1074. Number of Submatrices That Sum to Target

Hard · Array · Hash Table · Matrix · Prefix Sum · Leetcode Link

Problem Description

The problem provides us with a rectangular grid of numbers—a matrix—and a target sum. We are asked to count the total number of distinct submatrices where the sum of their elements is equal to the given target value. A submatrix is any contiguous block of cells within the original matrix, denoted by its top-left coordinate `(x1, y1)` and bottom-right coordinate `(x2, y2)`. Each submatrix is considered unique if it differs in at least one coordinate value from any other submatrix, even if they have the same elements. This means that we're not just looking for different sets of values that add up to the target, but also different positions of those values within the matrix.

Intuition

To solve this problem, an efficient approach is required since a brute force method of checking all possible submatrices would result in an impractical solution time, particularly for large matrices. The key is to recognize that we can construct submatrices by spanning vertically from any row `i` to any row `j` and then considering all possible horizontal slices within this vertical span.

The solution uses a function `f(nums)` that takes a list of numbers representing the sum of elements in a column for the rows from `i` to `j`. It then utilizes a hash map (dictionary in Python) to efficiently count the submatrices which sum to `target`. This is done by keeping a cumulative sum `s` as we iterate through `nums`, where `s` is the sum of elements from the start of `nums` to the current position `k`. If `s - target` is found in the dictionary `d`, it means a submatrix ending at the current column whose sum is `target` has been found (since `s - (s - target) = target`). Every time this occurs, we add the count of `s - target` found so far to our total count. The dictionary is updated with the current sum, effectively storing the count of all possible sums encountered up to that position.

The outer loops iterate over the matrix to set the vertical boundaries, `i` and `j` of the submatrices. For every such vertical span, we compute the running sum of columns as if they're a single array and use `f(nums)` to find eligible horizontal slices. The total count from each `f(nums)` call accumulates in the variable `ans`, which gives us the final number of submatrices meeting the condition.

In essence, by breaking down the problem into a series of one-dimensional problems, where each one-dimensional problem is a vertical slice of our matrix, we can use the hash map strategy to efficiently count submatrices summing to `target`.

Solution Approach

To tackle the problem, the given Python solution employs a clever use of prefix sums along with hashing to efficiently count the number of submatrices that sum to the target. Here's a walkthrough of the implementation, aligned with the algorithm and the patterns used:

- We start by initializing `ans` to zero, which will hold the final count of submatrices adding up to the target.
- The outer two loops fix the vertical boundaries of our submatrices. `i` represents the starting row, and `j` iterates from `i` to the last row, `m`. For each pair `(i,j)`, we are considering a horizontal slab of the matrix from row `i` to row `j`.
- For each of these horizontal slabs, we construct an array `col` which will hold the cumulative sums for the `k`-th column from row `i` to row `j`. This transformation essentially 'flattens' our 2D submatrix slab into a 1D array of sums.
- With this 1D array `col`, we invoke the function `f(nums)`. This function uses a dictionary `d` to keep track of the number of times a specific prefix sum has occurred. We initialize `d` with the base case `d[0] = 1`, representing a submatrix with a sum of zero, which is a virtual prefix before the start of any actual numbers.
- As we loop through `nums` (which are the column sums in `col`), we add each number to a running sum `s`. For each element, we look at the current sum `s` and check how many times we have seen a sum of `s - target`. If `s - target` is in `d`, it means that there is a submatrix ending at the current element which adds up to `target`. We increment `cnt` by the count of `s - target` from `d`.
- After checking for the count of `s - target`, we update `d` by incrementing the count of `s` by 1. This captures the idea that we now have one more submatrix (counted till the current column) that sums to ``s`.
- The return value of `f(nums)` gives us the number of submatrices that sum to `target` for our current slab of rows between `i` and `j`. We add this to our total `ans`.
- After the loops are finished, `ans` contains the total count of submatrices that add up to `target` across the entire matrix.

In terms of data structures, the solution relies on a 1D array to store the column sums and a dictionary to act as a hash map for storing prefix sums. The use of a hash map enables constant time checks and updates, significantly optimizing the process. This approach eliminates the need for naive checking of every possible submatrix, which would be computationally intensive.

This solution approach leverages dynamic programming concepts, particularly the idea of storing intermediary results (prefix sums and their counts) to avoid redundant calculations. This pattern is useful for problems involving contiguous subarrays or submatrices and target sums, and is a powerful tool in the competitive programming space.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Suppose we have the following matrix and target sum:

```
Matrix:     Target sum:
1 2 1     ->    4
3 2 0
1 1 1
```

For this example, we'll be looking to count the number of submatrices that add up to the target sum of 4.

1. **Initialize Ans**: Start by initializing `ans` to 0. This will be used to store the final count of submatrices.
2. **Iterate Over Rows**: Set up two nested loops to iterate over the rows to determine the vertical boundaries of potential submatrices. The variable `i` is the top row and `j` iterates from `i` to the bottom row.
3. **Transform to 1D `col` Array**: For each fixed vertical boundary `(i, j)`, we create a 1D `col` array representing cumulative sums of each column from row `i` to row `j`.

   For `i = 0` and `j = 1`, `col` would be:

   ```
   1st iteration (i=0, j=0): col = [1, 2, 1]
   2nd iteration (i=0, j=1): col = [4, 4, 1]  // Sum of rows 0 and 1
   ```

4. **Function `f(nums)` Calculation**: Call function `f(cols)` which uses a dictionary `d` to keep track of prefix sums.

   When we consider `i = 0` and `j = 1`, we invoke `f([4, 4, 1])`:

   ```
   Initialize d={0: 1} and `s` to 0.
   Iterate the col `nums`:
   - At col 0: s = 4. Check if s-target, which is 0, exists in d. It does. Increment ans by 1.
   - Update d with the new sum: d = {0: 1, 4: 1}.
   - At col 1: s = 8. There's no s-target, which is 4, in d.
   - Update d: d = {0: 1, 4: 2}.
   - At col 2: s = 9. There's no s-target, which is 5, in d.
   - Update d: d = {0: 1, 4: 2, 9: 1}.
   ```

   For this `(i, j)` pair, `f(nums)` finds 1 submatrix that adds to the target.

5. **Update Ans**: Add the count from the function `f(nums)` to `ans`. Repeat the process for each vertical slab defined by `(i, j)`.
6. **Final Answer**: After iterating through all pairs of `(i, j)`, summing up the counts of submatrices from each `f(nums)`, we end up with the total `ans`.

In our example, we can find the following submatrices that sum to the target:

- Single submatrix from (0,0) to (1,0) with elements `1, 3` which adds up to 4.

Thus, `ans` for this example would be 1, indicating there is one distinct submatrix where the sum of the elements equals the target sum of 4.

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def numSubmatrixSumTarget(self, matrix: List[List[int]], target: int) -> int:
        # Helper function to find the number of contiguous subarrays
        # that sum up to the target value.
        def count_subarrays_with_target_sum(nums: List[int]) -> int:
            prefix_sum_counts = defaultdict(int)
            prefix_sum_counts[0] = 1
            # `count` stores the number of valid subarrays found.
            # `prefix_sum` stores the ongoing sum of elements in the array.
            count = prefix_sum = 0
            for num in nums:
                prefix_sum += num
                # Increase count by the number of times (prefix_sum - target)
                # has occurred before, as it represents a valid subarray.
                count += prefix_sum_counts[prefix_sum - target]
                # Update the count of prefix sum occurrences.
                prefix_sum_counts[prefix_sum] += 1
            return count

        num_rows, num_cols = len(matrix), len(matrix[0])
        total_count = 0  # This variable will store the total submatrices found.
        # Loop over the start row for the submatrix.
        for start_row in range(num_rows):
            column_sums = [0] * num_cols
            # Loop over the end row for the submatrix.
            for end_row in range(start_row, num_rows):
                # Update the sum for each column to include the new row.
                for col in range(num_cols):
                    column_sums[col] += matrix[end_row][col]
                # Add the count of valid subarrays in the current column sums.
                total_count += count_subarrays_with_target_sum(column_sums)
        # Return the total number of submatrices that sum up to the target.
        return total_count
```

Java Solution

```java
class Solution {
    public int numSubmatrixSumTarget(int[][] matrix, int target) {
        int numRows = matrix.length;
        int numCols = matrix[0].length;
        int answer = 0;

        // Loop through each row, starting from the top
        for (int topRow = 0; topRow < numRows; ++topRow) {
            // Initialize a cumulative column sum array for the submatrix sum
            int[] cumulativeColSum = new int[numCols];

            // Extend the submatrix down by increasing the bottom row from the top row
            for (int bottomRow = topRow; bottomRow < numRows; ++bottomRow) {
                // Update the cumulative sum for each column in the submatrix
                for (int col = 0; col < numCols; ++col) {
                    cumulativeColSum[col] += matrix[bottomRow][col];
                }
                // Count the submatrices with the sum equals the target using the helper function
                answer += countSubarraysWithSum(cumulativeColSum, target);
            }
        }
        return answer;
    }

    // Helper function to count the subarrays which sum up to the target
    private int countSubarraysWithSum(int[] nums, int target) {
        // Initialize a map to store the sum and frequency
        Map<Integer, Integer> sumFrequency = new HashMap<>();
        sumFrequency.put(0, 1);
        int currentSum = 0;
        int count = 0;

        // Iterate through each element in the array
        for (int num : nums) {
            currentSum += num; // Update the running sum
            // Increment the count by the number of times (currentSum - target) has appeared
            count += sumFrequency.getOrDefault(currentSum - target, 0);
            // Update the frequency map with the current sum as the key
            // If the key exists, increment its value by 1
            sumFrequency.merge(currentSum, 1, Integer::sum);
        }
        return count;
    }
}
```

C++ Solution

```cpp
// Including necessary headers
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function that returns the number of submatrices that sum up to the target value.
    int numSubmatrixSumTarget(vector<vector<int>>& matrix, int target) {
        int rows = matrix.size(), columns = matrix[0].size();  // Matrix dimensions
        int answer = 0;  // Initialize the count of submatrices with sum equal to target

        // Iterate over each pair of rows to consider submatrices that span from row i to j
        for (int i = 0; i < rows; ++i) {
            vector<int> colSums(columns, 0);  // Initialize a vector to store column sums
            for (int j = i; j < rows; ++j) {
                // Accumulating sum for each column between rows i and j
                for (int k = 0; k < columns; ++k) {
                    colSums[k] += matrix[j][k];
                }
                // Add to answer the count of subarrays in the summed columns that meet the target
                answer += countSubarraysWithTarget(colSums, target);
            }
        }

        return answer;  // Return the total count of submatrices that have sums equal to the target
    }

private:
    // Helper function that counts the number of subarrays within a 1D array with sum equal to target
    int countSubarraysWithTarget(vector<int>& nums, int target) {
        unordered_map<int, int> sumFrequencies{{0, 1}};  // Initialize map with zero-sum frequency
        int count = 0;  // Count of subarrays with sum equal to target
        int sum = 0;    // Current sum of elements

        // Iterate over the elements in the array
        for (int num : nums) {
            sum += num;  // Update running sum
            // If (current sum - target) exists in the map, increment count by the number of times
            // the (current sum - target) has been seen (this number of previous subarrays
            // contribute to current sum equals target).
            if (sumFrequencies.count(sum - target)) {
                count += sumFrequencies[sum - target];
            }
            // Increment the frequency of the current sum in the map
            ++sumFrequencies[sum];
        }

        return count;  // Return the count of subarrays with sum equal to target
    }
};

// Example usage:
// int main() {
//     Solution sol;
//     vector<vector<int>> matrix{{0,1,0},{1,1,1},{0,1,0}};
//     int target = 0;
//     int result = sol.numSubmatrixSumTarget(matrix, target);
//     // result will hold the number of submatrices that sum up to the target
// }
```

Typescript Solution

```typescript
// This function counts the number of submatrices that sum up to the 'target'
function numSubmatrixSumTarget(matrix: number[][], target: number): number {
    const numRows = matrix.length;
    const numCols = matrix[0].length;
    let count = 0;

    // Iterate through rows
    for (let startRow = 0; startRow < numRows; ++startRow) {
        const columnSum: number[] = new Array(numCols).fill(0);

        // Accumulate sums for all possible submatrices starting at startRow
        for (let endRow = startRow; endRow < numRows; ++endRow) {
            for (let col = 0; col < numCols; ++col) {
                // Add the current row's values to the column sums
                columnSum[col] += matrix[endRow][col];
            }
            // Use the helper function to count subarrays in this contiguous slice of rows that sum to target
            count += countSubarraysWithSum(columnSum, target);
        }
    }
    return count;
}

// Helper function to count the number of subarrays with sum equal to 'target'
function countSubarraysWithSum(nums: number[], target: number): number {
    const sumOccurrences: Map<number, number> = new Map();
    sumOccurrences.set(0, 1); // A sum of 0 occurs once initially
    let count = 0;
    let currentSum = 0;

    // Iterate through the array elements
    for (const num of nums) {
        currentSum += num;

        // If the required sum that would lead to the target is found, add its occurrences to count
        if (sumOccurrences.has(currentSum - target)) {
            count += sumOccurrences.get(currentSum - target)!;
        }

        // Record the current sum's occurrence count, incrementing it if it already exists
        sumOccurrences.set(currentSum, (sumOccurrences.get(currentSum) || 0) + 1);
    }
    return count;
}
```

Time and Space Complexity

The time complexity of the algorithm can be broken down into two parts: iterating through submatrices and calculating the sum for each submatrix to find the number of occurrences that add up to the target.

- Iterating through submatrices: It uses two nested loops that go through the rows of the matrix, which will be `O(m^2)` where `m` is the number of rows. Inside these loops, we iterate through the columns for each submatrix, which adds another factor of `n`, where `n` is the number of columns. Thus, the iteration through the submatrices is `O(m^2 * n)`.
- Calculating the sum for each submatrix: The function `f(nums: List[int])` is called for each submatrix. Inside this function, there is a for-loop of `O(n)` complexity because it iterates through the column cumulative sums. The operations inside the loop (accessing and updating the dictionary) have an average-case complexity of `O(1)`.

Therefore, combining these together, the total average-case time complexity of the algorithm is `O(m^2 * n^2)`.

For space complexity:

- The `col` array uses `O(n)` space, where `n` is the number of columns. This array stores the cumulative sum of the columns for the current submatrix.
- The `d` dictionary in function `f` will store at most `n + 1` key-value pairs, where `n` is the length of `nums` (number of columns). This is because it records the cumulative sum of the numbers we've seen so far plus an initial zero sum. Hence, space complexity for `d` is `O(n)`.

As each of the above is not dependent on one another, we take the larger of the two, which is `O(n)`, for the overall space complexity.

In summary, the final computational complexities are:

- Time Complexity: `O(m^2 * n^2)`
- Space Complexity: `O(n)`