1992. Find All Groups of Farmland

Problem Description

In this LeetCode problem, we are working with a 2D grid representing a piece of land, where each cell in this grid can either be forested land (denoted by `0`) or farmland (denoted by `1`). The grid is 0-indexed, meaning that the rows and columns are numbered starting from 0. Groups of farmland are rectangles within the grid that are composed entirely of 1s. No two groups are adjacent to each other, which means the sides of the rectangles of farmland don't touch each other horizontally or vertically.

We are tasked with finding the coordinates of the top-left and bottom-right corners of each group of farmland. The coordinates of a corner are represented as ((r, c)), where (r) and (c) are the row and column indexes of the cell within the grid. The result is expected to be a list of 4-item lists, each containing the coordinates of the top-left and bottom-right corners of a group, denoted as ([r1, c1, r2, c2]).

To summarize, we need to scan the grid, identify the separate rectangular farmland groups, record the coordinates of their top-left and bottom-right corners, and return this information in a 2D list.

Intuition

The intuition behind the solution is based on finding the starting point (the top-left corner) for each group of farmland. We can iterate through the grid cell by cell. When we find a cell with a value of 1 that hasn't been identified as part of a group yet (which means it's neither directly below nor to the right of another farmland cell), we consider it as the top-left corner of a new group.

Once we have identified a top-left corner, we need to find the corresponding bottom-right corner for the group. We do this by scanning downwards from the top-left corner until we reach a row where the farmland ends (the next cell is 0 or we've reached the grid boundary). The same approach is used horizontally from the top-left corner to find the end of the farmland in that particular row. The last cell before the farmland ends in both directions will be the bottom-right corner.

The key steps for the algorithm are as follows:

1. Traverse all cells in the grid row by row and column by column.
2. Skip the inspection for a cell if:
   - It is forested land (a 0).
   - It is immediately to the right of another farmland cell (as this would be within the same farmland group).
   - It is directly below another farmland cell (also indicating the same group).
3. Upon finding the starting point of a new group (farmland cell that does not meet the above conditions), mark it and then expand downwards and rightwards to find the bottom right corner.
4. Once the bottom right corner is identified, append the coordinates to the results list.
5. After finishing the traversal, return the list of coordinates for each group.

Solution Approach

The implementation of the solution uses a straightforward approach to process the `land` matrix and identify the farmland groups without additional data structures for storing intermediate states. The algorithm relies heavily on the properties of the matrix and the characteristics of the groups.

Here is a step-by-step explanation of the algorithm, referencing the solution code provided:

1. Initialize a list `ans` to store the answer.
2. Start two nested loops to iterate over each cell in the matrix:
   - The outer loop goes through each row `i`.
   - The inner loop goes through each column `j`.
3. Within the inner loop, for each cell check if the cell can be the top-left corner of a new group:
   - Check if the cell is a forested area (`land[i][j] == 0`); If true, continue to the next iteration.
   - Check if the cell is immediately to the right (`j > 0 and land[i][j - 1] == 1`) or below (`i > 0 and land[i - 1][j] == 1`) another farmland cell; if true, this means the cell is part of an existing group, so continue to the next iteration.
4. If the current cell (`land[i][j]`) is indeed the top-left corner of a new group, find the corresponding bottom-right corner:
   - Initialize variables `r` and `c` with the current coordinates (`i, j`).
   - Expand downwards from the top-left corner along the column `i` until reaching a cell where the next cell below is 0 or the boundary of the matrix is reached. Update the `r` to the row index of the last farmland cell in this column.
   - Expand rightwards from the top-left corner along the row `i` until reaching a cell where the next cell to the right is 0 or the boundary of the matrix is reached. Update the `p` to the column index of the last farmland cell in this row.
5. Append the coordinates `i, j, r, p` to the `ans` list. These coordinates represent the top-left (`i, j`) and bottom-right (`r, p`) corners of the new farmland group.
6. After processing all cells, return the `ans` list containing the 4-length arrays which provide the coordinates for each group of farmland.

The implementation does not require additional data structures and works efficiently due to the constraints put in place, such as non-adjacency of the farmland groups and the binary nature of the `land` matrix. As the algorithm processes each cell at most twice, once in the overall iteration and once during the expansion to find the bottom-right corner, the time complexity of the algorithm is $O(mn)$, where m and n are the dimensions of the `land` matrix. The space complexity is $O(1)$, not counting the input and output, since no extra space is used beyond variables for iteration and expansion.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider a small 2D grid, representing land as follows:

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

In this grid:

- `1` represents a farmland cell,
- `0` represents a forested area.

We want to find the coordinates of the top-left and bottom-right corners of each group of farmland.

Following the steps of the algorithm:

1. Initialize an empty list `ans` to store the coordinates of farmland groups.
2. Starting from the top-left corner of the grid, go row by row and column by column. We initiate an outer loop for rows and an inner loop for columns.
3. When we reach the first `1` at position `(0, 0)`, we check:
   - It's not forested land.
   - It's not directly to the right or below another farmland cell. Hence, it could be a top-left corner.
4. From `(0, 0)`, move downwards in the same column until we reach a `0`. The last farmland cell is `(1, 0)`, so `x = 1`.
5. From `(0, 0)`, move rightwards in the same row until we reach a farmland cell followed by a `0`. The last farmland cell is `(0, 0)`, so `y = 0`.
6. Append `(0, 0, 1, 0)` to the `ans` list, representing the top-left `(0, 0)` and bottom-right `(1, 0)` coordinates of this group.
7. Continue to `(0, 3)`, which again could be a top-left corner. Repeat the expansion process to find the bottom-right corner at `(1, 4)`. Append `[0, 3, 1, 4]` to the `ans`.
8. Skip cells in the middle of groups like `(0, 1)`, `(0, 4)`, `(1, 1)`, and so on.
9. Finally, the cell `(3, 0)` is identified as the top-left of the third group, and `(3, 2)` as its bottom-right corner. Append `[3, 0, 3, 2]`.
10. Once all cells have been processed, we have `ans = [[0, 0, 1, 0], [0, 3, 1, 4], [3, 0, 3, 2]]`.

The resulting `ans` list provides the coordinates for each group of farmland in the grid as required.

Python Solution

```python
from typing import List

class Solution:
    def findFarmland(self, land: List[List[int]]) -> List[List[int]]:
        # Get the dimensions of the input grid.
        num_rows, num_columns = len(land), len(land[0])

        # Initialize an output list to store the coordinates of each farmland.
        farmlands = []

        # Iterate over each cell in the grid.
        for row in range(num_rows):
            for col in range(num_columns):
                # Skip the cell if it's not part of a farmland, or if it is not the top-left cell
                # of a farmland (if it's a continuation of a row or column of a previous farmland).
                if (land[row][col] == 0 or
                    (col > 0 and land[row][col - 1] == 1) or
                    (row > 0 and land[row - 1][col] == 1)):
                    continue

                # Initialize farmland boundaries with the current cell.
                farmland_end_row, farmland_end_col = row, col

                # Expand vertically downwards to find the bottom boundary of the farmland.
                while farmland_end_row + 1 < num_rows and land[farmland_end_row + 1][col] == 1:
                    farmland_end_row += 1

                # Expand horizontally rightwards to find the right boundary of the farmland.
                while farmland_end_col + 1 < num_columns and land[farmland_end_row][farmland_end_col + 1] == 1:
                    farmland_end_col += 1

                # Add the found farmland coordinates to the list.
                # It includes the top-left and bottom-right coordinates of the rectangle.
                farmlands.append([row, col, farmland_end_row, farmland_end_col])

                # Mark the found farmland on the map to not count it again
                for i in range(row, farmland_end_row + 1):
                    for j in range(col, farmland_end_col + 1):
                        land[i][j] = 0

        # Return the list of coordinates for all farmlands found.
        return farmlands
```

Java Solution

```java
class Solution {

    // Method to find farmland blocks in a given grid of land.
    public int[][] findFarmland(int[][] land) {
        // Initialize a list to store the results.
        List<int[]> farmlandList = new ArrayList<>();

        // Get the number of rows and columns in the land grid.
        int rowCount = land.length;
        int colCount = land[0].length;

        // Iterate over each cell in the grid.
        for (int row = 0; row < rowCount; ++row) {
            for (int col = 0; col < colCount; ++col) {

                // Skip the current cell if it is not land (0), or it's not the top-left corner of a farmland block.
                if (land[row][col] == 0 || (col > 0 && land[row][col - 1] == 1) || (row > 0 && land[row - 1][col] == 1)) {
                    continue;
                }

                // Initialize variables for the bottom-right corner of the farmland block.
                int bottomRow = row;
                int rightCol = col;

                // Extend the farmland block towards the bottom (row-wise).
                while (bottomRow + 1 < rowCount && land[bottomRow + 1][col] == 1) {
                    ++bottomRow;
                }

                // Extend the farmland block towards the right (column-wise).
                while (rightCol + 1 < colCount && land[bottomRow][rightCol + 1] == 1) {
                    ++rightCol;
                }

                // Add the top-left and bottom-right corner coordinates of the farmland block to the result list.
                farmlandList.add(new int[] {row, col, bottomRow, rightCol});
            }
        }

        // Convert the list of farmland blocks to an array and return the array.
        return farmlandList.toArray(new int[farmlandList.size()][4]);
    }
}
```

C++ Solution

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

class Solution {
public:
    vector<vector<int>> findFarmland(vector<vector<int>>& land) {
        vector<vector<int>> farmlands; // Variable to store the list of farmlands found
        int rows = land.size(); // Number of rows in the input matrix
        int cols = land[0].size(); // Number of columns in the input matrix

        // Loop through each cell in the matrix
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Skip the cell if it is not part of farmland, or if it is already part of an identified farmland
                if (land[i][j] == 0 || (j > 0 && land[i][j - 1] == 1) || (i > 0 && land[i - 1][j] == 1)) {
                    continue;
                }

                // Initialize the top-left corner of the current farmland
                int topLeftR = i;
                int topLeftC = j;
                int bottomRightR = i;
                int bottomRightC = j;

                // Expand in the downward direction (increment rows)
                while (bottomRightR + 1 < rows && land[bottomRightR + 1][j] == 1) {
                    bottomRightR++;
                }

                // Expand in the rightward direction (increment columns)
                while (bottomRightC + 1 < cols && land[i][bottomRightC + 1] == 1) {
                    bottomRightC++;
                }

                // Store the coordinates of the current farmland
                farmlands.push_back({topLeftR, topLeftC, bottomRightR, bottomRightC});
            }
        }

        // Return the list of farmlands identified in the matrix
        return farmlands;
    }
};
```

Typescript Solution

```typescript
// Type alias for better readability, denotes the matrix of land plots
type LandMatrix = number[][];

// Defines the signature of a rectangle representing a farmland plot with its top-left and bottom-right coordinates
type FarmLand = [number, number, number, number];

/**
 * Finds all rectangular farmlands within a given land matrix.
 * A farmland is a contiguous rectangular block of land marked with 1s.
 *
 * @param land - A matrix of numbers representing plots, where 1 is a farmland plot and 0 is not part of a farmland.
 * @returns Farmlands - An array of rectangles representing distinct farmlands.
 */
function findFarmland(land: LandMatrix): FarmLand[] {
    const farmlands: FarmLand[] = []; // Variable to store the list of farmlands found
    const rowCount: number = land.length; // Number of rows in the input matrix
    const colCount: number = land[0].length; // Number of columns in the input matrix

    // Loop through each cell in the matrix
    for (let i = 0; i < rowCount; ++i) {
        for (let j = 0; j < colCount; ++j) {
            // Skip the cell if it is not part of farmland, or if it is already part of an identified farmland
            if (land[i][j] === 0 || (j > 0 && land[i][j - 1] === 1) || (i > 0 && land[i - 1][j] === 1)) {
                continue;
            }

            // Initialize the top-left corner of the current farmland
            let topLeftR = i;
            let topLeftC = j;
            let bottomRightR = i;
            let bottomRightC = j;

            // Expand in the downward direction (increment rows)
            while (bottomRightR + 1 < rowCount && land[bottomRightR + 1][j] === 1) {
                bottomRightR++;
            }

            // Expand in the rightward direction (increment columns)
            while (bottomRightC + 1 < colCount && land[i][bottomRightC + 1] === 1) {
                bottomRightC++;
            }

            // Store the coordinates of the current farmland as a tuple [topLeftR, topLeftC, bottomRightR, bottomRightC]
            farmlands.push([topLeftR, topLeftC, bottomRightR, bottomRightC]);

            // Mark the identified farmland in the matrix to ensure it is not processed again
            for (let x = topLeftR; x <= bottomRightR; x++) {
                for (let y = topLeftC; y <= bottomRightC; y++) {
                    land[x][y] = 0;
                }
            }
        }
    }

    // Return the list of farmlands identified in the matrix
    return farmlands;
}
```

Time and Space Complexity

The time complexity of the provided code 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 scenario, you have to visit each cell once to check if it's the start of a new farmland and then possibly extend the search downwards and to the right to find the extent of that farmland. However, since an extension in one direction for one farm doesn't check cells belonging to another potential farm, each cell is visited at most twice (once in the main iteration and once when expanding downwards or rightwards).

The space complexity of the code is $O(1)$ if we don't count the output space. The algorithm creates only a few variables to keep track of the current rectangle (farmland) being discovered (`r, y, i, j`). It does not use any additional data structure that grows with the input size. The `ans` list is the output and typically isn't counted in space complexity calculations. However, in the context where the space used by the output can be taken into consideration, the space complexity would be $O(f)$ where f is the number of farmlands discovered, since this is the size of the `ans` list that is being returned.