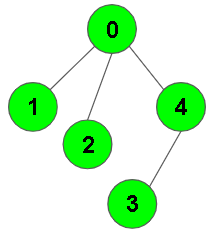
**############################################################**

**1. DFS of Graph**

Given a connected undirected graph. Perform a Depth First Traversal of the graph.  
**Note:**Use recursive approach to find the DFS traversal of the graph starting from the 0th vertex from left to right according to the graph..

**Example 1:**

**Input:**



**Output:** 0 1 2 4 3

**Explanation**:

0 is connected to 1, 2, 4.

1 is connected to 0.

2 is connected to 0.

3 is connected to 0.

4 is connected to 0, 3.

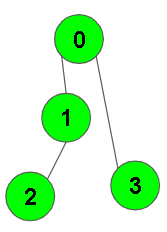
so starting from 0, it will go to 1 then 2

then 4, and then from 4 to 3.

Thus dfs will be 0 1 2 4 3.

**Example 2:**

**Input:**



**Output:** 0 1 2 3

**Explanation**:

0 is connected to 1 , 3.

1 is connected to 2.

2 is connected to 1.

3 is connected to 0.

so starting from 0, it will go to 1 then 2

then back to 0 then 0 to 3

thus dfs will be 0 1 2 3.

**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **dfsOfGraph()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns  a list containing the DFS traversal of the graph starting from the 0th vertex from left to right according to the graph.

**Expected Time Complexity:**O(V + E)  
**Expected Auxiliary Space:**O(V)

**Constraints:**  
1 ≤ V, E ≤ 104

void dfs(int node, vector<int> adj[], vector<int> &ans, vector<bool> &vis)

{ vis[node] = true;

ans.push\_back(node);

for(auto i: adj[node])

if(vis[i] == false)

dfs(i, adj, ans, vis);

}

vector<int>dfsOfGraph(int v, vector<int> adj[])

{ vector<int> ans;

vector<bool> vis(v, false);

dfs(0, adj, ans, vis);

return ans;

}

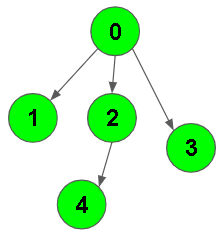
**############################################################**

**2. BFS of graph**

Given a directed graph. The task is to do Breadth First Traversal of this graph starting from 0.  
**Note:**One can move from node u to node v only if there's an edge from u to v and find the BFS traversal of the graph starting from the 0th vertex, from left to right according to the graph. Also, you should only take nodes directly or indirectly connected from Node 0 in consideration.

**Example 1:**

**Input:**



**Output:** 0 1 2 3 4

**Explanation**:

0 is connected to 1 , 2 , 3.

2 is connected to 4.

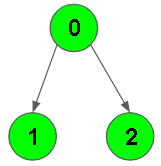
so starting from 0, it will go to 1 then 2

then 3.After this 2 to 4, thus bfs will be

0 1 2 3 4.

**Example 2:**

**Input:**



**Output:** 0 1 2

**Explanation**:

0 is connected to 1 , 2.

so starting from 0, it will go to 1 then 2,

thus bfs will be 0 1 2 3 4.

**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **bfsOfGraph()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns  a list containing the BFS traversal of the graph starting from the 0th vertex from left to right.

**Expected Time Complexity:**O(V + E)  
**Expected Auxiliary Space:**O(V)

**Constraints:**  
1 ≤ V, E ≤ 104

vector<int>bfsOfGraph(int v, vector<int> adj[])

{ vector<int> ans;

vector<bool> vis(v, false);

queue<int> q;

q.push(0);

vis[0] = true;

while(!q.empty())

{ int node = q.front();

q.pop();

ans.push\_back(node);

for(auto i: adj[node])

if(vis[i] == false)

{ vis[i] = true;

q.push(i); }

}

return ans;

}

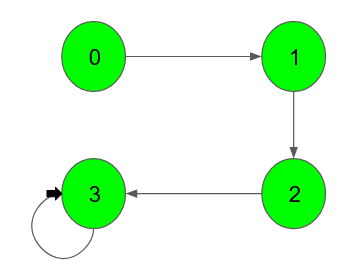
**############################################################**

**3. Detect cycle in a directed graph**

Given a Directed Graph with **V** vertices (Numbered from **0** to **V-1**) and **E** edges, check whether it contains any cycle or not.

**Example 1:**

**Input:**

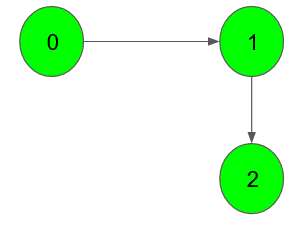


**Output:** 1

**Explanation**: 3 -> 3 is a cycle

**Example 2:**

**Input:**



**Output:** 0

**Explanation**: no cycle in the graph

**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **isCyclic()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns a boolean value denoting if the given directed graph contains a cycle or not.

**Expected Time Complexity:**O(V + E)  
**Expected Auxiliary Space:**O(V)

**Constraints:**  
1 ≤ V, E ≤ 105

bool check(vector<int> adj[], int node, vector<bool> &v\_start, vector<bool> &v\_sub)

{ v\_sub[node] = true;

for(int i: adj[node])

if(v\_start[i] == false)

if(v\_sub[i] || check(adj, i, v\_start, v\_sub))

return true;

v\_start[node] = true;

v\_sub[node] = false;

return false;

}

bool isCyclic(int v, vector<int> adj[])

{ vector<bool> v\_start(v, false), v\_sub(v, false);

for(int i=0; i<v; i++)

if(v\_start[i] == false)

if(check(adj, i, v\_start, v\_sub))

return true;

return false;

}

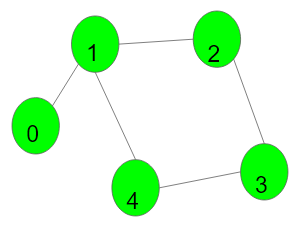
**############################################################**

**4. Detect cycle in an undirected graph**

Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

**Example 1:**

**Input:**

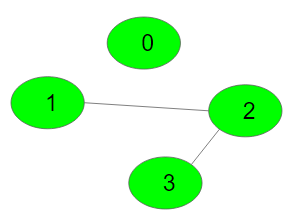


**Output:** 1

**Explanation:** 1->2->3->4->1 is a cycle.

**Example 2:**

**Input:**



**Output:** 0

**Explanation:** No cycle in the graph.

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **isCycle()**which takes V denoting the number of vertices and adjacency list as input parameters and returns a boolean value denoting if the undirected graph contains any cycle or not.

**Expected Time Complexity:**O(V + E)  
**Expected Space Complexity:**O(V)

**Constraints:**  
1 ≤ V, E ≤ 105

bool dfs(vector<int> adj[], int node, int par, vector<bool> &vis)

{ vis[node] = true;

for(auto i: adj[node])

{ if(vis[i] == false)

{ if(dfs(adj, i, node, vis))

return true;

}

else if(par != i)

return true; }

return false;

}

bool isCycle(int v, vector<int>adj[])

{ vector<bool> vis(v, false);

for(int i=0; i<v; i++)

if(vis[i] == false)

{ if(dfs(adj, i, -1, vis))

return true;

}

return false;

}

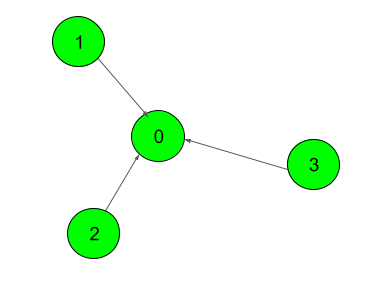
**############################################################**

**5. Topological sort**

Given a Directed Graph with V vertices and E edges, Find any Topological Sorting of that Graph.

**Example 1:**

**Input:**



**Output:**

1

**Explanation**:

The output 1 denotes that the order is

valid. So, if you have, implemented

your function correctly, then output

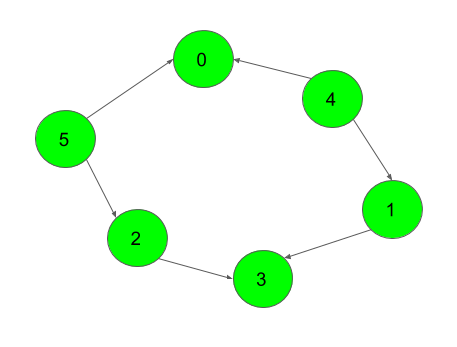
would be 1 for all test cases.

One possible Topological order for the

graph is 3, 2, 1, 0.

**Example 2:**

**Input:**



**Output:**

1

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **topoSort()**  which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns an array consisting of a the vertices in Topological order. As there are multiple Topological orders possible, you may return any of them.

**Expected Time Complexity:** O(V + E).  
**Expected Auxiliary Space:** O(V).

**Constraints:**  
2 ≤ V ≤ 104  
1 ≤ E ≤ (N\*(N-1))/2

vector<int> topoSort(int v, vector<int> adj[])

{ vector<int> deg(v,0);

vector<int> ans;

queue<int> q;

for(int i=0; i<v; i++)

for(int j: adj[i])

deg[j]++;

for(int i=0; i<v; i++)

if(deg[i] == 0)

q.push(i);

while(!q.empty())

{ int node = q.front();

q.pop();

ans.push\_back(node);

for(int i: adj[node])

{ deg[i]--;

if(deg[i] == 0)

q.push(i);

}

}

return ans;

}

**############################################################**

**6. Find the number of islands**

Given a grid consisting of '0's(Water) and '1's(Land). Find the number of islands.  
**Note:**An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

**Example 1:**

**Input:**

grid = {{0,1},{1,0},{1,1},{1,0}}

**Output:**

1

**Explanation:**

The grid is-

0 1

1 0

1 1

1 0

All lands are connected.

**Example 2:**

**Input:**

grid = {{0,1,1,1,0,0,0},{0,0,1,1,0,1,0}}

**Output:**

2

**Expanation:**

The grid is-

0 1 1 1 0 0 0

0 0 1 1 0 1 0

There are two islands one is colored in blue

and other in orange.

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **numIslands()**which takes grid as input parameter and returns the total number of islands.

**Expected Time Compelxity:**O(n\*m)  
**Expected Space Compelxity:**O(n\*m)

**Constraints:**  
1 ≤ n, m ≤ 500

void dfs(vector<vector<char>>& grid, vector<vector<bool>> &vis, int i, int j, int n, int m)

{ if(i < 0 || i >= n || j < 0 || j >= m)

return;

if(grid[i][j] == '0')

return;

if(vis[i][j] == false)

{ vis[i][j] = true;

dfs(grid, vis, i+1 ,j ,n, m);

dfs(grid, vis, i-1 ,j ,n, m);

dfs(grid, vis, i ,j+1 ,n, m);

dfs(grid, vis, i ,j-1 ,n, m);

dfs(grid, vis, i+1 ,j+1 ,n, m);

dfs(grid, vis, i-1 ,j-1 ,n, m);

dfs(grid, vis, i+1 ,j-1 ,n, m);

dfs(grid, vis, i-1 ,j+1 ,n, m);

}

}

int numIslands(vector<vector<char>>& grid)

{ int n = grid.size();

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

int count = 0;

vector<vector<bool>> vis(n, vector<bool>(m,false));

for(int i=0; i<n; i++)

{ for(int j=0; j<m; j++)

{ if(vis[i][j] == false && grid[i][j] == '1')

{ dfs(grid, vis, i ,j ,n, m);

count++;

}

}

}

return count;

}

**############################################################**

**7. Unit Area of largest region of 1's**

Given a grid of dimension **nxm**containing 0s and 1s. Find the unit area of the largest region of 1s.  
Region of 1's is a group of 1's connected 8-directionally(horizontally, vertically, dioganally).

**Example 1:**

**Input:** grid = {{1,1,1,0},{0,0,1,0},{0,0,0,1}}

**Output:** 5

**Explanation:** The grid is-

1 1 1 0

0 0 1 0

0 0 0 1

The largest region of 1's is colored

in orange.

**Example 2:**

**Input:** grid = {{0,1}}

**Output:** 1

**Explanation:** The grid is-

0 1

The largest region of 1's is colored in

orange.

**Your Task:**  
You don't need to read or print anyhting. Your task is to complete the function **findMaxArea()**which takes grid as input parameter and returns the area of the largest region of 1's.

**Expected Time Complexity:**O(n\*m)  
**Expected Auxiliary Space:**O(n\*m)

**Constraints:**  
1 ≤ n, m ≤ 500

void dfs(vector<vector<int>>& grid, vector<vector<bool>> &vis, int i, int j, int n, int m, int &sum)

{ if(i < 0 || i >= n || j < 0 || j >= m)

return;

if(grid[i][j] == 0)

return;

if(vis[i][j] == false)

{ vis[i][j] = true;

sum++;

dfs(grid, vis, i+1 ,j ,n, m, sum);

dfs(grid, vis, i-1 ,j ,n, m, sum);

dfs(grid, vis, i ,j+1 ,n, m, sum);

dfs(grid, vis, i ,j-1 ,n, m, sum);

dfs(grid, vis, i+1 ,j+1 ,n, m, sum);

dfs(grid, vis, i-1 ,j-1 ,n, m, sum);

dfs(grid, vis, i+1 ,j-1 ,n, m, sum);

dfs(grid, vis, i-1 ,j+1 ,n, m, sum);

}

}

int findMaxArea(vector<vector<int>>& grid) {

int n = grid.size();

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

int count = INT\_MIN;

vector<vector<bool>> vis(n, vector<bool>(m,false));

for(int i=0; i<n; i++)

{ for(int j=0; j<m; j++)

{ if(vis[i][j] == false && grid[i][j] == 1)

{ int sum = 0;

dfs(grid, vis, i ,j ,n, m, sum);

count = max(count, sum);

}

}

}

return count;

}

**############################################################**

**8. Steps by Knight**

Given a square chessboard, the initial position of Knight and position of a target. Find out the minimum steps a Knight will take to reach the target position.

**Note:**  
The initial and the target position co-ordinates of Knight have been given accoring to 1-base indexing.

**Example 1:**

**Input:**

N=6

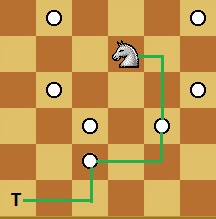
knightPos[ ] = {4, 5}

targetPos[ ] = {1, 1}

**Output:**

3

**Explanation:**



Knight takes 3 step to reach from

(4, 5) to (1, 1):

(4, 5) -> (5, 3) -> (3, 2) -> (1, 1).

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **minStepToReachTarget()** which takes the inital position of Knight (KnightPos), the target position of Knight (TargetPos) and the size of the chess board (N) as an input parameters and returns the minimum number of steps required by the knight to reach from its current position to the given target position.

**Expected Time Complexity:** O(N2).  
**Expected Auxiliary Space:** O(N2).

**Constraints:**  
1 <= N <= 1000  
1 <= Knight\_pos(X, Y), Targer\_pos(X, Y) <= N

int minStepToReachTarget(vector<int>&start, vector<int>&end,int n)

{ if(start[0] == end[0] && start[1] == end[1])

return 0;

int board[n][n];

for(int i = 0; i < n; i++)

for(int j = 0; j < n; j++)

board[i][j] = 0;

queue<pair<int, int>> q;

q.push(make\_pair(start[0] - 1, start[1] - 1));

while(!q.empty())

{ pair<int, int> cur = q.front();

int x = cur.first;

int y = cur.second;

q.pop();

if((x-1 >= 0 && x-1 < n) && (y-2 >= 0 && y-2 < n) && (board[x-1][y-2] == 0))

{ q.push(make\_pair(x-1, y-2));

board[x-1][y-2] = board[x][y] + 1;

}

if((x+1 >= 0 && x+1 < n) && (y-2 >= 0 && y-2 < n) && (board[x+1][y-2] == 0))

{ q.push(make\_pair(x+1, y-2));

board[x+1][y-2] = board[x][y] + 1;

}

if((x-2 >= 0 && x-2 < n) && (y-1 >= 0 && y-1 < n) && (board[x-2][y-1] == 0))

{ q.push(make\_pair(x-2, y-1));

board[x-2][y-1] = board[x][y] + 1;

}

if((x+2 >= 0 && x+2 < n) && (y-1 >= 0 && y-1 < n) && (board[x+2][y-1] == 0))

{ q.push(make\_pair(x+2, y-1));

board[x+2][y-1] = board[x][y] + 1;

}

if((x-2 >= 0 && x-2 < n) && (y+1 >= 0 && y+1 < n) && (board[x-2][y+1] == 0))

{ q.push(make\_pair(x-2, y+1));

board[x-2][y+1] = board[x][y] + 1;

}

if((x+2 >= 0 && x+2 < n) && (y+1 >= 0 && y+1 < n) && (board[x+2][y+1] == 0))

{ q.push(make\_pair(x+2, y+1));

board[x+2][y+1] = board[x][y] + 1;

}

if((x-1 >= 0 && x-1 < n) && (y+2 >= 0 && y+2 < n) && (board[x-1][y+2] == 0))

{ q.push(make\_pair(x-1, y+2));

board[x-1][y+2] = board[x][y] + 1;

}

if((x+1 >= 0 && x+1 < n) && (y+2 >= 0 && y+2 < n) && (board[x+1][y+2] == 0))

{ q.push(make\_pair(x+1, y+2));

board[x+1][y+2] = board[x][y] + 1;

}

}

return board[end[0]-1][end[1]-1];

}

**############################################################**

**9. Word Boggle**

Given a dictionary of distinct words and an M x N board where every cell has one character. Find all possible words from the dictionary that can be formed by a sequence of adjacent characters on the board. We can move to any of 8 adjacent characters, but a word should not have multiple instances of the same cell.

**Example 1:**

**Input:**

N = 1

dictionary = {"CAT"}

R = 3, C = 3

board = {{C,A,P},{A,N,D},{T,I,E}}

**Output:**

CAT

**Explanation**:

C A P

A N D

T I E

Words we got is denoted using same color.

**Example 2:**

**Input:**

N = 4

dictionary = {"GEEKS","FOR","QUIZ","GO"}

R = 3, C = 3

board = {{G,I,Z},{U,E,K},{Q,S,E}}

**Output:**

GEEKS QUIZ

**Explanation**:

G I Z

U E K

Q S E

Words we got is denoted using same color.

**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **wordBoggle()** which takes the dictionary contaning N space-separated strings and R\*C board as input parameters and returns a list of words that exist on the board.

**Expected Time Complexity:**O(N\*W + R\*C^2)  
**Expected Auxiliary Space:**O(N\*W + R\*C)

**Constraints:**  
1 ≤ N ≤ 15  
1 ≤ R, C ≤ 50  
1 ≤ length of Word ≤ 60

bool dfs(vector<vector<char>> &board, vector<vector<bool>> &vis, string str, int i, int j, int index)

{

if(index == str.size())

return true;

if(i >= 0 && j >= 0 && i < board.size() && j < board[0].size() && vis[i][j] == false && board[i][j] == str[index])

{

vis[i][j] = true;

if( dfs(board, vis, str, i+1, j, index + 1) ||

dfs(board, vis, str, i-1, j, index + 1) ||

dfs(board, vis, str, i, j+1, index + 1) ||

dfs(board, vis, str, i, j-1, index + 1) ||

dfs(board, vis, str, i+1, j+1, index + 1) ||

dfs(board, vis, str, i-1, j-1, index + 1) ||

dfs(board, vis, str, i+1, j-1, index + 1) ||

dfs(board, vis, str, i-1, j+1, index + 1) )

return true;

vis[i][j] = false;

}

return false;

}

bool check(vector<vector<char>> &board, string str)

{

int n = board.size();

int m = board[0].size();

vector<vector<bool>> vis(n, vector<bool>(m, false));

for(int i=0; i<n; i++)

for(int j=0; j<m; j++)

if(str[0] == board[i][j])

if(dfs(board, vis, str, i ,j, 0))

return true;

return false;

}

vector<string> wordBoggle(vector<vector<char>> &board, vector<string>& dictionary)

{

vector<string> ans;

for(string str: dictionary)

{ if(check(board, str) == true)

ans.push\_back(str);

}

return ans;

}

**############################################################**

**10. Rotten Oranges**

Given a grid of dimension **nxm** where each cell in the grid can have values 0, 1 or 2 which has the following meaning:  
**0**: Empty cell  
**1** : Cells have fresh oranges  
**2** : Cells have rotten oranges

We have to determine what is the minimum time required to rot all oranges. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (**up**, **down**, **left** and **right**) in unit time. 

**Example 1:**

**Input:** grid = {{0,1,2},{0,1,2},{2,1,1}}

**Output:** 1

**Explanation:** The grid is-

0 1 2

0 1 2

2 1 1

Oranges at positions (0,2), (1,2), (2,0)

will rot oranges at (0,1), (1,1), (2,2) and

(2,1) in unit time.

**Example 2:**

**Input:** grid = {{2,2,0,1}}

**Output:** -1

**Explanation:** The grid is-

2 2 0 1

Oranges at (0,0) and (0,1) can't rot orange at

(0,3).

**Your Task:**  
You don't need to read or print anything, Your task is to complete the function **orangesRotting()**which takes grid as input parameter and returns the minimum time to rot all the fresh oranges. If not possible returns -1.

**Expected Time Compelxity:**O(n\*m)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ n, m ≤ 500

int orangesRotting(vector<vector<int>>& grid)

{

int n = grid.size();

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

//vector<vector<bool>> vis(n, vector<bool>(m,false));

queue<pair<int,int>> q1,q2;

for(int i=0; i<n; i++)

for(int j=0; j<m; j++)

if(grid[i][j] == 2)

q1.push(make\_pair(i,j));

int count = 0;

while(!q1.empty() || !q2.empty())

{ if(!q1.empty())

count++;

while(!q1.empty())

{

pair<int, int> x = q1.front();

q1.pop();

int i = x.first;

int j = x.second;

grid[i][j]++;

if(i+1 >=0 && i+1 < n && j >= 0 && j < m && grid[i+1][j] == 1)

{

q2.push(make\_pair(i+1,j));

grid[i+1][j]++;

}

if(i >=0 && i < n && j+1 >= 0 && j+1 < m && grid[i][j+1] == 1)

{

q2.push(make\_pair(i,j+1));

grid[i][j+1]++;

}

if(i-1 >=0 && i-1 < n && j >= 0 && j < m && grid[i-1][j] == 1)

{

q2.push(make\_pair(i-1,j));

grid[i-1][j]++;

}

if(i >=0 && i < n && j-1 >= 0 && j-1 < m && grid[i][j-1] == 1)

{

q2.push(make\_pair(i,j-1));

grid[i][j-1]++;

}

}

if(!q2.empty())

count++;

while(!q2.empty())

{

pair<int, int> x = q2.front();

q2.pop();

int i = x.first;

int j = x.second;

grid[i][j]++;

if(i+1 >=0 && i+1 < n && j >= 0 && j < m && grid[i+1][j] == 1)

{

q1.push(make\_pair(i+1,j));

grid[i+1][j]++;

}

if(i >=0 && i < n && j+1 >= 0 && j+1 < m && grid[i][j+1] == 1)

{

q1.push(make\_pair(i,j+1));

grid[i][j+1]++;

}

if(i-1 >=0 && i-1 < n && j >= 0 && j < m && grid[i-1][j] == 1)

{

q1.push(make\_pair(i-1,j));

grid[i-1][j]++;

}

if(i >=0 && i < n && j-1 >= 0 && j-1 < m && grid[i][j-1] == 1)

{

q1.push(make\_pair(i,j-1));

grid[i][j-1]++;

}

}

}

for(int i=0; i<n; i++)

for(int j=0; j<m; j++)

if(grid[i][j] == 1)

return -1;

return count-1;

}

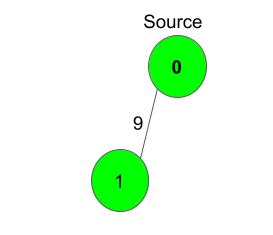
**############################################################**

**11. Implementing Dijkstra Algorithm**

Given a weighted, undirected and connected graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S.  
**Note:**The Graph doesn't contain any negative weight cycle.

**Example 1:**

**Input:**



**S** = 0

**Output:**

0 9

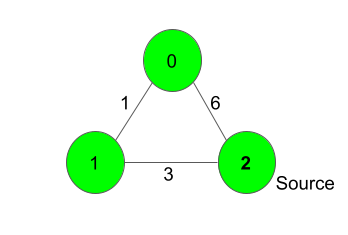
**Explanation**:

Shortest distance of all nodes from

source is printed.

**Example 2:**

**Input:**



**S** = 2

**Output:**

4 3 0

**Explanation**:

For nodes 2 to 0, we can follow the path-

2-1-0. This has a distance of 1+3 = 4,

whereas the path 2-0 has a distance of 6. So,

the Shortest path from 2 to 0 is 4.

The other distances are pretty straight-forward.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **dijkstra()**  which takes number of vertices Vandan adjacency list adj as input parameters and returns a list of integers, where ith integer denotes the shortest distance of the ith node from Source node. Here adj[i] contains a list of lists containing two integers where the first integer j denotes that there is an edge between i and j and second integer w denotes that the weight between edge i and j is w.

**Expected Time Complexity:** O(V2).  
**Expected Auxiliary Space:** O(V2).

**Constraints:**  
1 ≤ V ≤ 1000  
0 ≤ adj[i][j] ≤ 1000  
0 ≤ S < V

vector<int> dijkstra(int v, vector<vector<int>> adj[], int s)

{

vector<int> val(v, INT\_MAX);

val[s] = 0;

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;

q.push({0, s});

while(!q.empty())

{ int par = q.top().second;

q.pop();

for(auto child: adj[par])

{ if(val[child[0]] > val[par] + child[1])

{ val[child[0]] = val[par] + child[1];

q.push({val[child[0]], child[0]});

}

}

}

return val;

}

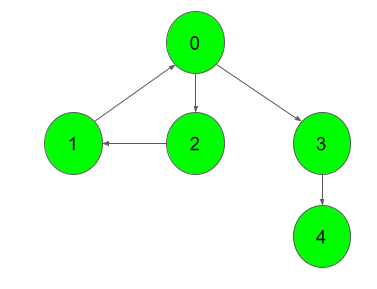
**############################################################**

**12. Strongly Connected Components (Kosaraju's Algo)**

Given a Directed Graph with**V**vertices **(**Numbered from**0 to V-1)** and **E** edges, Find the number of strongly connected components in the graph.

**Example 1:**

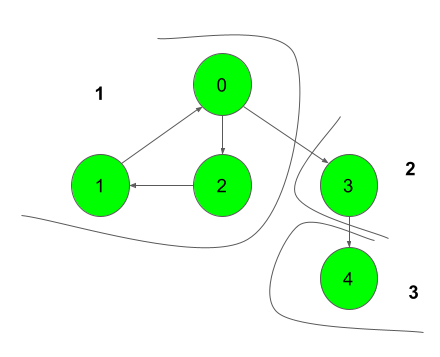
**Input:**



**Output:**

3

**Explanation**:

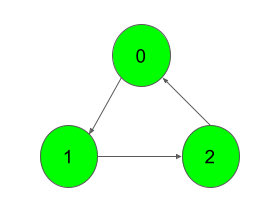


We can clearly see that there are 3 Strongly

Connected Components in the Graph

**Example 2:**

**Input:**



**Output:**

1

**Explanation**:

All of the nodes are connected to each other.

So, there's only one SCC.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **kosaraju()** which takes the number of vertices V and adjacency list of the graph as inputs and returns an integer denoting the number of strongly connected components in the given graph.

**Expected Time Complexity:** O(V+E).  
**Expected Auxiliary Space:** O(V).

**Constraints:**  
1 ≤ V ≤ 5000  
0 ≤ E ≤ (V\*(V-1))  
0 ≤ u, v ≤ N-1  
Sum of E over all testcases will not exceed 25\*106

void dfs(int node, vector<bool> &vis, stack<int> &stk, vector<int> adj[])

{ vis[node] = true;

for(auto i: adj[node])

if(vis[i] == false)

dfs(i, vis, stk, adj);

stk.push(node);

}

void dfs2(vector<int> rev[], int node, vector<bool> &vis)

{ vis[node] = true;

for(auto i: rev[node])

if(vis[i] == false)

dfs2(rev, i, vis);

}

int kosaraju(int v, vector<int> adj[])

{ stack<int> stk;

vector<bool> vis(v,false);

for(int i=0; i<v; i++)

if(vis[i] == false)

dfs(i, vis, stk, adj);

vector<int> rev[v];

for(int i=0; i<v; i++)

for(auto j: adj[i])

rev[j].push\_back(i);

vector<bool> vis2(v, false);

int count = 0;

while(!stk.empty())

{ int node = stk.top();

stk.pop();

if(vis2[node] == false)

{ dfs2(rev, node, vis2);

count++;

}

}

return count;

}

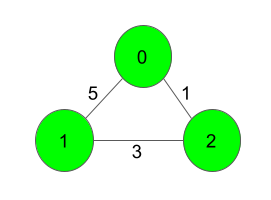
**############################################################**

**13. Minimum Spanning Tree**

Given a weighted, undirected and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.

**Example 1:**

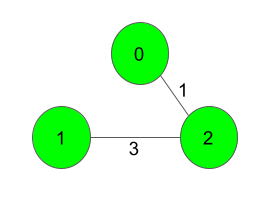
**Input:**



**Output:**

4

**Explanation**:

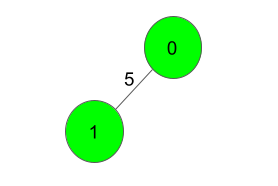


The Spanning Tree resulting in a weight

of 4 is shown above.

**Example 2:**

**Input:**



**Output:**

5

**Explanation**:

Only one Spanning Tree is possible

which has a weight of 5.

**Your task:**  
Since this is a functional problem you don't have to worry about input, you just have to complete the function  **spanningTree()** which takes number of vertices Vandan adjacency matrix adj as input parameters and returns an integer denoting the sum of weights of the edges of the Minimum Spanning Tree. Here adj[i] contains a list of lists containing two integers where the first integer j denotes that there is an edge between i and j and second integer w denotes that the distance between edge i and j is w.

**Expected Time Complexity:**O(ElogV).  
**Expected Auxiliary Space:**O(V2).

**Constraints:**  
2 ≤ V ≤ 1000  
V-1 ≤ E ≤ (V\*(V-1))/2  
1 ≤ w ≤ 1000  
Graph is connected and doesn't contain self loops & multiple edges.

int spanningTree(int v, vector<vector<int>> adj[]) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;

vector<bool> vis(v, false);

int par = 0;

int cost = 0;

q.push({0, par});

while(!q.empty()) {

pair<int, int> cur = q.top();

q.pop();

par = cur.second;

if(vis[par] == true)

continue;

cost += cur.first;

vis[par] = true;

for(auto child: adj[par]) {

if(vis[child[0]] == false)

q.push({child[1], child[0]});

}

}

return cost;

}

**############################################################**