1594. Maximum Non Negative Product in a Matrix Medium Array Dynamic Programming Matrix

Leetcode Link

Problem Description

In this problem, you are given a two-dimensional matrix where each cell contains an integer. Starting from the top-left corner of the matrix (0, 0), you need to find a path that leads you to the bottom-right corner (m - 1, n - 1). The only allowed moves are to the right or down from your current position. Your goal is to find the path that results in the maximum non-negative product of all the numbers on that path. The product of a path

is calculated by multiplying all the values of the cells visited on that path. Finally, you'll need to return this maximum product modulo 10^9 + 7. If no such non-negative product exists, you should return -1.

Intuition The problem can be tackled using dynamic programming. The key observation is that the maximum product at any given cell will either be the result of multiplying the maximum or the minimum product up to that point by the value of the current cell. This is

because multiplying a negative value by another negative value could give a positive product, which might be the maximum.

1. dp[i][j][0]: The minimum product to reach cell (i, j). This is important to keep track of because a negative minimum product can become a positive product if we multiply it by a negative number in the grid.

2. dp[i][j][1]: The maximum product to reach cell (i, j). For the first row and first column, the maximum and minimum products can only come from one direction, so we just multiply the current cell value by the value in the dp array of the previous cell in the same row or column.

We use a 3-dimensional dynamic programming (dp) array where each cell at (i, j) holds two values:

As we iterate through the inner cells of the grid, we calculate the possible minimum and maximum products using the previously filled cells above and to the left. We calculate the new values to store by considering the current cell's value: If the current cell's value is non-negative, we achieve the minimum product by multiplying the cell's value by the minimum of the

two possible pre-calculated minimum products. We obtain the maximum product by multiplying the cell's value by the maximum

of the two possible pre-calculated maximum products. If the cell value is negative, the minimum product could result in the largest product (if later multiplied by another negative

value), hence we take the maximum of the pre-calculated maximum products and multiply it by the cell's value. Conversely, the maximum value is achieved by taking the minimum of the pre-calculated minimum products and multiplying by the cell's value. After calculating dp values for the entire grid, we look at the maximum product of the last cell. If the maximum is negative, return -1.

Solution Approach The solution to this problem involves understanding the intricacies of dynamic programming and how to effectively handle both positive and negative numbers while searching for the maximum product path.

The dp array is initialized with a size of m x n x 2, where m and n are the dimensions of the input grid. Each cell in dp will hold

2. Filling the first row and column:

grid value.

- 1][0] (left).

- 1] [0] (left).

and thus we return -1.

requirement.

Example Walkthrough

1 Grid:

2 [1, -2, 1]

1. Initialization:

2. Filling the first row and column:

3. Filling the rest of the dp array:

First row: Iterate over j from 1 to 2.

First column: Iterate over i from 1 to 2.

4. Getting the result:

1. Initialization:

Otherwise, return the maximum product modulo 10^9 + 7.

 Each cell in the first row and column can only be reached from one direction. Thus, for the first row, we iterate over j from 1 to n, and set dp[0][j] equal to the product of dp[0][j - 1][0] (the previous cell) and grid[0][j]. We do the same for the first column, iterating over i from 1 to m, setting each dp[i][0] according to the product of the previous cell and the current

We iterate over the cells of the grid starting from (1, 1) and for each cell at (i, j), we calculate the minimum and

■ Find dp[i][j][0] (min product) by multiplying grid[i][j] with the smaller of either dp[i - 1][j][0] (above) or dp[i][j

Calculate dp[i][j][1] (max product) by multiplying grid[i][j] with the larger of either dp[i - 1][j][1] (above) or

dp[i][j - 1][1] (left). If grid[i][j] is negative, we switch the above, since minimum times a negative can lead to a

■ Find dp[i][j][1] (max product) by multiplying grid[i][j] with the smaller of either dp[i - 1][j][0] (above) or dp[i][j]

The first cell, dp[0][0], is initialized with the value of the first cell of the grid, grid[0][0], as both the minimum and

3. Filling the rest of the dp array:

The algorithm uses a 3-dimensional dynamic programming array called dp. Here's the step-by-step breakdown:

two numbers: the minimum and maximum product up to that point in the grid.

maximum, since there's only one number in the path at this point.

The crux of the approach: If grid[i][j] is non-negative, we:

- maximum product paths to that point. This involves comparisons and multiplications using the values from the top (1 1, j) and left (i, j-1) cells.
 - maximum product, and vice versa: ■ Calculate dp[i][j][0] (min product) by multiplying grid[i][j] with the larger of either dp[i - 1][j][1] (above) or dp[i][j - 1][1] (left).

After filling in the entire dp array, we look at the bottom-right corner (last cell) of the grid dp[m - 1][n - 1]. Specifically,

If the maximum product path is negative, this indicates there are no non-negative product paths to the bottom-right corner,

with both positive and negative integers, ensuring the computations adhere to the maximum non-negative product principle.

∘ If the maximum product is non-negative, we return the maximum product modulo 10^9 + 7 as per the problem's

The solution demonstrates the utilization of dynamic programming to maintain and compare potential product paths through a grid

we're interested in the maximum product path, which is stored in dp[m-1][n-1][1].

Let's consider a small 3×3 matrix as an example to illustrate the solution approach:

Here's a step-by-step application of the solution approach:

o Initialize dp as a 3-dimensional array with the dimensions of the grid and an extra dimension to hold both minimum and maximum products. For our 3×3 grid, dp will have dimensions $3\times3\times2$.

■ Calculate minimum as $min(dp[0][1][0], dp[1][0][0]) * 3 <math>\Rightarrow min(-2, 1) * 3 \Rightarrow 1 * 3 \Rightarrow 3$.

■ Calculate maximum as $\max(dp[0][1][1], dp[1][0][1]) * 3 ⇒ \max(-2, 1) * 3 ⇒ 1 * 3 ⇒ 3.$

■ Calculate the minimum as $\max(dp[0][2][1], dp[1][1][1]) * -2 \Rightarrow \max(-2, 3) * -2 \Rightarrow 3 * -2 \Rightarrow -6$.

■ Calculate the maximum as min(dp[0][2][0], dp[1][1][0]) * -2 ⇒ <math>min(-2, 3) * -2 ⇒ -2 * -2 ⇒ 4.

Looking at the last cell dp[2][2], we have the maximum product as 16. Since it is non-negative, we return 16 % (10^9 + 7),

This walk-through demonstrates how the dynamic programming array is filled out in order to maximize the product path from the

Set dp [0] [0] to [1, 1] since the first cell of the grid is 1, and there's only one path with one cell at the starting point.

Now, we iterate over the inner cells starting from (1, 1). \circ For i = 1 and j = 1 (grid[1][1] is 3, a positive value):

Set dp[1][1] to [3, 3]. \circ For i = 1 and j = 2 (grid[1][2] is -2, a negative value):

■ Set dp[2][1] to [-6, 2].

which is 16.

Python Solution

1 class Solution:

3

4

5

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

30

31

32

33

34

35

36

37

38

4

5

6

8

9

10

11

12

13

14

15

16

17

18

19

top-left to the bottom-right of the grid.

mod = 10**9 + 7

for i in range(1, rows):

for j in range(1, cols):

for i in range(1, rows):

ans = dp[-1][-1][1]

for j in range(1, cols):

if value >= 0:

value = grid[i][j]

private static final int MOD = (int) 1e9 + 7;

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

// Get the dimensions of the grid.

int rows = grid.length;

int cols = grid[0].length;

dp[0][0][0] = grid[0][0];

dp[0][0][1] = grid[0][0];

// Function to calculate the maximum product of paths.

// dp[i][j][1] will store the maximum product.

long[][][] dp = new long[rows][cols][2];

// Initialize the first cell of the grid.

 \circ For i = 2 and j = 2 (grid[2][2] is 4, a positive value):

def maxProductPath(self, grid: List[List[int]]) -> int:

Define the modulo value for the final result.

Initialize the first column of the grid.

Initialize the first row of the grid.

dp = [[[0] * 2 for _ in range(cols)] for _ in range(rows)]

Compute the min and max products for the whole grid.

The answer is the max product for the bottom-right cell.

If the answer is negative, there is no non-negative product path.

// 3D DP array to store the min and max product up to each cell.

// dp[i][j][0] will store the minimum product up to grid[i][j],

return -1 if ans < 0 else ans % mod # Return the result modulo 10^9+7.

Get the dimensions of the grid.

rows, cols = len(grid), len(grid[0])

Set dp[1][2] to [-6, 4]. \circ For i = 2 and j = 1 (grid[2][1] is -2, a negative value): ■ Minimum: $\max(dp[1][1][1], dp[2][0][1]) * -2 \Rightarrow \max(3, -1) * -2 \Rightarrow 3 * -2 \Rightarrow -6$. ■ Maximum: min(dp[1][1][0], dp[2][0][0]) * -2 ⇒ <math>min(3, -1) * -2 ⇒ -1 * -2 ⇒ 2.

■ For j = 1, dp[0][1] is initialized as $[1 * (-2), 1 * (-2)] \Rightarrow [-2, -2]$.

■ For j = 2, dp[0][2] becomes $[-2 * 1, -2 * 1] \Rightarrow [-2, -2]$.

■ For i = 2, dp[2][0] becomes $[1 * (-1), 1 * (-1)] \Rightarrow [-1, -1]$.

■ For i = 1, dp[1][0] is $[1 * 1, 1 * 1] \Rightarrow [1, 1]$.

Set dp[2][2] to [-24, 16]. 4. Getting the result:

Initialize a 3D dp array where dp[i][j] will store the min and max product up to (i, j).

dp[i][j][0] = max(dp[i-1][j][1], dp[i][j-1][1]) * value

dp[i][j][1] = min(dp[i - 1][j][0], dp[i][j - 1][0]) * value

dp[0][0] = [grid[0][0], grid[0][0]] # Base case: min and max products for the starting cell.

dp[i][0] = [dp[i-1][0][0] * grid[i][0]] * 2 # Copy min and max since they are the same here.

dp[0][j] = [dp[0][j-1][0] * grid[0][j]] * 2 # Copy min and max since they are the same here.

■ Minimum: min(dp[1][2][0], dp[2][1][0]) * 4 ⇒ <math>min(-6, -6) * 4 ⇒ -6 * 4 ⇒ -24.

■ Maximum: $max(dp[1][2][1], dp[2][1][1]) * 4 \Rightarrow max(4, 2) * 4 \Rightarrow 4 * 4 \Rightarrow 16.$

The current value is non-negative, so the min product is the min of min products from above and left cells. 24 25 dp[i][j][0] = min(dp[i - 1][j][0], dp[i][j - 1][0]) * value26 # The max product is the max of max products from above and left cells. 27 dp[i][j][1] = max(dp[i - 1][j][1], dp[i][j - 1][1]) * value28 else: 29 # The current value is negative, so the min product becomes the max product from above/left, and vice versa.

Java Solution 1 class Solution { // Define modulus for the result to keep the result within integer range.

1 using ll = long long;

class Solution {

public:

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

48

49

50

52

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

51 };

2 const int MOD = 1e9 + 7;

// Function to find the maximum product path in a given grid

// dp[row][col][0] for minimum product up to (row, col)

// dp[row][col][1] for maximum product up to (row, col)

// Initialize the first cell with the value in the grid

// Creating a 3D vector to store the minimum and maximum products

vector<vector<vector<ll>>>> dp(rows, vector<vector<ll>>>(cols, vector<ll>(2, 0)));

int maxProductPath(vector<vector<int>>& grid) {

dp[0][0][0] = dp[0][0][1] = grid[0][0];

for (int i = 1; i < rows; ++i) {

for (int j = 1; j < cols; ++j) {

for (int i = 1; i < rows; ++i) {

if (value >= 0) {

} else {

// Fill the first column (all rows, column 0)

// Fill the first row (row 0, all columns)

for (int j = 1; j < cols; ++j) {

int value = grid[i][j];

ll result = dp[rows - 1][cols - 1][1];

// Calculate dp values for the rest of the grid

// If current cell value is non-negative

int rows = grid.size();

int cols = grid[0].size();

```
20
             // Initialize the first column of the grid.
 21
             for (int i = 1; i < rows; ++i) {
 22
                 dp[i][0][0] = dp[i - 1][0][0] * grid[i][0];
 23
                 dp[i][0][1] = dp[i - 1][0][1] * grid[i][0];
 24
 25
 26
             // Initialize the first row of the grid.
 27
             for (int j = 1; j < cols; ++j) {
 28
                 dp[0][j][0] = dp[0][j - 1][0] * grid[0][j];
 29
                 dp[0][j][1] = dp[0][j - 1][1] * grid[0][j];
 30
 31
 32
             // Fill the DP table.
 33
             for (int i = 1; i < rows; ++i) {
                 for (int j = 1; j < cols; ++j) {
 34
 35
                     int value = grid[i][j];
 36
 37
                     // When the current value is non-negative.
 38
                     if (value >= 0) {
                         dp[i][j][0] = Math.min(dp[i - 1][j][0], dp[i][j - 1][0]) * value;
 39
                         dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i][j - 1][1]) * value;
 40
 41
                     } else {
 42
                         // When the current value is negative, flip the min and max.
 43
                         dp[i][j][0] = Math.max(dp[i - 1][j][1], dp[i][j - 1][1]) * value;
 44
                         dp[i][j][1] = Math.min(dp[i - 1][j][0], dp[i][j - 1][0]) * value;
 45
 46
 47
 48
 49
             // Get the max product from the bottom-right cell of the grid.
 50
             long maxProduct = dp[rows - 1][cols - 1][1];
 51
 52
             // If the max product is negative, return -1, otherwise return the max product modulo MOD.
 53
             return maxProduct < 0 ? -1 : (int) (maxProduct % MOD);
 54
 55 }
 56
C++ Solution
```

// Number of rows in the grid

// Number of columns in the grid

dp[i][0][0] = dp[i - 1][0][0] * grid[i][0]; // Product could be positive or negative

dp[0][j][0] = dp[0][j - 1][0] * grid[0][j]; // Product could be positive or negative

dp[0][j][1] = dp[0][j][0]; // Both min and max are the same for the first row

dp[i][j][0] = min(dp[i-1][j][0], dp[i][j-1][0]) * value;

dp[i][j][1] = max(dp[i - 1][j][1], dp[i][j - 1][1]) * value;

dp[i][j][0] = max(dp[i-1][j][1], dp[i][j-1][1]) * value;

dp[i][j][1] = min(dp[i - 1][j][0], dp[i][j - 1][0]) * value;

// If current cell value is negative, min and max swap

// The final result is the maximum product path to the bottom-right cell

return result < 0 ? -1 : static_cast<int>(result % MOD);

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

const dp: ll[][][] = Array.from({ length: rows }, () =>

// Initialize the first cell with the value in the grid

dp[i][0][0] = dp[i - 1][0][0] * BigInt(grid[i][0]);

dp[0][j][0] = dp[0][j - 1][0] * BigInt(grid[0][j]);

// If current cell value is non-negative

dp[0][0][0] = dp[0][0][1] = BigInt(grid[0][0]);

// Fill in the first column (all rows, column 0)

// Fill in the first row (row 0, all columns)

// Calculate dp values for the rest of the grid

const value: ll = BigInt(grid[i][j]);

for (let j = 1; j < cols; ++j) {

if (value >= BigInt(0)) {

for (let i = 1; i < rows; ++i) {

for (let j = 1; j < cols; ++j) {

for (let i = 1; i < rows; ++i) {

} else {

// Create a 3D array to store the minimum and maximum products

Array.from({ length: cols }, () => Array(2).fill(BigInt(0)))

dp[i][0][1] = dp[i][0][0]; // Both min and max are the same for the first column

dp[0][j][1] = dp[0][j][0]; // Both min and max are the same for the first row

dp[i][j][0] = llMin(dp[i - 1][j][0], dp[i][j - 1][0]) * value;

dp[i][j][1] = llMax(dp[i - 1][j][1], dp[i][j - 1][1]) * value;

dp[i][j][0] = llMax(dp[i - 1][j][1], dp[i][j - 1][1]) * value;

dp[i][j][1] = llMin(dp[i - 1][j][0], dp[i][j - 1][0]) * value;

// If current cell value is negative, min and max swap

// If the result is negative, return -1, else return the result modulo MOD

dp[i][0][1] = dp[i][0][0]; // Both min and max are the same for the first column

1 type ll = bigint; // Using TypeScript alias for bigint 2 const MOD: ll = BigInt(1e9 + 7); // MOD value as a bigint // Function to find the maximum product path in a given grid function maxProductPath(grid: number[][]): number { const rows = grid.length; // Number of rows in the grid 6

);

Typescript Solution

```
42
 43
 44
        // The final result is the maximum product path to the bottom-right cell
         const result: ll = dp[rows - 1][cols - 1][1];
 45
 46
        // If the result is negative, return -1, else return the result modulo MOD
 47
         return result < BigInt(0) ? -1 : Number(result % MOD);</pre>
 48
 49
    // Helper function to find the minimum of two bigints
    function llMin(a: ll, b: ll): ll {
 52
         return a < b ? a : b;
 53 }
 54
    // Helper function to find the maximum of two bigints
    function llMax(a: ll, b: ll): ll {
         return a > b ? a : b;
 57
 58
 59
Time and Space Complexity
Time Complexity
```

The time complexity of the code is determined by the nested loops that iterate over each cell of the matrix grid. Since there are two

loops, one going through all rows (m) and the other through all columns (n), and the operations inside the inner loop take O(1) time,

size of the input grid.

product paths up to each cell. The size of this list dictates the space complexity, which is 0(m * n) as it's directly proportional to the

Space Complexity For space complexity, the code is using a 3-dimensional list dp with dimensions m x n x 2 to store the minimum and maximum

the time complexity is 0(m * n).