ASSIGNMENT 9 (Fast Learner)

Student Name: Ramit Chaturvedi **UID:22BCS15597**
Branch: CSE **Section/Group:614-B**
Semester: 6 **Date of Performance:10-04-25**
Subject Name: AP LAB-II **Subject Code: 22CSP-351**

Q 1) **Number of Islands**

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*.

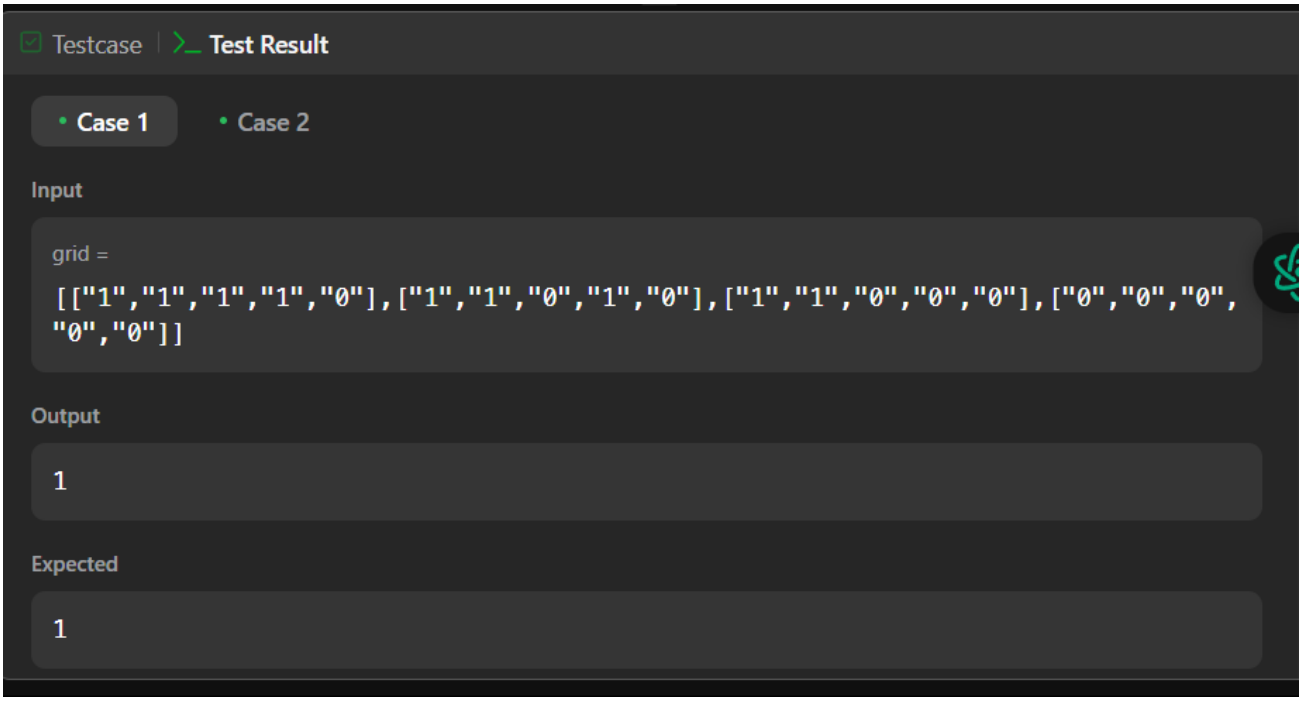
An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Code)

```
class Solution {
 public:
  int numIslands(vector<vector<char>>& grid) {
    constexpr int kDirs[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    const int m = grid.size();
    const int n = grid[0].size();
    int ans = 0;

    auto bfs = [&](int r, int c) {
      queue<pair<int, int>> q{{{r, c}}};
      grid[r][c] = '2';  // Mark '2' as visited.
      while (!q.empty()) {
        const auto [i, j] = q.front();
        q.pop();
        for (const auto& [dx, dy] : kDirs) {
          const int x = i + dx;
          const int y = j + dy;
```

```
            if (x < 0 || x == m || y < 0 || y == n)
              continue;
            if (grid[x][y] != '1')
              continue;
            q.emplace(x, y);
            grid[x][y] = '2';  // Mark '2' as visited.
          }
        }
      };

      for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
          if (grid[i][j] == '1') {
            bfs(i, j);
            ++ans;
          }

      return ans;
    }
};
```

Output)



Q 2) **Word Ladder**

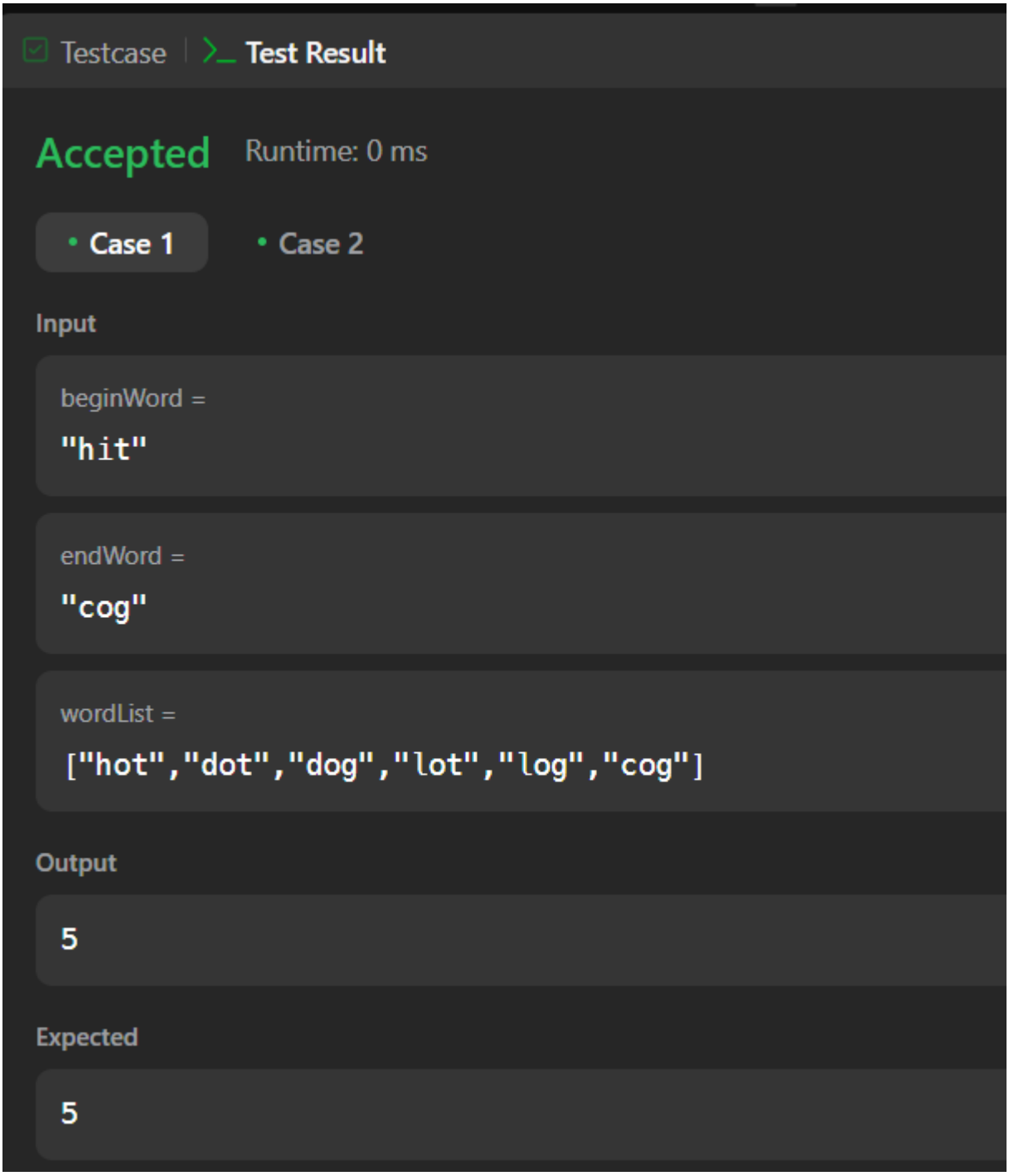
A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k such that:

Code)

```
class Solution {
 public:
  int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
    unordered_set<string> wordSet(wordList.begin(), wordList.end());
    if (!wordSet.contains(endWord))
      return 0;

    queue<string> q{{beginWord}};

    for (int step = 1; !q.empty(); ++step)
      for (int sz = q.size(); sz > 0; --sz) {
```

```
        string word = q.front();
        q.pop();
        for (int i = 0; i < word.length(); ++i) {
          const char cache = word[i];
          for (char c = 'a'; c <= 'z'; ++c) {
            word[i] = c;
            if (word == endWord)
              return step + 1;
            if (wordSet.contains(word)) {
              q.push(word);
              wordSet.erase(word);
            }
          }
          word[i] = cache;
        }
      }

    return 0;
  }
};
```

Output)



3) **Surrounded Regions**

You are given an m x n matrix board containing **letters** 'X' and 'O', **capture regions** that are **surrounded**:

- **Connect**: A cell is connected to adjacent cells horizontally or vertically.

- **Region**: To form a region **connect every** 'O' cell.
- **Surround**: The region is surrounded with 'X' cells if you can **connect the region** with 'X' cells and none of the region cells are on the edge of the board.

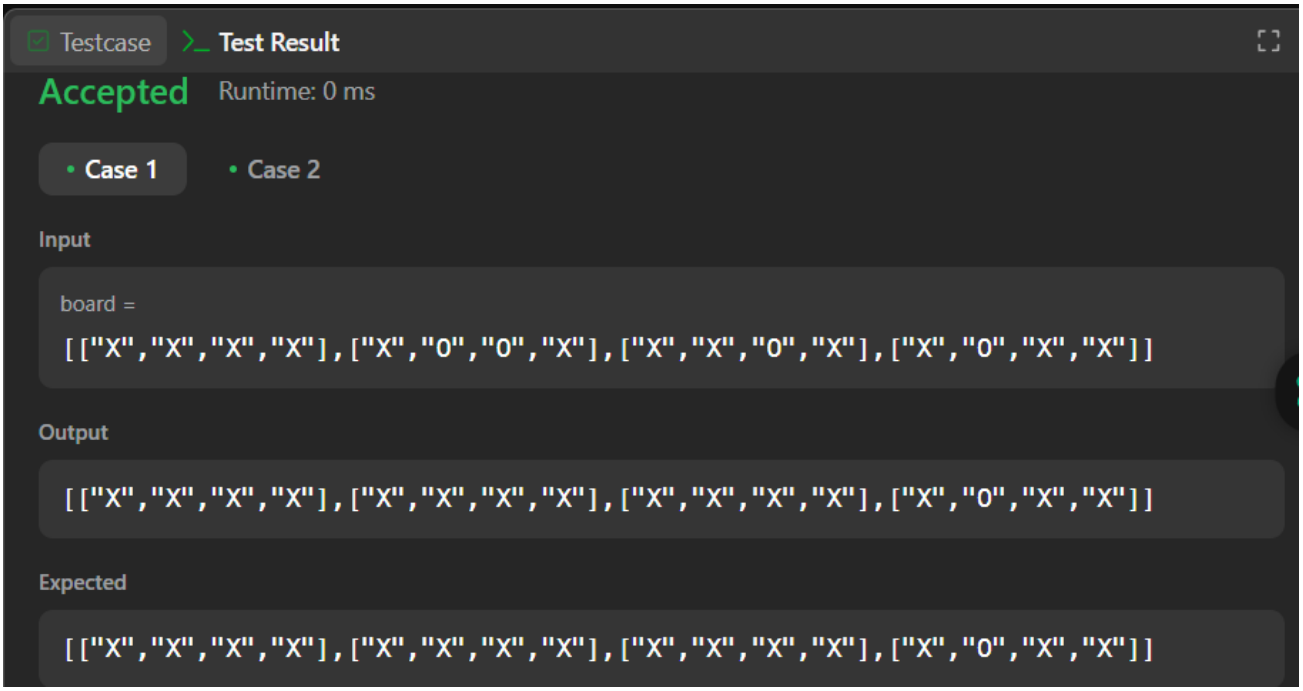
To capture a **surrounded region**, replace all 'O's with 'X's **in-place** within the original board. You do not need to return anything.

Code)

```
class Solution {
 public:
  void solve(vector<vector<char>>& board) {
    if (board.empty())
      return;

    constexpr int kDirs[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    const int m = board.size();
    const int n = board[0].size();

    queue<pair<int, int>> q;

    for (int i = 0; i < m; ++i)
      for (int j = 0; j < n; ++j)
        if (i * j == 0 || i == m - 1 || j == n - 1)
          if (board[i][j] == 'O') {
            q.emplace(i, j);
            board[i][j] = '*';
          }

    // Mark the grids that stretch from the four sides with '*'.
    while (!q.empty()) {
      const auto [i, j] = q.front();
      q.pop();
      for (const auto& [dx, dy] : kDirs) {
        const int x = i + dx;
```

```
        const int y = j + dy;
        if (x < 0 || x == m || y < 0 || y == n)
          continue;
        if (board[x][y] != 'O')
          continue;
        q.emplace(x, y);
        board[x][y] = '*';
      }
    }

    for (vector<char>& row : board)
      for (char& c : row)
        if (c == '*')
          c = 'O';
        else if (c == 'O')
          c = 'X';
  }
};
```

Output)



4) **Binary Tree Maximum Path Sum**

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

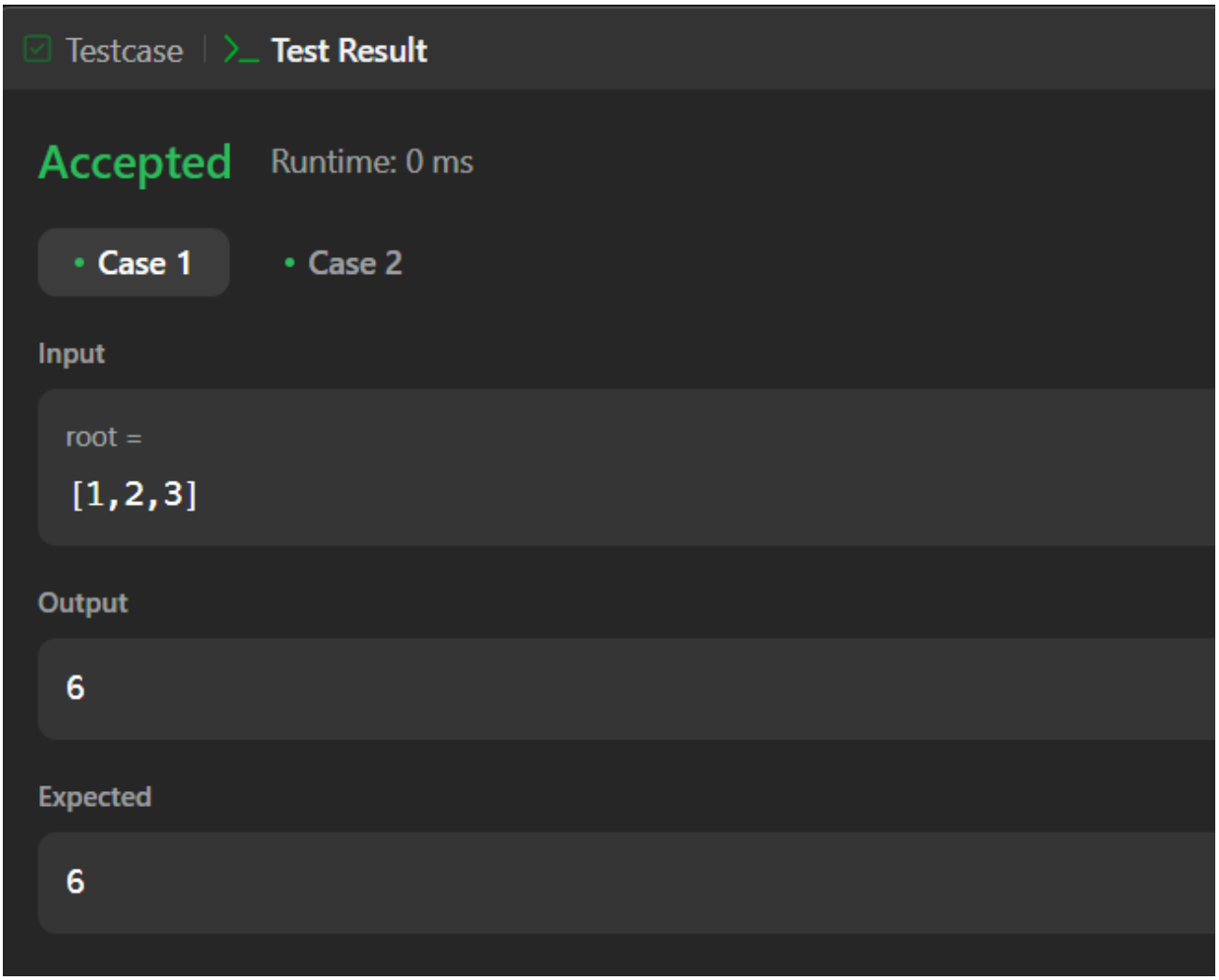
Given the root of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

Code)

```
class Solution {
 public:
  int maxPathSum(TreeNode* root) {
    int ans = INT_MIN;
    maxPathSumDownFrom(root, ans);
    return ans;
  }

 private:
  // Returns the maximum path sum starting from the current root, where
  // root->val is always included.
```

```
  int maxPathSumDownFrom(TreeNode* root, int& ans) {
    if (root == nullptr)
      return 0;

    const int l = max(0, maxPathSumDownFrom(root->left, ans));
    const int r = max(0, maxPathSumDownFrom(root->right, ans));
    ans = max(ans, root->val + l + r);
    return root->val + max(l, r);
  }
};
```

Output)



5) **Number of Provinces**

There are n cities. Some of them are connected, while some are not. If city a is connected

directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

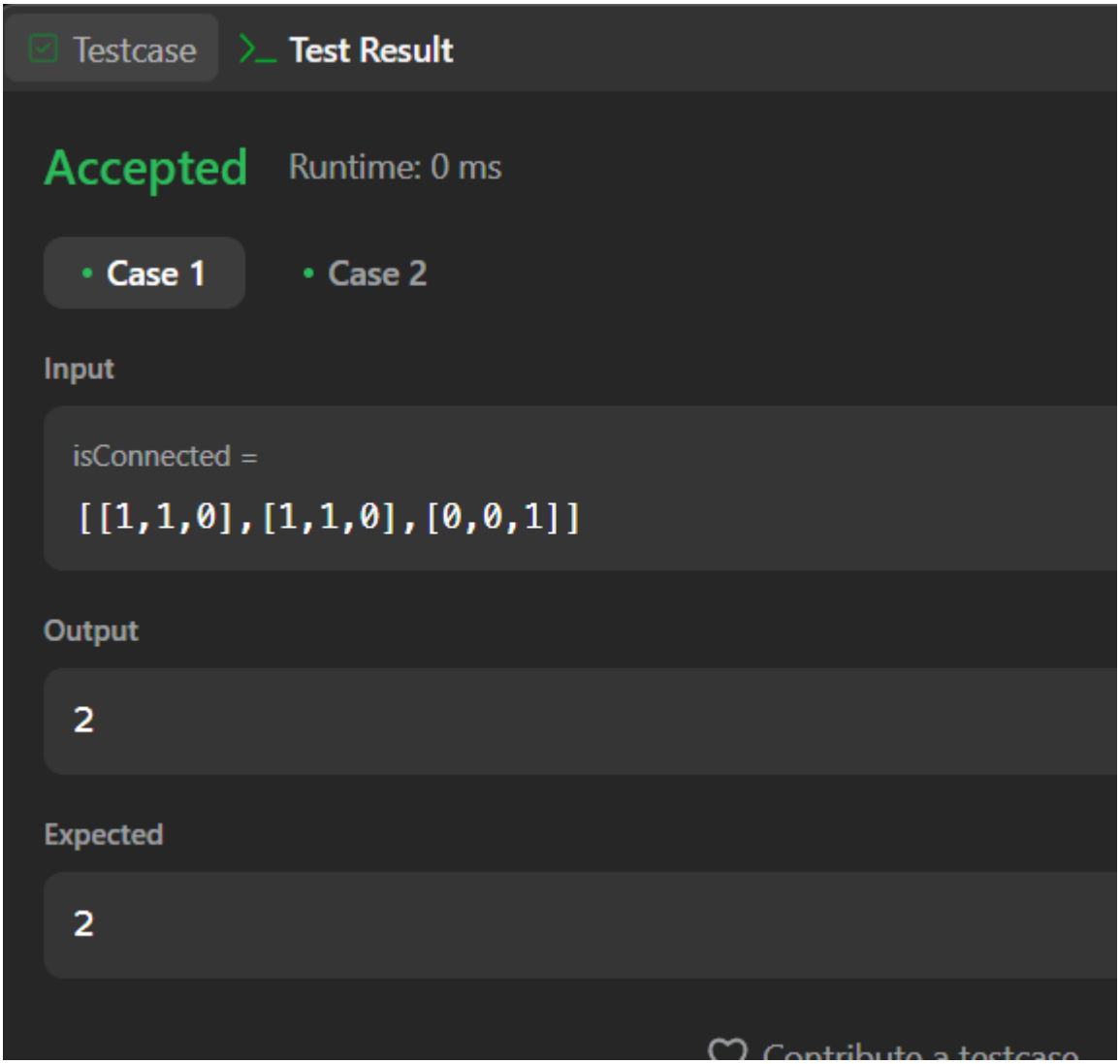
You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i^{th} city and the j^{th} city are directly connected, and isConnected[i][j] = 0 otherwise.

Return *the total number of* ***provinces***.

Code)

```
class UnionFind {
 public:
  UnionFind(int n) : count(n), id(n), rank(n) {
    iota(id.begin(), id.end(), 0);
  }

  void unionByRank(int u, int v) {
    const int i = find(u);
    const int j = find(v);
    if (i == j)
      return;
    if (rank[i] < rank[j]) {
      id[i] = j;
    } else if (rank[i] > rank[j]) {
      id[j] = i;
    } else {
      id[i] = j;
      ++rank[j];
    }
    --count;
  }

  int getCount() const {
    return count;
```

```
  }

 private:
  int count;
  vector<int> id;
  vector<int> rank;

  int find(int u) {
    return id[u] == u ? u : id[u] = find(id[u]);
  }
};

class Solution {
 public:
  int findCircleNum(vector<vector<int>>& isConnected) {
    const int n = isConnected.size();
    UnionFind uf(n);

    for (int i = 0; i < n; ++i)
      for (int j = i; j < n; ++j)
        if (isConnected[i][j] == 1)
          uf.unionByRank(i, j);

    return uf.getCount();
  }
};Output)
```


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
isConnected =
[[1,1,0],[1,1,0],[0,0,1]]
Output
2
Expected
2
Contribute a testcase