DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

EXPERIMENT - 9

Student Name: Sameer **UID:** 22BCS15631
Branch: Computer Science & Engineering **Section/Group:** IOT-614/B
Semester: 6th **Date of Performance:** 10/04/2025
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSP-351

Q.1. 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.

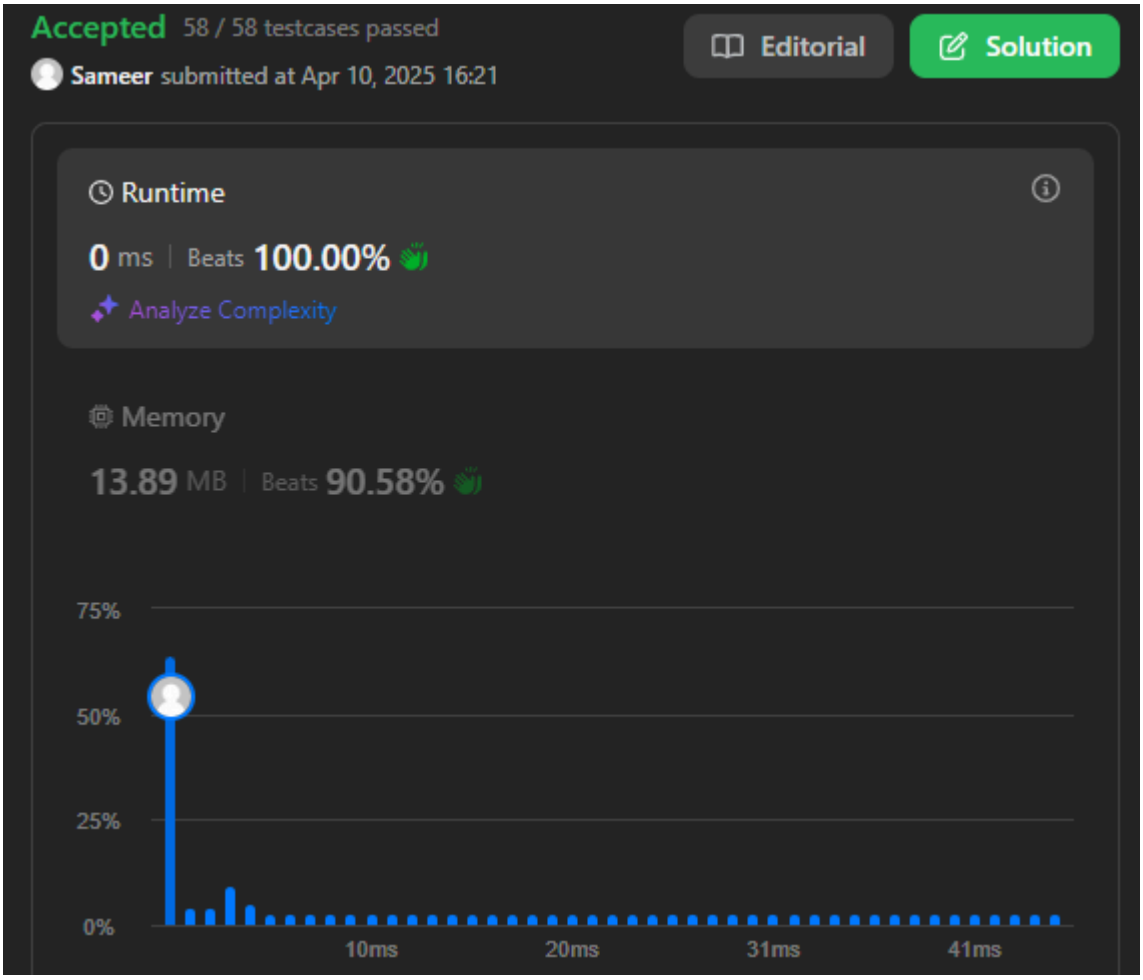
Code:

```
class Solution {
public:
    void dfs(vector<vector<char>>& board, int i, int j) {
        int m = board.size();
        int n = board[0].size();

        if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] != 'O')
            return;

        board[i][j] = '#';

        dfs(board, i + 1, j);
        dfs(board, i - 1, j);
        dfs(board, i, j + 1);
        dfs(board, i, j - 1);
    }

    void solve(vector<vector<char>>& board) {
```

```
        if (board.empty())
            return;

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

        for (int i = 0; i < m; ++i) {
            if (board[i][0] == 'O')
                dfs(board, i, 0);

            if (board[i][n - 1] == 'O')
                dfs(board, i, n - 1);
        }

        for (int j = 0; j < n; ++j) {
            if (board[0][j] == 'O')
                dfs(board, 0, j);

            if (board[m - 1][j] == 'O')
                dfs(board, m - 1, j);
        }

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == 'O')
                    board[i][j] = 'X';

                else if (board[i][j] == '#')
                    board[i][j] = 'O';
            }
        }
    }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Input
board =
[["X","X","X","X"],["X","0","0","X"],["X","X","0","X"],["X","0","X","X"]]
Output
[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","0","X","X"]]
Expected
[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","0","X","X"]]


Accepted
58 / 58 testcases passed
Sameer submitted at Apr 10, 2025 16:21
Editorial
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
13.89 MB | Beats 90.58%
75%
50%
25%
0%
10ms
20ms
31ms
41ms

Q.2. Word Ladder

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.

Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.

sk == endWord

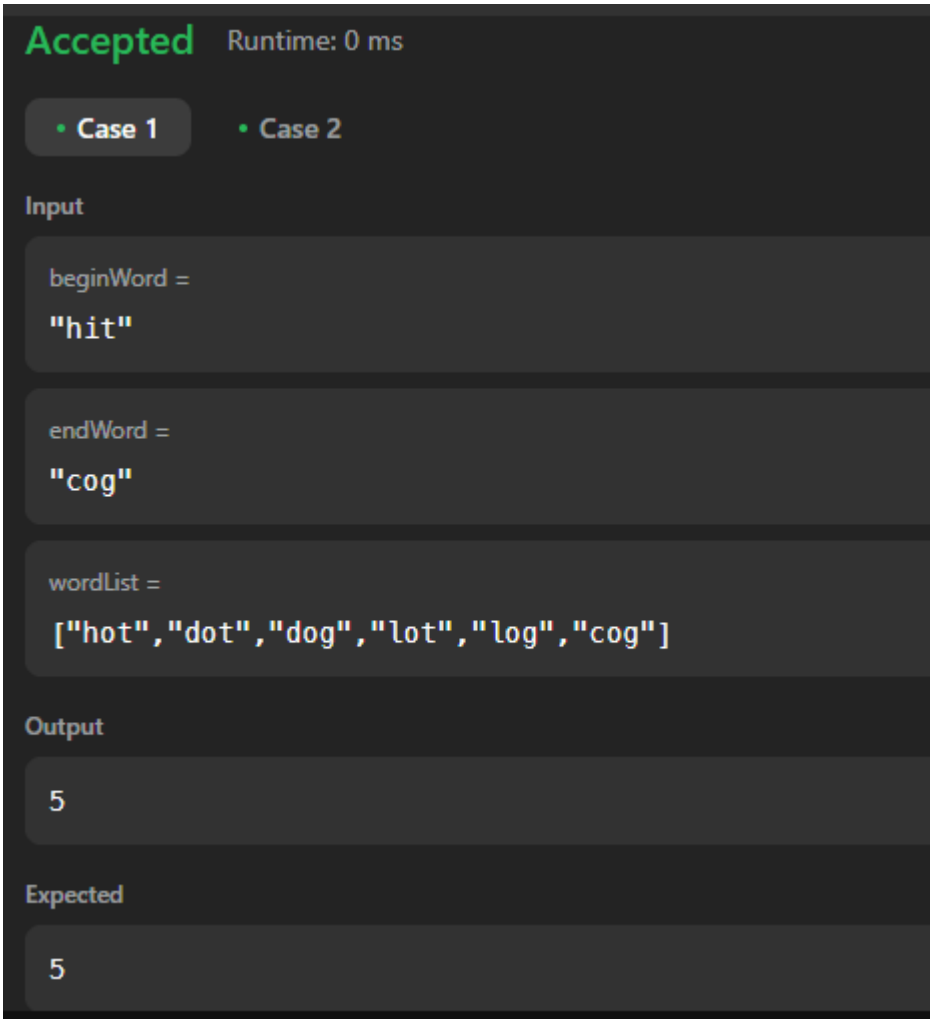
Given two words, beginWord and endWord, and a dictionary wordList, return the number of words in the shortest transformation sequence from beginWord to endWord, or 0 if no such sequence exists.

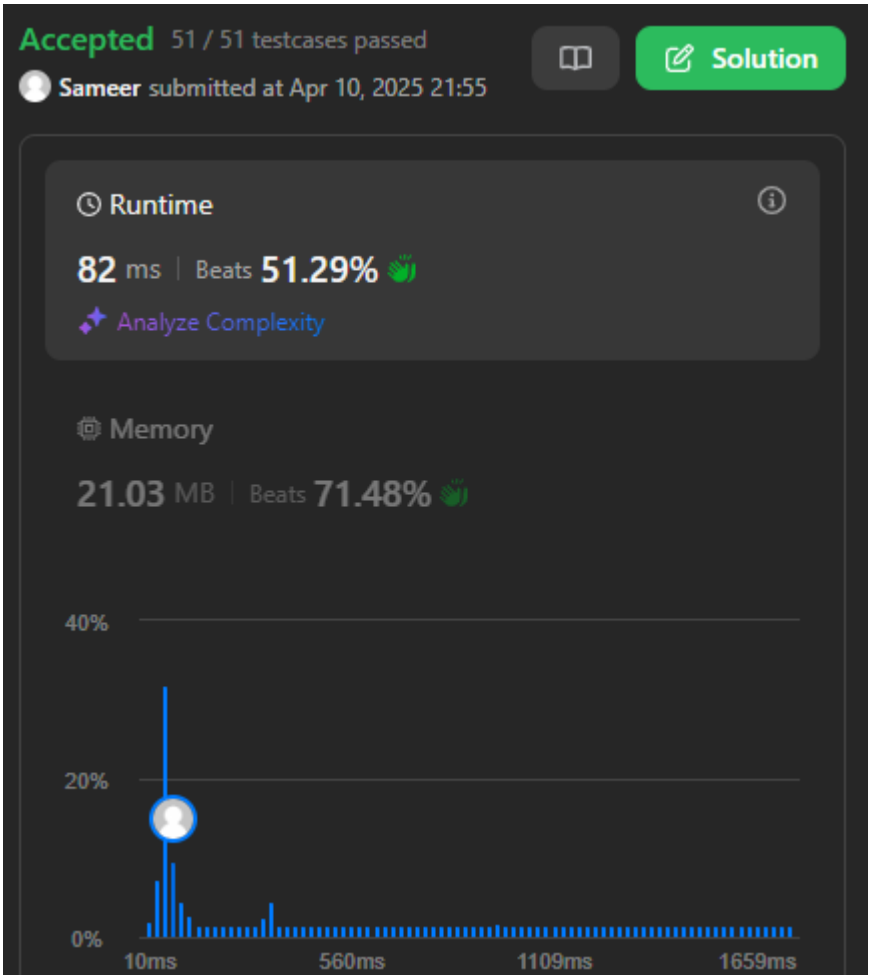
Code:

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

        queue<pair<string, int>> q;
        q.push({beginWord, 1});

        while (!q.empty()) {
            auto [currentWord, steps] = q.front();
            q.pop();

            for (int i = 0; i < currentWord.length(); ++i) {
                string temp = currentWord;

                for (char c = 'a'; c <= 'z'; ++c) {
                    temp[i] = c;

                    if (temp == currentWord)
                        continue;

                    if (temp == endWord)
                        return steps + 1;
```

```
                if (wordSet.find(temp) != wordSet.end()) {
                    q.push({temp, steps + 1});
                    wordSet.erase(temp);
                }
            }
        }
    }

    return 0;
  }
};
```

Output:




Accepted 51 / 51 testcases passed
Sameer submitted at Apr 10, 2025 21:55
Solution
Runtime
82 ms | Beats 51.29%
Analyze Complexity
Memory
21.03 MB | Beats 71.48%
40%
20%
0%
10ms
560ms
1109ms
1659ms

Q.3. Course Schedule

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

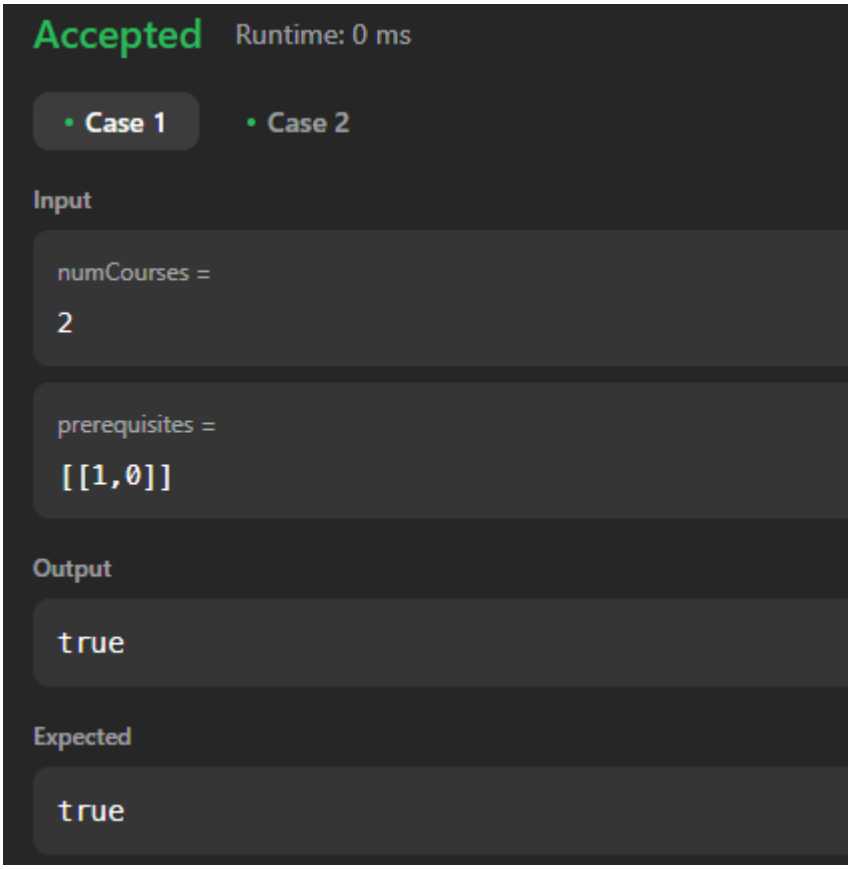
Return true if you can finish all courses. Otherwise, return false.

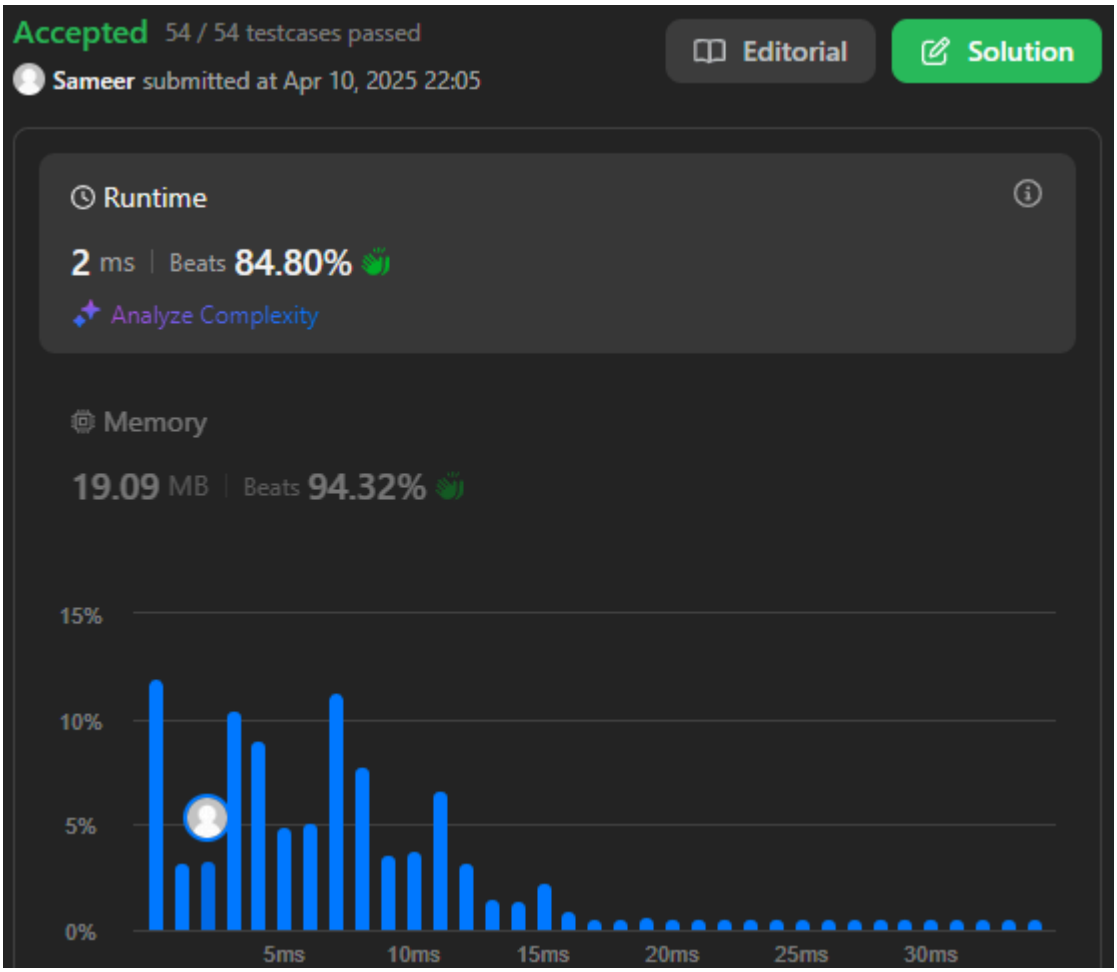
Code:

```
class Solution {
public:
   bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
      vector<vector<int>> adj(numCourses);
      vector<int> indegree(numCourses, 0);

      for (auto& pre : prerequisites) {
         adj[pre[1]].push_back(pre[0]);
         indegree[pre[0]]++;
      }

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

      int count = 0;
      while (!q.empty()) {
         int curr = q.front();
         q.pop();
         count++;

         for (int neighbor : adj[curr]) {
            indegree[neighbor]--;

            if (indegree[neighbor] == 0) {
               q.push(neighbor);
```

```
            }
        }
    }

    return count == numCourses;
  }
};
```

Output:




Accepted 54 / 54 testcases passed
Sameer submitted at Apr 10, 2025 22:05
Editorial
Solution
Runtime
2 ms | Beats 84.80%
Analyze Complexity
Memory
19.09 MB | Beats 94.32%
15%
10%
5%
0%
5ms
10ms
15ms
20ms
25ms
30ms

Q.4. Longest Increasing Path in a Matrix

Given an m x n integers matrix, return the length of the longest increasing path in matrix.

From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

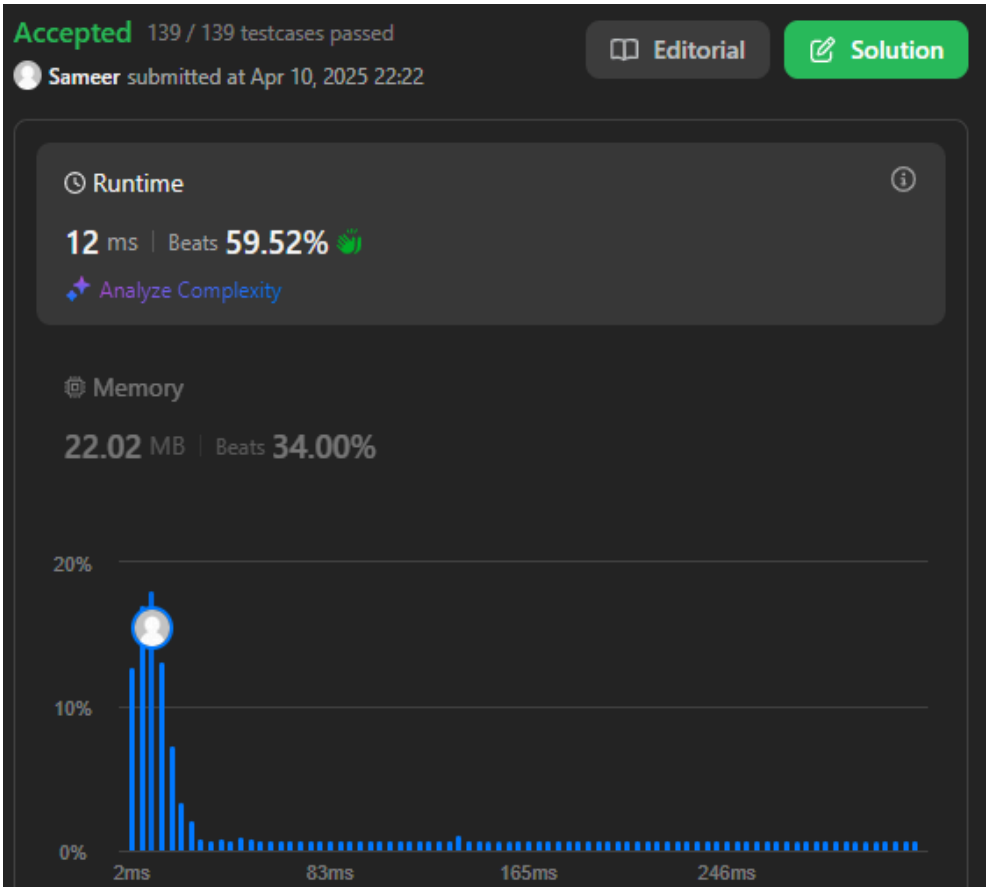
Code:

```
class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        int m = matrix.size(), n = matrix[0].size();
        vector<vector<int>> dp(m, vector<int>(n, 0));

        int res = 0;
        vector<int> dirs = {0, 1, 0, -1, 0};

        function<int(int, int)> dfs = [&](int i, int j) {
            if (dp[i][j]) {
                return dp[i][j];
            }

            int maxLen = 1;
            for (int d = 0; d < 4; ++d) {
                int x = i + dirs[d], y = j + dirs[d + 1];

                if (x >= 0 && x < m && y >= 0 && y < n &&
                    matrix[x][y] > matrix[i][j]) {
                    maxLen = max(maxLen, 1 + dfs(x, y));
                }
            }
            return dp[i][j] = maxLen;
        };

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                res = max(res, dfs(i, j));
            }
        }

        return res;
```

```
    }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
matrix =
[[9,9,4],[6,6,8],[2,1,1]]
Output
4
Expected
4


Accepted
139 / 139 testcases passed
Sameer submitted at Apr 10, 2025 22:22
Editorial
Solution
Runtime
12 ms | Beats 59.52%
Analyze Complexity
Memory
22.02 MB | Beats 34.00%
20%
10%
0%
2ms
83ms
165ms
246ms