695. Max Area of Island

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

Problem Description

In the given LeetCode problem, we are provided with a 2D grid of `0`s and `1`s, where each `1` represents a piece of land, and `0`s represent water. The grid represents a map where islands are formed by connecting adjacent `1`s horizontally or vertically. We need to determine the size of the largest island in the grid, with the island's size being the count of `1`s that make up the island. If no islands are present in the grid, the result should be `0`.

An example grid might look like this:

```
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

In this grid, there are three islands with sizes 4, 1, and 2, respectively. The goal is to return the size of the largest island, which would be `4` in this case.

Intuition

To solve this problem, we can use Depth-First Search (DFS) to explore each piece of land (`1`) and count its area. We iterate through each cell of the grid; when we encounter a `1`, we start a DFS traversal from that cell. As we visit each `1`, we mark it as visited by setting it to `0` to ensure that each land cell is counted only once. This also helps to avoid infinite loops.

The DFS algorithm explores the land in all four directions: up, down, left, and right. For each new land cell we find, we add `1` to the area of the current island and recursively continue the search from that new cell. Once we can't explore further (we hit `0`s, or we reach the grid's boundaries), the recursive calls will return the total area of that particular island to the initial call.

By performing DFS on each `1` we find, we can calculate the area of each island. We keep track of the maximum area encountered during these searches. Once we've processed the whole grid, we have the largest island's area captured, and we return this as our result.

Solution Approach

The solution uses Depth-First Search (DFS), a classical algorithm for exploring all elements in a connected component of a grid, graph, or network. In this scenario, "connected components" are the individual islands within the grid.

The implementation consists of:

1. A helper function, `dfs`, which is a recursive function that takes the row and column indices `(i, j)` of a point in the grid as arguments.
2. Within `dfs`, we first check if the current cell contains a `1`. If it contains a `0`, it's either water or already visited, so we return an area of `0` for that cell.
3. If the current cell is a `1`, we initiate the area of this part of the island with `1`, and then set the cell to `0` to mark it as visited.
4. We define the possible directions we can explore from the current cell using the array `dirs` which contains the relative movements to visit top, right, bottom, and left adjacent cells.
5. We loop through each direction and calculate the new coordinates `(x, y)` for the adjacent cells. For each adjacent cell that is within the boundaries of the grid, we recursively call `dfs`.
6. The recursive `dfs` calls will return the area of the connected `1`s, which we add to the area of the current island.
7. After exploring all directions, the total area of the island, including the current cell, is returned.

At the top level of the `maxAreaOfIsland` function:

1. We get the number of rows `m` and columns `n` of the grid.
2. Then, we initiate a comprehensive search across all cells in the grid using list comprehension together with `max` function. Here, we only start a `dfs` traversal when we find a `1` (land cell).
3. Whichever cell starts a new DFS, the area of the connected island will be calculated completely before moving on to the next cell in the comprehension.
4. Finally, the maximum area found during the DFS traversals is returned.

By marking visited cells and only initiating DFS on unvisited land cells, we ensure that each island's area is calculated once, which gives us the efficiency and correctness of the algorithm.

This pattern of search and marking is common in problems dealing with connected components in a grid and is a handy technique to remember for similar problems.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following 2D grid:

```
0 1
1 0
```

In this grid, there are two islands, each consisting of a single piece of land (`1`). We aim to find the size of the largest island, although in this case, as both islands are of size `1`, the result should be `1`.

1. Begin by initializing `maxArea` to `0`. This variable will keep track of the largest island area found.
2. The algorithm starts scanning the grid from the top-left cell. When it encounters a `1`, it performs a DFS from that cell.
3. Let's start with the cell at (0,1). Since it's a land cell (`1`), we call the `dfs` function.
4. Inside `dfs`, we set the current cell to `0` to mark it visited and initialize the `area` to `1`, since we already found one piece of land.
5. The `dfs` function will check all adjacent cells (in our case, there is only one at (1,0)) and perform `dfs` on them if they are part of the land (if they contain `1`).
6. The `dfs` function is called on cell (1,0). Again, it will set the cell to `0`, increment the `area` to `2`, and check surrounding cells.
7. Since the adjacent cells are either water (`0`) or out of bounds, there are no further recursive calls, and the total area for this island is `1`.
8. We return to the top level of the `maxAreaOfIsland` function and continue checking the next cells. Since all `1`s have been visited, there are no new DFS calls.
9. The `maxArea` of `1` found is the size of the largest island, which is returned.

In this example, the algorithm correctly identifies the size of the largest island in the grid, which is `1`, and demonstrates the typical flow of search using DFS in this context.

Python Solution

```python
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        def dfs(row: int, col: int) -> int:
            # If the current cell is water (0), return area 0
            if grid[row][col] == 0:
                return 0

            # Current cell is land, so mark it as visited by setting it to 0,
            # and start area count at 1 (for the current cell)
            area = 1
            grid[row][col] = 0

            # Directions for exploring neighboring cells: up, right, down, left
            directions = (-1, 0, 1, 0, -1)

            # Iterate over the (row, col) pairs of neighboring cells
            for delta_row, delta_col in zip(directions, directions[1:]):
                next_row, next_col = row + delta_row, col + delta_col

                # Check if the neighboring cell is within bounds and not visited
                if 0 <= next_row < row_count and 0 <= next_col < col_count:
                    # Increase the area count by the area of the neighboring island part
                    area += dfs(next_row, next_col)

            # Return the total area found for this island
            return area

        # Get the dimensions of the grid
        row_count, col_count = len(grid), len(grid[0])

        # Use a list comprehension to apply DFS on each cell of the grid
        # Only cells with value 1 (land) will contribute to the area
        max_area = max(dfs(row, col) for row in range(row_count) for col in range(col_count) if grid[row][col] == 1)

        # Return the maximum area found among all islands
        return max_area
```

Java Solution

```java
public class Solution {
    private int rows;            // Number of rows in the grid
    private int cols;            // Number of columns in the grid
    private int[][] grid;        // The grid itself

    public int maxAreaOfIsland(int[][] grid) {
        rows = grid.length;              // Set the total number of rows in the grid
        cols = grid[0].length;           // Set the total number of columns in the grid
        this.grid = grid;                // Assign the input grid to the instance variable
        int maxArea = 0;                 // To keep track of the maximum area found so far

        // Iterate over every cell in the grid
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Update the maximum area after performing DFS on current cell
                maxArea = Math.max(maxArea, dfs(i, j));
            }
        }
        return maxArea;                  // Return the maximum area found
    }

    // Helper method to perform Depth-First Search (DFS)
    private int dfs(int row, int col) {
        // If the current cell is water (0), or it is already visited, then the area is 0
        if (grid[row][col] == 0) {
            return 0;
        }

        int area = 1;                     // Start with a size of 1 for the current land cell
        grid[row][col] = 0;               // Mark the land cell as visited by sinking it (set to 0)
        int[] dirs = {-1, 0, 1, 0, -1};  // Array to represent the four directions (up, right, down, left)

        // Iterate over the four directions
        for (int k = 0; k < 4; ++k) {
            int nextRow = row + dirs[k];      // Calculate the row for adjacent cell
            int nextCol = col + dirs[k + 1];  // Calculate the column for adjacent cell

            // Check if adjacent cell is within the bounds and then perform DFS
            if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols) {
                area += dfs(nextRow, nextCol); // Add the area found from DFS to the total area
            }
        }
        return area;                      // Return the total area found from the current cell
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional> // For std::function
#include <algorithm>  // For std::max

class Solution {
public:
    // Function to find the maximum area of an island in a given grid
    int maxAreaOfIsland(std::vector<std::vector<int>>& grid) {
        // Obtain the number of rows and columns of the grid
        int rows = grid.size(), cols = grid[0].size();
        // Directions array to explore all 4 neighbors (up, right, down, left)
        int directions[5] = {-1, 0, 1, 0, -1};
        // Variable to store the final maximum area of island found
        int maxArea = 0;

        // Depth-first search function using lambda and std::function for ease of recursion
        std::function<int(int, int)> depthFirstSearch = [&](int i, int j) -> int {
            // Base case: if the current cell is water (0), return 0 area
            if (grid[i][j] == 0) {
                return 0;
            }

            // Mark the current cell as visited by setting it to 0 and start counting the area from 1
            int area = 1;
            grid[i][j] = 0;

            // Explore all 4 neighbor directions
            for (int k = 0; k < 4; ++k) {
                int x = i + directions[k], y = j + directions[k + 1];
                // Check if the neighbor coordinates are within grid bounds
                if (x >= 0 && x < rows && y >= 0 && y < cols) {
                    // Increment the area based on this recursive depth-first search
                    area += depthFirstSearch(x, y);
                }
            }
            // Return the area found for this island
            return area;
        };

        // Iterate over all cells in the grid
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Update maxArea with the maximum between current maxArea and newly found area
                maxArea = std::max(maxArea, depthFirstSearch(i, j));
            }
        }

        // Return the maximum area of island found in the grid
        return maxArea;
    }
};
```

Typescript Solution

```typescript
function maxAreaOfIsland(grid: number[][]): number {
    const rows = grid.length;
    const cols = grid[0].length;
    // Define the directions for exploring adjacent cells (up, right, down, left)
    const directions = [-1, 0, 1, 0, -1];

    // Helper function to perform DFS and calculate the area of the island
    const exploreIsland = (row: number, col: number): number => {
        if (grid[row][col] === 0) {
            // If the current cell is water (0), then there's no island to explore
            return 0;
        }

        // Initialize area for the current island
        let area = 1;
        // Mark the current cell as visited by setting it to water (0)
        grid[row][col] = 0;
        // Explore all adjacent cells
        for (let k = 0; k < 4; ++k) {
            const nextRow = row + directions[k];
            const nextCol = col + directions[k + 1];
            if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols) {
                // Increment the area by the area of adjacent lands
                area += exploreIsland(nextRow, nextCol);
            }
        }
        return area;
    };

    // Initialize maximum area of an island to be 0
    let maxArea = 0;
    // Loop through every cell in the grid
    for (let row = 0; row < rows; ++row) {
        for (let col = 0; col < cols; ++col) {
            // Update the maxArea if a larger island is found
            maxArea = Math.max(maxArea, exploreIsland(row, col));
        }
    }
    // Return the maximum area of an island found in the grid
    return maxArea;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is `O(M * N)`, where `M` is the number of rows and `N` is the number of columns in the `grid`. This is because in the worst case, the entire grid could be filled with land (1's), and we would need to explore every cell exactly once. The function `dfs` is called for each cell, but each cell is flipped to `0` once visited to avoid revisiting, ensuring each cell is processed only once.

Space Complexity

The space complexity is `O(M * N)` in the worst case, due to the call stack size in the case of a deep recursion caused by a large contiguous island. This would happen if the grid is filled with land (1's) and we start the depth-first search from one corner of the grid, the recursion would reach the maximum depth equal to the number of cells in the grid before it begins to unwind.