200. Number of Islands

Medium | Depth-First Search | Breadth-First Search | Union Find | Array | Matrix | Leetcode Link

Problem Description

In this problem, we are provided with a two-dimensional grid where each cell has a binary value; either '1' representing land or '0' representing water. The task is to calculate the number of islands on this grid. An island is defined as a cluster of adjacent '1's, and two cells are considered adjacent if they are directly north, south, east, or west of each other (no diagonal connection). The boundary of the grid is surrounded by water, which means there are '0's all around the grid.

To sum up, we need to identify the groups of connected '1's within the grid and count how many separate groups (islands) are present.

Intuition

The intuition behind the solution is to scan the grid cell by cell. When we encounter a '1', we treat it as a part of an island which has not been explored yet. To mark this exploration, we perform a Depth-First Search (DFS) starting from that cell, turning all connected '1's to '0's, effectively "removing" the island from the grid to prevent counting it more than once.

The core approach involves the following steps:

1. Iterate over each cell of the grid.
2. When a cell with value '1' is found, increment our island count as we've discovered a new island.
3. Initiate a DFS from that cell, changing all connected '1's (i.e., the entire island) to '0's to mark them as visited.
4. Continue the grid scan until all cells are processed.

During the DFS, we explore in four directions: up, down, left, and right. We ensure the exploration step remains within grid bounds to avoid index errors.

This approach allows us to traverse each island only once, and by marking cells as visited, we avoid recounting any part of an island. Thus, we effectively count each distinct island just once, leading us to the correct solution.

Solution Approach

The solution approach utilizes Depth-First Search (DFS) as the main algorithm for solving the problem of finding the number of islands in the grid. Here's a step-by-step explanation of the implementation:

1. Define a nested function called `dfs` inside the `numIslands` method which takes the grid coordinates `(i, j)` of a cell as parameters. This function will be used to perform the DFS from a given starting cell that is part of an island.

2. The `dfs` function first sets the current cell's value to '0' to mark it as visited. This prevents the same land from being counted again when iterating over other parts of the grid.

3. Then, for each adjacent direction defined by `dirs` (which are the four possible movements: up, down, left, and right), the `dfs` function calculates the adjacent cell's coordinates `(x, y)`.

4. If the adjacent cell `(x, y)` is within the grid boundaries (`0 <= x < m` and `0 <= y < n`) and it's a '1' (land), the `dfs` function is recursively called for this new cell.

5. Outside of the `dfs` function, the main part of the `numIslands` method initializes an `ans` counter to keep track of the number of islands found.

6. The variable `dirs` is a tuple containing the directions used in the DFS to traverse the grid in a cyclic order. By using `pairwise(dirs)`, we always get a pair of directions that represent a straight line movement (either horizontal or vertical) without diagonal moves.

7. We iterate over each cell of the grid using a nested loop. When we encounter a '1', we call the `dfs` function on that cell, marking the entire connected area, and then increment the `ans` counter since we have identified an entire island.

8. At the end of the nested loops, we have traversed the entire grid and recursively visited all connected '1's, marking them as '0's, thus avoiding multiple counts of the same land.

9. The `ans` variable now contains the total number of islands, which is returned as the final answer.

The main data structures used in the implementation are:

- The input `grid`, which is a two-dimensional list used to represent the map.
- The `dirs` tuple, which stores the possible directions of movement within the grid.

By using DFS, we apply a flood fill technique, similar to the "bucket" fill tool in paint programs, where we fill an entire connected region with a new value. In our case, we fill connected '1's with '0's to "remove" the island from being counted more than once. This pattern ensures that each separate island is counted exactly one time.

Example Walkthrough

Imagine we are given the following grid:

```
grid = [
    ["1","1","0","0","0"],
    ["1","1","0","0","0"],
    ["0","0","1","0","0"],
    ["0","0","0","1","1"]
]
```

Here is a step-by-step walkthrough of the solution approach for the given example:

1. We start from the top-left corner of the grid and iterate through each cell.

2. The first cell `grid[0][0]` is '1', indicating land, so we increase our island count to `1`.

3. We then call the `dfs` function on this cell which will mark all the connected '1's as '0's as follows:

   - `dfs(0, 0)` changes `grid[0][0]` to '0'. It then recursively explores its neighbors which are `grid[0][1]`, `grid[1][0]` (since `grid[-1][0]` and `grid[0][-1]` are out of bounds).
   - `dfs(0, 1)` changes `grid[0][1]` to '0'. Its neighbors `grid[0][2]` and `grid[1][1]` are then explored but they are already '0' and '1' respectively but since `grid[1][1]` is connected, `dfs(1, 1)` is called.
   - `dfs(1, 1)` changes `grid[1][1]` to '0'. There are no more '1's connected to it directly.

   This results in the grid becoming:

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

4. Continuing the iteration, we find the next '1' at `grid[2][2]`. We increment our island count to `2` and perform `dfs(2, 2)` which results in:

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

5. Finally, the last '1' is encountered at `grid[3][3]`. After incrementing the island count to `3`, we perform `dfs(3, 3)` which also changes `grid[3][4]` to '0' as it is connected:

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

   With no more '1's left in the grid, we have identified all the islands - a total of `3`.

6. Now that we have processed the entire grid, we return the `ans` counter value which is `3`.

This walkthrough demonstrates the application of DFS in marking each discovered island and preventing overlap in counting by converting all connected '1' cells to '0'. By the completion of the iteration over the entire grid, we have the total number of separate islands encountered.

Python Solution

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def dfs(row, col):
            # Mark the current cell as '0' to indicate the land is visited
            grid[row][col] = '0'
            # Explore all four directions from the current cell
            for dx, dy in zip(directions[:-1], directions[1:]):
                new_row, new_col = row + dx, col + dy
                # Check if the new cell is within grid bounds and is land ('1')
                if 0 <= new_row < rows and 0 <= new_col < cols and grid[new_row][new_col] == '1':
                    # Perform DFS on the new cell
                    dfs(new_row, new_col)

        # Initialize count of islands
        island_count = 0
        # Define the directions to explore
        directions = (-1, 0, 1, 0, -1)
        # Get the dimensions of the grid
        rows, cols = len(grid), len(grid[0])
        # Iterate over each cell in the grid
        for row in range(rows):
            for col in range(cols):
                # If the cell is land ('1'), it's a new island
                if grid[row][col] == '1':
                    # Perform DFS to mark all connected land for the current island
                    dfs(row, col)
                    # Increment the island count
                    island_count += 1
        # Return the total number of islands
        return island_count

```

Java Solution

```java
class Solution {
    // Define grid and its dimensions
    private char[][] grid;
    private int numRows;
    private int numCols;

    // Method to count the number of islands in the given grid
    public int numIslands(char[][] grid) {
        numRows = grid.length;
        numCols = grid[0].length;
        this.grid = grid;

        int numIslands = 0; // Initialize island count

        // Iterate through each cell in the grid
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // If cell contains '1', it is part of an island
                if (grid[i][j] == '1') {
                    // Use DFS to mark the entire island as visited
                    depthFirstSearch(i, j);
                    // Increase the island count
                    ++numIslands;
                }
            }
        }
        return numIslands;
    }

    // Helper method to perform DFS to mark all cells of an island as visited
    private void depthFirstSearch(int row, int col) {
        // Mark the current cell as visited by setting it to '0'
        grid[row][col] = '0';

        // Array to facilitate the exploration of adjacent directions (up, right, down, left)
        int[] directions = {-1, 0, 1, 0, -1};

        // Explore all 4 adjacent directions
        for (int k = 0; k < 4; ++k) {
            int newRow = row + directions[k];
            int newCol = col + directions[k + 1];
            // Check boundaries and if the adjacent cell is part of an island
            if (newRow >= 0 && newRow < numRows && newCol >= 0 && newCol < numCols && grid[newRow][newCol] == '1') {
                // Continue DFS exploration for the adjacent cell
                depthFirstSearch(newRow, newCol);
            }
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
using namespace std;

class Solution {
public:
    // Function to count the number of islands in a given grid
    int numIslands(vector<vector<char>>& grid) {
        int rowCount = grid.size();      // Number of rows in the grid
        int colCount = grid[0].size();   // Number of columns in the grid
        int islandCount = 0;             // Number of islands found

        // Directions array for moving up, right, down, left
        int directions[5] = {-1, 0, 1, 0, -1};

        // Depth-First Search (DFS) to traverse the island and mark visited parts
        function<void(int, int)> dfs = [&](int row, int col) {
            grid[row][col] = '0'; // Mark the current cell as visited
            // Traverse adjacent cells
            for (int k = 0; k < 4; ++k) {
                int newRow = row + directions[k];
                int newCol = col + directions[k + 1];
                // Check if the new position is within bounds and has a '1' (unvisited land)
                if (newRow >= 0 && newRow < rowCount && newCol >= 0 && newCol < colCount && grid[newRow][newCol] == '1') {
                    dfs(newRow, newCol); // Recursively call DFS for the adjacent cell
                }
            }
        };

        // Iterate through the entire grid
        for (int row = 0; row < rowCount; ++row) {
            for (int col = 0; col < colCount; ++col) {
                // If the cell contains a '1', it is a new island
                if (grid[row][col] == '1') {
                    dfs(row, col);     // Perform DFS to mark all connected lands
                    islandCount++;     // Increment island count
                }
            }
        }
        return islandCount; // Return the total count of islands
    }
};

```

Typescript Solution

```typescript
function numIslands(grid: string[][]): number {
    // m is the number of rows in the grid
    const numberOfRows = grid.length;
    // n is the number of columns in the grid (assuming the grid is not empty)
    const numberOfColumns = grid[0].length;
    // ans will hold the number of islands found
    let numberOfIslands = 0;

    // The Depth-First Search function, which marks visited land sections as '0'
    function depthFirstSearch(row: number, column: number) {
        // Set the current location to '0' to mark as visited
        grid[row][column] = '0';
        // Array representing the 4 directions (up, right, down, left)
        const directions = [-1, 0, 1, 0, -1];

        // Iterate over each direction
        for (let k = 0; k < 4; ++k) {
            // Calculate the new coordinates based on the current direction
            const newRow = row + directions[k];
            const newColumn = column + directions[k + 1];

            // Check if the new coordinates are within bounds and the cell contains '1'
            if (newRow >= 0 && newRow < numberOfRows && newColumn >= 0 && newColumn < numberOfColumns && grid[newRow][newColumn] ===
                // If so, perform DFS on the adjacent cell
                depthFirstSearch(newRow, newColumn);
            }
        }
    }

    // Iterate over every cell in the grid
    for (let row = 0; row < numberOfRows; ++row) {
        for (let column = 0; column < numberOfColumns; ++column) {
            // If the cell contains '1' (land), an island is found
            if (grid[row][column] === '1') {
                // Perform DFS to mark the entire island
                depthFirstSearch(row, column);
                // Increment the island count
                numberOfIslands++;
            }
        }
    }

    // Return the total number of islands found
    return numberOfIslands;
}

```

Time and Space Complexity

The given code implements the Depth-First Search (DFS) algorithm to count the number of islands (`'1'`) in a grid. The time complexity and space complexity analysis are as follows:

Time Complexity

The time complexity of the code is `O(m * n)`, where `m` is the number of rows in the grid, and `n` is the number of columns. This is because the algorithm must visit each cell in the entire grid once to ensure all parts of the islands are counted and marked. The DFS search is invoked for each land cell (`'1'`) that hasn't yet been visited, and it traverses all its adjacent land cells. Although the outer loop runs for `m * n` iterations, each cell is visited once by the DFS, ensuring that the overall time complexity remains linear concerning the total number of cells.

Space Complexity

The space complexity is `O(m * n)` in the worst case. This worst-case scenario occurs when the grid is filled with land cells (`'1'`), where the depth of the recursion stack (DFS) potentially equals the total number of cells in the grid if we are dealing with one large island. Since the DFS can go as deep as the largest island, and in this case, that's the entire grid, the stack space used by the recursion is proportionate to the total number of cells.