**Algorithm all question:**

1.. <https://leetcode.com/problems/keys-and-rooms/description/>

There are n rooms labeled from 0 to n - 1 and all the rooms are locked except for room 0. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of **distinct keys** in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array rooms where rooms[i] is the set of keys that you can obtain if you visited room i, return true if you can visit ***all*** the rooms, or false otherwise.

**Example 1:**

**Input:** rooms = [[1],[2],[3],[]]

**Output:** true

**Explanation:**

We visit room 0 and pick up key 1.

We then visit room 1 and pick up key 2.

We then visit room 2 and pick up key 3.

We then visit room 3.

Since we were able to visit every room, we return true.

**Example 2:**

**Input:** rooms = [[1,3],[3,0,1],[2],[0]]

**Output:** false

**Explanation:** We can not enter room number 2 since the only key that unlocks it is in that room.

class Solution {

public:

    bool vis[1005];

    void dfs(int src,vector<vector<int>>& rooms){

        vis[src]=true;

        for(int child:rooms[src]){

            if(vis[child]==false){

                dfs(child,rooms);

            }

        }

    }

    bool canVisitAllRooms(vector<vector<int>>& rooms) {

       int n=rooms.size();

       memset(vis,false,sizeof(vis));

       dfs(0,rooms);

       for(int i=0;i<n;i++){

        if(vis[i]==false){

            return false;

        }

       }

       return true;

    }

};

2. <https://cses.fi/problemset/task/1667>

Syrjälä's network has nnn computers and mmm connections. Your task is to find out if Uolevi can send a message to Maija, and if it is possible, what is the minimum number of computers on such a route.

# Input

The first input line has two integers nnn and mmm: the number of computers and connections. The computers are numbered 1,2,…,n1,2,\dots,n1,2,…,n. Uolevi's computer is 111 and Maija's computer is nnn.

Then, there are mmm lines describing the connections. Each line has two integers aaa and bbb: there is a connection between those computers.

Every connection is between two different computers, and there is at most one connection between any two computers.

# Output

If it is possible to send a message, first print kkk: the minimum number of computers on a valid route. After this, print an example of such a route. You can print any valid solution.

If there are no routes, print "IMPOSSIBLE".

# Constraints

* 2≤n≤1052 \le n \le 10^52≤n≤105
* 1≤m≤2⋅1051 \le m \le 2 \cdot 10^51≤m≤2⋅105
* 1≤a,b≤n1 \le a,b \le n1≤a,b≤n

# Example

Input:

5 5

1 2

1 3

1 4

2 3

5 4

Output:

3

1 4 5

Code:

#include <bits/stdc++.h>

using namespace std;

const int MAX\_N = 100005;

vector<int> adj\_list[MAX\_N];

bool vis[MAX\_N];

int level[MAX\_N], pa[MAX\_N];

void bfs(int src) {

queue<int> q;

q.push(src);

level[src] = 1; // Start counting from 1 (including the starting node)

vis[src] = true;

pa[src] = -1; // No parent for the source

while (!q.empty()) {

int par = q.front();

q.pop();

for (int child : adj\_list[par]) {

if (!vis[child]) {

vis[child] = true;

level[child] = level[par] + 1;

pa[child] = par;

q.push(child);

}

}

}

}

int main() {

int n, m;

cin >> n >> m;

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

int a, b;

cin >> a >> b;

adj\_list[a].push\_back(b);

adj\_list[b].push\_back(a);

}

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

memset(level, -1, sizeof(level));

bfs(1); // Start BFS from node 1

if (!vis[n]) {

cout << "IMPOSSIBLE\n";

return 0;

}

// Output the shortest distance (number of nodes in the path)

cout << level[n] << "\n";

// Reconstruct the path from n to 1

vector<int> path;

for (int v = n; v != -1; v = pa[v]) {

path.push\_back(v);

}

reverse(path.begin(), path.end());

// Print the path

for (int node : path) {

cout << node << " ";

}

cout << "\n";

return 0;

}

3. <https://leetcode.com/problems/flood-fill/description/>

You are given an image represented by an m x n grid of integers image, where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc, and color. Your task is to perform a **flood fill** on the image starting from the pixel image[sr][sc].

To perform a **flood fill**:

1. Begin with the starting pixel and change its color to color.
2. Perform the same process for each pixel that is **directly adjacent** (pixels that share a side with the original pixel, either horizontally or vertically) and shares the **same color** as the starting pixel.
3. Keep **repeating** this process by checking neighboring pixels of the *updated* pixels and modifying their color if it matches the original color of the starting pixel.
4. The process **stops** when there are **no more** adjacent pixels of the original color to update.

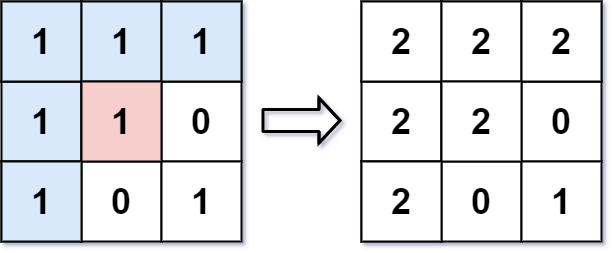
Return the **modified** image after performing the flood fill.

**Example 1:**

**Input:** image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

**Explanation:**



From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is **not** colored 2, because it is not horizontally or vertically connected to the starting pixel.

**Example 2:**

**Input:** image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

**Output:** [[0,0,0],[0,0,0]]

**Explanation:**

The starting pixel is already colored with 0, which is the same as the target color. Therefore, no changes are made to the image.

**Constraints:**

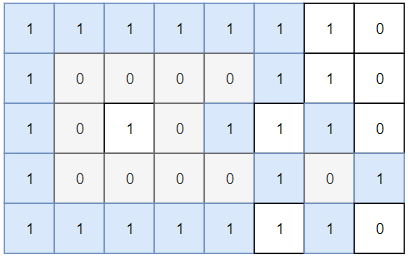
* m == image.length
* n == image[i].length
* 1 <= m, n <= 50
* 0 <= image[i][j], color < 216
* 0 <= sr < m
* 0 <= sc < n

4.. <https://leetcode.com/problems/number-of-closed-islands/description/>

Given a 2D grid consists of 0s (land) and 1s (water).  An *island* is a maximal 4-directionally connected group of 0s and a *closed island* is an island **totally** (all left, top, right, bottom) surrounded by 1s.

Return the number of *closed islands*.

**Example 1:**



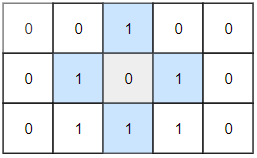
**Input:** grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],[1,0,0,0,0,1,0,1],[1,1,1,1,1,1,1,0]]

**Output:** 2

**Explanation:**

Islands in gray are closed because they are completely surrounded by water (group of 1s).

**Example 2:**



**Input:** grid = [[0,0,1,0,0],[0,1,0,1,0],[0,1,1,1,0]]

**Output:** 1

**Example 3:**

**Input:** grid = [[1,1,1,1,1,1,1],

  [1,0,0,0,0,0,1],

  [1,0,1,1,1,0,1],

  [1,0,1,0,1,0,1],

  [1,0,1,1,1,0,1],

  [1,0,0,0,0,0,1],

[1,1,1,1,1,1,1]]

**Output:** 2

**Constraints:**

* 1 <= grid.length, grid[0].length <= 100
* 0 <= grid[i][j] <=1

Code:

class Solution {

    int m, n;

    vector<vector<bool>> vis;

    vector<pair<int, int>> d = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    bool valid(int i, int j) {

        return i >= 0 && i < m && j >= 0 && j < n;

    }

    bool dfs(int si, int sj, vector<vector<int>>& grid) {

        if (si < 0 || si >= m || sj < 0 || sj >= n) {

            return false; // Touching boundary means it's not a closed island

        }

        if (vis[si][sj] || grid[si][sj] == 1) {

            return true;

        }

        vis[si][sj] = true;

        bool isClosed = true;

        for (auto &[dx, dy] : d) {

            int ci = si + dx;

            int cj = sj + dy;

            if (!dfs(ci, cj, grid)) {

                isClosed = false;

            }

        }

        return isClosed;

    }

5.. <https://leetcode.com/problems/maximum-number-of-fish-in-a-grid/description/>

You are given a **0-indexed** 2D matrix grid of size m x n, where (r, c) represents:

* A **land** cell if grid[r][c] = 0, or
* A **water** cell containing grid[r][c] fish, if grid[r][c] > 0.

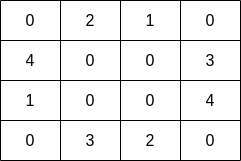
A fisher can start at any **water** cell (r, c) and can do the following operations any number of times:

* Catch all the fish at cell (r, c), or
* Move to any adjacent **water** cell.

Return *the* ***maximum*** *number of fish the fisher can catch if he chooses his starting cell optimally, or* 0 if no water cell exists.

An **adjacent** cell of the cell (r, c), is one of the cells (r, c + 1), (r, c - 1), (r + 1, c) or (r - 1, c) if it exists.

**Example 1:**

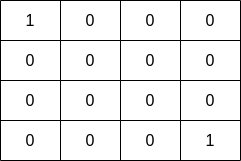


**Input:** grid = [[0,2,1,0],[4,0,0,3],[1,0,0,4],[0,3,2,0]]

**Output:** 7

**Explanation:** The fisher can start at cell (1,3) and collect 3 fish, then move to cell (2,3) and collect 4 fish.

**Example 2:**



**Input:** grid = [[1,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,1]]

**Output:** 1

**Explanation:** The fisher can start at cells (0,0) or (3,3) and collect a single fish.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 10
* 0 <= grid[i][j] <= 10

class Solution {

public:

   int n, m;

   int c = 0;

   bool vis[15][15];  // Visited array

   vector<pair<int, int>> d = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};  // 4 directions

   bool valid(int i, int j) {

       return i >= 0 && i < m && j >= 0 && j < n;

   }

   int dfs(int si, int sj, vector<vector<int>>& grid) {

       vis[si][sj] = true;  // Mark as visited

       c += grid[si][sj];   // Add current cell's fish count

       cout << "adding with c and total c is " << grid[si][sj] << " " << c << endl;

       for (int i = 0; i < 4; i++) {

           int ci = si + d[i].first;

           int cj = sj + d[i].second;

           if (valid(ci, cj) && !vis[ci][cj] && grid[ci][cj] > 0) {

               cout << ".....i and j " << ci << "  " << cj << endl;

               dfs(ci, cj, grid);

           }

       }

       return c;

   }

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

       m = grid.size();

       n = grid[0].size();

       int ans = 0;

       memset(vis, false, sizeof(vis));  // Reset visited array

       for (int i = 0; i < m; i++) {

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

               if (grid[i][j] > 0 && !vis[i][j]) {

                   c = 0;  // Reset `c` for each new DFS call

                   cout << "ans, c are " << ans << " " << c;

                   ans = max(ans, dfs(i, j, grid));

                   cout << " new answer and c is " << ans << " " << c << endl;

               }

           }

       }

       return ans;

   }

};

6.. <https://www.geeksforgeeks.org/problems/detect-cycle-in-an-undirected-graph/1?utm_source=geeksforgeeks&utm_medium=article_practice_tab&utm_campaign=article_practice_tab>

(do it with bfs and dfs)

Given an undirected graph with V vertices labelled from 0 to V-1 and E edges, check whether the graph contains any cycle or not. The Graph is represented as an adjacency list, where adj[i] contains all the vertices that are directly connected to vertex i.

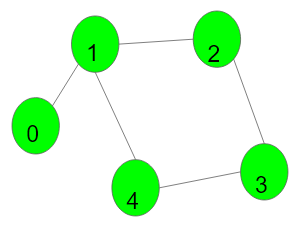
**NOTE:**The adjacency list represents undirected edges, meaning that if there is an edge between vertex i and vertex j, both j will be adj[i] and i will be in adj[j].

**Examples:**

**Input:** adj = [[1], [0,2,4], [1,3], [2,4], [1,3]]

**Output:** 1

**Explanation:**

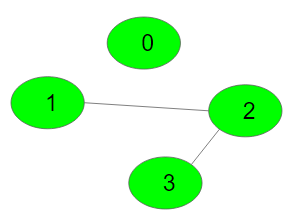


1->2->3->4->1 is a cycle.

**Input:** adj = [[], [2], [1,3], [2]]

**Output:** 0

**Explanation:**



No cycle in the graph.

**Constraints:**  
1 ≤ adj.size() ≤ 105

public:

bool vis[100005];

int pa[100005];

void dfs(int src,vector<vector<int>>& adj,bool &flag){

if(flag)return;

vis[src]=true;

for(auto child:adj[src]){

if(vis[child] && pa[src]!=child){

flag=true;

return ;

}

if(!vis[child]){

pa[child]=src;

dfs(child,adj,flag);

}

}

}

// Function to detect cycle in an undirected graph.

bool isCycle(vector<vector<int>>& adj) {

// Code here

bool flag=false;

int n=adj.size();

fill(vis, vis + 100005, false);

fill(pa, pa + 100005, -1);

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

if(!vis[i]){

dfs(i,adj,flag);

}

}

return flag;

}

7.. <https://leetcode.com/problems/island-perimeter/>

You are given row x col grid representing a map where grid[i][j] = 1 represents land and grid[i][j] = 0 represents water.

Grid cells are connected **horizontally/vertically** (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

**Example 1:**



**Input:** grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]

**Output:** 16

**Explanation:** The perimeter is the 16 yellow stripes in the image above.

**Example 2:**

**Input:** grid = [[1]]

**Output:** 4

**Example 3:**

**Input:** grid = [[1,0]]

**Output:** 4

**Constraints:**

* row == grid.length
* col == grid[i].length
* 1 <= row, col <= 100
* grid[i][j] is 0 or 1.
* There is exactly one island in grid

Code:

class Solution {

public:

int c = 0;

bool vis[105][105]; // Visited array

int m, n;

vector<pair<int, int>> d = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}; // Directions

void dfs(vector<vector<int>>& grid, int si, int sj) {

vis[si][sj] = true;

// Check all 4 directions

for (auto [dx, dy] : d) {

int ci = si + dx, cj = sj + dy;

// If out of bounds or water, contribute to perimeter

if (ci < 0 || ci >= m || cj < 0 || cj >= n || grid[ci][cj] == 0) {

c++;

} else if (!vis[ci][cj]) {

dfs(grid, ci, cj); // Continue DFS

}

}

}

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

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

m = grid.size();

n = grid[0].size();

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

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

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

dfs(grid, i, j);

return c; // Return perimeter after finding the first land cell

}

}

}

return 0; // No land found

}

};

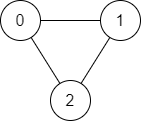
8.. <https://leetcode.com/problems/find-if-path-exists-in-graph/>

There is a **bi-directional** graph with n vertices, where each vertex is labeled from 0 to n - 1 (**inclusive**). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex source to vertex destination.

Given edges and the integers n, source, and destination, return true *if there is a* ***valid path*** *from* source *to* destination*, or* false *otherwise.*

**Example 1:**



**Input:** n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2

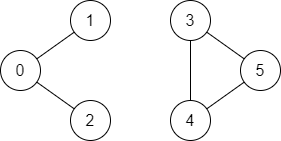
**Output:** true

**Explanation:** There are two paths from vertex 0 to vertex 2:

- 0 → 1 → 2

- 0 → 2

**Example 2:**



**Input:** n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5

**Output:** false

**Explanation:** There is no path from vertex 0 to vertex 5.

**Constraints:**

* 1 <= n <= 2 \* 105
* 0 <= edges.length <= 2 \* 105
* edges[i].length == 2
* 0 <= ui, vi <= n - 1
* ui != vi
* 0 <= source, destination <= n - 1
* There are no duplicate edges.
* There are no self edges.

Code:

class Solution {

public:

void dfs(vector<vector<int>>& edges,int src,vector<bool>&vis){

vis[src]=true;

for(int c:edges[src]){

if(vis[c]==false)

dfs(edges,c,vis);

}

}

bool validPath(int n, vector<vector<int>>& edges, int source, int destination){

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

for (auto& edge : edges) {

adj[edge[0]].push\_back(edge[1]);

adj[edge[1]].push\_back(edge[0]); // Since the graph is undirected

}

vector<bool>vis(n);

dfs(adj,source,vis);

if(vis[destination]){

return true;

}

else{

return false;

}

}

};

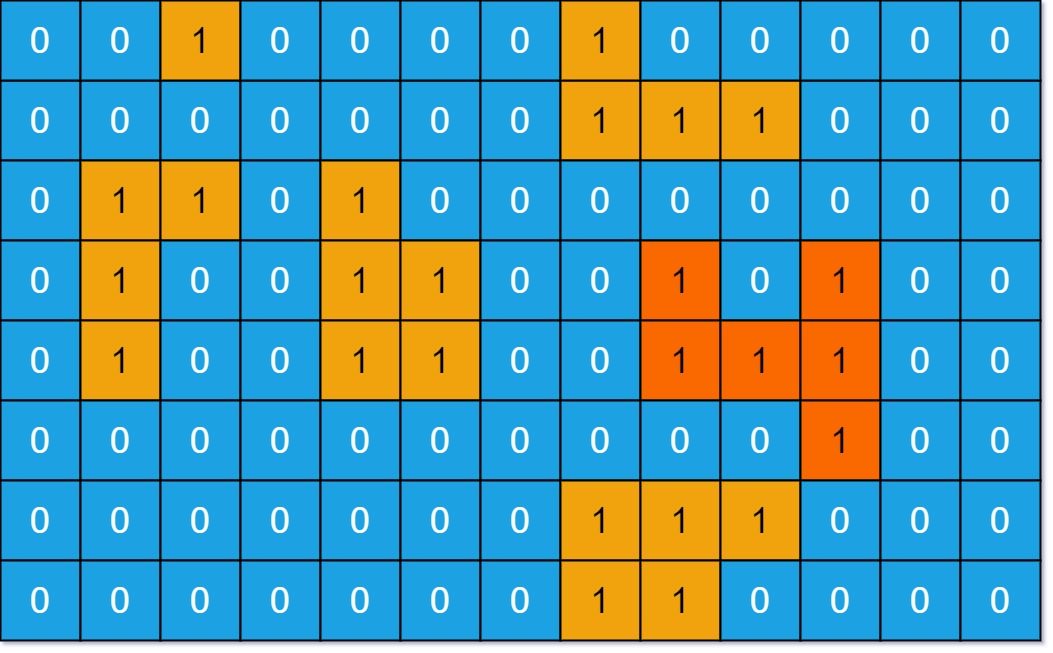
9.. <https://leetcode.com/problems/max-area-of-island/description/>

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value 1 in the island.

Return *the maximum* ***area*** *of an island in* grid. If there is no island, return 0.

**Example 1:**



**Input:** grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]

**Output:** 6

**Explanation:** The answer is not 11, because the island must be connected 4-directionally.

**Example 2:**

**Input:** grid = [[0,0,0,0,0,0,0,0]]

**Output:** 0

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 50
* grid[i][j] is either 0 or 1.

Code:

class Solution {

public:

int m, n;

int cf = 0;

bool vis[105][105]; // Global visited array

vector<pair<int, int>> d = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

void dfs(vector<vector<int>>& grid, int si, int sj, int& c) {

vis[si][sj] = true;

c++; // Start with counting the current cell

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

int ci = si + d[i].first;

int cj = sj + d[i].second;

// Check if the new cell is valid and within bounds

if (ci >= 0 && cj >= 0 && ci < m && cj < n && !vis[ci][cj] && grid[ci][cj] == 1) {

dfs(grid, ci, cj, c);

}

}

}

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

m = grid.size();

n = grid[0].size();

memset(vis, false, sizeof(vis)); // Initialize global visited array to false

int maxArea = 0;

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

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

if (!vis[i][j] && grid[i][j] == 1) { // Only start DFS if the cell is land (1)

int c = 0; // Temporary counter for the current island area

dfs(grid, i, j, c);

maxArea = max(maxArea, c); // Update max area

}

}

}

return maxArea;

}

};

10. <https://leetcode.com/problems/number-of-islands/description/>

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

**Example 2:**

**Input:** grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

**Output:** 3

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 300
* grid[i][j] is '0' or '1'.

Code:

class Solution {

public:

int m,n;

bool vis[400][400];

vector<pair<int,int>>d={{1,0},{0,1},{-1,0},{0,-1}};

void dfs(vector<vector<char>>& grid,int si,int sj){

vis[si][sj]=true;

cout<<"in dsf vis[i][j] "<<si<<" "<<sj<<"...>"<<vis[si][sj]<<endl;

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

int ci=si+d[i].first;

int cj=sj+d[i].second;

if(ci>=0 && cj>=0 && ci<m && cj<n && !vis[ci][cj] && grid[ci][cj]=='1' ){

cout<<"\*\*\*\*";

dfs(grid,ci,cj);

}

}

}

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

m=grid.size();

n=grid[0].size();

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

int cf=0;

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

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

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

cf++;

cout<<"i & j"<<i<<" , "<<j<<endl;

dfs(grid,i,j);

}

}

}

return cf;

}

};

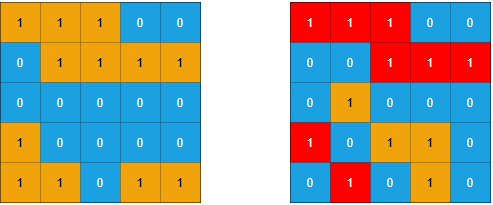
11. <https://leetcode.com/problems/count-sub-islands/description/>

You are given two m x n binary matrices grid1 and grid2 containing only 0's (representing water) and 1's (representing land). An **island** is a group of 1's connected **4-directionally** (horizontal or vertical). Any cells outside of the grid are considered water cells.

An island in grid2 is considered a **sub-island** if there is an island in grid1 that contains **all** the cells that make up **this** island in grid2.

Return the ***number*** *of islands in* grid2 *that are considered* ***sub-islands***.

**Example 1:**



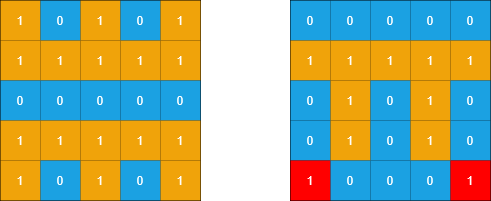
**Input:** grid1 = [[1,1,1,0,0],[0,1,1,1,1],[0,0,0,0,0],[1,0,0,0,0],[1,1,0,1,1]], grid2 = [[1,1,1,0,0],[0,0,1,1,1],[0,1,0,0,0],[1,0,1,1,0],[0,1,0,1,0]]

**Output:** 3

**Explanation:** In the picture above, the grid on the left is grid1 and the grid on the right is grid2.

The 1s colored red in grid2 are those considered to be part of a sub-island. There are three sub-islands.

**Example 2:**



**Input:** grid1 = [[1,0,1,0,1],[1,1,1,1,1],[0,0,0,0,0],[1,1,1,1,1],[1,0,1,0,1]], grid2 = [[0,0,0,0,0],[1,1,1,1,1],[0,1,0,1,0],[0,1,0,1,0],[1,0,0,0,1]]

**Output:** 2

**Explanation:** In the picture above, the grid on the left is grid1 and the grid on the right is grid2.

The 1s colored red in grid2 are those considered to be part of a sub-island. There are two sub-islands.

**Constraints:**

* m == grid1.length == grid2.length
* n == grid1[i].length == grid2[i].length
* 1 <= m, n <= 500
* grid1[i][j] and grid2[i][j] are either 0 or 1.

Code:

class Solution {

int m,n,c=0;

int flag;

vector<vector<bool>>vis;

vector<pair<int,int>>d={{-1,0},{1,0},{0,-1},{0,1}};

bool valid(int i,int j){

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

return false;

}

return true;

}

void dfs(int si,int sj,vector<vector<int>>& grid1, vector<vector<int>>& grid2){

vis[si][sj]=true;

if(grid1[si][sj]==0){

flag=1;

}

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

int ci=si+d[i].first;

int cj=sj+d[i].second;

if(valid(ci,cj)==true && vis[ci][cj]==false && grid2[ci][cj]==1){

dfs(ci,cj,grid1,grid2);

}

}

}

public:

int countSubIslands(vector<vector<int>>& grid1, vector<vector<int>>& grid2) {

m=grid1.size();

n=grid1[0].size();

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

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

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

if(grid2[i][j]==1 && vis[i][j]==false){

flag=0;

dfs(i,j,grid1,grid2);

if(flag==0){

c++;

}

}

}

}

return c;

}

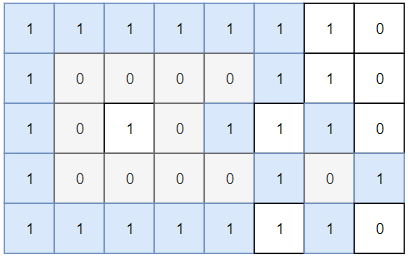
};

12. <https://leetcode.com/problems/number-of-closed-islands/description/>

Given a 2D grid consists of 0s (land) and 1s (water).  An *island* is a maximal 4-directionally connected group of 0s and a *closed island* is an island **totally** (all left, top, right, bottom) surrounded by 1s.

Return the number of *closed islands*.

**Example 1:**



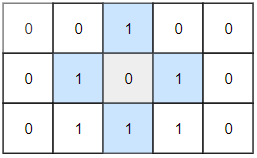
**Input:** grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],[1,0,0,0,0,1,0,1],[1,1,1,1,1,1,1,0]]

**Output:** 2

**Explanation:**

Islands in gray are closed because they are completely surrounded by water (group of 1s).

**Example 2:**



**Input:** grid = [[0,0,1,0,0],[0,1,0,1,0],[0,1,1,1,0]]

**Output:** 1

**Example 3:**

**Input:** grid = [[1,1,1,1,1,1,1],

  [1,0,0,0,0,0,1],

  [1,0,1,1,1,0,1],

  [1,0,1,0,1,0,1],

  [1,0,1,1,1,0,1],

  [1,0,0,0,0,0,1],

[1,1,1,1,1,1,1]]

**Output:** 2

**Constraints:**

* 1 <= grid.length, grid[0].length <= 100
* 0 <= grid[i][j] <=1

Code:

class Solution {

    int m, n;

    vector<vector<bool>> vis;

    vector<pair<int, int>> d = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    bool valid(int i, int j) {

        return i >= 0 && i < m && j >= 0 && j < n;

    }

    bool dfs(int si, int sj, vector<vector<int>>& grid) {

        if (si < 0 || si >= m || sj < 0 || sj >= n) {

            return false; // Touching boundary means it's not a closed island

        }

        if (vis[si][sj] || grid[si][sj] == 1) {

            return true;

        }

        vis[si][sj] = true;

        bool isClosed = true;

        for (auto &[dx, dy] : d) {

            int ci = si + dx;

            int cj = sj + dy;

            if (!dfs(ci, cj, grid)) {

                isClosed = false;

            }

        }

        return isClosed;

    }

public:

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

        m = grid.size();

        n = grid[0].size();

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

        int count = 0;

        for (int i = 0; i < m; i++) {

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

                if (grid[i][j] == 0 && !vis[i][j]) {

                    if (dfs(i, j, grid)) {

                        count++;

                    }

                }

            }

        }

        return count;

    }

};

13. <https://cses.fi/problemset/task/1192>

You are given a map of a building, and your task is to count the number of its rooms. The size of the map is n×mn \times mn×m squares, and each square is either floor or wall. You can walk left, right, up, and down through the floor squares.

# Input

The first input line has two integers nnn and mmm: the height and width of the map.

Then there are nnn lines of mmm characters describing the map. Each character is either . (floor) or # (wall).

# Output

Print one integer: the number of rooms.

# Constraints

* 1≤n,m≤10001 \le n,m \le 10001≤n,m≤1000

# Example

Input:

5 8

########

#..#...#

####.#.#

#..#...#

########

Output:

3

Code:

14.. <https://www.geeksforgeeks.org/problems/implementing-dijkstra-set-1-adjacency-matrix/1?utm_source=geeksforgeeks&utm_medium=article_practice_tab&utm_campaign=article_practice_tab>

Given a weighted, undirected and connected graph where you have given [adjacency list](https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/) **adj.** You have to find the shortest distance of all the vertices from the source vertex **src**, and return a list of integers denoting the shortest distance between **each node** and source vertex **src**.

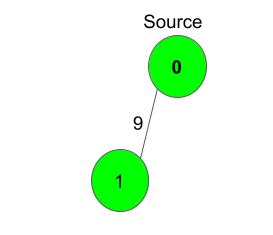
**Note:** The Graph doesn't contain any negative weight edge.

**Examples:**

**Input:** adj=[[[1, 9]], [[0, 9]]], src= 0

**Output:** [0, 9]

**Explanation**:

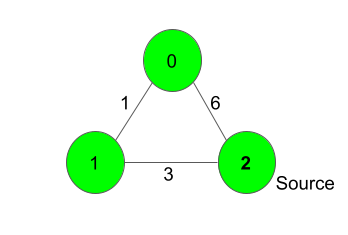


The source vertex is 0. Hence, the shortest distance of node 0 is 0 and the shortest distance from node 0 to 1 is 9.

**Input:** adj = [[[1, 1], [2, 6]], [[2, 3], [0, 1]], [[1, 3], [0, 6]]], src= 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 shortest distance from 0 to 1 is 1 .

**Constraints:**  
1 ≤ no. of vertices ≤ 1000  
0 ≤ adj[i][j] ≤ 1000

0 ≤ src < no. of vertices

Code:

public:

// Function to find the shortest distance of all the vertices

// from the source vertex src.

vector<int> dijkstra(vector<vector<pair<int, int>>> &adj, int src) {

// Code here

int n=adj.size();

vector<int>dis(n,INT\_MAX);

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

dis[src]=0;

pq.push({0,src});

while(!pq.empty()){

pair<int,int>par=pq.top();

pq.pop();

int par\_node=par.second;

int par\_dis=par.first;

for(auto child:adj[par\_node]){

int child\_node=child.first;

int child\_dis=child.second;

if(child\_dis+par\_dis<dis[child\_node]){

dis[child\_node]=par\_dis+child\_dis;

pq.push({dis[child\_node],child\_node});

}

}

}

return dis;

}

15.. <https://cses.fi/problemset/task/1666/>

Byteland has nnn cities, and mmm roads between them. The goal is to construct new roads so that there is a route between any two cities.

Your task is to find out the minimum number of roads required, and also determine which roads should be built.

# Input

The first input line has two integers nnn and mmm: the number of cities and roads. The cities are numbered 1,2,…,n1,2,\dots,n1,2,…,n.

After that, there are mmm lines describing the roads. Each line has two integers aaa and bbb: there is a road between those cities.

A road always connects two different cities, and there is at most one road between any two cities.

# Output

First print an integer kkk: the number of required roads.

Then, print kkk lines that describe the new roads. You can print any valid solution.

# Constraints

* 1≤n≤1051 \le n \le 10^51≤n≤105
* 1≤m≤2⋅1051 \le m \le 2 \cdot 10^51≤m≤2⋅105
* 1≤a,b≤n1 \le a,b \le n1≤a,b≤n

# Example

Input:

4 2

1 2

3 4

Output:

1

2 3

Code:

#include<bits/stdc++.h>

using namespace std;

vector<int>adj\_list[100005];

bool vis[100005];

void dfs(int src){

vis[src]=true;

for(auto child:adj\_list[src]){

if(!vis[child]){

dfs(child);

}

}

}

int main(){

int n,e,dst,src;

cin>>n>>e;

vector<int>v;

int c;

while(e--){

int a,b;

cin>>a>>b;

adj\_list[a].push\_back(b);

adj\_list[b].push\_back(a);

}

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

for(int i=1;i<=n;i++){

if(!vis[i]){

v.push\_back(i);

dfs(i);

}

}

c=v.size();

cout<<c-1<<endl;

for(int i=1;i<v.size();i++){

cout<<v[i-1]<<" "<<v[i]<<endl;

}

return 0;

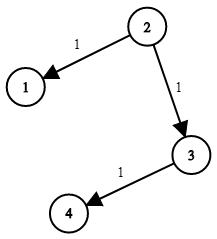
}

16. <https://leetcode.com/problems/network-delay-time/description/>

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return *the* ***minimum*** *time it takes for all the* n *nodes to receive the signal*. If it is impossible for all the n nodes to receive the signal, return -1.

**Example 1:**



**Input:** times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

**Output:** 2

**Example 2:**

**Input:** times = [[1,2,1]], n = 2, k = 1

**Output:** 1

**Example 3:**

**Input:** times = [[1,2,1]], n = 2, k = 2

**Output:** -1

**Constraints:**

* 1 <= k <= n <= 100
* 1 <= times.length <= 6000
* times[i].length == 3
* 1 <= ui, vi <= n
* ui != vi
* 0 <= wi <= 100
* All the pairs (ui, vi) are **unique**. (i.e., no multiple edges.)

**Code:**

class Solution {

public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {

        vector<int>dis(n+1,INT\_MAX);

        vector<vector<pair<int,int>>>adj(n+1);

        for(auto x:times){//carefulllllllllllll.............

        adj[x[0]].push\_back({x[1], x[2]});

        }

        dis[k]=0;

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

       pq.push({0,k});

       while(!pq.empty()){

        pair<int,int>par=pq.top();

        pq.pop();

        int par\_node=par.second;

        int par\_dis=par.first;

        for(auto child:adj[par\_node]){

            int child\_node=child.first;

            int child\_dis=child.second;

            if(par\_dis+child\_dis<dis[child\_node]){

                dis[child\_node]=par\_dis+child\_dis;

                pq.push({dis[child\_node],child\_node});

            }

        }

       }

        int ans = \*max\_element(dis.begin() + 1, dis.end()); ///carefulllllllllll

        return (ans == INT\_MAX) ? -1 : ans;

    }

};

17. <https://www.hackerrank.com/contests/assignment-01-a-introduction-to-algorithms-a-batch-06/challenges/connected-or-not-1-1>

You will be given a directed graph as input. Then you will receive queries. For each query, you will be given two nodes, and . You need to determine whether you can go from to

directly without using any other nodes.

**Input Format**

* The first line will contain

and , the number of nodes and the number of edges, respectively. The values of the nodes range from to

 .

 Next

 lines will contain two node values which means there is a connection from first node to second node.

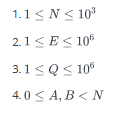
 The next line will contain

 .

 The following lines will each contain and

* .

**Constraints**



**Output Format**

* For each query output **YES** if it is possible to go from

to

* directly without using any other nodes, **NO** otherwise. Don't forget to put a new line after each query.

**Sample Input 0**

5 6

0 1

1 2

2 3

3 4

1 4

0 2

10

0 1

1 0

2 2

2 3

0 3

3 0

1 4

4 1

4 3

1 2

**Sample Output 0**

YES

NO

YES

YES

NO

NO

YES

NO

NO

YES

Code:

#include<bits/stdc++.h>

using namespace std;

int main(){

int n,e;

cin>>n>>e;

int a[n][n];

memset(a,0,sizeof(a));

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

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

if(i==j){

a[i][j]=1;

}

}

}

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

int x,y;

cin>>x>>y;

a[x][y]=1;

}

int q;

cin>>q;

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

int x,y;

cin>>x>>y;

if(a[x][y]==1){

cout<<"YES"<<endl;

}

else{

cout<<"NO"<<endl;

}

}

return 0;

}

18. <https://www.hackerrank.com/contests/assignment-01-a-introduction-to-algorithms-a-batch-06/challenges/connected-nodes-1>

You will be given an undirected graph as input. Then you will be given queries. For each query you will be given a node . You need to print the nodes that are connected with

in **descending** order.

Note: If there is no node connected to

, then print -1.

**Input Format**

* The first line will contain

and , the number of nodes and the number of edges, respectively. The values of the nodes range from to

 .

 Next

 lines will contain two node values which means there is a connection between first node and second node.

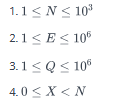
 The next line will contain

 .

 The following lines will each contain

* .

**Constraints**



**Output Format**

* Output the nodes that are connected with
* in descending order.

**Sample Input 0**

6 8

0 4

0 5

4 2

4 3

5 3

2 0

0 1

1 3

6

0

1

2

3

4

5

**Sample Output 0**

5 4 2 1

3 0

4 0

5 4 1

3 2 0

3 0

**Sample Input 1**

5 3

0 1

1 2

0 4

2

3

0

**Sample Output 1**

-1

4 1

Code:

#include<bits/stdc++.h>

using namespace std;

vector<int>adj\_list[1005];

bool vis[1005];

int level[1005];

int main(){

int n,e;

cin>>n>>e;

while(e--){

int a,b;

cin>>a>>b;

adj\_list[a].push\_back(b);

adj\_list[b].push\_back(a);

}

int x;

cin>>x;

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

vector<int>v;

int a;

cin>>a;

if(adj\_list[a].empty()==true){

cout<<"-1"<<endl;

continue;

}

for(auto p:adj\_list[a]){

v.push\_back(p);

}

sort(v.begin(), v.end(), greater<int>());

for(auto p:v){

cout<<p<<" ";

}

cout<<endl;

}

return 0;

}

19. <https://www.hackerrank.com/contests/assignment-01-a-introduction-to-algorithms-a-batch-06/challenges/can-go-1>

You are given an sized 2D matrix that represents a map of a building. Each cell represents a wall, a floor or a room. You will be given two rooms and . You need to tell if you can go from room to

by passing through the floors. You can walk **left**, **right**, **up**, and **down** through the floor cells. You can't pass through walls.

**Input Format**

* The first input line has two integers

and

 : the height and width of the map.

 Then there are lines of

* characters describing the map. Each character is .(floor), #(wall), A or B (rooms).

**Constraints**

**Output Format**

* Output **YES** if you can go from room

to

* , **NO** otherwise.

**Sample Input 0**

5 8

########

#.A#...#

#.##.#B#

#......#

########

**Sample Output 0**

YES

Code:

#include <bits/stdc++.h>

using namespace std;

int m, n;

char grid[1005][1005];

bool vis[1005][1005];

vector<pair<int, int>> d = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

void dfs(int si, int sj) {

vis[si][sj] = true;

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

int ci = si + d[i].first;

int cj = sj + d[i].second;

if (ci >= 0 && ci < m && cj >= 0 && cj < n && !vis[ci][cj] && (grid[ci][cj] == '.' || grid[ci][cj] == 'B')) {

dfs(ci, cj);

}

}

}

int main() {

cin >> m >> n;

int fi, fj, li, lj;

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

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

cin >> grid[i][j];

if (grid[i][j] == 'A') {

fi = i;

fj = j;

}

if (grid[i][j] == 'B') {

li = i;

lj = j;

}

}

}

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

dfs(fi, fj);

if (vis[li][lj]) {

cout << "YES" << endl;

} else {

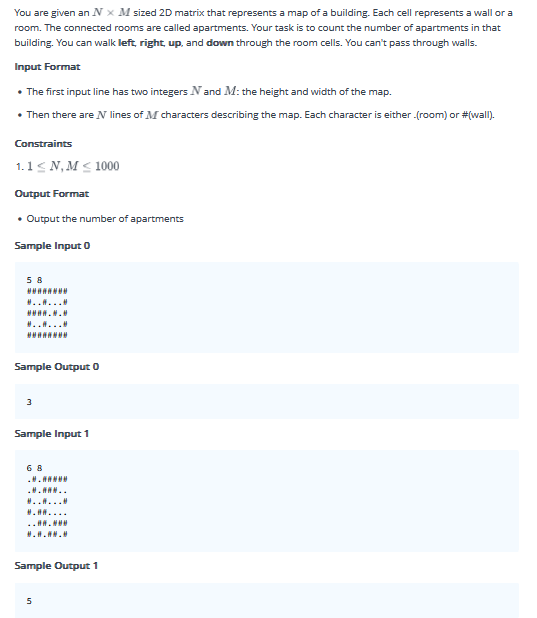
cout << "NO" << endl;

}

return 0;

}

20. <https://www.hackerrank.com/contests/assignment-01-a-introduction-to-algorithms-a-batch-06/challenges/count-apartments>



Code:

#include <bits/stdc++.h>

using namespace std;

int m, n;

char grid[1005][1005];

bool vis[1005][1005];

vector<pair<int, int>> d = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

void dfs(int si, int sj) {

vis[si][sj] = true;

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

int ci = si + d[i].first;

int cj = sj + d[i].second;

if (ci >= 0 && ci < m && cj >= 0 && cj < n && !vis[ci][cj] && (grid[ci][cj] == '.' )) {

dfs(ci, cj);

}

}

}

int main() {

cin >> m >> n;

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

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

cin >> grid[i][j];

}

}

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

int c=0;

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

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

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

dfs(i,j);

c++;

}

}

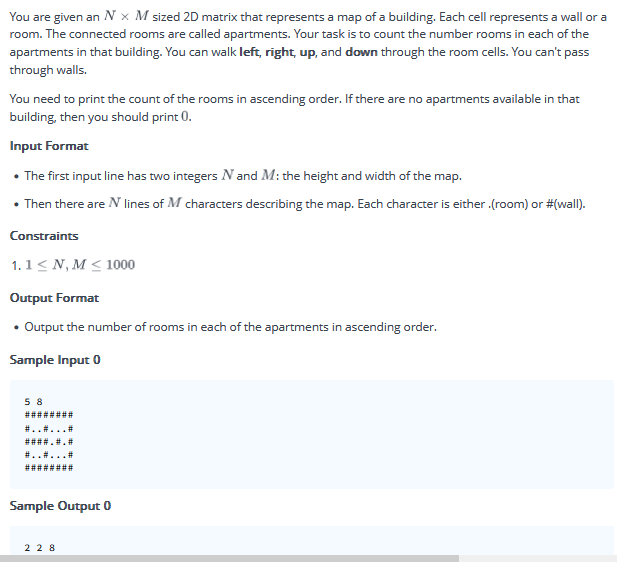
}

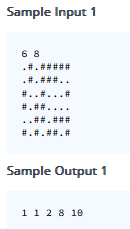
cout<<c<<endl;

return 0;

}

21. <https://www.hackerrank.com/contests/assignment-01-a-introduction-to-algorithms-a-batch-06/challenges/count-apartments-ii>





Code:

#include <bits/stdc++.h>

using namespace std;

int m, n;

char grid[1005][1005];

bool vis[1005][1005];

vector<pair<int, int>> d = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

void dfs(int si, int sj, int &c1) {

c1++;

vis[si][sj] = true;

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

int ci = si + d[i].first;

int cj = sj + d[i].second;

if (ci >= 0 && ci < m && cj >= 0 && cj < n && !vis[ci][cj] && grid[ci][cj] == '.') {

dfs(ci, cj, c1);

}

}

}

int main() {

cin >> m >> n;

vector<int> v;

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

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

cin >> grid[i][j];

}

}

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

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

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

if (!vis[i][j] && grid[i][j] == '.') {

int c1 = 0;

dfs(i, j, c1);

v.push\_back(c1);

}

}

}

if (v.empty()) {

cout << "0" << endl;

} else {

sort(v.begin(), v.end());

for (int x : v) {

cout << x << " ";

}

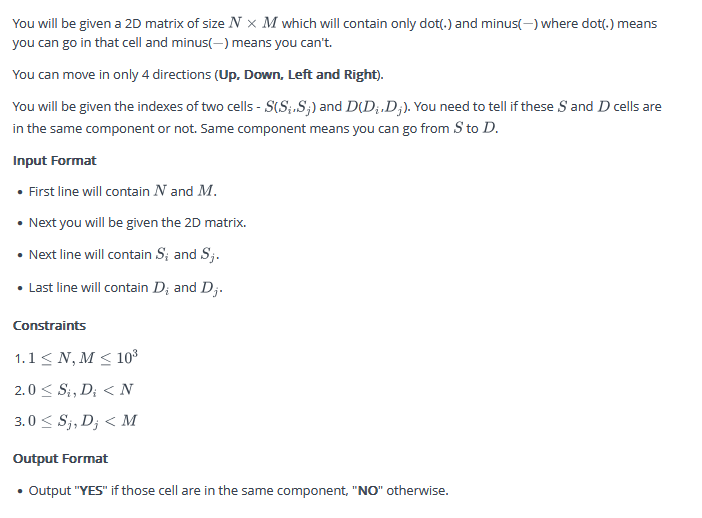
cout << endl;

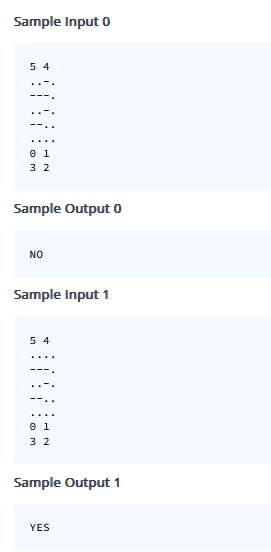
}

return 0;

}

22. <https://www.hackerrank.com/contests/mid-term-exam-a-introduction-to-algorithms-a-batch-06/challenges/same-component>





#include <bits/stdc++.h>

using namespace std;

int n, m;

int si, sj;

char grid[1005][1005];

vector<pair<int, int>> d = {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};

bool vis[1005][1005];

bool valid(int i, int j) {

return (i >= 0 && i < n && j >= 0 && j < m);

}

void dfs(int si, int sj) {

vis[si][sj] = true;

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

int ci = si + d[i].first;

int cj = sj + d[i].second;

if (valid(ci, cj) && !vis[ci][cj] && grid[ci][cj] == '.') {

dfs(ci, cj);

}

}

}

int main() {

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

cin >> n >> m;

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

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

cin >> grid[i][j];

}

}

cin >> si >> sj;

dfs(si, sj);

int xi, xj;

cin >> xi >> xj;

if (vis[xi][xj]) {

cout << "YES" << endl;

} else {

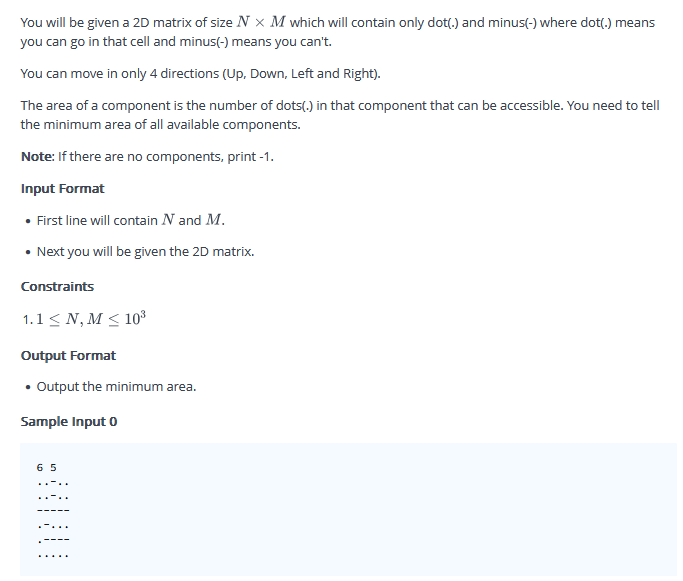
cout << "NO" << endl;

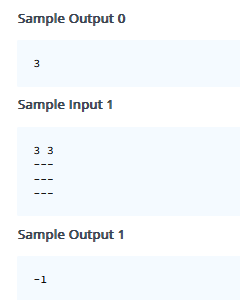
}

return 0;

}

23. <https://www.hackerrank.com/contests/mid-term-exam-a-introduction-to-algorithms-a-batch-06/challenges/area-of-component>





Code:

#include <bits/stdc++.h>

using namespace std;

int n, m;

int si, sj;

char grid[1005][1005];

int c;

vector<pair<int, int>> d = {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};

bool vis[1005][1005];

bool valid(int i, int j) {

return (i >= 0 && i < n && j >= 0 && j < m);

}

void dfs(int si, int sj) {

vis[si][sj] = true;

c=c+1;

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

int ci = si + d[i].first;

int cj = sj + d[i].second;

if (valid(ci, cj) && !vis[ci][cj] && grid[ci][cj] == '.') {

dfs(ci, cj);

}

}

}

int main() {

int ans=INT\_MAX;

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

cin >> n >> m;

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

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

cin >> grid[i][j];

}

}

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

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

if(!vis[i][j] && grid[i][j]=='.'){

c=0;

dfs(i,j);

ans=min(c,ans);

}

}

}

if(ans!=INT\_MAX){

cout<<ans<<endl;

}

else{

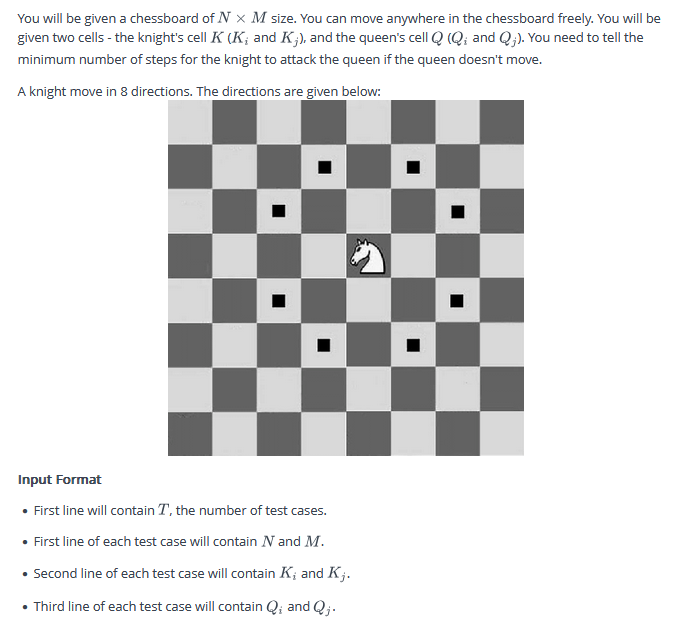
cout<<-1<<endl;

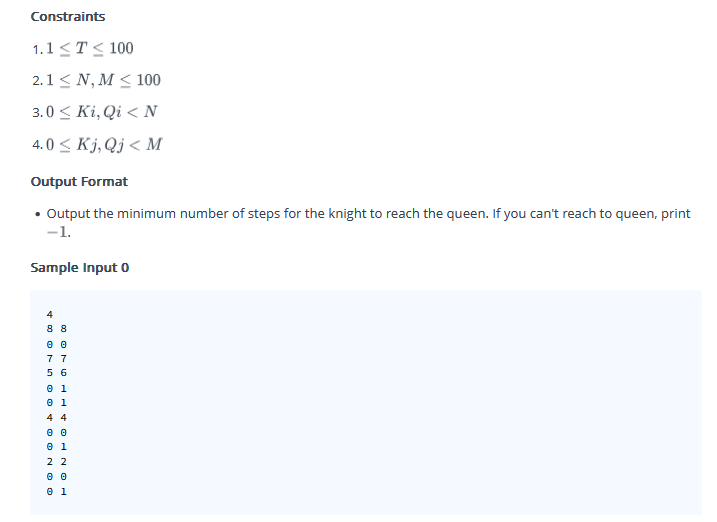
}

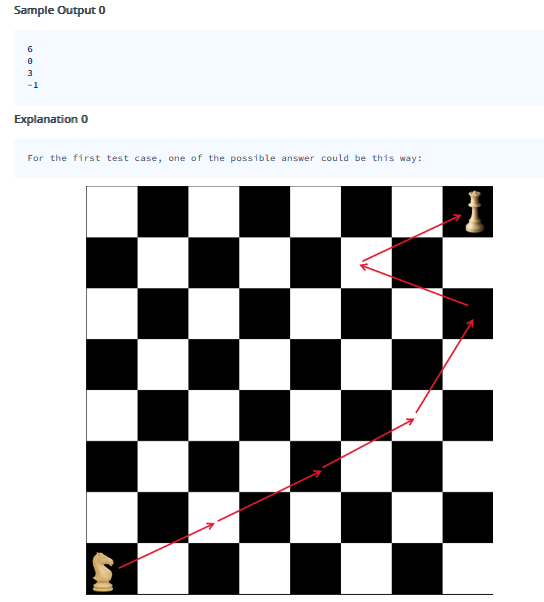
return 0;

}

24. <https://www.hackerrank.com/contests/mid-term-exam-a-introduction-to-algorithms-a-batch-06/challenges/knight-moves-2>







Code:  
#include <bits/stdc++.h>

using namespace std;

vector<pair<int, int>> d = {{1,-2},{2,-1},{-1,-2},{-2,-1},{-2,1},{-1,2},{1,2},{2,1}};

bool vis[105][105];

int level[105][105];

int n,m;

bool valid(int i, int j) {

return (i >= 0 && i < n && j >= 0 && j < m);

}

void bfs(int si, int sj) {

queue<pair<int,int>>q;

q.push({si,sj});

vis[si][sj] = true;

level[si][sj]=0;

while(!q.empty()){

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

q.pop();

int par\_i=par.first;

int par\_j=par.second;

for(int i=0;i<8;i++){

int ci=par\_i+d[i].first;

int cj=par\_j+d[i].second;

if (valid(ci, cj) && !vis[ci][cj]) {

q.push({ci,cj});

level[ci][cj] = level[par\_i][par\_j] + 1;

vis[ci][cj]=true;

}

}

}

}

int main() {

int N;

cin>>N;

while(N--){

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

memset(level,-1,sizeof(level));

cin >> n >> m;

int ki,kj,qi,qj;

cin>>ki>>kj>>qi>>qj;

bfs(ki,kj);

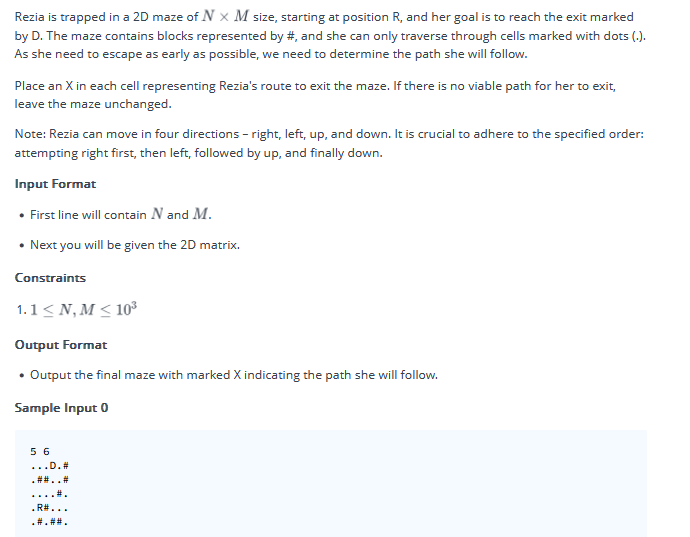
cout<<level[qi][qj]<<endl;

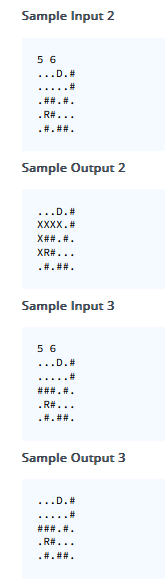
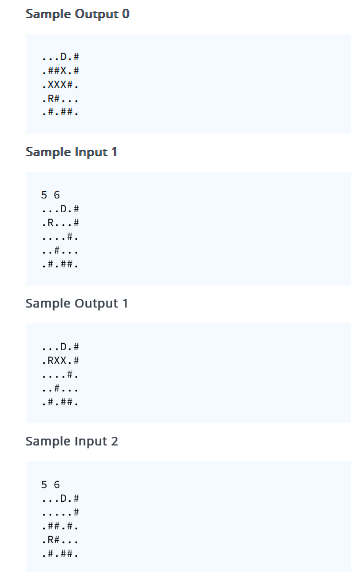
}

return 0;

}

25. <https://www.hackerrank.com/contests/mid-term-exam-a-introduction-to-algorithms-a-batch-06/challenges/maze-19>





#include <bits/stdc++.h>

using namespace std;

vector<pair<int, int>> d = {{0,1}, {0,-1}, {-1,0}, {1,0}};

bool vis[1005][1005];

int level[1005][1005];

pair<int, int> p[1005][1005];

int n, m;

char grid[1005][1005];

bool valid(int i, int j) {

return (i >= 0 && i < n && j >= 0 && j < m && grid[i][j] != '#');

}

void bfs(int si, int sj) {

queue<pair<int,int>> q;

q.push({si, sj});

vis[si][sj] = true;

level[si][sj] = 0;

p[si][sj] = {-1, -1};

while (!q.empty()) {

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

q.pop();

int par\_i = par.first;

int par\_j = par.second;

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

int ci = par\_i + d[i].first;

int cj = par\_j + d[i].second;

if (valid(ci, cj) && !vis[ci][cj]) {

q.push({ci, cj});

level[ci][cj] = level[par\_i][par\_j] + 1;

p[ci][cj] = {par\_i, par\_j};

vis[ci][cj] = true;

}

}

}

}

int main() {

cin >> n >> m;

int xi, xj, yi, yj;

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

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

cin >> grid[i][j];

if (grid[i][j] == 'R') {

xi = i;

xj = j;

} else if (grid[i][j] == 'D') {

yi = i;

yj = j;

}

}

}

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

memset(p, -1, sizeof(p));

bfs(xi, xj);

if (vis[yi][yj]) {

int i = yi, j = yj;

while (p[i][j] != make\_pair(-1, -1)) {

int qi = p[i][j].first;

int qj = p[i][j].second;

if (grid[qi][qj] != 'R' && grid[qi][qj] != 'D') {

grid[qi][qj] = 'X';

}

i = qi;

j = qj;

}

}

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

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

cout << grid[i][j];

}

cout << endl;

}

return 0;

}