2088. Count Fertile Pyramids in a Land Matrix Array **Dynamic Programming** Hard

Problem Description

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 its narrowest point at the bottom and gets wider towards the top. A few conditions must be satisfied for the plots:

The problem requires us to find and count all the possible pyramidal and inverse pyramidal plots in a farmer's rectangular grid land,

Leetcode Link

• 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 add 1 to that minimum height to obtain the height of the pyramid for which this cell could be the apex.

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

cell.

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 [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

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

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:

2. Use two for-loop blocks in the code to iterate over the entire grid twice — once for pyramids and once for inverse pyramids.

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

3. In the first for-loop block, calculate the number of levels for pyramidal plots:

at each cell.

 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 f at the three cells below and adjacent, plus one (to account for the current cell being a potential apex of a new pyramid level). 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:
- 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.

Reset the state of f before starting this block, taking care of the borders where the pyramid cannot be formed.

Start from the top row and move downwards, as inverse pyramids widen towards the top.

5. Return the total count ans as the sum of pyramid and inverse pyramid counts.

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

This solution ensures that every possible pyramid, as well as its sub-pyramids resulting from truncating the top levels, is accounted

 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.

Minimizing function: Using min to find the smallest pyramid that can be built on the level below, ensuring the pyramid property is

Matrix Manipulation: Iterating through and updating a matrix to track the state of the problem at each step.

Example Walkthrough Consider a grid represented by the following 2D array:

This approach relies on knowing that the pyramid's property is cumulative; any larger pyramid's base includes several smaller

solution approach to illustrate how the count is determined. Let's first initialize our dynamic programming array f to match the grid dimensions:

We initialize with ∅ for potentially fertile plots (not on the outer boundary initially) and -1 on the outer boundary to avoid considering

We want to count all possible pyramidal and inverse pyramidal plots within this grid. In this small example, we will walk through the

Counting Pyramidal Plots

them as potential apices of pyramids.

[-1, 0, 0, -1],

[-1, 0, 0, 0,],

[-1, 0, 0, -1],

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).

1. Starting from the top row (row 0) moving downwards:

computed, we achieve a solution that is much more efficient.

pyramids, which can be immediately inferred from the cell's f value.

In summary, the provided solution makes use of:

sustained.

1 grid = [

1 f = [

```
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

2. Next row (row 1):

 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):

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.

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.

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

 \circ At (1, 1), f[1][1] = min(f[2][1], f[2][2], f[2][1]) + 1 = min(0, 0, 0) + 1 = 1.

 \circ At (1, 2), f[1][2] = min(f[2][2], f[2][1], f[2][2]) + 1 = min(0, 0, 0) + 1 = 1.

 \circ At (1, 1), f[1][1] = min(f[0][1], f[0][2], f[0][1]) + 1 = min(0, 0, 0) + 1 = 1.

 \circ At (1, 2), f[1][2] = min(f[0][2], f[0][1], f[0][2]) + 1 = min(0, 0, 0) + 1 = 1.

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])

Counting the pyramids upside down.

if grid[row][col] == 0:

 $pyramid_heights[row][col] = -1$

pyramid_heights[row][col] = min(

elif row != rows - 1 and col != 0 and col != cols - 1:

pyramid_heights[row + 1][col - 1],

pyramid_heights[row + 1][col + 1]

total_count += pyramid_heights[row][col]

total_count += pyramid_heights[row][col]

pyramid_heights[row + 1][col],

for row in range(rows -1, -1, -1):

for col in range(cols):

) + 1

) + 1

public int countPyramids(int[][] grid) {

int[][] pyramidSizes = new int[rows][cols];

for (int row = rows -1; row ≥ 0 ; row--) {

for (int row = 0; row < rows; row++) {</pre>

for (int col = 0; col < cols; col++) {</pre>

pyramidSizes[row][col] = 0;

for (int col = 0; col < cols; col++) {</pre>

pyramidSizes[row][col] = 0;

// Bottom-up traversal to count all "upside-down" pyramids.

// Add to the total count of pyramids.

// Top-down traversal to count all "right-side-up" pyramids.

// Add to the total count of pyramids,

// since this is a separate traversal,

pyramidCount += pyramidSizes[row][col];

pyramidCount += pyramidSizes[row][col];

// If the cell is empty or at the boundary, it cannot form a pyramid.

// 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) {

// counts both the upside-down and right-side-up pyramids.

// Return the total count of both upside-down and right-side-up pyramids.

pyramidSizes[row][col] = Math.min(pyramidSizes[row - 1][col - 1],

if (grid[row][col] == 0 || row == rows - 1 || col == 0 || col == cols - 1) {

pyramidSizes[row][col] = Math.min(pyramidSizes[row + 1][col - 1],

+ 1;

// The size of the pyramid is limited by the size of pyramids in the three cells below.

// The size of the pyramid is limited by the size of pyramids in the three cells above.

Math.min(pyramidSizes[row + 1][col], pyramidSizes[row + 1][col + 1]))

Math.min(pyramidSizes[row - 1][col], pyramidSizes[row - 1][col + 1]))

int rows = grid.length;

int pyramidCount = 0;

} else {

} else {

return pyramidCount;

int cols = grid[0].length;

return total_count

Python Solution

O

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

43

44

45

46

47

5

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

Java Solution

class Solution {

```
# Initialize a matrix to store the heights of the pyramids.
 8
 9
            pyramid_heights = [[0] * cols for _ in range(rows)]
10
11
            # Variable to store the total count of pyramids.
12
            total_count = 0
13
```

get the minimum height of pyramids formed below the current cell

Add the count of smaller pyramids contained in the current pyramid.

ensuring we can place a pyramid with a valid shape on top

```
30
           # Flipping the grid to count regular (upright) pyramids by reusing the heights matrix.
31
           for row in range(rows):
32
                for col in range(cols):
33
                   if grid[row][col] == 0:
34
                        pyramid_heights[row][col] = -1
35
                    elif row == 0 or col == 0 or col == cols - 1:
36
                        pyramid_heights[row][col] = 0
37
                   else:
38
                        # This is similar to the above, but we're moving in the opposite direction.
39
                        pyramid_heights[row][col] = min(
40
                            pyramid_heights[row - 1][col - 1],
41
                            pyramid_heights[row - 1][col],
42
                            pyramid_heights[row - 1][col + 1]
```

48 C++ Solution

```
1 #include <vector>
    #include <algorithm> // For min function
    using namespace std;
    class Solution {
  6 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
 10
             vector<vector<int>> dp(rowCount, vector<int>(colCount, 0)); // Dynamic programming table
 11
 12
             int totalCount = 0; // Counts total number of pyramids
 13
             // Count pyramids that are pointing upwards
 14
 15
             for (int i = rowCount - 1; i >= 0; --i) {
 16
                 for (int j = 0; j < colCount; ++j) {</pre>
 17
                     if (grid[i][j] == 0) {
 18
                         // If the cell is 0, it can't be part of a pyramid
 19
                         dp[i][j] = -1;
                     } else if (i == rowCount - 1 || j == 0 || j == colCount - 1) {
 20
 21
                         // Border cells can't be the vertex of an upright pyramid
 22
                         dp[i][j] = 0;
 23
                     } else {
 24
                         // 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], dp[i + 1][j], dp[i + 1][j + 1]}) + 1;
 25
 26
                         // Add the number of pyramids to totalCount
 27
                         totalCount += dp[i][j];
 28
 29
 30
 31
 32
             // Count pyramids that are pointing downwards
 33
             for (int i = 0; i < rowCount; ++i) {</pre>
 34
                 for (int j = 0; j < colCount; ++j) {</pre>
 35
                     if (grid[i][j] == 0) {
 36
                         // If the cell is 0, it can't be part of a pyramid
                         dp[i][j] = -1;
 37
 38
                     } else if (i == 0 || j == 0 || j == colCount - 1) {
                         // Border cells can't be the vertex of an inverted pyramid
 39
 40
                         dp[i][j] = 0;
                     } else {
 41
 42
                         // Calculate the largest pyramid that can be formed with cell [i,j] as the top vertex
 43
                         dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i - 1][j + 1]}) + 1;
                         // Add the number of pyramids to totalCount (without counting the single-cell pyramids again)
 44
                         totalCount += dp[i][j];
 45
 46
 47
 48
 49
             return totalCount; // Return the total number of pyramids in both directions
 50
 51
 52 };
 53
Typescript Solution
```

// Number of rows in the grid

// 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], dp[i + 1][j], dp[i + 1][j + 1]) + 1;

const dp: number[][] = Array.from({ length: rowCount }, () => Array(colCount).fill(0)); // Dynamic programming table

// Calculate the largest pyramid that can be formed with cell [i, j] as the top vertex 38 39 dp[i][j] = Math.min(dp[i - 1][j - 1], dp[i - 1][j], dp[i - 1][j + 1]) + 1;40 // Add the number of pyramids to totalCount (without counting the single-cell pyramids again) 41 // Subtract 1 to avoid double-counting single-cell pyramids totalCount += (dp[i][j] > 0 ? dp[i][j] - 1 : 0); 42 43

} else {

return totalCount;

Time and Space Complexity

1 function countPyramids(grid: number[][]): number {

for (let $i = rowCount - 1; i >= 0; --i) {$

for (let j = 0; j < colCount; ++j) {</pre>

const colCount = grid[0].length; // Number of columns in the grid

let totalCount = 0; // Counts the total number of pyramids

// If the cell is 0, it can't be part of a pyramid

// Add the number of pyramids to totalCount

// Initialize dp for counting downward-pointing pyramids

// If the cell is 0, it can't be part of a pyramid

} else if (i === 0 || j === 0 || j === colCount - 1) {

// Return the total count of pyramids in both directions

// Border cells can't be the vertex of an inverted pyramid

} else if (i === rowCount - 1 || j === 0 || j === colCount - 1) {

// Border cells can't be the vertex of an upright pyramid

const rowCount = grid.length;

// Count pyramids pointing upwards

totalCount += dp[i][j];

dp.forEach(row => row.fill(0));

if (grid[i][j] === 0) {

dp[i][j] = -1;

dp[i][j] = 0;

// Count pyramids pointing downwards

for (let i = 0; i < rowCount; ++i) {</pre>

for (let j = 0; j < colCount; ++j) {</pre>

if (grid[i][j] === 0) {

dp[i][j] = -1;

dp[i][j] = 0;

} else {

5

6

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

44

45

46

47

48

49

50

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. There are m 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 0(m * n) + 0(m * n), which simplifies to 0(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.

Therefore, the overall time complexity of the given code is 0(m * n).

Thus, the space complexity of the code is 0(m * n).