# 2906. Construct Product Matrix

Medium Array Matrix Prefix Sum Leetcode Link

## Problem Description

This problem provides a matrix `grid` of size `n * m` and requires us to construct a new matrix `p`, also of size `n * m`. The new matrix `p` is defined as the product matrix of `grid`, where each element `p[i][j]` is the product of all elements in `grid` excluding the element `grid[i][j]`, with each product taken modulo `12345`.

To elaborate, if we were to calculate `p[i][j]`, we would take the product of all the numbers in the matrix `grid` except for `grid[i][j]`, and then take the result modulo `12345`. Our task is to construct the entire matrix `p` that satisfies this condition for every single element.

## Intuition

Constructing the product matrix `p` can appear challenging at first because of the requirement to exclude the current element from the product. A naive approach might involve calculating the product of all elements for each position separately, which would lead to a time-consuming process because of the repeated multiplications. However, this problem can be efficiently solved by decomposing the operations into prefix and suffix products. This approach allows us to calculate the desired product for each element without redundancy.

The intuition for the solution is to first calculate the suffix product matrix, which stores the product of all elements that are below and to the right of the current position. This is done by traversing the matrix from the bottom right corner upwards. Following this, we can calculate the prefix product for each element by traversing the matrix from the top left corner downwards, multiplying the already calculated suffix product by the prefix and taking the modulo at each step. By doing these two separate traversals, we cleverly circumvent the need to directly exclude the current element (since it was never included in `suf` or `pre`, to begin with).

Ultimately, the `p[i][j]` equals to the multiplication of the suffix product (calculated in the first traversal) and the prefix product (calculated in the second traversal). Both of these are calculated modulo `12345` as per the problem statement. As we are making use of previously computed results, computational efficiency is greatly improved over the naive approach.

## Solution Approach

The problem dictates a classic use of Algorithmic Patterns in which the prefix and suffix multiplications are used to simplify the process of excluding a particular element from the product. The solution employs two important concepts: suffix and prefix product decomposition. To implement the solution, we initialize a product matrix `p` and two variables, `suf` and `pre`, which will hold the suffix and prefix products, respectively.

**Step 1**: Initialize the `suf` variable (`suffix product`) to 1. Here, `suf` acts as a running product of all elements to the right and below the current element, excluding it. Starting from the bottom right corner of the matrix, iterate over the matrix in reverse order. For each element `grid[i][j]`, we will set `p[i][j]` to the current `suf` value. Then, we update the `suf` by multiplying it by the current element in `grid` and taking the modulo `12345` as prescribed by the problem.

**Step 2**: The next traversal calculates the prefix product. Initialize the `pre` variable (`prefix product`) to 1. Starting from the top left corner of the matrix, iterate over the matrix in normal order. For each element `grid[i][j]`, we multiply the `p[i][j]` (which at this moment only contains the suffix product) by the current `pre`, take the modulo `12345`, and then update the `pre` with the current element in `grid`, also with a modulo operation.

The processes indicated in **Step 1** and **Step 2** ensure that each `p[i][j]` is calculated by multiplying the prefix product. Up to but not including `grid[i][j]`, with the suffix product of all elements after `grid[i][j]`, effectively excluding `grid[i][j]` from the product as required.

The reason we can do this in two separate steps is because of the mathematical property that allows us to separate the product of a series of numbers into separate parts, compute them independently, and then combine them to get the final product. This separation into prefix and suffix helps us avoid including the current element in either product. The modulo operation is an integral part of every product calculation due to the requirement to take each product modulo `12345`.

By employing this specific order of operation—suffix product calculation followed by prefix product multiplication—we avoid the use of division, which is beneficial when dealing with modulus operations as division is not well-defined under modulus without additional steps (like using a multiplicative inverse). Therefore, this technique not only meets the problem constraints but also adheres to the computational limitations when dealing with modular arithmetic.

Finally, after the completion of these two passes, the matrix `p` is returned, containing the desired product matrix. This approach efficiently computes the solution in `O(nxm)` time complexity with `O(nxm)` space complexity, where `n` and `m` are the dimensions of the input matrix `grid`.

### Example Walkthrough

Let's walk through the solution with a small example. Consider the following `grid` matrix of size `3x3`:

```
grid = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]
```

To construct the new matrix `p`, we will follow the solution approach detailed in the content.

**Step 1**: Initialize `suf` to 1 and traverse the matrix in reverse order. Let's start at the bottom-right corner:

```
Iteration 1: (i=2, j=2) => p[2][2] = suf (1) => p = [
  [?, ?, ?],
  [?, ?, ?],
  [?, ?, 1]
]
suf = suf * grid[2][2] % 12345 => suf = 1 * 9 % 12345 = 9
```

(This process repeats for the remaining elements of the matrix from the bottom-right to the top-left.)

After the first traversal, the `p` matrix will have the suffix products:

```
p = [
  [1, 1, 1],
  [1, 1, 2],
  [1, 9, 1]
]
```

**Step 2**: Initialize `pre` to 1 and traverse the matrix in normal order:

```
Iteration 1: (i=0, j=0) => p[0][0] = suf * pre % 12345 => p[0][0] = 1 * 1 % 12345 = 1
pre = pre * grid[0][0] % 12345 => pre = 1 * 1 % 12345 = 1
```

(In a similar fashion, we update each element in `p` for the whole matrix in normal order.)

After the second traversal, we fully calculate the `p` matrix:

```
p = [
  [(5*6*7*8*9) % 12345, (4*6*7*8*9) % 12345, (4*5*7*8*9) % 12345],
  [(2*3*7*8*9) % 12345, (1*3*7*8*9) % 12345, (1*2*7*8*9) % 12345],
  [(2*3*5*6) % 12345, (1*3*5*6) % 12345, (1*2*5*6) % 12345]
]
```

With some arithmetic, we find:

```
p = [
  [15120 % 12345, 60480 % 12345, 30240 % 12345],
  [60480 % 12345, 30240 % 12345, 40320 % 12345],
  [360 % 12345, 180 % 12345, 60 % 12345]
]
```

And, finally, the resulting p matrix is:

```
p = [
  [2775, 10530, 7895],
  [10530, 7895, 10080],
  [360, 180, 60]
]
```

The new `p` matrix is the product matrix of `grid`, where each entry is the product of all other elements except itself, modulo `12345`. Thus, by using the suffix and prefix product computation, we've successfully found the solution without directly excluding each `grid[i][j]` from the multiplication, proving the efficiency of this approach.

## Python Solution

```python
class Solution:
    def constructProductMatrix(self, grid: List[List[int]]) -> List[List[int]]:
        # Determine the number of rows and columns in the given grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Initialize a product matrix with zeros, of the same dimensions as the grid
        product_matrix = [[0] * num_cols for _ in range(num_rows)]

        # Define the modulo value as stated in the problem
        modulo = 12345

        # Initialize the suffix product variable
        suffix_product = 1

        # Calculate suffix products (right to left, bottom to top)
        for row in range(num_rows - 1, -1, -1):
            for col in range(num_cols - 1, -1, -1):
                # Store the current suffix product in the product_matrix at [row][col]
                product_matrix[row][col] = suffix_product
                # Update the suffix product (include the current grid value)
                suffix_product = suffix_product * grid[row][col] % modulo

        # Initialize the prefix product variable
        prefix_product = 1

        # Calculate the final product matrix values using prefix products
        for row in range(num_rows):
            for col in range(num_cols):
                # Multiply the prefix product with the already stored suffix product and apply modulo
                product_matrix[row][col] = product_matrix[row][col] * prefix_product % modulo
                # Update the prefix product (include the current grid value)
                prefix_product = prefix_product * grid[row][col] % modulo

        # Return the final product matrix
        return product_matrix

# Example usage:
# solution = Solution()
# matrix = solution.constructProductMatrix([[1, 2], [3, 4]])
# print(matrix)

```

## Java Solution

```java
class Solution {
    public int[][] constructProductMatrix(int[][] grid) {
        final int MOD = 12345; // Constant representing the modulo value
        int rows = grid.length;
        int cols = grid[0].length;
        int[][] productMatrix = new int[rows][cols]; // Initialize product matrix
        long suffixProduct = 1; // Used for suffix product calculation

        // Calculate suffix products for each element and store them in the product matrix
        for (int i = rows - 1; i >= 0; --i) {
            for (int j = cols - 1; j >= 0; --j) {
                productMatrix[i][j] = (int) suffixProduct; // Cast long to int
                suffixProduct = suffixProduct * grid[i][j] % MOD; // Modulo to prevent overflow
            }
        }

        long prefixProduct = 1; // Used for prefix product calculation

        // Calculate prefix product and multiply with the corresponding suffix products
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                productMatrix[i][j] = (int) (productMatrix[i][j] * prefixProduct % MOD); // Final product with modulo
                prefixProduct = prefixProduct * grid[i][j] % MOD; // Update prefix product
            }
        }

        return productMatrix; // Return the constructed product matrix
    }
}

```

## C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to construct a product matrix based on the input grid.
    // Each element in the product matrix will contain the product of all
    // elements in the input grid, modulo a predefined value.
    std::vector<std::vector<int>> constructProductMatrix(std::vector<std::vector<int>>& grid) {
        // The modulo constant as specified.
        const int MODULO = 12345;

        // Get the dimensions of the grid.
        int rowCount = grid.size();
        int colCount = grid[0].size();

        // Initialize the product matrix with the same dimensions as the input grid.
        std::vector<std::vector<int>> productMatrix(rowCount, std::vector<int>(colCount));

        // This variable will hold the suffix product as we traverse the matrix
        // from bottom-right to top-left.
        long long suffixProduct = 1;

        // Compute the suffix product for each element in the grid.
        for (int row = rowCount - 1; row >= 0; --row) {
            for (int col = colCount - 1; col >= 0; --col) {
                productMatrix[row][col] = suffixProduct;
                suffixProduct = suffixProduct * grid[row][col] % MODULO;
            }
        }

        // This variable will hold the prefix product as we traverse the matrix
        // from top-left to bottom-right.
        long long prefixProduct = 1;

        // Compute the prefix product for each element in the grid and multiply
        // it with the corresponding suffix product.
        // Finally, take the modulo of the product to fit the result within the specified range.
        for (int row = 0; row < rowCount; ++row) {
            for (int col = 0; col < colCount; ++col) {
                productMatrix[row][col] = productMatrix[row][col] * prefixProduct % MODULO;
                prefixProduct = prefixProduct * grid[row][col] % MODULO;
            }
        }

        // Return the resulting product matrix.
        return productMatrix;
    }
};

```

## Typescript Solution

```typescript
/**
 * Creates a product matrix from the given grid where each element is the product of all other elements in the grid excluding the cur
 * @param {number[][]} grid - The input grid of numbers.
 * @returns {number[][]} - The product matrix.
 */
function constructProductMatrix(grid: number[][]): number[][] {
    // Define a modulo constant.
    const mod = 12345;
    // Extract grid dimensions.
    const numRows = grid.length;
    const numCols = grid[0].length;
    // Initialize the product matrix with zeroes.
    const productMatrix: number[][] = Array.from({ length: numRows }, () => Array(numCols).fill(0));
    // Variable to track suffix product.
    let suffixProduct = 1;

    // Compute the product of elements to the right and below the current element.
    for (let row = numRows - 1; row >= 0; row--) {
        for (let col = numCols - 1; col >= 0; col--) {
            productMatrix[row][col] = suffixProduct;
            suffixProduct = (suffixProduct * grid[row][col]) % mod;
        }
        // Reset the suffix product for the next row.
        suffixProduct = 1;
    }

    // Variable to track prefix product.
    let prefixProduct = 1;

    // Multiply the prefix product to the existing product in productMatrix and
    // Compute the product of elements to the left and above the current element.
    for (let row = 0; row < numRows; row++) {
        for (let col = 0; col < numCols; col++) {
            productMatrix[row][col] = (productMatrix[row][col] * prefixProduct) % mod;
            prefixProduct = (prefixProduct * grid[row][col]) % mod;
        }
        // Reset the prefix product for the next row.
        prefixProduct = 1;
    }

    // Return the completed product matrix.
    return productMatrix;
}

```

## Time and Space Complexity

The time complexity of the provided code is `O(n * m)`, where `n` is the number of rows and `m` is the number of columns in the input `grid`. This complexity arises because the code contains two nested loops, each of which iterates over every cell in the matrix exactly once.

As for the space complexity, aside from the space needed for the output matrix `p`, the space complexity is `O(1)`. This is because there are only a finite number of additional variables (`n`, `m`, `suf`, `pre`, and `mod`) that do not depend on the size of the input matrix, therefore occupying constant space.