3070. Count Submatrices with Top-Left Element and Sum Less Than k

Medium Array Matrix Prefix Sum

Problem Description

You are provided with a matrix of integers, `grid`, where indexing begins from 0. Additionally, you are given an integer `k`. The task at hand is to determine how many rectangular submatrices (i.e., continuous sections of the grid) include the top-left element of the `grid` and have a sum of their elements that does not exceed the value of `k`.

This is essentially about finding particular submatrices in a grid. These submatrices must:

- Start from the top-left corner, extending to any end point in the grid.
- Have a total sum of elements that is less than or equal to the given threshold `k`.

Intuition

The solution is built on the concept of prefix sums, a common technique used to quickly calculate the sum of elements in a given subarray, or in this case, submatrices.

In a two-dimensional grid, a prefix sum at a point `(i, j)` represents the sum of all elements in the rectangular area from the top-left corner `(0, 0)` to the point `(i, j)`. The formula for calculating the prefix sum of a point `(i, j)` in a matrix is as follows:

```
s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + grid[i][j]
```

By dynamic programming, this approach builds a matrix of prefix sums, `s`, where each cell contains the sum of the submatrix that ends at that cell.

The reason for using the prefix sum technique is that it simplifies the computation of the sum of any submatrix in constant time after the initial calculation of the prefix sum matrix. In essence, once we know the prefix sums, we can deduce the sum of any submatrix by subtracting the relevant prefix sums that overlap.

The provided code implements the following steps:

- First, it initializes a new matrix `s` with dimensions one greater than those of `grid` in both directions, in order to accommodate the prefix sum without needing to handle bounds checking separately.
- It then calculates the two-dimensional prefix sum for each element in `grid`.
- For each element `x` in `grid`, we also check if the sum of the submatrix from the top-left corner to `x` is less than or equal to `k`. If it is, we increment the answer (`ans`) by 1.
- The answer is the total number of such submatrices, which is what we aim to return.

The insight to take away is that by using prefix sums, it is possible to check very quickly whether each submatrix that starts at the top-left corner satisfies the condition of having a sum less than or equal to `k`.

Solution Approach

The solution uses the two-dimensional prefix sum concept for efficient computation of submatrix sums. Here is how the algorithm works, step by step:

1. **Initialize Prefix Sum Matrix (`s`):** A matrix `s` is created, initialized with zeros. Its dimensions are one greater than the input `grid` in both rows and columns. This additional size is for an easier calculation so that the corner cases when `i` or `j` is 0 can be handled without condition checking. Essentially, the extra row and column act as padding.

2. **Calculate Prefix Sums:** To calculate the prefix sums, we iterate through each cell in the input `grid`, and for each cell located at `(i, j)` in `grid`, we compute the corresponding prefix sum at `(i + 1, j + 1)` in `s`. The formula used is:

   ```
   s[i][j] = s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1] + x
   ```

   where `x` is the value at `grid[i - 1][j - 1]` since we are indexing `s` from 1 to include the padding.

   This step dynamically builds up the prefix sum matrix by adding the current cell's value to the sum of the values above and to the left of the current cell, subtracting the overlapping area that was added twice (which is the cell to the top-left), to maintain the correct sum.

3. **Count Valid Submatrices:** Each time we calculate a prefix sum for a cell `(i, j)` in `s`, we immediately check whether this sum is less than or equal to `k`. If it is, it means that the submatrix starting from the top-left corner of `grid` and extending to `(i - 1, j - 1)` has a valid sum; hence, we increment our answer `ans` by 1.

   The reason we check after each prefix sum calculation is that we are interested in all the prefix sub matrices, and each submatrix is uniquely identified by its bottom-right corner.

In terms of data structures, the additional 2D array `s` functions as both storage for the prefix sums as well as the dynamic programming table from which we derive our solution. No other auxiliary data structures are used.

To illustrate the algorithm with an example: consider a 2×2 `grid` with values `[[1,2], [3,4]]` and `k = 6`. The corresponding `s` matrix after calculating prefix sums will be:

```
[[0, 0, 0],
 [0, 1, 3],
 [0, 4, 10]]
```

We will find two valid submatrices, those ending at `(1, 1)` and `(1,2)`, with sums `1` and `3`, respectively.

Overall, this approach efficiently computes the number of qualifying submatrices in a matrix using dynamic programming and the concept of two-dimensional prefix sums.

Example Walkthrough

Let's take a small `grid` example to illustrate the solution approach:

Suppose we have a grid:

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

and `k = 6`.

Following the solution approach let's walk through the algorithm:

1. **Initialize Prefix Sum Matrix (`s`):** We first create a prefix sum matrix `s` with dimensions one greater than `grid` to handle the boundary conditions seamlessly:

   ```
   s = [
     [0, 0, 0],
     [0, 0, 0],
     [0, 0, 0]
   ]
   ```

2. **Calculate Prefix Sums:** Now, let's compute the prefix sums. We iterate over `grid` and update `s` starting from the top-left corner:

   - For `grid[0][0] = 1`, the corresponding cell in `s` is `s[1][1]`. Hence, `s[1][1] = s[0][1] + s[1][0] - s[0][0] + grid[0][0] = 0 + 0 - 0 + 1 = 1`.
   - For `grid[0][1] = 2`, we calculate `s[1][2]: s[1][2] = s[0][2] + s[1][1] - s[0][1] + grid[0][1] = 0 + 1 - 0 + 2 = 3`.
   - For `grid[1][0] = 3`, we calculate `s[2][1]: s[2][1] = s[1][1] + s[2][0] - s[1][0] + grid[1][0] = 1 + 0 - 0 + 3 = 4`.
   - For `grid[1][1] = 4`, we update `s[2][2]: s[2][2] = s[1][2] + s[2][1] - s[1][1] + grid[1][1] = 3 + 4 - 1 + 4 = 10`.

   After calculating, the prefix sums matrix `s` will look like this:

   ```
   s = [
     [0, 0, 0],
     [0, 1, 3],
     [0, 4, 10]
   ]
   ```

3. **Count Valid Submatrices:** We now have the prefix sums for all possible submatrices that start from the top-left corner. We consider all points in `s` as the bottom-right corner of potential submatrices and check if the sum is less than or equal to `k`:

   - `s[1][1] = 1` which is `<= k`, so we have one valid submatrix.
   - `s[1][2] = 3` which is `<= k`, so we have one more valid submatrix.
   - `s[2][1] = 4` which is `<= k`, but since submatrices must be rectangular, this cell doesn't generate a new submatrix starting from the top-left corner. We do not count it.
   - `s[2][2] = 10` which is `> k`, so the submatrix with bottom-right corner at `s[2][2]` is not a valid submatrix.

We end up with two valid submatrices, those ending at `(1, 1)` and `(1,2)`, with sums `1` and `3` respectively.

The total number of valid submatrices in this `grid`, considering the given `k`, is 2.

Solution Implementation

Python

```python
class Solution:
    def countSubmatrices(self, grid: List[List[int]], k: int) -> int:
        # Initialize a prefix sum matrix with an extra row and column of zeros
        prefix_sum = [[0] * (len(grid[0]) + 1) for _ in range(len(grid) + 1)]
        # Initialize counter for the number of submatrices
        num_submatrices = 0

        # Populate the prefix sum matrix with the cumulative sums
        for i, row in enumerate(grid, start=1):
            for j, cell_value in enumerate(row, start=1):
                # Calculate the cumulative sum up to the current cell
                prefix_sum[i][j] = (prefix_sum[i - 1][j] + prefix_sum[i][j - 1]
                                    - prefix_sum[i - 1][j - 1] + cell_value)
                # Increment the counter if the sum of the current submatrix is less than or equal to k
                num_submatrices += prefix_sum[i][j] <= k

        # Return the total number of submatrices with a sum less than or equal to k
        return num_submatrices
```

Java

```java
class Solution {
    public int countSubmatrices(int[][] grid, int k) {
        int rowCount = grid.length; // The number of rows in the grid.
        int colCount = grid[0].length; // The number of columns in the grid.
        // Prefix sum array with extra row and column to handle boundaries.
        int[][] prefixSum = new int[rowCount + 1][colCount + 1];
        int count = 0; // Variable to keep track of the count of submatrices.

        // Build the prefix sum matrix.
        for (int i = 1; i <= rowCount; ++i) {
            for (int j = 1; j <= colCount; ++j) {
                // Calculate prefix sum at each cell.
                prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1]
                                  - prefixSum[i - 1][j - 1] + grid[i - 1][j - 1];

                // If the sum of the current single-cell matrix is less than or equal to k,
                // increment the count.
                if (prefixSum[i][j] <= k) {
                    ++count;
                }
            }
        }

        // Return the total count of submatrices.
        return count;
    }
}
```

C++

```cpp
#include <vector>
#include <string>

class Solution {
public:
    int countSubmatrices(std::vector<std::vector<int>>& grid, int k) {
        int m = grid.size(); // Row count
        int n = grid[0].size(); // Column count
        int prefixSum[m + 1][n + 1]; // Create a 2D array to store prefix sums
        std::memset(prefixSum, 0, sizeof(prefixSum)); // Initialize prefixSum array with 0
        int ans = 0; // Initialize counter for submatrices with sum <= k

        // Build prefix sums for submatrices and check if any submatrix equals k
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                // Calculate prefix sum for current submatrix
                prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1]
                                  - prefixSum[i - 1][j - 1] + grid[i - 1][j - 1];

                // Check if the sum of this single cell submatrix is <= k
                if (prefixSum[i][j] <= k) {
                    ++ans; // Increment counter if condition is met
                }
            }
        }

        // Return the total number of submatrices with sum <= k
        return ans;
    }
};
```

TypeScript

```typescript
// Counts the number of submatrices with a sum less than or equal to k.
function countSubmatrices(grid: number[][], k: number): number {
    const numRows = grid.length;
    const numCols = grid[0].length;

    // Initialize the matrix for storing cumulative sums with extra padding for easy calculations.
    const cumulativeSums: number[][] = Array.from({ length: numRows + 1 }, () => Array(numCols + 1).fill(0));

    let count: number = 0; // This variable will keep count of our result.

    // Fill the cumulativeSums matrix with the cumulative sum of submatrices.
    for (let i = 1; i <= numRows; ++i) {
        for (let j = 1; j <= numCols; ++j) {
            // Calculate the cumulative sum for the position (i, j).
            cumulativeSums[i][j] =
                cumulativeSums[i - 1][j] +
                cumulativeSums[i][j - 1] -
                cumulativeSums[i - 1][j - 1] +
                grid[i - 1][j - 1];

            // If the sum of the submatrix ending at (i-1, j-1) is less than or equal to k,
            // increment the count.
            if (cumulativeSums[i][j] <= k) {
                ++count;
            }
        }
    }

    return count; // Return the final count of submatrices.
}

// You can then use the function like this:
// let result = countSubmatrices([[1, 2], [3, 4]], 6);
// console.log(result); // Outputs the count of submatrices with sum <= k
```

```python

class Solution:
    def countSubmatrices(self, grid: List[List[int]], k: int) -> int:
        # Initialize a prefix sum matrix with an extra row and column of zeros
        prefix_sum = [[0] * (len(grid[0]) + 1) for _ in range(len(grid) + 1)]
        # Initialize counter for the number of submatrices
        num_submatrices = 0

        # Populate the prefix sum matrix with the cumulative sums
        for i, row in enumerate(grid, start=1):
            for j, cell_value in enumerate(row, start=1):
                # Calculate the cumulative sum up to the current cell
                prefix_sum[i][j] = (prefix_sum[i - 1][j] + prefix_sum[i][j - 1]
                                    - prefix_sum[i - 1][j - 1] + cell_value)
                # Increment the counter if the sum of the current submatrix is less than or equal to k
                num_submatrices += prefix_sum[i][j] <= k

        # Return the total number of submatrices with a sum less than or equal to k
        return num_submatrices
```

Time and Space Complexity

The given code calculates the number of submatrices in a grid whose sum is less than or equal to `k`. It does so by using a 2D prefix sum to compute the sum of any submatrix in constant time after an initial setup phase.

The time complexity of the code cannot be $O(m * n)$ as the reference answer suggests, because there is a lack of logic to compute submatrices other than those starting at the (0,0) coordinate. For every element `(i, j)` in the grid, the algorithm calculates the sum from the starting point `(0, 0)` to `(i, j)` and increases the count of submatrices if the sum is less than or equal to `k`. This process is done in constant time for each element, resulting in a time complexity of $O(m * n)$, where m and n are the number of rows and columns, respectively. However, this only counts the submatrices starting at `(0, 0)`, and does not count all possible submatrices.

The space complexity of the code involves the creation of an auxiliary matrix `s` of size `(m+1) * (n+1)` to store the prefix sums. Therefore, the space complexity is also $O(m * n)$.

To summarize, the actual time complexity for the given code logic should be stated as $O(m * n)$ for the described prefix sum computation but not for the overall problem of counting all valid submatrices less than `k`, and the space complexity is correctly stated as $O(m * n)$.