
                        dist[newr][newc]=neweffort;
                        pq.push({dist[newr][newc],{newr,newc}});
                    }
                }
            }
        }
        return 0;
    }
};

// TC ~ O(Elog(V)) -- dijkstra time complexity
// SC ~ O(N*M)
```

34.) CHEAPEST FLIGHTS WITHIN K STOPS

```
class Solution {
public:
    int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int
dst, int k) {
        vector<pair<int,int>> adj[n];
        for(auto it:flights){
            adj[it[0]].push_back({it[1],it[2]});
        }

        queue<pair<int,pair<int,int>>> q;
        // {stops, {node, dist}};
        q.push({0,{src,0}});

        vector<int> dist(n, 1e9);
        dist[src]=0;

        while(!q.empty()){
            auto it=q.front();
            q.pop();

            int stops=it.first;
            int node=it.second.first;
            int cost=it.second.second;

            if(stops>k)
            continue;

            for(auto iter:adj[node]){
                int adjnode=iter.first;
                int edgeweight=iter.second;
                if(cost+edgeweight<dist[adjnode] && stops<=k){
                    dist[adjnode]=cost+edgeweight;
                    q.push({stops+1,{adjnode,cost+edgeweight}});
                }
            }
        }
        if(dist[dst]==1e9)
        return -1;
        return dist[dst];
    }
};

// TC ~ O(total no of edges = flights.size())
```

35.) MINIMUM MULTIPLICATIONS TO REACH END

```
class Solution {

  public:
    int minimumMultiplications(vector<int>& arr, int start, int end) {
        queue<pair<int,int>>q;
        q.push({0,start});
        vector<int>dis(100000,1e9);
        dis[start]=0;
        while(!q.empty()){
            int steps=q.front().first;
            int val=q.front().second;
            q.pop();

            if(val==end)return steps;

            for(auto it:arr){
               int num=(it*val)%100000;
               if(steps+1<dis[num]){
                   dis[num]=steps+1;
                   q.push({steps+1,num});
               }
            }
        }
        return -1;
    }
};
```

36.) NUMBER OF WAYS TO REACH AT DESTINATION

```cpp
class Solution {

public:
    int countPaths(int n, vector<vector<int>>& roads) {
        vector<pair<long long int,long long int>> adj[n];
        for(auto it:roads){
            adj[it[0]].push_back({it[1],it[2]});
            adj[it[1]].push_back({it[0],it[2]});
        }
        priority_queue<pair<long long int,long long int>,
                       vector<pair<long long int,long long int>>,
                       greater<pair<long long int,long long int>>> pq;

        vector<long long int> dist(n,1e15);
        vector<long long int> ways(n,0);

        dist[0]=0;
        ways[0]=1;
        pq.push({0,0});
        int mod=(int)(1e9+7);

        while(!pq.empty()){
            long long int dis=pq.top().first;
            long long int node=pq.top().second;
            pq.pop();

            for(auto it:adj[node]){
                long long int adjnode=it.first;
                long long int edgewt=it.second;
                if(dis+edgewt<dist[adjnode]){
                    dist[adjnode]=dis+edgewt;
                    pq.push({dis+edgewt,adjnode});
                    ways[adjnode]=ways[node];
                }
                else if(dis+edgewt==dist[adjnode]){
                    ways[adjnode]=(ways[adjnode]+ways[node])%mod;
                }
            }
        }
        return ways[n-1]%mod;
    }
};

// TC ~ O(E log(V))
// SC ~ extra O(N)
```

37.) BELLMAN FORD ALGORITHM

```
class Solution {
  public:
    /*  Function to implement Bellman Ford
    *   edges: vector of vectors which represents the graph
    *   S: source vertex to start traversing graph with
    *   V: number of vertices
    */
    vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S) {
        vector<int> dist(V, 1e8);
        dist[S]=0;

        for(int i=0;i<V-1;i++){
            for(auto it:edges){
                int u=it[0];
                int v=it[1];
                int wt=it[2];
                if(dist[u]!=1e8 && dist[u]+wt<dist[v]){
                    dist[v]=dist[u]+wt;
                }
            }
        }

        for(auto it:edges){
            int u=it[0];
            int v=it[1];
            int wt=it[2];
            if(dist[u]!=1e8 && dist[u]+wt<dist[v]){
                return {-1};
            }
        }
        return dist;
    }
};
```

38.) FLOYD WARSHALL ALGORITHM

```
class Solution {
  public:
    void shortest_distance(vector<vector<int>>&matrix){
        int n=matrix.size();

        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                if(matrix[i][j]==-1){
                    matrix[i][j]=1e9;
                }
                if(i==j)
                matrix[i][j]=0;
            }
        }

        for(int k=0;k<n;k++){
            for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                    matrix[i][j]=min(matrix[i][j],
                                    matrix[i][k]+matrix[k][j]);
                }
            }
        }

        // FOR HANDLING -ve CYCLE
        // for(int i=0;i<n;i++){

        //      if(matrix[i][i]<0){
        //          // SOMETHING
        //      }
        // }

        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                if(matrix[i][j]==1e9){
                    matrix[i][j]=-1;
                }
            }
        }
    }
};
```

39.) FIND THE CITY WITH THE SMALLEST NUMBER OF NEIGHBOURS AT A THRESHOLD DISTANCE

```
class Solution {
public:
    int findTheCity(int n, vector<vector<int>>& edges, int distanceThreshold)
{
        vector<vector<int>> dist(n, vector<int> (n, INT_MAX));

        for(auto it:edges){
            dist[it[0]][it[1]]=it[2];
            dist[it[1]][it[0]]=it[2];
        }

        for(int i=0;i<n;i++)
        dist[i][i]=0;

        for(int k=0;k<n;k++){
            for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                    if(dist[i][k]==INT_MAX || dist[k][j]==INT_MAX)
                    continue;
                    dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j]);
                }
            }
        }

        int cntCity=n;
        int cityNo=-1;
        for(int city=0;city<n;city++){
            int cnt=0;

            for(int adjCity=0;adjCity<n;adjCity++){
                if(dist[city][adjCity]<=distanceThreshold)
                cnt++;
            }

            if(cnt<=cntCity){
                cntCity=cnt;
                cityNo=city;
            }
        }
        return cityNo;
    }
};
```

40.) PRIM'S ALGORITHM

```cpp
class Solution
{
    public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        priority_queue<pair<int,int>,
                       vector<pair<int,int>>,
                       greater<pair<int,int>>> pq;
        vector<int> vis(V,0);
        pq.push({0,0});
        int sum=0;

        while(!pq.empty()){
            auto it=pq.top();
            pq.pop();

            int node=it.second;
            int wt=it.first;

            if(vis[node]==1)
            continue;

            vis[node]=1;
            sum=sum+wt;

            for(auto it:adj[node]){
                int adjNode=it[0];
                int edWt=it[1];
                if(!vis[adjNode]){
                    pq.push({edWt, adjNode});
                }
            }
        }
        return sum;
    }
};
```

41.) DISJOINT SET | UNION BY SIZE | UNION BY RANK

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

    // #ifndef ONLINE_JUDGE
    // freopen("input1.txt","r",stdin);
    // freopen("output1.txt","w",stdout);
    // #endif

class DisjointSet{
    vector<int> rank,parent,size;
public:
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }
    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};


int main() {

    #ifndef ONLINE_JUDGE
    freopen("input1.txt","r",stdin);
    freopen("output1.txt","w",stdout);
    #endif

    DisjointSet ds(7);
    ds.unionBySize(1,2);
    ds.unionBySize(2,3);
    ds.unionBySize(4,5);
    ds.unionBySize(6,7);
    ds.unionBySize(5,6);
    if(ds.findPar(3)==ds.findPar(7)){
        cout<<"same\n";
    }
    else{
        cout<<"Not same\n";
    }
    ds.unionBySize(3,7);
    if(ds.findPar(3)==ds.findPar(7)){
        cout<<"same\n";
    }
    else{
        cout<<"Not same\n";
    }
    return 0;
}
```

42.) KRUSKAL’S ALGORITHM

```
class DisjointSet{
    vector<int> rank,parent,size;
public:
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }

    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
```

```cpp
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution
{
    public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        vector<pair<int,pair<int,int>>> edges;
        for(int i=0;i<V;i++){
            for(auto it:adj[i]){
                int adjNode=it[0];
                int wt=it[1];
                int node=i;

                edges.push_back({wt,{node,adjNode}});

            }
        }

        DisjointSet ds(V);

        sort(edges.begin(),edges.end());
        int mstWt=0;

        for(auto it:edges){
            int wt=it.first;
            int u=it.second.first;
            int v=it.second.second;

            if(ds.findPar(u)!=ds.findPar(v)){
                mstWt+=wt;
                ds.unionBySize(u, v);
            }
        }
        return mstWt;
    }
};
```

43.) NUMBER OF PROVINCES (DISJOINT SET DATA STRUCTURE)

```
class DisjointSet{
public:
    vector<int> rank,parent,size;
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }
    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution {
public:
    int findCircleNum(vector<vector<int>>& adj) {
        int V=adj.size();
        DisjointSet ds(V);
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                if(adj[i][j]==1){
                    ds.unionBySize(i, j);
                }
            }
        }
        int count=0;
        for(int i=0;i<V;i++){
            if(ds.parent[i]==i)
            count++;
        }
        return count;
    }
};
```

44.) NUMBER OF OPERATIONS TO MAKE NETWORK CONNECTED

```
class DisjointSet{
public:
    vector<int> rank,parent,size;
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }
    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution {
public:
    int makeConnected(int n, vector<vector<int>>& edge) {
        DisjointSet ds(n);
        int countExtra=0;

        for(auto it:edge){
            int u=it[0];
            int v=it[1];
            if(ds.findPar(u)==ds.findPar(v)){
                countExtra++;
            }
            else{
                ds.unionBySize(u,v);
            }
        }

        int countC=0;
        for(int i=0;i<n;i++){
            if(ds.parent[i]==i)
            countC++;
        }

        int ans=countC-1;
        if(countExtra>=ans)
        return ans;
        return -1;
    }
};
```

45.) ACCOUNTS MERGE

```
class DisjointSet{
public:
    vector<int> rank,parent,size;
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }
    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>>& account) {
        int n=account.size();
        DisjointSet ds(n);
        unordered_map<string,int> mpp;
        sort(account.begin(),account.end());

        for(int i=0;i<n;i++){
            for(int j=1;j<account[i].size();j++){
                string mail=account[i][j];
                if(mpp.find(mail)==mpp.end()){
                    mpp[mail]=i;
                }
                else{
                    ds.unionBySize(i, mpp[mail]);
                }
            }
        }

        vector<string> mergeMail[n];
        for(auto it:mpp){
            string mail=it.first;
            int node=ds.findPar(it.second);
            mergeMail[node].push_back(mail);
        }

        vector<vector<string>> ans;
        for(int i=0;i<n;i++){
            if(mergeMail[i].size()==0)
            continue;
            sort(mergeMail[i].begin(),mergeMail[i].end());
            vector<string> temp;
            temp.push_back(account[i][0]);
            for(auto it:mergeMail[i]){
                temp.push_back(it);
            }
            ans.push_back(temp);
        }
        return ans;
    }
};
```

46.) NUMBER OF ISLANDS II

```
// User function Template for C++

class DisjointSet{
public:
    vector<int> rank,parent,size;
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }

    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution {
  public:
    vector<int> numOfIslands(int n, int m, vector<vector<int>> &operators) {
        DisjointSet ds(n*m);
        vector<vector<int>> vis(n, vector<int> (m,0));
        int count=0;
        vector<int> ans;

        for(auto it:operators){
            int row=it[0];
            int col=it[1];
            if(vis[row][col]==1){
                ans.push_back(count);
                continue;
            }
            vis[row][col]=1;
            count++;

            vector<int> dr={-1,0,1,0};
            vector<int> dc={0,1,0,-1};

            for(int ind=0;ind<4;ind++){
                int adjRow=row+dr[ind];
                int adjCol=col+dc[ind];
                if(adjRow>=0 && adjRow<n && adjCol>=0 && adjCol<m){
                    if(vis[adjRow][adjCol]==1){
                        int nodeNo=row*m+col;
                        int adjNodeNo=adjRow*m+adjCol;
                        if(ds.findPar(nodeNo)!=ds.findPar(adjNodeNo)){
                            count--;
                            ds.unionBySize(nodeNo, adjNodeNo);
                        }
                    }
                }
            }
            ans.push_back(count);
        }
        return ans;
    }
};
```

47.) MAKING A LARGE ISLANDS

```
class DisjointSet{
public:
    vector<int> rank,parent,size;
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }

    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution {
public:
    int largestIsland(vector<vector<int>>& grid) {
        int n=grid.size();
        DisjointSet ds(n*n);

        for(int row=0;row<n;row++){
            for(int col=0;col<n;col++){
                if(grid[row][col]==0)
                continue;

                vector<int> dr={-1,0,1,0};
                vector<int> dc={0,-1,0,1};

                for(int ind=0;ind<4;ind++){
                    int newr=row+dr[ind];
                    int newc=col+dc[ind];
                    if((newr>=0 && newr<n && newc>=0 && newc<n)
                        && grid[newr][newc]==1){
                        int nodeNo=row*n+col;
                        int adjNodeNo=newr*n+newc;
                        ds.unionBySize(nodeNo, adjNodeNo);
                    }
                }
            }
        }

        int mx=0;
        for(int row=0;row<n;row++){
            for(int col=0;col<n;col++){
                if(grid[row][col]==1)
                continue;

                vector<int> dr={-1,0,1,0};
                vector<int> dc={0,-1,0,1};
                set<int> components;

                for(int ind=0;ind<4;ind++){
                    int newr=row+dr[ind];
                    int newc=col+dc[ind];
                    if(newr>=0 && newr<n && newc>=0 && newc<n){
                        if(grid[newr][newc]==1){
                            components.insert(ds.findPar(newr*n+newc));
                        }
                    }
                }
                int sizeTotal=0;
                for(auto it:components){
                    sizeTotal+=ds.size[it];
                }
                mx=max(mx,sizeTotal+1);
            }
        }

        for(int cellNo=0;cellNo<n*n;cellNo++){
            mx=max(mx, ds.size[ds.findPar(cellNo)]);
        }
        return mx;
    }
};
```

48.) MOST STONES REMOVED WITH SAME ROW OR COLUMN

```
class DisjointSet{
public:
    vector<int> rank,parent,size;
    DisjointSet(int n){
        rank.resize(n+1, 0);
        parent.resize(n+1);
        size.resize(n+1);
        for(int i=0;i<=n;i++){
            parent[i]=i;
            size[i]=1;
        }
    }

    int findPar(int node){
        if(node==parent[node])
            return node;
        return parent[node]=findPar(parent[node]);
    }

    void unionByRank(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(rank[ulp_u]<rank[ulp_v]){
            parent[ulp_u]=ulp_v;
        }
        else if(rank[ulp_v]<rank[ulp_u]){
            parent[ulp_v]=ulp_u;
        }
        else{
            parent[ulp_v]=ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v){
        int ulp_u=findPar(u);
        int ulp_v=findPar(v);
        if(ulp_u==ulp_v)
            return;
        if(size[ulp_u]<size[ulp_v]){
            parent[ulp_u]=ulp_v;
            size[ulp_v]+=size[ulp_u];
        }
        else{
            parent[ulp_v]=ulp_u;
            size[ulp_u]+=size[ulp_v];
        }
    }
};

class Solution {
public:
    int removeStones(vector<vector<int>>& stones) {
        int n=stones.size();
        int maxRow=0;
        int maxCol=0;

        for(auto it:stones){
            maxRow=max(maxRow, it[0]);
            maxCol=max(maxCol, it[1]);
        }

        DisjointSet ds(maxRow+maxCol+1);
        unordered_map<int,int> mpp;

        for(auto it:stones){
            int nodeRow=it[0];
            int nodeCol=it[1]+maxRow+1;
            ds.unionBySize(nodeRow, nodeCol);
            mpp[nodeRow]=1;
            mpp[nodeCol]=1;
        }

        int count=0;
        for(auto it:mpp){
            if(ds.findPar(it.first)==it.first){
                count++;
            }
        }
        return n-count;
    }
};
```

49.) STRONGLY CONNECTED COMPONENTS
(KOSARAJU’S ALGORITHM)

```
class Solution
{
    public:
    void dfs(int node,vector<bool>
             &visited,vector<vector<int>>& adj,stack<int> &st){
        visited[node] = true;
        for(auto neigh:adj[node]){
            if(visited[neigh] == false){
                dfs(neigh,visited,adj,st);
            }
        }
        st.push(node);
    }

    void revDFS(int node,vector<bool>
                &visited,vector<vector<int>> &transpose){
        visited[node] = true;
        for(auto neigh:transpose[node]){
            if(!visited[neigh])
                revDFS(neigh,visited,transpose);
        }
    }


    int kosaraju(int V, vector<vector<int>>& adj){
        vector<bool> visited(V,false);
        stack<int> st;

        for(int i=0;i<V;i++{
           if(!visited[i]){
               dfs(i,visited,adj,st);
           }
        }

        vector<vector<int>> transpose(V);

        for(int i=0;i<V;i++){
            visited[i] = false;
            for(auto neigh:adj[i]){
                transpose[neigh].push_back(i);
            }
        }

        int count =0;
        while(!st.empty()){
            int top = st.top();
            st.pop();

           if(visited[top] == false){
               count++;
                revDFS(top,visited,transpose);
           }
        }
        return count;
    }
};
```

50.) BRIDGES IN GRAPH – TARJAN'S ALGORITHM OF TIME IN AND LOW TIME

(CRITICAL CONNECTIONS IN A NETWORK)

```
class Solution {
public:
int timer=1;
    void dfs(int node, int parent, vector<int> &vis,
             vector<int> adj[], vector<int> &tin,
             vector<int> &low, vector<vector<int>> &bridges){
        vis[node]=1;
        tin[node]=low[node]=timer;
        timer++;
        for(auto it:adj[node]){
            if(it==parent)
            continue;
            if(vis[it]==0){
                dfs(it, node, vis, adj, tin, low, bridges);
                low[node]=min(low[node], low[it]);
                // node -- it
                if(low[it]>tin[node]){
                    bridges.push_back({it, node});
                }
            }
            else{
                low[node]=min(low[node], low[it]);
            }
        }
    }
    vector<vector<int>> criticalConnections(int n,
                                    vector<vector<int>>& connections) {
        vector<int> adj[n];
        for(auto it:connections){
            adj[it[0]].push_back(it[1]);
            adj[it[1]].push_back(it[0]);
        }

        vector<int> vis(n, 0);
        vector<int> tin(n);
        vector<int> low(n);
        vector<vector<int>> bridges;

        dfs(0, -1, vis, adj, tin, low, bridges);
        return bridges;
    }
};
```

51.) ARTICULATION POINTS

```cpp
class Solution {
  public:
    int timer=0;
    void dfs(int node, int parent, vector<int> &vis,
             vector<int> &tin, vector<int> &low,
             vector<int> &mark, vector<int> adj[]){
        vis[node]=1;
        tin[node]=low[node]=timer;
        timer++;
        int child=0;

        for(auto it:adj[node]){
            if(it==parent)
            continue;
            if(!vis[it]){
                dfs(it, node, vis, tin, low, mark, adj);
                low[node]=min(low[node], low[it]);
                if(low[it]>=tin[node] && parent!=-1){
                    mark[node]=1;
                }
                child++;
            }
            else{
                low[node]=min(low[node], tin[it]);
            }
            if(child>1 && parent==-1){
                mark[node]=1;
            }
        }
    }

    vector<int> articulationPoints(int n, vector<int>adj[]) {
        vector<int> vis(n, 0);
        vector<int> tin(n);
        vector<int> low(n);
        vector<int> mark(n, 0);

        for(int i=0;i<n;i++){
            if(!vis[i]){
                dfs(i, -1, vis, tin, low, mark, adj);
            }
        }

        vector<int> ans;
        for(int i=0;i<n;i++){
            if(mark[i]==1){
                ans.push_back(i);
            }
        }
        if(ans.size()==0)
        return {-1};
        return ans;
    }
};
```

THANK YOU !