2658. Maximum Number of Fish in a Grid

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

Problem Description

The problem presents a 2D matrix `grid` representing a 'fishing grid' where each cell is either land or water. The cells containing water have a non-zero value representing the number of fish present. A cell with the value `0` indicates that it is a land cell, and no fishing can be done there. The objective is to determine the maximum number of fish that a fisher can catch when starting from an optimally chosen water cell. The conditions for fishing are:

- A fisher can harvest all fish from the water cell he is currently on.
- The fisher can move from one water cell to any directly adjacent water cell (up, down, left, right), provided it isn't land.
- The fisher can perform these actions repetitively and in any order.

The desired output is the maximum number of fish that can be caught, with the possibility of a `0` when no water cell (cells with fish) is available.

Intuition

To solve the problem, we need to consider all water regions (areas of connected water cells) and the fish available within them. Since the fisher can only move to adjacent water cells, whenever he moves, he 'engages' a specific water region. It's important to recognize that once a fisher starts fishing in a region, they should catch all the fish in that region before moving to another since returning to a region is not beneficial—fish already caught won't respawn.

We reach an intuitive approach by focusing on the following points:

- Once the fisher starts at a water cell, they should collect all fish in the connected water region.
- We should account for all fish present in a standalone water region in one go—there's no coming back for more once they are caught.
- By iterating over all water cells and simulating this fishing approach, we can find the largest possible catch.

With these points in mind, we turn to Depth-First Search (DFS)—a perfect technique for exploring all cells in a connected region starting from any given water cell. The DFS counts the fish, marks the cells as `0` once visited (to avoid recounting), and moves to adjacent water cells until the entire region is depleted of fish. We keep track of the maximum number of fish caught in any single run of the DFS across the entire grid. By applying this process to each water cell, we can identify the optimal starting cell that yields the most fish.

Solution Approach

To achieve the task, we use a recursive Depth-First Search (DFS) algorithm to navigate through the grid. The implementation relies on a helper `dfs` function that takes the coordinates `(i, j)` of the current cell and returns the count of fish caught starting from that cell.

Here is the approach broken down:

1. The `dfs` function initiates with the grid cell coordinates `(i, j)` and proceeds to capture all fish at this location. It sets the fish count at this cell to `0` to mark the cell as "fished out" and to prevent revisiting during the same DFS traversal.

2. We then iterate over the four possible directions (up, down, left, right) to move from the current cell to an adjacent one. This is done efficiently by using a loop and the `pairwise` utility, iterating over pairs of directional movements encoded as `(dx, dy)` deltas.

3. For each potential move to an adjacent cell `(x, y)`, we check if the cell is within bounds of the grid, and if it is a water cell (`grid[x][y] > 0`). If it fits these criteria, we call the `dfs` function recursively for that cell.

4. The counts of fish from the current cell and from recursively applied DFS calls are summed up to get the total count of fish in this connected water region.

5. The outer loop of the `Solution` class iterates through all cells `(i, j)` in the grid. For each water cell identified (`grid[i][j] > 0`), it invokes the `dfs` function and captures the return value.

6. We maintain a variable `ans` to keep track of the maximum fish count seen so far. Every time we get a count back from a `dfs` call, we update `ans` with the maximum of its current value and the newly obtained count.

7. After the loop completes, we return the value of `ans` as the result, representing the maximum number of fish that can be caught starting from the best water cell.

The combination of recursive DFS and careful updates ensures that we thoroughly explore each connected water region, never double count, and always remember the "best" starting cell found in terms of fish count.

Mentioned in the Reference Solution Approach, the DFS method inherently works well for this problem because it naturally follows the constraints of the fisher's movement and fishing actions. Furthermore, by traversing and marking cells within the `grid`, we avoid the need for an auxiliary data structure to keep track of visited cells, thereby utilizing the `grid` for dual purposes—both as the map of the fishing area and as the record of visited water cells.

Here's the pseudocode that encapsulates this solution:

```
function dfs(i, j):
    cnt = grid[i][j]
    grid[i][j] = 0  // Mark as visited (fished out)
    for each direction (up, down, left, right) do:
        x, y = new coordinates in direction
        if (x, y) is in bounds and is water cell then:
            cnt += dfs(x, y)
    return cnt

for each cell (i, j) in grid do:
    if grid[i][j] > 0 then:
        ans = max(ans, dfs(i, j))

return ans
```

The final external `for` loop ensures that every water region is considered, while the recursive `dfs` functions guarantee a thorough exploration of each region.

Example Walkthrough

Let's take a 3×3 grid to illustrate the solution approach.

```
grid = [
  [2, 3, 1],
  [0, 0, 4],
  [7, 6, 0]
]
```

In this grid, `0` represents land cells, and non-zero values like `2`, `3`, `1`, etc., represent water cells with the corresponding number of fish in them.

Let's apply the DFS algorithm on the grid.

1. Starting at the top left corner, `(0, 0)` has `2` fish. We start here and set `grid[0][0]` to `0` which now looks like:

```
grid = [
  [0, 3, 1],
  [0, 0, 4],
  [7, 6, 0]
]
```

2. From `(0, 0)`, we look at adjacent cells up, down, left and right. Only two cells are adjacent and they are water cells: `(0, 1)` and `(1, 2)`.

3. Next, we go to `(0, 1)`, where there are `3` fish. After setting `grid[0][1]` to `0`, the grid is:

```
grid = [
  [0, 0, 1],
  [0, 0, 4],
  [7, 6, 0]
]
```

4. From `(0, 1)`, we only have one water cell adjacent which is `(0, 2)`, so we move there and add `1` fish to our count and mark the cell as visited:

```
grid = [
  [0, 0, 0],
  [0, 0, 4],
  [7, 6, 0]
]
```

This completes the DFS for one water region originating from `(0, 0)`. The total fish caught so far is `2+3+1 = 6` fish.

5. We then continue the external loop and come across the cell `(1, 2)` and start a new DFS because `grid[1][2] > 0`. This cell is isolated and only has `4` fish. After fishing, the grid updates as follows:

```
grid = [
  [0, 0, 0],
  [0, 0, 0],
  [7, 6, 0]
]
```

6. Finally, we explore the cell `(2, 0)`, triggering another DFS call. This region includes the cells `(2, 0)` and `(2, 1)` with a total of `7 + 6 = 13` fish.

```
grid = [
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

Following these steps, we have the maximum fish counts from each standalone water region:

- Starting at `(0, 0)` - we caught `6` fish.
- Starting at `(1, 2)` - we caught `4` fish.
- Starting at `(2, 0)` - we caught `13` fish.

Our answer `ans` is the maximum of these, which is `13` fish.

The procedure ensures that we never revisited any water cell and always accounted for the whole region before moving to the next. The final answer is the maximum fish caught across all regions, taking into account that only one region can be fully fished by starting at the optimal water cell.

Python Solution

```python
from typing import List

class Solution:
    def findMaxFish(self, grid: List[List[int]]) -> int:
        # Depth-first search function to explore the grid and count fish
        def dfs(i: int, j: int) -> int:
            fish_count = grid[i][j]  # Number of fish at the current location
            grid[i][j] = 0  # Mark the current location as visited by setting it to 0
            # Explore all four adjacent cells (up, down, left, right)
            for dx, dy in [(-1, 0), (0, 1), (1, 0), (0, -1)]:
                x, y = i + dx, j + dy
                # Check if the new position is within the grid bounds and has fish
                if 0 <= x < m and 0 <= y < n and grid[x][y]:
                    fish_count += dfs(x, y)  # Add fish from the neighboring cell
            return fish_count

        m, n = len(grid), len(grid[0])  # Get the dimensions of the grid
        max_fish = 0  # Initialize the maximum fish count
        # Iterate over all cells in the grid
        for i in range(m):
            for j in range(n):
                # If the current cell has fish, perform DFS from here
                if grid[i][j]:
                    max_fish = max(max_fish, dfs(i, j))  # Update the maximum fish count
        return max_fish  # Return the maximum number of fish that can be found in one group

# Example usage:
# sol = Solution()
# print(sol.findMaxFish([[2, 1], [1, 1], [0, 1]]))

```

Java Solution

```java
class Solution {

    private int[][] grid; // The grid representing the pond.
    private int rows; // Number of rows in the pond grid.
    private int cols; // Number of columns in the pond grid.

    // This method calculates the maximum number of fish that can be found in a straight line.
    public int findMaxFish(int[][] grid) {
        rows = grid.length; // Assigns the number of rows of the grid.
        cols = grid[0].length; // Assigns the number of columns of the grid.
        this.grid = grid; // Stores the grid in the instance variable for easy access.
        int maxFishCount = 0; // Starts with zero as the maximum number of fish found.

        // Iterates through each cell in the grid.
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If the current cell contains fish, perform a DFS to find all connected fish.
                if (grid[i][j] > 0) {
                    maxFishCount = Math.max(maxFishCount, dfs(i, j));
                }
            }
        }
        // Return the largest group of connected fish found in the pond.
        return maxFishCount;
    }

    // This method performs a depth-first search (DFS) to find all connected fish starting from cell (i, j).
    private int dfs(int i, int j) {
        int fishCount = grid[i][j]; // Counts the fish at the current cell.
        grid[i][j] = 0; // Marks the current cell as "visited" by setting its fish count to zero.
        // Array to calculate adjacent cell coordinates (up, right, down, left).
        int[] directions = {-1, 0, 1, 0, -1};

        // Explore all four adjacent cells using the directions array.
        for (int k = 0; k < 4; ++k) {
            int x = i + directions[k]; // Row index of the adjacent cell.
            int y = j + directions[k + 1]; // Column index of the adjacent cell.

            // Check whether the adjacent cell is within grid bounds and contains fish.
            if (x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] > 0) {
                fishCount += dfs(x, y); // Accumulate fish count and continue DFS.
            }
        }
        // Return the total count of fish connected to cell (i, j).
        return fishCount;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <functional>

class Solution {
public:
    int findMaxFish(std::vector<std::vector<int>>& grid) {
        int numRows = grid.size();
        int numCols = grid[0].size();
        int maxFishCount = 0;

        // Depth-first search function to search for connected fishes.
        std::function<int(int, int)> depthFirstSearch = [&](int row, int col) -> int {
            int fishCount = grid[row][col];
            grid[row][col] = 0; // Mark as visited by setting to 0.

            // Directions: up, right, down, left
            int directions[5] = {-1, 0, 1, 0, -1};
            for (int k = 0; k < 4; ++k) {
                int newRow = row + directions[k];
                int newCol = col + directions[k + 1];

                // Check if the new coordinates are valid and not visited.
                if (newRow >= 0 && newRow < numRows && newCol >= 0 && newCol < numCols && grid[newRow][newCol]) {
                    fishCount += depthFirstSearch(newRow, newCol); // Recurse.
                }
            }
            return fishCount;
        };

        // Iterate over each cell in the grid.
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // If cell is not visited and there are fishes, perform DFS.
                if (grid[i][j]) {
                    maxFishCount = std::max(maxFishCount, depthFirstSearch(i, j));
                }
            }
        }
        return maxFishCount; // Return the maximum number of fishes in a connected region.
    }
};

```

Typescript Solution

```typescript
function findMaxFish(grid: number[][]): number {
    const rowCount = grid.length;    // Number of rows in the grid.
    const colCount = grid[0].length; // Number of columns in the grid.

    // Direction vectors to help navigate through grid neighbors.
    const directions = [-1, 0, 1, 0, -1];

    // Deep First Search function to count fish in a connected region of the grid.
    const deepFirstSearch = (row: number, col: number): number => {
        let count = grid[row][col]; // Start with the initial count of fish in the given cell.
        grid[row][col] = 0; // Mark the cell as visited by setting it to 0.

        // Search through all four possible directions: up, right, down, and left.
        for (let dirIndex = 0; dirIndex < 4; ++dirIndex) {
            const nextRow = row + directions[dirIndex];
            const nextCol = col + directions[dirIndex + 1];

            // If the next cell is within the grid bounds and has fish, perform DFS on it.
            if (nextRow >= 0 && nextRow < rowCount && nextCol >= 0 && nextCol < colCount && grid[nextRow][nextCol] > 0) {
                count += deepFirstSearch(nextRow, nextCol);
            }
        }

        return count; // Return the total fish count for this region.
    };

    let maxFish = 0; // Keep track of the maximum fish count found.

    // Iterate through each cell in the grid.
    for (let row = 0; row < rowCount; ++row) {
        for (let col = 0; col < colCount; ++col) {
            // If the cell is not visited and has fish, use DFS to find total fish count.
            if (grid[row][col] > 0) {
                // Update maxFish with the maximum of the current max and newly found count.
                maxFish = Math.max(maxFish, deepFirstSearch(row, col));
            }
        }
    }

    return maxFish; // Return the maximum fish count found in any connected region.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is `O(m*n)` because the `dfs` function is called for every cell in the grid at most once. Each cell is visited and then it's marked as `0` (zero) to avoid revisiting. Consequently, every cell (where `m` is the number of rows and `n` is the number of columns of the grid) is accessed in the worst case.

Space Complexity

The space complexity is `O(m*n)` in the worst-case scenario, which occurs when the grid is fully filled with non-zero values, leading to the deepest recursion of `dfs` potentially reaching `m*n` calls in the call stack before returning.