2088. Count Fertile Pyramids in a Land

Hard Array Dynamic Programming Matrix

Leetcode Link

Problem Description

The problem requires us to find and count all the possible pyramidal and inverse pyramidal plots in a farmer's rectangular grid land, represented by a 2D array `grid`. Each cell of the grid can either be fertile (value `1`) or barren (value `0`). A pyramidal plot is a set of fertile cells that form a pyramid shape, with the narrowest point at the top and getting progressively wider until the base. An inverse pyramidal plot, on the other hand, has the narrowest point at the bottom and gets wider towards the top.

A few conditions must be satisfied for the plots:

- For a pyramidal plot, starting with the apex and moving down each row, the width of the plot increases by one cell on both sides. The base will be the widest part of the pyramid.
- For an inverse pyramidal plot, it's the opposite: starting with the apex and moving upwards each row, the width increases by one cell on both sides.
- The number of cells in each plot must be more than one and all must be fertile.
- The apexes of the pyramids must not be on the outer boundary of the grid to ensure there is room for the plot to expand.

The goal of the problem is to determine the total number of valid pyramidal and inverse pyramidal plots that can be found within the given grid.

Intuition

The solution requires a dynamic approach to efficiently count the number of pyramids and inverse pyramids in the grid without having to manually enumerate each possibility which would be computationally expensive.

We can use a dynamic programming array `f` that mirrors the dimensions of `grid` to hold the maximum height of a pyramid ending at a given cell `(i, j)`. A cell `f[i][j]` in `f` represents the maximum height of a pyramid whose apex is at the corresponding cell in `grid` if the cell is fertile, or `-1` if the cell is barren. This approach enables us to build larger pyramids based on smaller ones.

For pyramidal plots, we initialize `f` to zero and start from the bottom row moving upwards. If a cell is fertile (not on the boundary and `grid[i][j]` is `1`), we check the minimum heights of pyramids that could be formed in the cells below it and to the left and right, and we add `1` to that minimum height to obtain the height of the pyramid for which this cell could be the apex.

For inverse pyramidal plots, we do the opposite. We reset `f` and start from the top row, moving downwards, updating `f` in the same manner but based on the cells above each fertile cell.

In both cases, we accumulate the number of plots in a counter `ans` by adding the heights obtained in `f` for each fertile cell as we iterate over the grid. This is because for each pyramid with height `h`, there are `h` pyramids (nested within one another) ending at that cell.

This way, the final `ans` will contain the total count of pyramidal and inverse pyramidal plots within the grid.

Solution Approach

The solution provided uses a dynamic programming approach to efficiently calculate the total number of pyramids and inverse pyramids contained within the grid. Let's break down the key aspects of the implementation:

1. Initialize a matrix `f` with the same dimensions as `grid`, to store the number of levels of the largest possible pyramid with its apex at each cell.
2. Use two for-loop blocks in the code to iterate over the entire grid twice — once for pyramids and once for inverse pyramids.
3. In the first for-loop block, calculate the number of levels for pyramidal plots:
   - Start from the bottom row and move upwards, as pyramids narrow towards the top.
   - If the current cell is fertile, set `f[i][j]` to the minimum value of the three cells below and adjacent, plus one (to account for the current cell being a potential apex of a new pyramid). If it's on the border or barren, set `f[i][j]` to `-1`.
   - Increment `ans` by the value of `f[i][j]` (except when `-1`) to count all pyramids ending at that cell.
4. In the second for-loop block, calculate the number of levels for inverse pyramidal plots:
   - Start from the top row and move downwards, as inverse pyramids widen towards the top.
   - Reset the state of `f` before using this block, since the care of the borders where the pyramid cannot be formed.
   - Follow a similar process as with pyramids, but consider the cells above the current cell and adjacent to it.
   - Again, increment `ans` by the value of `f[i][j]` for each fertile cell.
5. Return the total count `ans` as the sum of pyramid and inverse pyramid counts.

This solution ensures that every possible pyramid, as well as its sub-pyramids resulting from truncating the top levels, is accounted for. Similarly, it takes into account every possible inverse pyramid and its sub-pyramids that can be formed from the bottom up.

The dynamic programming data structure helps bypass the need to examine each potential pyramid individually, which would be prohibitively time-consuming. Instead, by building up from the smallest to the largest pyramid and using the information already computed, we achieve a solution that is much more efficient.

In summary, the provided solution makes use of:

- Dynamic Programming: Storing intermediate results to avoid recalculating them and to build up to the final result.
- Two-pass approach: Performing a pass for standard pyramids and another for inverse pyramids.
- Matrix Manipulation: Iterating through and updating a matrix to track the state of the problem at each step.
- Minimizing function: Using `min` to find the smallest pyramid that can be built on the level below, ensuring the pyramid property is sustained.

This approach relies on knowing that the pyramid's property is cumulative; any larger pyramid's base includes several smaller pyramids, which can be immediately inferred from the cell's `1` value.

Example Walkthrough

Consider a grid represented by the following 2D array:

```
1 grid =
2 [ [1, 1, 0],
3   [1, 1, 1],
4   [0, 1, 0] ]
5
```

We want to count all possible pyramidal and inverse pyramidal plots within this grid. In this small example, we will walk through the solution approach to illustrate how the count is determined.

Let's first initialize our dynamic programming array `f` to match the grid dimensions:

```
1 f = [
2   [0, 0, 0],
3   [0, 0, 0],
4   [0, 0, 0]
5 ]
```

We initialize with `0` for potentially fertile plots (not on the outer boundary initially) and `-1` on the outer boundary to avoid considering them as potential apices of pyramids.

Counting Pyramidal Plots

1. Starting from the bottom row (row 2) and moving upwards:
   - At (2, 1), `f[2][1] = 0` (no pyramid possible as it's on the boundary).
   - At (2, 2), `f[2][2] = 0` (same reason as above).
2. Next row (row 1):
   - At (1, 1), `f[1][1] = min(f[2][1], f[2][2], f[2][1]) + 1 = min(0, 0, 0) + 1 = 1`.
   - At (1, 2), `f[1][2] = min(f[2][2], f[2][1], f[2][2]) + 1 = min(0, 0, 0) + 1 = 1`.

Counting the pyramids ending at each cell: `ans += f[1][1] + f[1][2] = 1 + 1 = 2`.

We stop here since the next row (row 0) has cells on the outer boundary and cannot be the apex of a pyramid.

Counting Inverse Pyramidal Plots

For inverse pyramids, reset `f` and follow a similar approach but starting from the top of the grid and moving downwards.

1. Starting from the top row (row 0) moving downwards:
   - At (0, 1), `f[0][1] = 0` (no inverse pyramid possible as it's on the boundary).
   - At (0, 2), `f[0][2] = 0` (same reason as above).
2. Next row (row 1):
   - At (1, 1), `f[1][1] = min(f[0][1], f[0][2], f[0][1]) + 1 = min(0, 0, 0) + 1 = 1`.
   - At (1, 2), `f[1][2] = min(f[0][2], f[0][1], f[0][2]) + 1 = min(0, 0, 0) + 1 = 1`.

Increment `ans` by the value of `f[1][1]` and `f[1][2]` for each fertile cell: `ans += f[1][1] + f[1][2] = 1 + 1 = 2`.

Therefore, the total count of all the pyramidal and inverse pyramidal plots is `ans = 2 (pyramidal) + 2 (inverse pyramidal) = 4`.

This simple example using a small grid illustrates how the dynamic approach efficiently computes the number of valid pyramidal and inverse pyramidal plots without enumerating each possibility. The approach can be applied to larger grids following the same logic.

Python Solution

```python
from typing import List

class Solution:
    def countPyramids(self, grid: List[List[int]]) -> int:
        # Dimensions of the grid
        rows, cols = len(grid), len(grid[0])

        # Initialize a matrix to store the heights of the pyramids.
        pyramid_heights = [[0] * cols for _ in range(rows)]

        # Variable to store the total count of pyramids.
        total_count = 0

        # Counting the pyramids upside down.
        for row in range(rows - 2, -1, -1):
            for col in range(1, cols - 1):
                # If the cell is fertile
                if grid[row][col] == 1:
                    pyramid_heights[row][col] = -1
                elif row > rows - 3 and col != cols - 1:
                    # Calculate the height of the pyramid with the current cell
                    # as the apex or place a pyramid with a valid shape on top
                    pyramid_heights[row][col] = min(
                        pyramid_heights[row + 1][col - 1],
                        pyramid_heights[row + 1][col],
                        pyramid_heights[row + 1][col + 1]
                    ) + 1
                    # Add the count of smaller pyramids contained in the current pyramid.
                    total_count += pyramid_heights[row][col]

        # Flipping the grid to count regular (upright) pyramids by reusing the heights matrix.
        for row in range(rows):
            for col in range(cols):
                if grid[row][col] == 0:
                    pyramid_heights[row][col] = -1
                elif row == 0 or col == 0 or col == cols - 1:
                    pyramid_heights[row][col] = 1
                else:
                    # This is similar to the above, but we're moving in the opposite direction.
                    pyramid_heights[row][col] = min(
                        pyramid_heights[row - 1][col - 1],
                        pyramid_heights[row - 1][col],
                        pyramid_heights[row - 1][col + 1]
                    ) + 1
                total_count += pyramid_heights[row][col]

        return total_count
```

Java Solution

```java
class Solution {
    public int countPyramids(int[][] grid) {
        int rows = grid.length;
        int cols = grid[0].length;
        int[][] pyramidSizes = new int[rows][cols];
        int pyramidCount = 0;

        // Bottom-up traversal to count all "upside-down" pyramids.
        for (int row = rows - 1; row >= 0; --row) {
            for (int col = 0; col < cols; ++col) {
                // If the cell is empty or at the boundary, it cannot form a pyramid.
                if (grid[row][col] == 0 || row == rows - 1 || col == 0 || col == cols - 1) {
                    pyramidSizes[row][col] = 0;
                } else {
                    // The size of the pyramid is limited by the size of pyramids in the three cells below.
                    pyramidSizes[row][col] = Math.min(pyramidSizes[row + 1][col - 1],
                                                      Math.min(pyramidSizes[row + 1][col], pyramidSizes[row + 1][col + 1])) + 1;
                    // Add to the total count of pyramids.
                    pyramidCount += pyramidSizes[row][col];
                }
            }
        }

        // Top-down traversal to count all "right-side-up" pyramids.
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                // If the cell is empty or at the boundary, it cannot form a pyramid.
                if (grid[row][col] == 0 || row == 0 || col == 0 || col == cols - 1) {
                    pyramidSizes[row][col] = 0;
                } else {
                    // The size of the pyramid is limited by the size of pyramids in the three cells above.
                    pyramidSizes[row][col] = Math.min(pyramidSizes[row - 1][col - 1],
                                                      Math.min(pyramidSizes[row - 1][col], pyramidSizes[row - 1][col + 1])) + 1;
                    // Add to the total count of pyramids,
                    // since this is a separate traversal,
                    // counts both the upside-down and right-side-up pyramids.
                    pyramidCount += pyramidSizes[row][col];
                }
            }
        }

        // Return the total count of both upside-down and right-side-up pyramids.
        return pyramidCount;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For min function
using namespace std;

class Solution {
public:
    // Counts the number of pyramids that can be found in the given grid
    int countPyramids(vector<vector<int>>& grid) {
        int rowCount = grid.size(); // Number of rows in the grid
        int colCount = grid[0].size(); // Number of columns in the grid
        vector<vector<int>> dp(rowCount, vector<int>(colCount, 0)); // Dynamic programming table
        int totalCount = 0; // Counts total number of pyramids

        // Count pyramids that are pointing upwards
        for (int i = rowCount - 1; i >= 0; --i) {
            for (int j = 0; j < colCount; ++j) {
                // Edge cases
                if (i == rowCount - 1 || j == 0 || j == colCount - 1) {
                    dp[i][j] = 0; // If the cell is 0, it can't be part of a pyramid
                } else if (grid[i][j] == 0) {
                    dp[i][j] = -1; // If the cell is 0, it can't be part of a pyramid
                } else {
                    // Calculate the largest pyramid that can be formed with cell (i,j) as the bottom vertex
                    dp[i][j] = min(dp[i + 1][j - 1], min(dp[i + 1][j], dp[i + 1][j + 1])) + 1;
                    // Add the number of pyramids to totalCount
                    totalCount += dp[i][j];
                }
            }
        }

        // Count pyramids that are pointing downwards
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                // Edge cases
                if (i == 0 || j == 0 || j == colCount - 1) {
                    dp[i][j] = 0; // If the cell is 0, it can't be part of a pyramid
                } else if (i > 1 && grid[i][j] == 0 && grid[i - 1][j] == 0 && grid[i - 2][j] == 0) {
                    // Border cells can't be the vertex of an inverted pyramid
                    dp[i][j] = 0;
                } else {
                    // Calculate the largest pyramid that can be formed with cell (i,j) as the top vertex
                    dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i - 1][j + 1])) + 1;
                    // Add the number of pyramids to totalCount (without counting the single-cell pyramids again)
                    totalCount += max(0, dp[i][j] - 1);
                }
            }
        }

        return totalCount; // Return the total number of pyramids in both directions
    }
};
```

Typescript Solution

```typescript
function countPyramids(grid: number[][]): number {
    const rowCount = grid.length; // Number of rows in the grid
    const colCount = grid[0].length; // Number of columns in the grid
    const dp: number[][] = Array.from({ length: rowCount }, () => Array(colCount).fill(0)); // Dynamic programming table
    let totalCount = 0; // Counts the total number of pyramids

    // Count pyramids pointing upwards
    for (let i = rowCount - 1; i >= 0; --i) {
        for (let j = 0; j < colCount; ++j) {
            // Edge cases
            // If the cell is 0, it can't be part of a pyramid
            if (i === rowCount - 1 || j === 0 || j === colCount - 1) {
                dp[i][j] = 0;
            } else if (grid[i][j] === 0) {
                dp[i][j] = -1;
            } else {
                // Calculate the largest pyramid that can be formed with cell (i,j) as the bottom vertex
                dp[i][j] = Math.min(dp[i + 1][j - 1], Math.min(dp[i + 1][j], dp[i + 1][j + 1])) + 1;
                // Add the number of pyramids to totalCount
                totalCount += dp[i][j];
            }
        }
    }

    // Count pyramids that are pointing downwards
    for (let i = 0; i < rowCount; ++i) {
        for (let j = 0; j < colCount; ++j) {
            // Edge cases
            // If the cell is 0, it can't be part of a pyramid
            if (i === 0 || j === 0 || j === colCount - 1) {
                dp[i][j] = 0;
            } else if (i > 1 && grid[i][j] === 0 && grid[i - 1][j] === 0 && grid[i - 2][j] === 0) {
                // Border cells can't be the vertex of an inverted pyramid
                dp[i][j] = 0;
            } else {
                // Calculate the largest pyramid that can be formed with cell (i,j) as the top vertex
                dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i - 1][j + 1])) + 1;
                // Add the number of pyramids to totalCount (without counting the single-cell pyramids again)
                totalCount += Math.max(0, dp[i][j] - 1);
            }
        }
    }

    // Return the total count of pyramids in both directions
    return totalCount;
}
```

Time and Space Complexity

Time Complexity

The given code consists of two main loops that iterate over all cells of the `grid`. Each cell is processed in constant time, except for a comparison which is also done in constant time.

The outer loops run once for each row in the `grid`. The same is rows, and for each row, the inner loop runs for each column, of which there are `n`. Thus, each of these two loops has `m * n` iterations. Since the outer loops are not nested, the time complexity for both loops combined is `O(m * n) + O(m * n)`, which simplifies to `O(m * n)`.

Therefore, the overall time complexity of the given code is `O(m * n)`.

Space Complexity

The space complexity is determined by the additional space required by the code, not including the input `grid`. In this case, a 2D list `f` of the same dimensions as `grid` is created to store the extents of pyramids buildable at each point, which requires `m * n` space.

Thus, the space complexity of the code is `O(m * n)`.