dev.to

Solution: Number of Submatrices That Sum to Target

6-8 minutes

This is part of a series of Leetcode solution explanations (index). If you liked this solution or found it useful, ***please like*** *this post and/or* ***upvote*** *my solution post on Leetcode's forums.*

Leetcode Problem #1074 (*Hard*): Number of Submatrices That Sum to Target

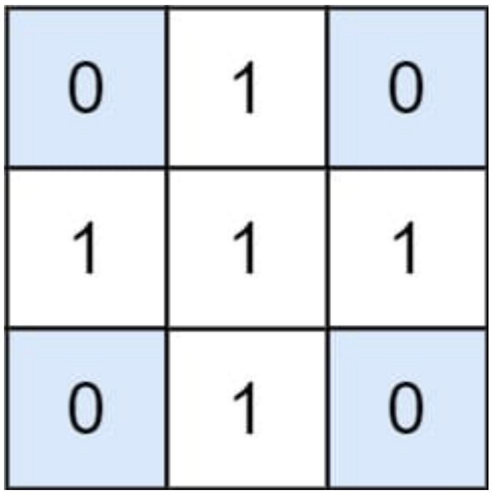
---

Description:

(*Jump to*: *Solution Idea* || *Code*: *JavaScript* | *Python* | *Java* | *C++*)

> Given a `matrix` and a `target`, return the number of non-empty submatrices that sum to `target`.
>
> A submatrix `x1, y1, x2, y2` is the set of all cells `matrix[x][y]` with `x1 <= x <= x2` and `y1 <= y <= y2`.
>
> Two submatrices `(x1, y1, x2, y2)` and `(x1', y1', x2', y2')` are different if they have some coordinate that is different: for example, if `x1 != x1'`.

Examples:

Example 1:	
Input:	matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
Output:	4
Explanation:	The four 1x1 submatrices that only contain 0.
Visual:	

Example 2:	
Input:	matrix = [[1,-1],[-1,1]], target = 0
Output:	5
Explanation:	The two 1x2 submatrices, plus the two 2x1 submatrices, plus the 2x2 submatrix.

Example 3:	
Input:	matrix = [[904]], target = 0
Output:	0

Constraints:

- `1 <= matrix.length <= 100`
- `1 <= matrix[0].length <= 100`
- `-1000 <= matrix[i] <= 1000`
- `-10^8 <= target <= 10^8`

Idea:

(*Jump to*: *Problem Description* || *Code*: *JavaScript* | *Python* | *Java* | *C++*)

This problem is essentially a **2-dimensional** version of **#560. Subarray Sum Equals K (S.S.E.K)**. By using a **prefix sum** on each row or each column, we can compress this problem down to either **N^2** iterations of the **O(M)** SSEK, or **M^2** iterations of the **O(N)** SSEK.

In the SSEK solution, we can find the number of subarrays with the target sum by utilizing a **result map** (**res**) to store the different values found as we iterate through the array while keeping a running sum (**csum**). Just as in the case with a prefix sum array, the sum of a subarray between **i** and **j** is equal to the sum of the subarray from **0** to **j** minus the sum of the subarray from **0** to **i-1**.

Rather than iteratively checking if **sum[0,j] - sum[0,i-1] = T** for every pair of **i, j** values, we can flip it around to **sum[0,j] - T = sum[0,i-1]** and since every earlier sum value has been stored in **res**, we can simply perform a lookup on **sum[0,j] - T** to see if there are any matches.

When extrapolating this solution to our **2-dimensional** matrix (**M**), we will need to first prefix sum the rows or columns, (which we can do **in-place** to avoid extra space, as we will not need the original values again). Then we should iterate through **M** again in the opposite order of rows/columns where the prefix sums will allow us to treat a group of columns or rows as if it were a **1-dimensional** array and apply the SSEK algorithm.

Implementation:

There are only minor differences in the code of all four languages.

Javascript Code:

(*Jump to*: [*Problem Description*]() || [*Solution Idea*]())

```
var numSubmatrixSumTarget = function(M, T) {
    let xlen = M[0].length, ylen = M.length,
        ans = 0, res = new Map()
    for (let i = 0, r = M[0]; i < ylen; r = M[++i])
        for (let j = 1; j < xlen; j++)
            r[j] += r[j-1]
    for (let j = 0; j < xlen; j++)
        for (let k = j; k < xlen; k++) {
            res.clear(), res.set(0,1), csum = 0
            for (let i = 0; i < ylen; i++) {
                csum += M[i][k] - (j ? M[i][j-1] : 0)
                ans += (res.get(csum - T) || 0)
                res.set(csum, (res.get(csum) || 0) +
1)
```

```
            }
        }
    return ans
};
```

Python Code:

(*Jump to*: [*Problem Description*]() || [*Solution Idea*]())

```
class Solution:
    def numSubmatrixSumTarget(self, M:
List[List[int]], T: int) -> int:
        xlen, ylen, ans, res = len(M[0]), len(M), 0,
defaultdict(int)
        for r in M:
            for j in range(1, xlen):
                r[j] += r[j-1]
        for j in range(xlen):
            for k in range(j, xlen):
                res.clear()
                res[0], csum = 1, 0
                for i in range(ylen):
                    csum += M[i][k] - (M[i][j-1] if j
else 0)
                    ans += res[csum - T]
                    res[csum] += 1
        return ans
```

Java Code:

(*Jump to*: [Problem Description](#) || [Solution Idea](#))

```
class Solution {
    public int numSubmatrixSumTarget(int[][] M, int
T) {
        int xlen = M[0].length, ylen = M.length, ans
= 0;
        Map<Integer, Integer> res = new HashMap<>();
        for (int[] r : M)
            for (int j = 1; j < xlen; j++)
                r[j] += r[j-1];
        for (int j = 0; j < xlen; j++)
            for (int k = j; k < xlen; k++) {
                res.clear();
                res.put(0,1);
                int csum = 0;
                for (int i = 0; i < ylen; i++) {
                    csum += M[i][k] - (j > 0 ?
M[i][j-1] : 0);
                    ans += res.getOrDefault(csum - T,
0);
                    res.put(csum,
res.getOrDefault(csum, 0) + 1);
                }
            }
```

```
        return ans;
    }
}
```

C++ Code:

(*Jump to*: [Problem Description](#) || [Solution Idea](#))

```
class Solution {
public:
    int numSubmatrixSumTarget(vector<vector<int>>& M,
int T) {
        int xlen = M[0].size(), ylen = M.size(), ans
= 0;
        unordered_map<int, int> res;
        for (int i = 0; i < ylen; i++)
            for (int j = 1; j < xlen; j++)
                M[i][j] += M[i][j-1];
        for (int j = 0; j < xlen; j++)
            for (int k = j; k < xlen; k++) {
                res.clear();
                res[0] = 1;
                int csum = 0;
                for (int i = 0; i < ylen; i++) {
                    csum += M[i][k] - (j ? M[i][j-1]
: 0);
                    ans += res.find(csum - T) !=
res.end() ? res[csum - T] : 0;
```

```
                        res[csum]++;
                    }
                }
            return ans;
        }
};
```