

Experiment 9 (Advance Programming)

Submitted By: Ankit Kharb

Class: IOT_614(B)

UID: 22BCS16964

Q-Number of Islands

Problem:

Given an $m \times n$ 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

Solution:

```
class Solution {
```

```
public:
```

```
    void dfs(vector<vector<char>>& grid, int i, int j) {
```

```
        int m = grid.size();
```

```
        int n = grid[0].size();
```

```
        // Boundary or water check
```

```

        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') {
            return;
        }

        // Mark the current land cell as visited
        grid[i][j] = '0';

        // Explore all 4 directions
        dfs(grid, i - 1, j); // up
        dfs(grid, i + 1, j); // down
        dfs(grid, i, j - 1); // left
        dfs(grid, i, j + 1); // right
    }

    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty()) return 0;

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

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

```

```

    }

}

return count;

}

};

```

The screenshot shows the LeetCode interface for the problem "200. Number of Islands". The problem description states: "Given an $m \times n$ 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

The solution is implemented in C++ using a Depth-First Search (DFS) approach. The code defines a `Solution` class with a `dfs` method that recursively visits all land cells connected to the starting point, marking them as visited by setting them to '0'. The `dfs` method returns 1 if the current cell is land, and 0 otherwise. The `numIslands` method iterates through the grid and counts the number of times `dfs` is called, which corresponds to the number of islands.

```

1 class Solution {
2 public:
3     void dfs(vector<vector<char>>& grid, int i, int j) {
4         int m = grid.size();
5         int n = grid[0].size();
6
7         // Boundary or water check
8         if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') {
9             return;
10        }
11    }

```

The test results show that the solution is "Accepted" with a runtime of 0 ms. The input for the test case is `grid =`.

Q-Word Ladder

Problem:

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:

- Every adjacent pair of words differs by a single letter.
- Every s_i for $1 \leq i \leq k$ is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- $s_k == \text{endWord}$

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.*

Solution:

```
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; // Pair of (current word, current depth)  
        q.push({beginWord, 1});  
  
        while (!q.empty()) {  
            auto [word, depth] = q.front();  
            q.pop();  
  
            if (word == endWord) return depth;  
  
            for (int i = 0; i < word.size(); ++i) {  
                string nextWord = word;  
                for (char c = 'a'; c <= 'z'; ++c) {  
                    nextWord[i] = c;  
                    if (wordSet.find(nextWord) != wordSet.end()) {  
                        q.push({nextWord, depth + 1});  
                        wordSet.erase(nextWord); // Mark as visited
```

```

    }

    }

    }

    }

    return 0; // No transformation sequence found
}

};

```

127. Word Ladder Solved ✓

Hard Topics Companies

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 s_i for $1 \leq i \leq k$ is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- $s_k == endWord$

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.

```

C++
20
21
22
23
24
25
26
27
28
29
30
if (wordSet.find(nextWord) != wordSet.end()) {
    q.push({nextWord, depth + 1});
    wordSet.erase(nextWord); // Mark as visited
}
}
}
return 0; // No transformation sequence found
};

```

Ln 30, Col 3 Saved Run Submit

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

beginWord =

Q-Surrounded Regions

You are given an $m \times n$ 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.

Solution:

```
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 || i >= m || j < 0 || j >= n || board[i][j] != 'O') {
            return;
        }

        board[i][j] = 'S'; // Temporarily mark as safe

        // Explore 4 directions
        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();

// Step 1: Mark border-connected 'O's
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);
}

// Step 2: Flip the regions
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        if (board[i][j] == 'O') {
            board[i][j] = 'X'; // Captured
        } else if (board[i][j] == 'S') {
            board[i][j] = 'O'; // Restore safe
        }
    }
}
};

```

130. Surrounded Regions Solved ✓

Medium Topics Companies

You are given an $m \times n$ 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.

Example 1:

9.1K 217 110 Online

```

38
39
40
41
42
43
44
45
46
47
48
C++
Auto
for (int j = 0; j < n; ++j) {
    if (board[i][j] == 'O') {
        board[i][j] = 'X'; // Captured
    } else if (board[i][j] == 'S') {
        board[i][j] = 'O'; // Restore safe
    }
}
};

```

Ln 48, Col 1 Saved Run Submit

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

board =

Q-Binary Tree Maximum Path Sum

Problem:

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*.

Solution:

```
class Solution {
```

```
public:
```

```
    int maxSum = INT_MIN;
```

```
    int maxGain(TreeNode* node) {
```

```
        if (!node) return 0;
```

```
        // Only consider positive gains; discard negative paths
```

```
        int leftGain = max(maxGain(node->left), 0);
```



```
int rightGain = max(maxGain(node->right), 0);
```

```
// Max path using this node as root of the path
```

```
int localMax = node->val + leftGain + rightGain;
```

```
// Update global max
```

```
maxSum = max(maxSum, localMax);
```

```
// Return gain to be used in parent call (only one path allowed)
```

```
return node->val + max(leftGain, rightGain);
```

```
}
```

```
int maxPathSum(TreeNode* root) {
```

```
    maxGain(root);
```

```
    return maxSum;
```

```
}
```

```
}
```

The screenshot shows the LeetCode interface for problem 124, "Binary Tree Maximum Path Sum". The problem is marked as "Solved" with a green checkmark. The description explains that a path in a binary tree is a sequence of nodes where each pair of adjacent nodes has an edge connecting them, and a node can only appear at most once. The path sum is the sum of the node's values in the path. The goal is to return the maximum path sum of any non-empty path.

Example 1: A diagram shows a single node with the value 1, representing a path sum of 1.

The code editor on the right shows the following C++ code:

```
27 maxSum = max(maxSum, localMax);
28
29 // Return gain to be used in parent call (only one path allowed)
30 return node->val + max(leftGain, rightGain);
31 }
32
33 int maxPathSum(TreeNode* root) {
34     maxGain(root);
35     return maxSum;
36 }
37 ;
```

The bottom of the interface shows a "Testcase" section with a "Test Result" table. The "Expected" value is 6. A note indicates that the visualization is not supported.

Q-Friend Circles

Problem:

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 $n \times n$ 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*.

Solution:

```
class Solution {
```

```
public:
```

```
    void dfs(vector<vector<int>>& isConnected, vector<bool>& visited, int city) {
```

```
        visited[city] = true;
```

```
        for (int j = 0; j < isConnected.size(); ++j) {
```

```
            if (isConnected[city][j] == 1 && !visited[j]) {
```

```
                dfs(isConnected, visited, j);
```

```
            }
```

```
        }
```

```
    }
```

```
    int findCircleNum(vector<vector<int>>& isConnected) {
```

```
        int n = isConnected.size();
```

```
        vector<bool> visited(n, false);
```

```
        int provinces = 0;
```

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

```

        if (!visited[i]) {
            dfs(isConnected, visited, i);
            ++provinces;
        }
    }

    return provinces;
}

}; vector<vector<int>> skyline = getSkyline(buildings);

cout << "Skyline: ";
for (const auto& point : skyline) {
    cout << "[" << point[0] << ", " << point[1] << "]" ";
}

cout << endl;

return 0;
}

```

The screenshot shows the LeetCode interface for the problem "547. Number of Provinces". The problem is marked as "Solved" and "Medium" difficulty. The description states: "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 $n \times n$ 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**." Example 1 is provided at the bottom left.

The code editor on the right shows the following C++ code:

```

16
17     for (int i = 0; i < n; ++i) {
18         if (!visited[i]) {
19             dfs(isConnected, visited, i);
20             ++provinces;
21         }
22     }
23     return provinces;
24 }
25
26

```

The test result section shows "Accepted" with a runtime of 0 ms. Below it, there are two test cases: "Case 1" and "Case 2". The input for Case 1 is shown as "isConnected =".

