3070. Count Submatrices with Top-Left Element and Sum Less Than k Medium <u>Array</u> Matrix **Prefix Sum**

Problem Description

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.

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

Have a total sum of elements that is less than or equal to the given threshold k.

- Intuition

• The answer is the total number of such submatrices, which is what we aim to return.

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.

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]

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-

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:

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.

• 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

starts at the top-left corner satisfies the condition of having a sum less than or equal to k. **Solution Approach**

The insight to take away is that by knowing the prefix sums, it is possible to check very quickly whether each submatrix that

works, step by step: Initialize Prefix Sum Matrix (s): A matrix s is created, initialized with zeros. Its dimensions are one greater than the input

be handled without condition checking. Essentially, the extra row and column act as padding.

programming table from which we derive our solution. No other auxiliary data structures are used.

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

The solution uses the two-dimensional <u>prefix sum</u> concept for efficient computation of submatrix sums. Here is how the algorithm

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

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 <u>prefix sum</u> 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

maintain the correct sum.

concept of two-dimensional prefix sums.

boundary conditions seamlessly:

Suppose we have a grid:

grid = [

[1, 2],

[3, 4]

submatrix is uniquely identified by its bottom-right corner.

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 <u>prefix sum</u> 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

The reason we check after each <u>prefix sum</u> calculation is that we are interested in all the prefix submatrices, and each

In terms of data structures, the additional 2D array s functions as both storage for the prefix sums as well as the dynamic

To illustrate the algorithm with an example: consider a 2×2 grid with values [[1,2], [3,4]] and k = 6. The corresponding s

- 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.
- matrix after calculating prefix sums will be: [[0, 0, 0], [0, 1, 3], [0, 4, 10]]

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

Overall, this approach efficiently computes the number of qualifying submatrices in a matrix using dynamic programming and the

```
and k = 6.
Following the solution approach let's walk through the algorithm:
```

Initialize Prefix Sum Matrix (s): We first create a prefix sum matrix s with dimensions one greater than grid to handle the

∘ 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

 \circ 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.

 \circ 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.

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

+ 1 = 1.

[0, 0, 0],

[0, 1, 3],

[0, 4, 10]

left corner. We do not count it.

Solution Implementation

return num_submatrices

Python

Java

class Solution {

class Solution:

s = [

[0, 0, 0],

[0, 0, 0],

[0, 0, 0]

 \circ 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:

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. \circ s[1][2] = 3 which is \ll k, so we have one more valid submatrix.

 \circ s[2][2] = 10 which is > k, so the submatrix with bottom-right corner at s[2][2] is not a valid submatrix.

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

Initialize a prefix sum matrix with an extra row and column of zeros

prefix sum = [[0] * (len(qrid[0]) + 1) for in range(len(grid) + 1)]

Calculate the cumulative sum up to the current cell

prefix sum[i][i] = (prefix sum[i - $\mathbf{1}$][i] + prefix sum[i][i - $\mathbf{1}$]

Return the total number of submatrices with a sum less than or equal to k

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

num_submatrices += prefix_sum[i][j] <= k</pre>

int rowCount = grid.length; // The number of rows in the grid.

int[][] prefixSum = new int[rowCount + 1][colCount + 1];

// Calculate prefix sum at each cell.

int colCount = grid[0].length; // The number of columns in the grid.

int count = 0; // Variable to keep track of the count of submatrices.

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,

// Prefix sum array with extra row and column to handle boundaries.

public int countSubmatrices(int[][] grid, int k) {

// Build the prefix sum matrix.

for (int i = 1; i <= rowCount; ++i) {</pre>

// increment the count.

++count;

if (prefixSum[i][j] <= k) {</pre>

// Return the total count of submatrices.

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

if (prefixSum[i][i] <= k) {</pre>

return ans;

const numRows = grid.length;

const numCols = grid[0].length;

};

TypeScript

// Calculate prefix sum for current submatrix

// Return the total number of submatrices with sum <= k

// Counts the number of submatrices with a sum less than or equal to k.

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

function countSubmatrices(grid: number[][], k: number): number {

Initialize counter for the number of submatrices

for i, row in enumerate(grid, start=1):

Populate the prefix sum matrix with the cumulative sums

num_submatrices += prefix_sum[i][j] <= k</pre>

Calculate the cumulative sum up to the current cell

prefix sum[i][i] = (prefix sum[i - $\mathbf{1}$][i] + prefix sum[i][i - $\mathbf{1}$]

Return the total number of submatrices with a sum less than or equal to k

- 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

for j, cell value in enumerate(row, start=1):

prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1]

// Check if the sum of this single cell submatrix is <= k

// 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));

++ans; // Increment counter if condition is met

- prefixSum[i - 1][j - 1] + grid[i - 1][j - 1];

for (int i = 1; i <= colCount; ++i) {</pre>

Initialize counter for the number of submatrices

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

- prefix sum[i - 1][i - 1] + cell value)

Increment the counter if the sum of the current submatrix is less than or equal to k

o 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-

Count Valid Submatrices: We now have the prefix sums for all possible submatrices that start from the top-left corner. We

```
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):
```

return count; C++ #include <vector> #include <cstring> 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</pre> // Build prefix sums for submatrices and check if any submatrix equals k for (int i = 1; $i \le m$; ++i) {

// Fill the cumulativeSums matrix with the cumulative sum of submatrices. for (let i = 1; i <= numRows; ++i) {</pre> for (let i = 1; i <= numCols; ++i) {</pre> // Calculate the cumulative sum for the position (i, j). cumulativeSums[i][j] = cumulativeSums[i - 1][i] + 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) {</pre> ++count; return count; // Return the final count of submatrices. // You can then use the function like this: // let result = countSubmatrices([[1, 1], [1, 1]], 4); // console.log(result); // Outputs the count of submatrices with sum <= k 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(qrid[0]) + 1) for in range(len(qrid) + 1)]

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.

count all possible submatrices.

return num_submatrices

Time and Space Complexity

num_submatrices = 0

The time complexity of the code cannot be 0(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 0(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

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 0(m * n). To summarize, the actual time complexity for the given code logic should be stated as 0(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 0(m * n).