<https://leetcode.com/problems/binary-subarrays-with-sum/description/>

Given a binary array nums and an integer goal, return the number of non-empty subarrays with a sum goal.

A subarray is a contiguous part of the array.

**Example 1:**

**Input:** nums = [1,0,1,0,1], goal = 2

**Output:** 4

**Explanation:**

The 4 subarrays are bolded and underlined below:

[**1,0,1**,0,1]

[**1,0,1,0**,1]

[1,**0,1,0,1**]

[1,0,**1,0,1**]

**Example 2:**

**Input:** nums = [0,0,0,0,0], goal = 0

**Output:** 15

**Constraints:**

1 <= nums.length <= 3 \* 10^4

nums[i] is either 0 or 1.

0 <= goal <= nums.length

**Attempt 1: 2025-02-01**

**Solution 1: Sliding Window (10 min, exactly same as** [L930.Binary Subarrays With Sum (Ref.L992,L1248)](note://WEB0c236239487de108cb6c215aee18f315)**)**

class Solution {

    public int numSubarraysWithSum(int[] nums, int goal) {

        return helper(nums, goal) - helper(nums, goal - 1);

    }

    private int helper(int[] nums, int target) {

        int count = 0;

        int sum = 0;

        int i = 0;

        for(int j = 0; j < nums.length; j++) {

            sum += nums[j];

            // Must include '&& i <= j'

            // Test out: nums = {0,0,0,0,0} and goal = 0

            while(sum > target && i <= j) {

                sum -= nums[i];

                i++;

            }

            count += (j - i + 1);

        }

        return count;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/binary-subarrays-with-sum/solutions/2866679/easy-and-detailed-sliding-window-approach-at-most-method-detailed-explanation-c-o-n/>

Hope you're doing well.

This problem can be solved by 3 methods generally.

Brute-Force -> Using nested loop -> T.C = O(n^2) ; S.C = O(1)

Optimised -> Using hash map -> T.C = O(n) ; S.C = O(n)

Optimal -> Using sliding window -> T.C = O(n) ; S.C =O(n)

First two are well explained and known to you, There are so many good articles regarding them in discussion section by others too. But personally I think third method is where all are facing doubts. So I will try my best to explain it properly.

So, let me write the code and then explain you the intution and how logic is working.

Note : I have maintained comments in code, what exactly that part is doing. Please refer that too.

class Solution {

public:

//This function will give us the number of subarrays

//which is having the sum<=goal, means atmost the sum

//can be goal, but additionally it will also count the

//subarrays having sum<goal.

int atmostSubarray(vector<int>&nums, int goal)

{

//checking if the goal is negative

//then return 0 as sum can't be negative

//0's and 1's are there in the array.

if(goal<0)

return 0;

int sum=0; // For calculating the current sum

int i=0;

int j=0;

int res=0; // storing the res

while(j<nums.size())

{

//we're iterating over the nums array and

//then adding the current element into the sum

sum+=nums[j];

//if our current sum is greater than our goal

//that means we don't need the extra element

//so we will shrink our window by subtracting

//the ith elements from sum till sum > goal

while(sum>goal)

{

sum-=nums[i];

i++;

}

//We're adding the length of each subarray

//to our result

res+=(j-i+1);

j++;

}

return res;

}

int numSubarraysWithSum(vector<int>& nums, int goal) {

return atmostSubarray(nums,goal)-atmostSubarray(nums,goal-1);

}

};

So, some questions are there, let us go through them.

What's the logic of **atmostSubarray** function?

This function actually calculates the number of subarrays which is having ***sum*** less than and equal to ***goal*** , now the catch is this function will return the number of subarrays having ***sum*** less than ***goal*** too.

How it's calculating ?

*For example :* ***Array*** = [ 1 , 0 , 1 , 0 , 1 ] and ***Goal*** = 2.

We'll iterate over it like below and calculate the current sum and check whether it's less than or equal to ***Goal*** and count the number of subarrays in *count* variable.

Initially : sum = 0 , i = 0 , count = 0.

Step 1 : **1** 0 1 0 1 ; i = 0 , sum = 1 , count = 1

Step 2 : **1 0** 1 0 1 ; i = 1 , sum = 1 , count = 1 + 2 = 3 , not 2

Now here's the catch , we aren't simply incrementing the count by 1 , we are adding 2, why?

See, till 1st index how many subarrays are there with sum <= 2 ( our goal ).

(a) 1

(b) 1 0

(c) 0

Total = 3 , not 2 , So from this we can observe that to cover all possible subarrays whose sum <= goal we will add length of our current window means ***( j - i + 1 )*** to count variable.

and after Step 2 , for each step you'll do this only and get count as follow :

1 + 2 + 3 + 4 + 4 ; at last 4 not 5 , as **1 0 1 0 1** , this window sum > goal , so we will shrink it like this 1 **0 1 0 1** , now the size is 4.

so total count = 14.

and similarly we will count the number of subarrays having **Sum <= Goal - 1.**

I won't go into calculations of that , that you can do, but result will be *10*.

That's the logic of this function.

Why calculating atmost goal or count of subarrays having *sum <= goal* , not directly equal to *goal*.

Let me explain you with an example :

Initially , i = 0 , j = 0 , sum = 0 , count = 0 , goal = 2

*Note : Here count will count the subarray having sum = goal.*

Array = [ 1 , 0 , 1 , 0 , 1 ]

**1** , 0 , 1 , 0 , 1 ; i = 0 , j = 0 , sum = 1 , count = 0

**1** , **0** , 1 , 0 , 1 ; i = 0 , j = 1 , sum = 1 , count = 0

**1** , **0** , **1** , 0 , 1 ; i = 0 , j = 2 , sum = 2 , count = 1

**1** , **0** , **1** , **0** , 1 ; i = 0 , j = 3 , sum = 2 , count = 1

**1** , **0** , **1** , **0** , **1** ; i = 0 , j = 4 , sum = 3 , count = 1

Now as sum > goal , we will shrink our window by incrementing our **i** pointer and subtracting ith element from sum till sum > goal and we will get.

1 , **0** , **1** , **0** , **1** ; i = 1 , j = 4 , sum = 2 , count = 2

And now observe, now loop ends as our j == n - 1. but we missed one subarray if you will see , **1 0 1** at last.

Why is this happening ?

As array is containing zeroes so it won't create any impact on current sum , as by adding or subtracting 0 from your sum , sum value won't change , so here we can say with having number of zeros in our array additional possiblity of having subarray of sum = goal is there.

So that's why we are counting atmost or sum <= goal , so we can include all subarray cases.

At last , main question you guys want to know now would be .

🚀 Why***atmostSubarray(nums,goal) - atmostSubarray(nums,goal-1)***;

**a)** atmostSubarray(nums,goal) will give you number of subarrays with sum <= goal

*i.e. goal, goal-1 , goal-2 , goal-3 ... 0*

**b)** atmostSubarray(nums,goal) will give you number of subarrays with sum <= goal -1

*i.e. goal-1 , goal-2 , goal-3 ... 0*

So we only need number of subarrays which having sum as goal ,

atmostSubarray(nums,goal) having that , but additionally having other count too of subarrays having sum < goal.

and we know atmostSubarray(nums,goal -1) having count of all subarrays having sum <= goal - 1 or we can say ***sum < goal*** too. So that means this contain that additional count which we want to remove from atmostSubarray(nums,goal) result.

So we subtract both.

Sorry if it's a quite lengthy , I tried to answer every question or doubt I think , Hope you guys will understand it better now.

***Note : There is a 4th solution of 3 pointers with binary implementation in which we track number of zeros in order to find the subarrays too. But this is quite simple one for optimal .*** 🙂

Please feel free to add some suggestions or having any doubts ask in comments.

Thanks 😊

**Refer to Deepseek**

Yes, the pattern helper(A, S) - helper(A, S - 1) is a common technique used in sliding window problems where you need to count the number of subarrays (or subsequences) that satisfy a certain condition, such as having a sum equal to a specific value. This pattern is particularly useful when the problem can be transformed into finding the number of subarrays with **a sum less than or equal to** a certain value, and then using subtraction to find the exact count for a specific sum.

Here’s another LeetCode problem that can be solved using this pattern:

**LeetCode 992: Subarrays with K Different Integers**

**Problem Statement:**

**Example:**

Input: A = [1,2,1,2,3], K = 2

Output: 7

Explanation: Subarrays with exactly 2 distinct integers are:

[1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]

**Solution Using Sliding Window with the Pattern:**

The problem can be solved by using the sliding window technique with the pattern helper(A, K) - helper(A, K - 1). Here, helper(A, K) counts the number of subarrays with **at most** K distinct integers, and helper(A, K - 1) counts the number of subarrays with **at most** K - 1 distinct integers. Subtracting the two gives the number of subarrays with **exactly K** distinct integers.

Here’s the code:

class Solution {

public int subarraysWithKDistinct(int[] A, int K) {

return helper(A, K) - helper(A, K - 1);

}

private int helper(int[] A, int K) {

int i = 0;

int count = 0;

Map<Integer, Integer> map = new HashMap<>();

for (int j = 0; j < A.length; j++) {

// Add the current element to the map

map.put(A[j], map.getOrDefault(A[j], 0) + 1);

// Shrink the window if the number of distinct elements exceeds K

while (map.size() > K) {

map.put(A[i], map.get(A[i]) - 1);

if (map.get(A[i]) == 0) {

map.remove(A[i]);

}

i++;

}

// Count the number of valid subarrays ending at j

count += j - i + 1;

}

return count;

}

}

**Explanation:**

**helper(A, K)**:

This function counts the number of subarrays with **at most** K distinct integers.

It uses a sliding window to maintain a window [i, j] where the number of distinct integers is less than or equal to K.

When the number of distinct integers exceeds K, the window is shrunk from the left (i++).

**helper(A, K - 1)**:

This function counts the number of subarrays with **at most** K - 1 distinct integers.

It works similarly to helper(A, K) but with a stricter condition.

**Final Result**:

The number of subarrays with **exactly** K distinct integers is given by helper(A, K) - helper(A, K - 1).

**Why This Pattern Works:**

The pattern works because:

helper(A, K) counts all subarrays with **at most** K distinct integers.

helper(A, K - 1) counts all subarrays with **at most** K - 1 distinct integers.

Subtracting the two gives the count of subarrays with **exactly** K distinct integers.

This pattern is also applicable to other problems where you need to count subarrays or subsequences that satisfy a specific condition, such as:

Subarrays with a sum in a specific range.

Subarrays with a specific number of odd/even numbers.

Subarrays with a specific number of elements satisfying a certain property.

Let me know if you'd like further clarification or more examples!

**Solution 2: Hash Table + Prefix Sum (10 min)**

**Refer to**

<https://leetcode.com/problems/binary-subarrays-with-sum/editorial/>

**Overview**

We are given a binary array nums and an integer goal. The task is to find the number of non-empty subarrays in the given binary array where the sum of elements in the subarray equals the specified goal.

**Key Observations:**

The array contains only binary values (0 or 1).

The goal is to find subarrays with a specific sum.

The subarrays should be non-empty and contiguous.

Consider the given example with nums = [1,0,1,0,1] and goal = 2:

Output: 4

Explanation: The 4 subarrays are bolded and underlined below:

[**1,0,1**,0,1]

[**1,0,1,0**,1]

[1,**0,1,0,1**]

[1,0,**1,0,1**]

Note that all these subarrays are contiguous parts of the given array, and the count of such subarrays is the output.

**Approach 1: Prefix Sum**

**Intuition**

The task involves identifying contiguous sequences of elements within an array whose sum equals a specific target value. Problems that require sequences of elements to meet criteria often utilize [prefix sums](https://en.wikipedia.org/wiki/Prefix_sum).

We begin by iterating through the array. As we encounter each element, we maintain a running total (current sum). This current sum represents the cumulative addition of all elements encountered so far in the array.

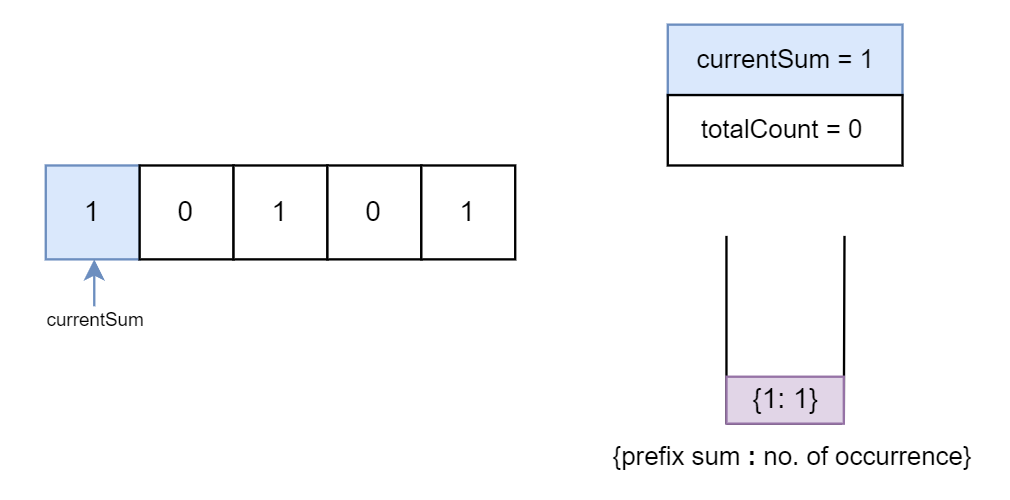
Next, we check if the current sum precisely matches the target value. If it does, we have found a subarray whose elements add up to the goal.

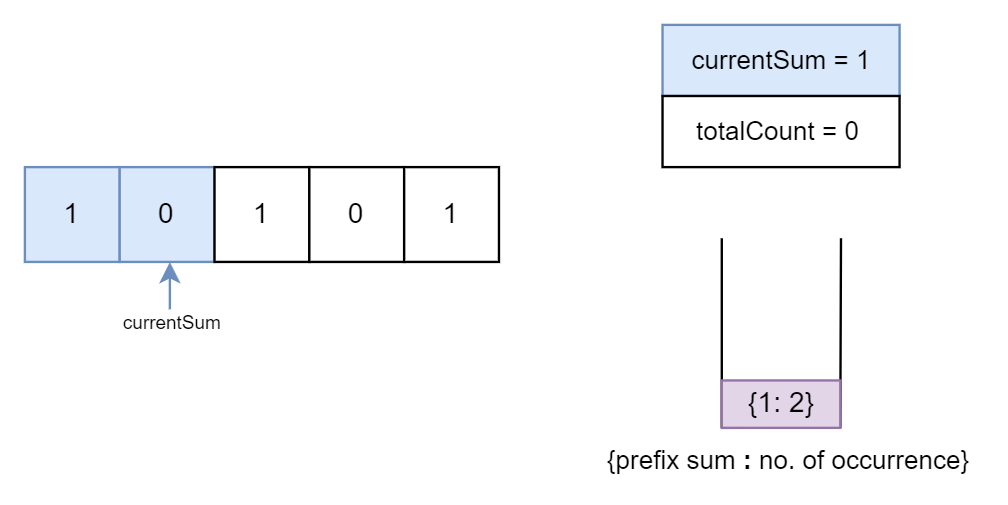
Now consider a scenario where the current sum exceeds the target value. This doesn't necessarily eliminate the possibility of finding a subarray that meets the criteria. We need a method to determine the sum of subarrays that begin after the first index of the original array.

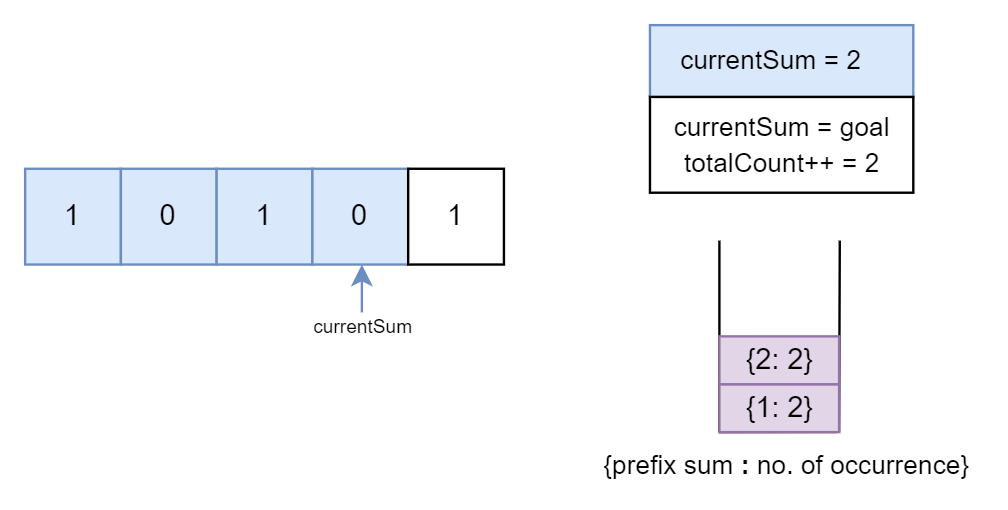
A prefix sum represents the cumulative sum of elements up to a specific point in the array. By subtracting the target value from the current sum, we obtain a new value, called as "prefix sum." If this value appears earlier in the array, it means a subarray starting later adds up to the target. In simpler terms, a subsequence of these elements adds up to the target sum value.

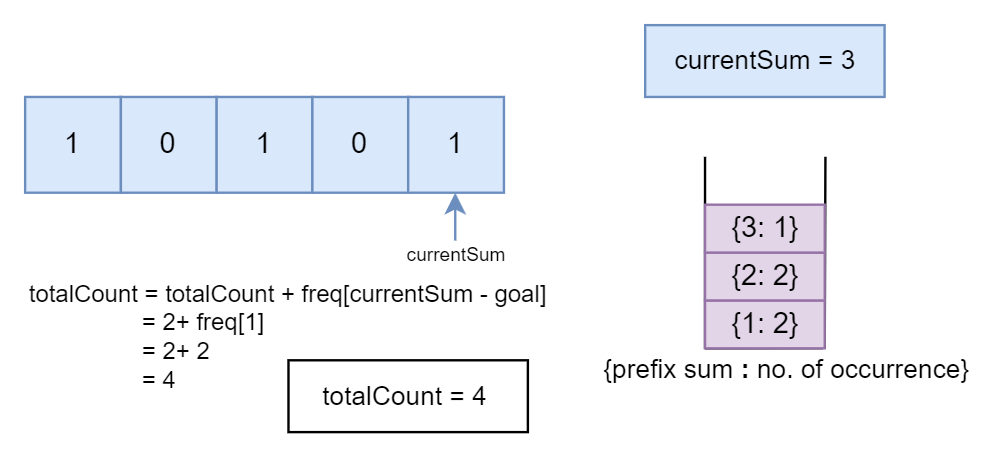
We can use a map to track the occurrences of prefix sums. If a prefix sum exists in the map, it indicates multiple groups that sum to the target. We update the map by adding the current sum. This ensures we can find any corresponding subarrays that leads to goal.

Refer to the visual slideshow demonstrating the algorithm with the example input [1, 0, 1, 0, 1] and goal = 2.









**Algorithm**

Initialize the totalCount variable to keep track of the number of subarrays with the desired sum and the currentSum variable to keep track of the cumulative sum of elements encountered so far.

Initialize a hash table, freq, to store the frequency of encountered prefix sums.

Iterate through the array nums.

Add the current element to the currentSum to get the updated running total. If the updated currentSum is equal to the goal, it means a subarray with a sum equal to the goal has been found. Increment totalCount by 1.

Check if the freq map contains a prefix sum currentSum - goal. This currentSum - goal represents the prefix sum of a subarray that, when added to the current element num, could potentially form a subarray with a sum equal to goal.

If freq[currentSum - goal] is in the hash table, it means there exists a subarray with a prefix sum equal to currentSum - goal. In this case:

Add the frequency of currentSum - goal (the number of subarrays with that prefix sum) to totalCount. These subarrays, when combined with num, would also result in a subarray with a sum equal to goal.

Update the frequency map by incrementing the frequency count for the current sum.

Return the totalCount variable.

**Implementation**

class Solution {

public int numSubarraysWithSum(int[] nums, int goal) {

int totalCount = 0;

int currentSum = 0;

// {prefix: number of occurrence}

Map<Integer, Integer> freq = new HashMap<>(); // To store the frequency of prefix sums

for (int num : nums) {

currentSum += num;

if (currentSum == goal) {

totalCount++;

}

// Check if there is any prefix sum that can be subtracted from the current sum to get the desired goal

if (freq.containsKey(currentSum - goal)) {

totalCount += freq.get(currentSum - goal);

}

freq.put(currentSum, freq.getOrDefault(currentSum, 0) + 1);

}

return totalCount;

}

}

**Complexity Analysis**

Let *n* be the length of the input array nums.

Time complexity: *O*(*n*)

We iterate through the array once to calculate the prefix sums and update the frequency map.

Space complexity: *O*(*n*)

We use an unordered map (freq) to store the frequency of prefix sums. In the worst case, all prefix sums can be distinct, resulting in *n* unique entries in the map. Therefore, the space required is proportional to the size of the input array.

**Refer to Deepseek**

Yes, **LeetCode 930: Binary Subarrays With Sum** can also be solved using the **Prefix Sum + HashMap** approach. This method is efficient and works in (O(n)) time with (O(n)) space complexity.

**Problem Recap:**

Given a binary array A (containing only 0s and 1s) and an integer S, return the number of non-empty subarrays with a sum equal to S.

**Example:**

Input: A = [1, 0, 1, 0, 1], S = 2

Output: 4

Explanation: The subarrays with sum 2 are:

[1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1]

**Prefix Sum + HashMap Approach:**

The idea is to use a **prefix sum** to keep track of the cumulative sum up to each index in the array. We then use a **HashMap** to store the frequency of each prefix sum encountered so far. This allows us to efficiently check if a subarray with the desired sum exists.

**Key Insight:**

If the cumulative sum up to index j is sum\_j, and the cumulative sum up to index i is sum\_i, then the sum of the subarray A[i+1 ... j] is sum\_j - sum\_i.

We want sum\_j - sum\_i = S, which can be rewritten as sum\_i = sum\_j - S.

We use the HashMap to count how many times sum\_i = sum\_j - S has occurred so far.

**Algorithm Steps:**

Initialize:

**A HashMap prefixSumCount to store the frequency of prefix sums. Initialize it with {0: 1} because a prefix sum of 0 occurs once before the array starts.**

A variable prefixSum to keep track of the cumulative sum.

A variable count to store the result.

Iterate through the array:

Update the prefixSum by adding the current element.

Check if prefixSum - S exists in the HashMap. If it does, add its frequency to count.

Update the HashMap with the current prefixSum.

Return the count.

**Code Implementation:**

import java.util.HashMap;

import java.util.Map;

class Solution {

public int numSubarraysWithSum(int[] A, int S) {

// HashMap to store the frequency of prefix sums

Map<Integer, Integer> prefixSumCount = new HashMap<>();

prefixSumCount.put(0, 1); // Initialize with prefix sum 0 occurring once

int prefixSum = 0;

int count = 0;

for (int num : A) {

prefixSum += num; // Update the prefix sum

// Check if (prefixSum - S) exists in the HashMap

if (prefixSumCount.containsKey(prefixSum - S)) {

count += prefixSumCount.get(prefixSum - S);

}

// Update the frequency of the current prefix sum in the HashMap

prefixSumCount.put(prefixSum, prefixSumCount.getOrDefault(prefixSum, 0) + 1);

}

return count;

}

}

**Explanation:**

Initialization:

prefixSumCount = {0: 1}: This accounts for the empty subarray before the array starts.

prefixSum = 0: Tracks the cumulative sum.

count = 0: Tracks the number of valid subarrays.

Iteration:

For each element in the array:

Update prefixSum by adding the current element.

Check if prefixSum - S exists in the HashMap. If it does, it means there are subarrays ending at the current index with sum S. Add the frequency of prefixSum - S to count.

Update the frequency of the current prefixSum in the HashMap.

Example Walkthrough:

For A = [1, 0, 1, 0, 1] and S = 2:

At index 0: prefixSum = 1, prefixSum - S = -1 (not in HashMap). Update HashMap: {0:1, 1:1}.

At index 1: prefixSum = 1, prefixSum - S = -1 (not in HashMap). Update HashMap: {0:1, 1:2}.

At index 2: prefixSum = 2, prefixSum - S = 0 (exists in HashMap). Add 1 to count. Update HashMap: {0:1, 1:2, 2:1}.

At index 3: prefixSum = 2, prefixSum - S = 0 (exists in HashMap). Add 1 to count. Update HashMap: {0:1, 1:2, 2:2}.

At index 4: prefixSum = 3, prefixSum - S = 1 (exists in HashMap). Add 2 to count. Update HashMap: {0:1, 1:2, 2:2, 3:1}.

Final count = 4.

**Time Complexity:**

(O(n)): We traverse the array once, and each HashMap operation (get/put) is (O(1)) on average.

**Space Complexity:**

(O(n)): The HashMap can store up to (n) distinct prefix sums in the worst case.

This approach is efficient and works well for this problem. Let me know if you have further questions!

**Refer to**

[L992.Subarrays with K Different Integers](note://WEBed9f16531d534d22184e6335b24e4cff)

[L1248.Count Number of Nice Subarrays (Ref.L930,L992)](note://WEBdc5ce9ddbc80dc6d2841bbc9e2674442)