1905. Count Sub Islands

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

Problem Description

In this problem, you are given two matrices `grid1` and `grid2` of the same dimensions, `m x n`, where each cell contains a `0` or a `1`. The value `1` represents a piece of land, while `0` represents water. An island is defined as a group of `1`s that are connected horizontally or vertically. The task is to count the number of islands in `grid2` that are sub-islands. A sub-island in `grid2` is characterized by every bit of land (`1`) that is also part of an island in `grid1`. In other words, we want to count the islands in `grid2` where every land cell of that island is also land in `grid1`.

Intuition

The solution to this problem lies in traversing the islands in `grid2` and checking whether they are completely contained within the islands of `grid1`. Depth-First Search (DFS) is a fitting approach for traversing islands, as we can explore all connected land cells (1s) from any starting land cell.

The DFS function will be the core of our approach. It is recursively called on adjacent land cells of `grid2`. For each land cell in `grid2`, the function checks whether the corresponding cell in `grid1` is also a land cell. If not, this land cell does not fulfill the condition of being part of a sub-island.

During each DFS call, we mark the visited cells in `grid2` as water (by setting them to `0`) to avoid revisiting them. The DFS will return `False` if any part of the current 'island' in `grid2` is not land in `grid1`, indicating that this island cannot be considered a sub-island. Only if all parts of an island in `grid2` are also land in `grid1`, it will return `True`.

This process is repeated for all cells in `grid2`. The sum of instances where DFS returns `True` gives us the count of sub-islands. This approach effectively traverses through all possible islands in `grid2`, and by comparing against `grid1`, it determines the count of sub-islands as specified in the problem.

Solution Approach

The solution makes use of Depth-First Search (DFS), which is a recursive algorithm used to explore all possible paths from a given point in a graph-like structure, such as our binary matrix. In this context, we use DFS to explore and mark the cells that make up each island in `grid2`.

We define a recursive function `dfs` inside our `Solution` class's `countSubIslands` method. This `dfs` function is crucial to the solution:

- It takes the current coordinate `(i, j)` as input, corresponding to the current cell in `grid2`.
- If the cell in `grid1` at `(i, j)` is not land (if `grid1[i][j]` is not `1`), the current path of DFS cannot be a sub-island, and it returns `False`.
- The function marks the cell in `grid2` as visited by setting `grid2[i][j]` to `0`.
- The DFS explores all 4-directionally adjacent cells (up, down, left, right) by calling itself on each neighboring land cell `(x, y)` in `grid2` that hasn't been visited yet.
- If any recursive call to `dfs` returns `False`, it means that not all cells of this island in `grid2` are present in `grid1`. Hence, the function propagates this failure by returning `False` as well.
- If all adjacent land cells of the current cell in `grid2` are also land cells in `grid1`, the function returns `True`, indicating that the current path is a valid sub-island.

In the `countSubIslands` method, we iterate through every cell of `grid2`:

- We initiate a DFS search from each unvisited land cell found in `grid2`.
- We use a list comprehension that counts how many times the `dfs` function returns `True` for the starting cells of potential sub-islands. This gives us the total number of sub-islands in `grid2`.

We gather this count and return it as the solution. The use of recursion via DFS allows us to explore each island in `grid2` exhaustively, easily comparing its cells with `grid1` to determine if it's a sub-island or not.

Example Walkthrough

Let's illustrate the solution approach using a small example.

Imagine we have the following two matrices (`grid1` and `grid2`):

```
grid1:      grid2:
1 1 0       1 1 0
0 1 1       0 1 0
1 0 1       1 0 1
```

Here, we want to find the number of sub-islands in `grid2` where every '1' (land) is also present on the same position in `grid1`.

When applying the DFS algorithm, we start at each unvisited cell containing a '1' in `grid2`, and we attempt to traverse the entire island to which this cell belongs.

We start at the first cell of `grid2`:

- The corresponding cell in `grid1` is also a '1', so we continue the DFS and mark the cell in `grid2` as visited by setting it to '0'.
- DFS explores adjacent cells. The cell to the right is a '1' in both `grid1` and `grid2`, so we continue and mark it in `grid2`.
- Now, all adjacent cells (up, down, left, right) are '0' in `grid2`, so this path's DFS concludes this is a valid sub-island.

Next, we skip the second cell on the first row since it has already been visited, and move on to unvisited '1's.

The next starting point for DFS will be the second cell of the second row:

- In `grid1`, the cell is '1', but we notice the cell below it which is '1' in `grid2` is '0' in `grid1`, which invalidates this path for a sub-island.
- Even though the DFS would explore the right cell in `grid2`, which is '1' in `grid1`, the previous failure means that the whole island is not a sub-island.

Finally, we visit the last row:

- Starting from the first cell, the DFS would recognize this cell as a '1' in both grids, but the cells right and down are '0' in `grid2`, so there is no need to continue the DFS from this point.
- Moving to the third cell, it's a '1' in both grids, and all adjacent cells are '0' in `grid2`, so this is considered a valid sub-island.

Now, our list comprehension would have counted two instances where `dfs` returned `True`, indicating that there are two sub-islands in `grid2`.

Thus, based on our DFS exploration and the rules specified, we return the count of sub-islands, which in this example is `2`.

Python Solution

```python
class Solution:
    def countSubIslands(self, grid1: List[List[int]], grid2: List[List[int]]) -> int:
        # Depth-first search function to explore the island in grid2 and check if it's a sub-island of grid1
        def dfs(row, col):
            is_sub_island = grid1[row][col] == 1  # Check if the current position is land in grid1
            grid2[row][col] = 0  # Mark the current position in grid2 as visited (water)
            # Explore in all 4 neighboring directions (left, right, up, down)
            for delta_row, delta_col in [[0, -1], [0, 1], [-1, 0], [1, 0]]:
                new_row, new_col = row + delta_row, col + delta_col
                # If the new position is within the bounds and is land in grid2
                if 0 <= new_row < num_rows and 0 <= new_col < num_cols and grid2[new_row][new_col] == 1:
                    # If any part of the island in grid2 is not in grid1, it's not a sub-island
                    if not dfs(new_row, new_col):
                        is_sub_island = False
            return is_sub_island

        # Get the number of rows and columns in either of the grids
        num_rows, num_cols = len(grid1), len(grid1[0])

        # Count the number of sub-islands in grid2 that are also in grid1
        sub_islands_count = 0
        for row in range(num_rows):
            for col in range(num_cols):
                # If the current position is land in grid2 and is also a sub-island
                if grid2[row][col] == 1 and dfs(row, col):
                    sub_islands_count += 1

        # Return the total count of sub-islands
        return sub_islands_count

```

Java Solution

```java
class Solution {

    // Method to count sub-islands
    public int countSubIslands(int[][] grid1, int[][] grid2) {
        int rows = grid1.length;    // Number of rows in the grid
        int cols = grid1[0].length; // Number of columns in the grid
        int subIslandsCount = 0;    // Initialize count of sub-islands

        // Iterate over all cells in grid2
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If we find a land cell in grid2, we perform DFS to check if it's a sub-island
                if (grid2[i][j] == 1 && isSubIsland(i, j, rows, cols, grid1, grid2)) {
                    subIslandsCount++; // Increment count if a sub-island is found
                }
            }
        }
        return subIslandsCount; // Return the total count of sub-islands
    }

    // Helper method to perform DFS and check if the current island in grid2 is a sub-island of grid1
    private boolean isSubIsland(int row, int col, int rows, int cols, int[][] grid1, int[][] grid2) {
        // Check if the current cell is also a land cell in grid1; initialize as a potential sub-island
        boolean isSub = grid1[row][col] == 1;
        grid2[row][col] = 0; // Mark the cell as visited by setting it to water

        // Directions for top, right, bottom, and left (for traversing adjacent cells)
        int[] dirRow = {-1, 0, 1, 0};
        int[] dirCol = {0, 1, 0, -1};

        // Explore all adjacent cells
        for (int k = 0; k < 4; ++k) {
            int newRow = row + dirRow[k];
            int newCol = col + dirCol[k];
            // Check if the adjacent cell is within grid bounds and has not been visited
            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && grid2[newRow][newCol] == 1
                // Recursively call DFS; if any part of the island is not a sub-island, mark as not a sub-island
                && !isSubIsland(newRow, newCol, rows, cols, grid1, grid2)) {
                isSub = false;
            }
        }
        return isSub; // Return true if all parts of the island are sub-islands, false otherwise
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of sub-islands
    int countSubIslands(vector<vector<int>>& grid1, vector<vector<int>>& grid2) {
        int rowCount = grid1.size(); // Number of rows in the grid
        int colCount = grid1[0].size(); // Number of columns in the grid
        int subIslandCount = 0; // Counter for sub-islands

        // Loop through every cell in grid2
        for (int row = 0; row < rowCount; ++row) {
            for (int col = 0; col < colCount; ++col) {
                // If cell is land and DFS confirms it's a sub-island, increment count
                if (grid2[row][col] == 1 && depthFirstSearch(row, col, rowCount, colCount, grid1, grid2)) {
                    ++subIslandCount;
                }
            }
        }

        return subIslandCount; // Return the total count of sub-islands
    }

    // Helper DFS function to explore the island and check if it is a sub-island
    bool depthFirstSearch(int row, int col, int rowCount, int colCount, vector<vector<int>>& grid1, vector<vector<int>>& grid2) {
        // Initialize as true if the corresponding cell in grid1 is also land
        bool isSubIsland = grid1[row][col] == 1;
        // Mark the cell as visited in grid2
        grid2[row][col] = 0;

        // Defining directions for exploring adjacent cells
        vector<int> directions = {-1, 0, 1, 0, -1};
        // Explore all four adjacent cells
        for (int k = 0; k < 4; ++k) {
            int nextRow = row + directions[k];
            int nextCol = col + directions[k + 1];

            // Continue DFS if the next cell is within bounds and is land
            if (nextRow >= 0 && nextRow < rowCount &&
                nextCol >= 0 && nextCol < colCount &&
                grid2[nextRow][nextCol] == 1) {
                // If any part of the island is not a sub-island, set the flag to false
                if (!depthFirstSearch(nextRow, nextCol, rowCount, colCount, grid1, grid2)) {
                    isSubIsland = false;
                }
            }
        }

        // Return true if all parts of the island are a sub-island
        return isSubIsland;
    }
};

```

Typescript Solution

```typescript
function countSubIslands(grid1: number[][], grid2: number[][]): number {
    let rowCount = grid1.length;
    let colCount = grid1[0].length;
    let subIslandCount = 0;  // This will hold the count of sub-islands.

    // Iterate over each cell in the second grid.
    for (let row = 0; row < rowCount; ++row) {
        for (let col = 0; col < colCount; ++col) {
            // Start DFS if we find land (1) on grid2.
            if (grid2[row][col] === 1 && dfs(grid1, grid2, row, col)) {
                subIslandCount++;  // Increment when a sub-island is found.
            }
        }
    }
    return subIslandCount;
}

function dfs(grid1: number[][], grid2: number[][], i: number, j: number): boolean {
    let rowCount = grid1.length;
    let colCount = grid1[0].length;
    let isSubIsland = true;  // Flag indicating if a piece of land is a sub-island.

    // If corresponding cell in grid1 isn't land, this piece can't be a sub-island.
    if (grid1[i][j] === 0) {
        isSubIsland = false;
    }

    grid2[i][j] = 0;  // Sink the visited land piece to avoid revisits.

    // The 4 possible directions we can move (right, left, down, up).
    const directions = [
        [0, 1],
        [0, -1],
        [1, 0],
        [-1, 0],
    ];

    // Explore all 4 directions.
    for (let [dx, dy] of directions) {
        let newX = i + dx;
        let newY = j + dy;

        // Check for valid grid bounds and if the cell is land in grid2.
        if (newX >= 0 && newX < rowCount && newY >= 0 && newY < colCount && grid2[newX][newY] === 1) {
            // Recursively call dfs. If one direction is not a sub-island, the whole is not.
            if (!dfs(grid1, grid2, newX, newY)) {
                isSubIsland = false;
            }
        }
    }

    return isSubIsland;  // Return the status of current piece being part of sub-island.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code primarily depends on the number of calls to the `dfs` function as it traverses `grid2`. In the worst case, every cell in `grid2` might be equal to `1`, requiring a `dfs` call for each. Since we iterate over each cell exactly once due to the modification of `grid2` in the `dfs` function (we set `grid2[i][j]` to `0` to avoid revisiting), the worst-case time complexity is `O(m * n)` where `m` is the number of rows and `n` is the number of columns in `grid2`. This is because the time complex for each `dfs` call can be bounded by the surrounding cells (at most 4 additional calls per cell), leading to each cell being visited only once.

Space Complexity

The space complexity is determined by the maximum depth of the recursion stack during the execution of the `dfs` function. In the worst case, the recursion could be as deep as the total number of cells in `grid2` if the grid represents one large island that needs to be traversed entirely. Therefore, the worst-case space complexity would be `O(m * n)` due to the depth of the recursive call stack. However, in practice, the space complexity is often less since not all cells will be part of a singular recursive chain.