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.

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

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.

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

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

1. **Initialization**:
   - 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 two numbers: the minimum and maximum product up to that point in the grid.
   - 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 maximum, since there's only one number in the path at this point.
2. **Filling the first row and column**:
   - 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 grid value.
3. **Filling the rest of the dp array**:
   - We iterate over the cells of the grid starting from `(1, 1)` and for each cell at `(i, j)`, we calculate the minimum and maximum product paths to that point. This involves comparisons and multiplications using the values from the top `(i - 1, j)` and left `(i, j - 1)` cells.
   - The crux of the approach: If `grid[i][j]` is non-negative, we:
     - 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 - 1][0]` (left).
     - 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 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).
     - 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 - 1][0]` (left).
4. **Getting the result**:
   - 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, we're interested in the maximum product path, which is stored in `dp[m - 1][n - 1][1]`.
   - If the maximum product path is negative, this indicates there are no non-negative product paths to the bottom-right corner, and thus we return `-1`.
   - If the maximum product is non-negative, we return the maximum product modulo `10^9 + 7` as per the problem's requirement.

The solution demonstrates the utilization of dynamic programming to maintain and compare potential product paths through a grid with both positive and negative integers, ensuring the computations adhere to the maximum non-negative product principle.

Example Walkthrough

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

```
1 Grid:
2 [1,  -2, 1]
3 [1,   3, -2]
4 [-1, -2, 4]
```

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

1. **Initialization**:
   - 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 x 3 x 2`.
   - 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.
2. **Filling the first row and column**:
   - First row: Iterate over `j` from 1 to 2.
     - For `j = 1`, `dp[0][1]` is initialized as `[1 * (-2), 1 * (-2)]` ⇒ `[-2, -2]`.
     - For `j = 2`, `dp[0][2]` becomes `[-2 * 1, -2 * 1]` ⇒ `[-2, -2]`.
   - First column: Iterate over `i` from 1 to 2.
     - For `i = 1`, `dp[1][0]` is `[1 * 1, 1 * 1]` ⇒ `[1, 1]`.
     - For `i = 2`, `dp[2][0]` becomes `[1 * (-1), 1 * (-1)]` ⇒ `[-1, -1]`.
3. **Filling the rest of the dp array**:
   - Now, we iterate over the inner cells starting from `(1, 1)`.
   - For `i = 1` and `j = 1` (`grid[1][1]` is `3`, a positive value):
     - Calculate minimum as `min(dp[0][1][0], dp[1][0][0]) * 3` ⇒ `min(-2, 1) * 3` ⇒ `1 * 3` ⇒ `3`.
     - Calculate maximum as `max(dp[0][1][1], dp[1][0][1]) * 3` ⇒ `max(-2, 1) * 3` ⇒ `1 * 3` ⇒ `3`.
     - Set `dp[1][1]` to `[3, 3]`.
   - For `i = 1` and `j = 2` (`grid[1][2]` is `-2`, a negative value):
     - Calculate the minimum as `max(dp[0][2][1], dp[1][1][1]) * -2` ⇒ `max(-2, 3) * -2` ⇒ `3 * -2` ⇒ `-6`.
     - Calculate the maximum as `min(dp[0][2][0], dp[1][1][0]) * -2` ⇒ `min(-2, 3) * -2` ⇒ `-2 * -2` ⇒ `4`.
     - Set `dp[1][2]` to `[-6, 4]`.
   - 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` ⇒ `max(3, -1) * -2` ⇒ `3 * -2` ⇒ `-6`.
     - Maximum: `min(dp[1][1][0], dp[2][0][0]) * -2` ⇒ `min(3, -1) * -2` ⇒ `-1 * -2` ⇒ `2`.
     - Set `dp[2][1]` to `[-6, 2]`.
   - For `i = 2` and `j = 2` (`grid[2][2]` is `4`, a positive value):
     - Minimum: `min(dp[1][2][0], dp[2][1][0]) * 4` ⇒ `min(-6, -6) * 4` ⇒ `-6 * 4` ⇒ `-24`.
     - Maximum: `max(dp[1][2][1], dp[2][1][1]) * 4` ⇒ `max(4, 2) * 4` ⇒ `4 * 4` ⇒ `16`.
     - Set `dp[2][2]` to `[-24, 16]`.
4. **Getting the result**:
   - 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)`, which is `16`.

This walk-through demonstrates how the dynamic programming array is filled out in order to maximize the product path from the top-left to the bottom-right of the grid.

Python Solution

```python
class Solution:
    def maxProductPath(self, grid: List[List[int]]) -> int:
        # Get the dimensions of the grid.
        rows, cols = len(grid), len(grid[0])
        # Define the modulo value for the final result.
        mod = 10**9 + 7
        # Initialize a 3D dp array where dp[i][j] will store the min and max product up to (i, j).
        dp = [[[0] * 2 for _ in range(cols)] for _ in range(rows)]
        dp[0][0] = [grid[0][0], grid[0][0]] # Base case: min and max products for the starting cell.

        # Initialize the first column of the grid.
        for i in range(1, rows):
            dp[i][0] = [dp[i - 1][0][0] * grid[i][0]] * 2 # Copy min and max since they are the same here.

        # Initialize the first row of the grid.
        for j in range(1, cols):
            dp[0][j] = [dp[0][j - 1][0] * grid[0][j]] * 2 # Copy min and max since they are the same here.

        # Compute the min and max products for the whole grid.
        for i in range(1, rows):
            for j in range(1, cols):
                value = grid[i][j]
                if value >= 0:
                    # The current value is non-negative, so the min product is the min of min products from above and left cells.
                    dp[i][j][0] = min(dp[i - 1][j][0], dp[i][j - 1][0]) * value
                    # The max product is the max of max products from above and left cells.
                    dp[i][j][1] = max(dp[i - 1][j][1], dp[i][j - 1][1]) * value
                else:
                    # The current value is negative, so the min product becomes the max product from above/left, and vice versa.
                    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

        # The answer is the max product for the bottom-right cell.
        ans = dp[-1][-1][1]

        # If the answer is negative, there is no non-negative product path.
        return -1 if ans < 0 else ans % mod # Return the result modulo 10^9+7.
```

Java Solution

```java
class Solution {
    // Define modulus for the result to keep the result within integer range.
    private static final int MOD = (int) 1e9 + 7;

    // Function to calculate the maximum product of paths.
    public int maxProductPath(int[][] grid) {
        // Get the dimensions of the grid.
        int rows = grid.length;
        int cols = grid[0].length;

        // 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],
        // dp[i][j][1] will store the maximum product.
        long[][][] dp = new long[rows][cols][2];

        // Initialize the first cell of the grid.
        dp[0][0][0] = grid[0][0];
        dp[0][0][1] = grid[0][0];

        // Initialize the first column of the grid.
        for (int i = 1; i < rows; ++i) {
            dp[i][0][0] = dp[i - 1][0][0] * grid[i][0];
            dp[i][0][1] = dp[i - 1][0][1] * grid[i][0];
        }

        // Initialize the first row of the grid.
        for (int j = 1; j < cols; ++j) {
            dp[0][j][0] = dp[0][j - 1][0] * grid[0][j];
            dp[0][j][1] = dp[0][j - 1][1] * grid[0][j];
        }

        // Fill the DP table.
        for (int i = 1; i < rows; ++i) {
            for (int j = 1; j < cols; ++j) {
                int value = grid[i][j];

                // When the current value is non-negative.
                if (value >= 0) {
                    dp[i][j][0] = Math.min(dp[i - 1][j][0], dp[i][j - 1][0]) * value;
                    dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i][j - 1][1]) * value;
                } else {
                    // When the current value is negative, flip the min and max.
                    dp[i][j][0] = Math.max(dp[i - 1][j][1], dp[i][j - 1][1]) * value;
                    dp[i][j][1] = Math.min(dp[i - 1][j][0], dp[i][j - 1][0]) * value;
                }
            }
        }

        // Get the max product from the bottom-right cell of the grid.
        long maxProduct = dp[rows - 1][cols - 1][1];

        // If the max product is negative, return -1, otherwise return the max product modulo MOD.
        return maxProduct < 0 ? -1 : (int) (maxProduct % MOD);
    }
}
```

C++ Solution

```cpp
using ll = long long;
const int MOD = 1e9 + 7;

class Solution {
public:
    // Function to find the maximum product path in a given grid
    int maxProductPath(vector<vector<int>>& grid) {
        int rows = grid.size();          // Number of rows in the grid
        int cols = grid[0].size();       // Number of columns in the grid
        // Creating a 3D vector to store the minimum and maximum products
        // dp[row][col][0] for minimum product up to (row, col)
        // dp[row][col][1] for maximum product up to (row, col)
        vector<vector<vector<ll>>> dp(rows, vector<vector<ll>>(cols, vector<ll>(2, 0)));

        // Initialize the first cell with the value in the grid
        dp[0][0][0] = dp[0][0][1] = grid[0][0];

        // Fill the first column (all rows, column 0)
        for (int i = 1; i < rows; ++i) {
            dp[i][0][0] = dp[i - 1][0][0] * grid[i][0]; // Product could be positive or negative
            dp[i][0][1] = dp[i][0][0]; // Both min and max are the same for the first column
        }

        // Fill the first row (row 0, all columns)
        for (int j = 1; j < cols; ++j) {
            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
        }

        // Calculate dp values for the rest of the grid
        for (int i = 1; i < rows; ++i) {
            for (int j = 1; j < cols; ++j) {
                int value = grid[i][j];
                if (value >= 0) {
                    // If current cell value is non-negative
                    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;
                } else {
                    // If current cell value is negative, min and max swap
                    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;
                }
            }
        }

        // The final result is the maximum product path to the bottom-right cell
        ll result = dp[rows - 1][cols - 1][1];
        // If the result is negative, return -1, else return the result modulo MOD
        return result < 0 ? -1 : static_cast<int>(result % MOD);
    }
};
```

Typescript Solution

```typescript
type ll = bigint; // Using TypeScript alias for bigint
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
    const cols = grid[0].length; // Number of columns in the grid
    // Create a 3D array to store the minimum and maximum products
    const dp: ll[][][] = Array.from({ length: rows }, () =>
        Array.from({ length: cols }, () => Array(2).fill(BigInt(0)))
    );

    // Initialize the first cell with the value in the grid
    dp[0][0][0] = dp[0][0][1] = BigInt(grid[0][0]);

    // Fill in the first column (all rows, column 0)
    for (let i = 1; i < rows; ++i) {
        dp[i][0][0] = dp[i - 1][0][0] * BigInt(grid[i][0]);
        dp[i][0][1] = dp[i][0][0]; // Both min and max are the same for the first column
    }

    // Fill in the first row (row 0, all columns)
    for (let j = 1; j < cols; ++j) {
        dp[0][j][0] = dp[0][j - 1][0] * BigInt(grid[0][j]);
        dp[0][j][1] = dp[0][j][0]; // Both min and max are the same for the first row
    }

    // Calculate dp values for the rest of the grid
    for (let i = 1; i < rows; ++i) {
        for (let j = 1; j < cols; ++j) {
            const value: ll = BigInt(grid[i][j]);
            if (value >= BigInt(0)) {
                // If current cell value is non-negative
                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;
            } else {
                // If current cell value is negative, min and max swap
                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;
            }
        }
    }

    // The final result is the maximum product path to the bottom-right cell
    const result: ll = dp[rows - 1][cols - 1][1];
    // If the result is negative, return -1, else return the result modulo MOD
    return result < BigInt(0) ? -1 : Number(result % MOD);
}

// Helper function to find the minimum of two bigints
function llMin(a: ll, b: ll): ll {
    return a < b ? a : b;
}

// Helper function to find the maximum of two bigints
function llMax(a: ll, b: ll): ll {
    return a > b ? a : b;
}
```

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, the time complexity is `O(m * n)`.

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