**Graph Problem Set**

1. DFS of Graph

Given a connected undirected graph. Perform a Depth First Traversal of the graph.  
**Note:** Use recursive approach.

**Input:**  
The first line of the input contains an integer **'T'** denoting the number of test cases. Then **'T'** test cases follow. Each test case consists of two lines. Description of testcases is as follows: The First line of each test case contains two integers **'N' and 'E'** which denotes the no of vertices and no of edges respectively. The Second line of each test case contains **'E'** space separated pairs **u** and **v** denoting that there is a edge from **u** to **v** .

**Output:**  
For each testcase, print the nodes while doing DFSstarting from node 0.

**Your task:**  
You don't need to read input or print anything. Your task is to complete the function **dfs**() which takes the Graph and the number of vertices as inputs and returns a list containing the DFS Traversal of the graph starting from the 0th node.

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

**Constraints:**  
1 <= T <= 100  
2 <= N <= 104  
1 <= E <= (N\*(N-1))/2  
Graph doesn't contain multiple edges and self loops.

**Example:  
Input:**  
2  
5 4  
0 1 0 2 0 3 2 4  
4 3  
0 1 1 2 0 3

**Output:**  
0 1 2 4 3     
0 1 2 3

**Explanation:  
Testcase 1:**  
0 is connected to 1 , 2 , 3  
1 is connected to 0  
2 is connected to 0 and 4  
3 is connected to 0  
4 is connected to 2  
so starting from 0 , dfs will be 0 1 2 4 3.

**-------------------------------------------------------------------------------------------------------------------------------**

**Code:**

#include<bits/stdc++.h>

using namespace std;

// } Driver Code Ends

/\* Function to do DFS of graph

g : adjacency list of graph

N : number of vertices

return a list containing the DFS traversal of the given graph

\*/

vector<int> ans;

void fun(vector<int> g[],int v,bool vis[])

{

vis[v]=true;

ans.push\_back(v);

for(auto x:g[v])

{

if(!vis[x])

fun(g,x,vis);

}

return;

}

vector <int> dfs(vector<int> g[], int N)

{

bool vis[N];

memset(vis,false,sizeof(vis));

ans.clear();

fun(g,0,vis);

// Your code here

return ans;

}

// { Driver Code Starts.

int main()

{

int T;

cin>>T;

while(T--)

{

int N, E;

cin>>N>>E;

vector<int> g[N];

bool vis[N];

memset(vis, false, sizeof(vis));

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

{

int u,v;

cin>>u>>v;

g[u].push\_back(v);

g[v].push\_back(u);

}

vector <int> res = dfs(g, N);

for (int i = 0; i < res.size (); i++)

cout << res[i] << " ";

cout<<endl;

}

} // } Driver Code Ends

2. BFS of graph

Given a directed graph. The task is to do Breadth First Search of this graph.

**Input:**  
The first line of the input contains an integer **'T'** denoting the number of test cases. Then **'T'** test cases follow. Each test case consists of two lines. Description of testcases is as follows: The First line of each test case contains two integers **'N'**and **'E'** which denotes the no of vertices and no of edges respectively. The Second line of each test case contains **'E'** space separated pairs **u** and **v** denoting that there is a edge from **u** to **v** .

**Output:**  
For each testcase, print the BFS of the graph starting from 0.

**Note:** The expected output button always produces BFS starting from node 0.

**User Task:**  
You don't need to read input or print anything. Your task is to complete the function **bfs**() takes the Graph and the number of vertices as its input and returns a list containing the BFS traversal of the graph starting from the 0th vertex.

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

**Constraints:**  
1 <= T <= 100  
2 <= N <= 104  
1 <= E <= (N\*(N-1))/2  
Graph doesn't contain multiple edges and self loops.

**Example:  
Input:**  
2  
5 4  
0 1 0 2 0 3 2 4  
3 2  
0 1 0 2

**Output:**  
0 1 2 3 4    // BFS from node 0  
0 1 2

**Explanation:  
Testcase 1:**   
0 is connected to 1 , 2 , 3  
2 is connected to 4  
so starting from 0 , bfs will be 0 1 2 3 4.

**-------------------------------------------------------------------------------------------------------------------------------**

**code:**

#include <bits/stdc++.h>

using namespace std;

vector <int> bfs(vector<int> g[], int N);

int main() {

int T;

cin >> T;

while (T--) {

int N, E;

cin >> N >> E;

vector<int> adj[N];

for (int i = 0; i < E; i++) {

int u, v;

cin >> u >> v;

adj[u].push\_back(v);

}

vector <int> res = bfs(adj, N);

for (int i = 0; i < res.size (); i++) cout << res[i] << " ";

cout << endl;

}

}// } Driver Code Ends

/\* You have to complete this function\*/

/\* Function to do BFS of graph

\* g[]: adj list of the graph

\* N : number of vertices

\*/

vector <int> bfs(vector<int> g[], int N) {

queue<int> q;

q.push(0);

bool vis[N];

vector<int> ans;

memset(vis,false,sizeof(vis));

while(!q.empty())

{

int curr=q.front();

q.pop();

vis[curr]=true;

ans.push\_back(curr);

for(auto x:g[curr])

{

if(!vis[x])

{

q.push(x);

vis[x]=true;

}

}

}

return ans;

// Your code here

}

3. Detect cycle in an undirected graph

Given a Undirected Graph. Check whether it contains a cycle or not.

**Input:**  
The first line of the input contains an integer **'T'** denoting the number of test cases. Then **'T'** testcases follow. Each testcase consists of two lines. Description of testcases are as follows: The First line of each testcase contains two integers **'N' and 'M'**which denotes the no of vertices and no of edges respectively. The Second line of each test case contains **'M'** space separated pairs **u** and **v** denoting that there is a bidirectional  edge from **u** to **v** .

**Output:**  
The method should return **1** if there is a cycle else it should return **0**.

**User task:**  
You don't need to read input or print anything. Your task is to complete the function **isCyclic**which takes the Graph and the number of vertices as inputs and returns true if the given undirected graph contains any cycle. Else, it returns false.

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

**Constraints:**  
1 <= T <= 100  
2 <= N <= 104  
1 <= M <= (N\*(N-1))/2  
0 <= u, v <=  N-1  
Graph doesn't contain multiple edges and self loops.

**Example:  
Input:**  
3  
2 1  
0 1  
4 3  
0 1 1 2 2 3  
5 4  
0 1 2 3 3 4 4 2

**Output:**  
0  
0  
1  
  
**Explanation:  
Testcase 1:** There is a graph with 2 vertices and 1 edge from 0 to 1. So there is no cycle.  
**Testcase 2:** There is a graph with 3 vertices and 3 edges from 0 to 1, 1 to 2 and 2 to 3.  
**Testcase 3:** There is a cycle in the given graph formed by vertices 2, 3 and 4.

**-------------------------------------------------------------------------------------------------------------------------------**

**Code:**

#include<bits/stdc++.h>

using namespace std;

// } Driver Code Ends

/\* This function is used to detect a cycle in undirected graph

\* g[]: array of vectors to represent graph

\* V: number of vertices

\*/

bool check(vector<int> g[],bool vis[],int v,int N)

{

queue<int> q;

q.push(0);

vis[0]=true;

int par[N]={-1};

while(!q.empty())

{

int cur=q.front();

q.pop();

vis[cur]=true;

for(auto x:g[cur])

{

if(!vis[x])

{

q.push(x);

vis[x]=true;

par[x]=cur;

}

else if(par[cur]!=x)

{

return true;

}

}

}

return false;

}

bool isCyclic(vector<int> g[], int N)

{

bool vis[N];

memset(vis,false,sizeof(vis));

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

{

if(!vis[i])

if(check(g,vis,i,N))

return true;

}

return false;

// Your code here

}

// { Driver Code Starts.

int main()

{

int T;

cin>>T;

while(T--)

{

int V, E;

cin>>V>>E;

// array of vectors to represent graph

vector<int> adj[V];

int u, v;

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

{

cin>>u>>v;

// adding edge to the graph

adj[u].push\_back(v);

adj[v].push\_back(u);

}

cout << isCyclic(adj, V)<<endl;

}

}

// } Driver Code Ends

4. Detect cycle in a directed graph

Given a Directed Graph. Check whether it contains any cycle or not.

**Input:** The first line of the input contains an integer **'T'** denoting the number of test cases. Then **'T'** test cases follow. Each test case consists of two lines. Description of testcases is as follows: The First line of each test case contains two integers **'N' and 'M'** which denotes the no of vertices and no of edges respectively. The Second line of each test case contains **'M'** space separated pairs **u** and **v** denoting that there is an uni-directed  edge from **u** to **v** .

**Output:**  
The method should return **1** if there is a cycle else it should return **0**.

**User task:**  
You don't need to read input or print anything. Your task is to complete the function **isCyclic** which takes the Graph and the number of vertices and inputs and returns true if the given directed graph contains a cycle. Else, it returns false.

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

**Constraints:**  
1 <= T <= 1000  
1<= N,M <= 1000  
0 <= u,v <= N-1

**Example:  
Input:**  
3  
2 2  
0 1 0 0  
4 3  
0 1 1 2 2 3  
4 3  
0 1 2 3 3 2  
**Output:**  
1  
0  
1

**-------------------------------------------------------------------------------------------------------------------------------**

**Code:**

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

/\* Function to check if the given graph contains cycle

\* V: number of vertices

\* adj[]: representation of graph

\*/

bool check(vector<int> adj[],int v,int n,bool vis[],bool rec[])

{

if(!vis[v])

{

vis[v]=true;

rec[v]=true;

for(auto x:adj[v])

{

if(!vis[x] && check(adj,x,n,vis,rec))

return true;

else if(rec[x])

return true;

}

}

rec[v]=false;

return false;

}

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

{

bool vis[n];

bool rec[n];

memset(vis,false,sizeof(vis));

memset(vis,false,sizeof(rec));

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

if(check(adj,i,n,vis,rec))

return true;

return false;

}

// { Driver Code Starts.

int main() {

int t;

cin >> t;

while(t--){

int v, e;

cin >> v >> e;

vector<int> adj[v];

for(int i =0;i<e;i++){

int u, v;

cin >> u >> v;

adj[u].push\_back(v);

}

cout << isCyclic(v, adj) << endl;

}

return 0;

} // } Driver Code Ends

5. Topological sort

Given a Directed Graph. Find any Topological Sorting of that Graph.

**Input:**  
The first line of input takes the number of test cases then T test cases follow . Each test case contains two lines. The first  line of each test case  contains two integers E and V representing no of edges and the number of vertices. Then in the next line are E  pairs of integers u, v representing an edge from u to v in the graph.

**Output:**  
For each test case output will be 1 if the topological sort is done correctly else it will be 0.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **topoSort()** which takes the adjacency list of the Graph and the number of vertices (N) as inputs are 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:**  
1 <= T <= 100  
2 <= V <= 104  
1 <= E <= (N\*(N-1))/2  
0 <= u, v <= N-1  
Graph doesn't contain multiple edges, self loops and cycles.  
Graph may be disconnected.

**Example:**  
**Input**  
2  
6 6  
5 0 5 2 2 3 4 0 4 1 1 3  
3 4  
3 0 1 0 2 0

**Output**:  
1  
1

**Explanation:  
Testcase 1:**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.

**-------------------------------------------------------------------------------------------------------------------------------**

**Code:**

#include <bits/stdc++.h>

using namespace std;

vector <int> topoSort(int N, vector<int> adj[]);

/\* Function to check if elements returned by user

\* contains the elements in topological sorted form

\* V: number of vertices

\* \*res: array containing elements in topological sorted form

\* adj[]: graph input

\*/

bool check(int V, vector <int> &res, vector<int> adj[]) {

vector<int> map(V, -1);

for (int i = 0; i < V; i++) {

map[res[i]] = i;

}

for (int i = 0; i < V; i++) {

for (int v : adj[i]) {

if (map[i] > map[v]) return false;

}

}

return true;

}

// Driver Code

int main() {

int T;

cin >> T;

while (T--) {

int N, E;

cin >> E >> N;

int u, v;

vector<int> adj[N];

for (int i = 0; i < E; i++) {

cin >> u >> v;

adj[u].push\_back(v);

}

vector <int> res = topoSort(N, adj);

cout << check(N, res, adj) << endl;

}

}// } Driver Code Ends

// The Graph structure is as folows

/\* Function which sorts the graph vertices in topological form

\* N: number of vertices

\* adj[]: input graph

\*/

void topo(vector<int> adj[], int u, bool visited[], stack<int> &s) {

visited[u] = true; // Mark the current node as visited

for (auto v : adj[u]) // Run for adjacent vertices

{

if (!visited[v]) // If adjacent vertex is not visited

topo(adj, v, visited,s); // Call the topo function for adjacent vertex

}

s.push(u); // Push the current node to the stack

}

vector <int> topoSort(int N, vector<int> adj[]) {

bool visited[N + 1]; // Make a visited array of N nodes

memset(visited, 0, sizeof visited); // Mark the visited array initially 0

stack<int> s;

for (int i = 0; i < N; i++) // Traverse for all the nodes from 0 to N.

{

if (!visited[i]) // If the current node is not visited

topo(adj, i, visited, s); // Call the topo function

}

vector <int> res;

while (!s.empty()) // Till stack is not empty

{

res.push\_back (s.top()); // Put the stack top value to the array

s.pop();

}

return res;

}

6. Find the number of islands

Given a Matrix consisting of 0s and 1s. Find the number of islands of connected 1s present in the matrix.   
**Note:**A 1 is said to be connected if it has another 1 around it (either of the 8 directions).

**Input:**  
The first line of input will be the number of testcases **T**, then T test cases follow. The first line of each testcase contains two space separated integers N and M. Then in the next line are the NxM inputs of the matrix A separated by space .

**Output:**  
For each testcase in a new line, print the number of islands present.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findIslands()** which takes the matrix A and its dimensions N and M as inputs and returns the number of islands of connected 1s present in the matrix. A 1 is said to be connected if it has another 1 around it (either of the 8 directions).

**Expected Time Complexity:** O(N\*M).  
**Expected Auxiliary Space:** O(N\*M).

**Constraints:**  
1 <= T <= 100  
1 <= N, M <= 100  
0 <= A[i][j] <= 1

**Example(To be used only for expected output) :  
Input**  
2  
3 3  
1 1 0 0 0 1 1 0 1  
4 4  
1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0

**Output**  
2  
2

**Explanation**:  
**Testcase 1:** The graph will look like  
1 1 0  
0 0 1  
1 0 1  
Here, two islands will be formed  
First island will be formed by elements {A[0][0] ,  A[0][1], A[1][2], A[2][2]}  
Second island will be formed by {A[2][0]}**.  
Testcase 2:**The graph will look like  
1 1 0 0  
0 0 1 0  
0 0 0 1  
0 1 0 0  
Here, two islands will be formed  
First island will be formed by elements {A[0][0] ,  A[0][1], A[1][2], A[2][3]}  
Second island will be formed by {A[3][1]}**.**

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

#include <bits/stdc++.h>

using namespace std;

int findIslands(vector<int> A[], int N, int M);

int main() {

int T;

cin >> T;

while (T--) {

int N, M;

cin >> N >> M;

vector<int> A[N];

for (int i = 0; i < N; i++) {

vector<int> temp(M);

A[i] = temp;

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

cin >> A[i][j];

}

}

cout << findIslands(A, N, M) << endl;

}

return 0;

}// } Driver Code Ends

/\*you are required to complete this method\*/

/\* Function to find number of islands in the given graph

\* A[]: input array

\* N, M: Row and column of given matrix

\*/

bool safe(vector<int> a[],int i,int j,int n,int m,vector<vector<bool>> &vis)

{

if(i>=0 && j>=0 && i<n && j<m && a[i][j] && !vis[i][j])

{

// cout<<i<<"-"<<j<<endl;

return true;

}

return false;

}

void check(vector<int> a[],int n,int m,int i,int j,vector<vector<bool>> &vis)

{

vis[i][j]=true;

// cout<<i<<"<->"<<j<<endl;

int row[8]={-1,-1,-1,0,1,1,1,0};

int col[8]={-1,0,1,1,1,0,-1,-1};

for(int k=0;k<8;k++)

{

if(safe(a,i+row[k],j+col[k],n,m,vis))

check(a,n,m,i+row[k],j+col[k],vis);

}

return;

}

int findIslands(vector<int> a[], int n, int m) {

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

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

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

vis[i][j]=false;

// memset(vis,false,sizeof(vis));

int cnt=0;

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

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

{

if(!vis[i][j] && a[i][j])

{

// cout<<i<<" "<<j<<endl;

check(a,n,m,i,j,vis);

cnt++;

}

}

return cnt;

// Your code here

}

7. Implementing Dijkstra | Set 1 (Adjacency Matrix)

Given a graph of **V** nodes represented in the form of the adjacency matrix. The task is to find the shortest distance of all the vertex's from the source vertex.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. The first line of each test case contains an integer V denoting the size of the adjacency matrix and in the next line are V\*V space-separated values, which denote the weight of an edge of the matrix (gr[i][j] represents the weight of an edge from ith node to jth node). The third line of each test case contains an integer denoting the source vertex s.

**Output:**  
For each test, case output will be V space-separated integers where the ith integer denotes the shortest distance of the ith vertex from source vertex.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **dijkstra()** which takes the adjacency matrix of the Graph **g**, the source vertex **src**and the number of vertices **V**as inputs and returns a list containing the minimum distance of all the vertices from the source vertex.

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

**Constraints:**  
1 <= T <= 20   
1 <= V <= 100  
0 <= graph[i][j] <= 1000  
0 <= s < V

**Example:**  
**Input**:  
2  
2  
0 25 25 0  
0  
3  
0 1 43 1 0 6 43 6 0  
2

**Output**:  
0 25  
7 6 0

**Explanation:  
Testcase 1:**Shortest distance of source node 0 to 1 is 25, and shortest distance of source to source is 0.  
  
**Note:**  
You must assume that graph[i][j] = 0 means that the path from i to j does not exist.

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

vector <int> dijkstra(vector<vector<int>> , int ,int );

int printSolution(vector <int> dist, int n)

{

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

printf("%d ", dist[i]);

}

int main()

{

int t;

int V;

cin>>t;

while(t--){

cin>>V;

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

for(int i = 0;i<V;i++){

vector<int> temp(V);

g[i] = temp;

}

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

{

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

{

cin>>g[i][j];

}

}

int s;

cin>>s;

vector <int> res = dijkstra(g, s, V);

printSolution (res, V);

cout<<endl;

}

return 0;

}// } Driver Code Ends

/\* Function to implement Dijkstra

\* g: vector of vectors which represents the graph

\* src: source vertex to start traversing graph with

\* V: number of vertices

\*/

int findMinimumVertex(int distance[], bool visited[], int V)

{

int minVertex=-1;

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

{

if(!visited[i] && (minVertex==-1 || distance[i]<distance[minVertex]))

minVertex=i;

}

return minVertex;

}

vector <int> dijkstra(vector<vector<int>> graph, int src, int V)

{

bool visited[V]={0};

int distance[V];

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

{

distance[i]=INT\_MAX;

}

distance[src]=0;

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

{

int minVertex= findMinimumVertex(distance,visited,V);

visited[minVertex]=1;

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

{

if(graph[minVertex][j]!=0 && !visited[j] )

{

int dist=graph[minVertex][j]+distance[minVertex];

if(dist<distance[j])

distance[j]=dist;

}

}

}

vector<int> vec;

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

{

vec.push\_back(distance[i]);

}

return vec;

}

**8.** Minimum Swaps to Sort

Given an array of integers. Find the minimum number of swaps required to sort the array in non-decreasing order.

**Input:**  
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow. Each test case contains an integer N denoting the no of element of the array A[ ]. In the next line are N space separated values of the array A[ ] .

**Output:**  
For each test case in a new line output will be an integer denoting  minimum umber of swaps that are  required to sort the array.

**Constraints:**  
1 <= T <= 100  
1 <= N <= 105  
1 <= A[] <= 106

**User Task:**  
You don't need to read input or print anything. Your task is to complete the function **minSwaps**() which takes the array arr[] and its size N as inputs and returns an integer denoting the minimum number of swaps required to sort the array. If the array is already sorted, return 0.

**Expected Time Complexity:**O(NlogN).  
**Expected Auxiliary Space:**O(N).

**Example(To be used only for expected output):  
Input:**  
2  
5  
1 5 4 3 2  
4  
8 9 16 15

**Output:**  
2  
1

**Explanation:**  
**Test Case 1:**We need two swaps, swap 2 with 5 and 3 with 4 to make it sorted.

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

**#include<bits/stdc++.h>**

**using namespace std;**

**int minSwaps(int A[], int N);**

**// Driver program to test the above function**

**int main()**

**{**

**int t;**

**cin>>t;**

**while(t--)**

**{**

**int n;**

**cin>>n;**

**int a[n];**

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

**cin>>a[i];**

**cout<<minSwaps(a,n)<<endl;**

**}**

**}**

**// } Driver Code Ends**

**/\* You are required to complete this method \*/**

**// return the minimum number of swaps required to sort the arra**

**int minSwaps(int arr[], int n){**

**pair<int,int> arrp[n];**

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

**{**

**arrp[i].first=arr[i];**

**arrp[i].second=i;**

**}**

**bool vis[n];**

**memset(vis,false,sizeof(vis));**

**sort(arrp,arrp+n);**

**int ans=0;**

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

**{**

**int j=i;**

**if(vis[i] || arrp[i].second==i)**

**continue;**

**int cycle=0;**

**while(!vis[j])**

**{**

**vis[j]=true;**

**j=arrp[j].second;**

**cycle++;**

**}**

**if(cycle>0)**

**ans+=cycle-1;**

**}**

**return ans;**

**/\*Your code here \*/**

**}**

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

Given a graph with **N** nodes and **M** directed edges. Your task is to complete the function **kosaraju()** which returns an integer denoting the number of [strongly connected components](https://en.wikipedia.org/wiki/Strongly_connected_component) in the graph.

**Input:**  
The first line of input contains an integer **T**. Then T test cases follow. Each test case contains two integers **N** and **M**. In the next line there are **M** space-separated values u,v denoting an edge from u to v.

**Output:**  
For each test case in a new line output will an integer denoting the no of strongly connected components present in the graph.

**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(N + M).  
**Expected Auxiliary Space:** O(N).

**Constraints:**  
1 <= T <= 100  
1 <= N <= 5000  
0 <= M <= (N\*(N-1))  
0 <= u, v <= N-1  
Sum of M over all testcases will not exceed 25\*106

**Example:  
Input**:  
2  
5 5  
1 0 0 2 2 1 0 3 3 4  
3 3  
0 1 1 2 2 0

**Output:**  
3  
1

**Explanation:  
Testcase 1:**  
There is a connected subgraph that includes 0-1-2 which satisfy the condition of strongly connecting components i.e each node is reachable from every other nodes.

Another subgraph includes individual nodes 4 and 3. That gives us a total of 3 subgraphs satisfying the condition of strongly connected components.

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

#include<bits/stdc++.h>

using namespace std;

// } Driver Code Ends

/\* Function to find the number of strongly connected components

\* using Kosaraju's algorithm

\* V: number of vertices

\* adj[]: array of vectors to represent graph

\*/

void dfs(int u, vector<int>adj[], bool \*vis, stack<int> &s){

vis[u] = true;

for(int v: adj[u]){

if(!vis[v]){

dfs(v, adj, vis, s);

}

}

s.push(u);

}

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

{

// Your code here

stack<int> s;

int cnt = 0;

bool vis[V]; memset(vis, false, sizeof(vis));

for(int i=0; i<V; i++){

if(!vis[i]){

dfs(i, adj, vis, s);

}

}

vector<int> transpose[5000];

for(int i=0; i<V; i++){

for(int v: adj[i]){

transpose[v].push\_back(i);

}

}

stack<int> st;

cnt = 0;

memset(vis, false, V);

while(!s.empty()){

if(!vis[s.top()]){

dfs(s.top(), transpose, vis, st);

cnt++;

}

s.pop();

}

return cnt;

}

// { Driver Code Starts.

int main() {

int t;

cin >> t;

while (t--) {

int a, b;

cin >> a >> b;

int m, n;

vector<int> adj[a];

for (int i = 0; i < b; i++) {

cin >> m >> n;

adj[m].push\_back(n);

}

cout << kosaraju(a, adj) << endl;

}

return 0;

}

// } Driver Code Ends

10.Shortest Source to Destination Path

Given a boolean 2D matrix (0-based index), find whether there is path from (0,0) to (x,y) and if there is one path, print the minimum no of steps needed to reach it, else print -1 if the destination is not reachable. You may move in only four direction ie up, down, left and right. The path can only be created out of a cell if its value is 1.  
  
**Input:**  
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow. Each test case contains two lines . The first line of each test case contains two integers n and m denoting the size of the matrix. Then in the next line are n\*m space separated values of the matrix. The following line after it contains two integers x and y denoting the index of the destination.  
  
**Output:**  
For each test case print in a new line the min no of steps needed to reach the destination.  
  
**Constraints:**  
1<=T<=100  
1<=n,m<=20  
  
**Example:  
Input:**  
2  
3 4  
1 0 0 0 1 1 0 1 0 1 1 1  
2 3  
3 4  
1 1 1 1 0 0 0 1 0 0 0 1  
0 3  
**Output:**  
5  
3

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

using namespace std;

class node{

public:

int r,c,dis;

node(int x,int y,int d)

{

r=x;

c=y;

dis=d;

}

};

int fun()

{

int n,m;

cin>>n>>m;

node src(0,0,0);

int arr[n][m];

bool vis[n][m];

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

{

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

{

cin>>arr[i][j];

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

vis[i][j]=true;

else

vis[i][j]=false;

}

}

int di,dj;

cin>>di>>dj;

if(arr[0][0]==0 || arr[di][dj]==0)

return -1;

queue<node> q;

q.push(src);

vis[0][0]=true;

while(!q.empty())

{

node p=q.front();

q.pop();

if(p.r==di && p.c==dj)

return p.dis;

if(p.r-1>=0 && !vis[p.r-1][p.c])

{

q.push(node(p.r-1,p.c,p.dis+1));

vis[p.r-1][p.c]=true;

}

if(p.r+1<n && !vis[p.r+1][p.c])

{

q.push(node(p.r+1,p.c,p.dis+1));

vis[p.r+1][p.c]=true;

}

if(p.c-1>=0 && !vis[p.r][p.c-1])

{

q.push(node(p.r,p.c-1,p.dis+1));

vis[p.r][p.c-1]=true;

}

if(p.c+1<m && !vis[p.r][p.c+1])

{

q.push(node(p.r,p.c+1,p.dis+1));

vis[p.r][p.c+1]=true;

}

}

return -1;

}

int main()

{

int t;

cin>>t;

while(t--)

{

cout<<fun()<<endl;

}

return 0;

}

**11.** Find whether path exist

Given a **N** X **N** matrix (**M**) filled with 1, 0, 2, 3. The task is to find whether there is a path possible from source to destination, while traversing through blank cells only. You can traverse up, down, right and left.

* A value of cell **1** means Source.
* A value of cell **2** means Destination.
* A value of cell **3** means Blank cell.
* A value of cell **0**means Blank Wall.

**Note**: there is only single source and single destination.

**Input:**  
The first line of input is an integer **T** denoting the no of testcases. Then T test cases follow. Each test case consists of 2 lines. The first line of each test case contains an integer N denoting the size of the square matrix. Then in the next line are N\*N space separated values of the matrix (M).

**Output:**  
For each test case in a new line print 1 if the path exist from source to destination else print 0.

**Constraints:**  
1 <= T <= 20  
1 <= N <= 20

**Example:  
Input:**  
2  
4  
3 0 0 0 0 3 3 0 0 1 0 3 0 2 3 3   
3  
0 3 2 3 0 0 1 0 0

**Output:**  
1  
0

**Explanation:  
Testcase 1:**The matrix for the above given input is:  
3 0 0 0  
0 3 3 0  
0 1 0 3  
0 2 3 3  
From the matrix we can see that there exists a path from to reach destination 2 from source 1.  
**Testcase 2:** The matrix for the above given input is:  
0 3 2  
3 0 0  
1 0 0  
From the matrix we can see that there does not exists any path to reach destination 2 from source 1.

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

using namespace std;

bool safe(int i,int j,vector<vector<int>> &adj,vector<vector<bool>> &vis,int n,pair<int,int> s,pair<int,int> d)

{

// cout<<"safe-";

//cout<<i<<"-"<<j<<endl;

if(i>=0 && j>=0 && i<n && j<n && (adj[i][j]==3 || adj[i][j]==2) && !vis[i][j])

return true;

return false;

}

bool dfs(int i,int j,vector<vector<int>> &adj,vector<vector<bool>> &vis,int n,pair<int,int> s,pair<int,int> d)

{

// cout<<"dfs-";

// cout<<i<<"-"<<j<<endl;

if(i==d.first && j==d.second)

return true;

vis[i][j]=true;

int row[4]={0,-1,0,1};

int col[4]={-1,0,1,0};

for(int k=0;k<4;k++)

{

if( safe(i+row[k],j+col[k],adj,vis,n,s,d) && dfs(i+row[k],j+col[k],adj,vis,n,s,d) )

return true;

}

return false;

}

bool check(vector<vector<int>> adj,int n,pair<int,int> s,pair<int,int> d)

{

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

return dfs(s.first,s.second,adj,vis,n,s,d);

}

int main()

{

int t;

cin>>t;

while(t--)

{

int n;

cin>>n;

vector<vector<int>> adj(n,vector<int> (n));

pair<int,int> s;

pair<int,int> d;

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

{

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

{

cin>>adj[i][j];

// adj[i][j];

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

s={i,j};

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

d={i,j};

}

}

// cout<<"source & dest-"<<endl;

// cout<<s.first<<" "<<s.second<<endl;

// cout<<d.first<<" "<<d.second<<endl;

if(check(adj,n,s,d))

cout<<1<<endl;

else

cout<<0<<endl;

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

// {

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

// cout<<adj[i][j]<<" ";

// cout<<endl;

// }

}

return 0;

}

12. Minimum Cost Path

Given a square grid of size **N**, each cell of which contains integer cost which represents a cost to traverse through that cell, we need to find a path from top left cell to bottom right cell by which total cost incurred is minimum. You can move in 4 directions : up, down, left an right.

**Note :**It is assumed that negative cost cycles do not exist in input matrix.

**Input:**  
The first line of input will contain number of testcases **T**. Then T test cases follow. Each test case contains 2 lines. The first line of each test case contains an integer N denoting the size of the grid. Next line of each test contains a single line containing N\*N space separated integers depicting the cost of respective cell from (0,0) to (N,N).

**Output:**  
For each test case output a single integer depecting the minimum cost to reach the destination.

**Constraints:**  
1 <= T <= 100  
1 <= N <= 100  
1 <= grid[i][j] <= 104

**Example:  
Input:**  
2  
5  
31 100 65 12 18 10 13 47 157 6 100 113 174 11 33 88 124 41 20 140 99 32 111 41 20  
2  
42 93 7 14

**Output**:  
327  
63

**Explanation:**  
**Testcase 1:**  
Grid is:  
31, 100, 65, 12, 18,  
10, 13, 47, 157, 6,  
100. 113, 174, 11, 33,  
88, 124, 41, 20, 140,  
99, 32, 111, 41, 20  
A cost grid is given in below diagram, minimum  
cost to reach bottom right from top left  
is 327 (31 + 10 + 13 + 47 + 65 + 12 + 18 + 6 + 33 + 11 + 20 + 41 + 20)

**---------------------------------------------------------------------------------------------------------------------**

**Code:**

int dx[]={0,1,0,-1}; //possible changes in present coordinates

int dy[]={1,0,-1,0};

int minCostPath(int g[100][100],int n)

{

int dist[n][n]; //min distance stored array

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

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

dist[i][j]=INT\_MAX;

}

}

dist[0][0]=g[0][0]; //intialization

set<pair<int,pair<int,int>>>st;

//distance=0 at (0,0)

st.insert({0,{0,0}});

while(!st.empty()){

pair<int,pair<int,int>>temp=\*(st.begin());

st.erase(st.begin());

int fst=temp.second.first;

int sec=temp.second.second;

for(int i=0;i<4;i++){

int x=fst+dx[i], y=sec+dy[i];

if(x>=0 && y>=0 && x<n && y<n && dist[x][y]>dist[fst][sec]+g[x][y]){

dist[x][y]=dist[fst][sec]+g[x][y];

//updating the set with new cost and coordinate

st.insert({dist[x][y],{x,y}});

}

}

}

return dist[n-1][n-1];

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int t,n;

cin>>t;

while(t--){

cin>>n;

int g[100][100];

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

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

cin>>g[i][j];

}

}

cout<<minCostPath(g,n)<<endl;

}

return 0;

}