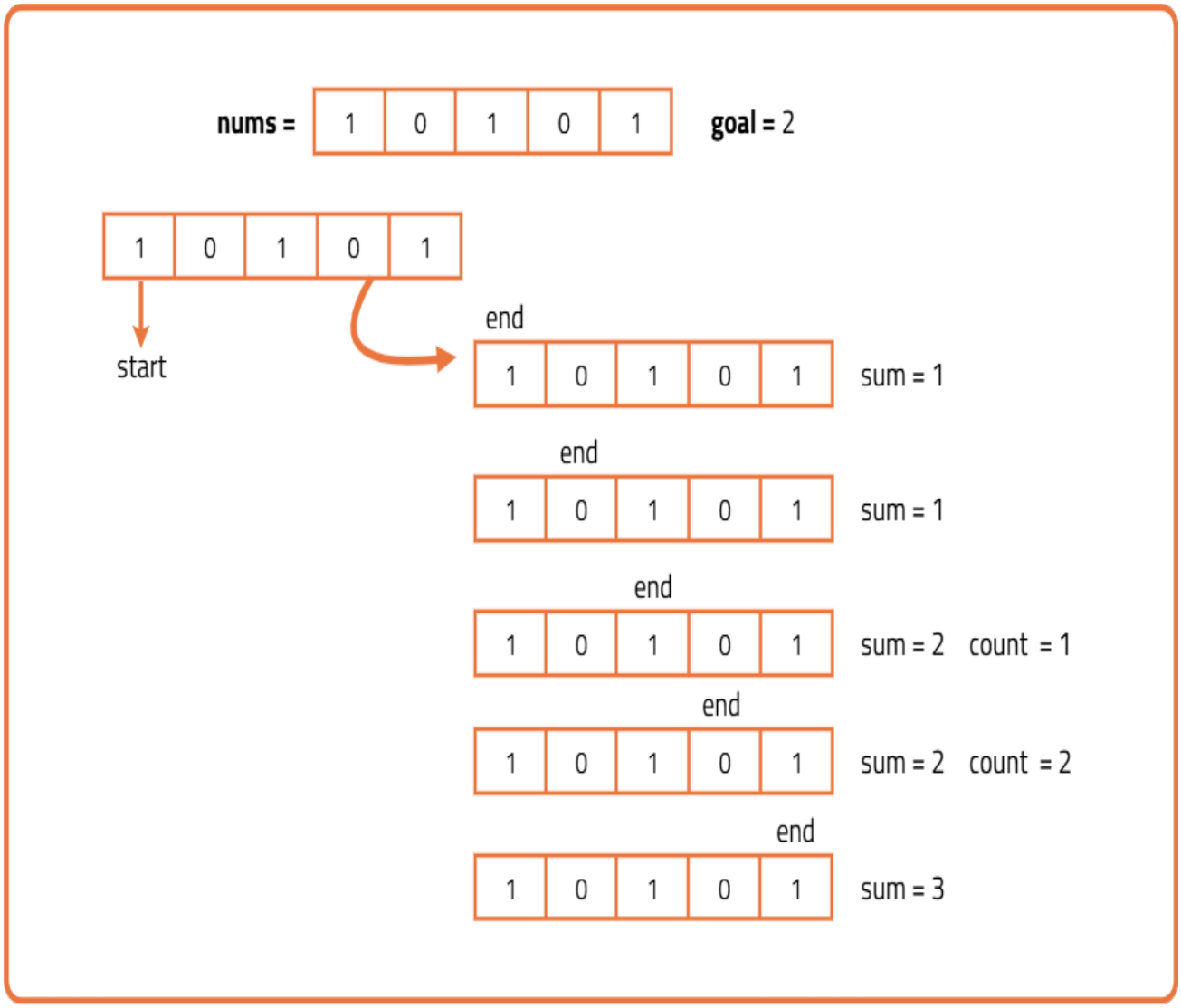
Binary subarray with sum

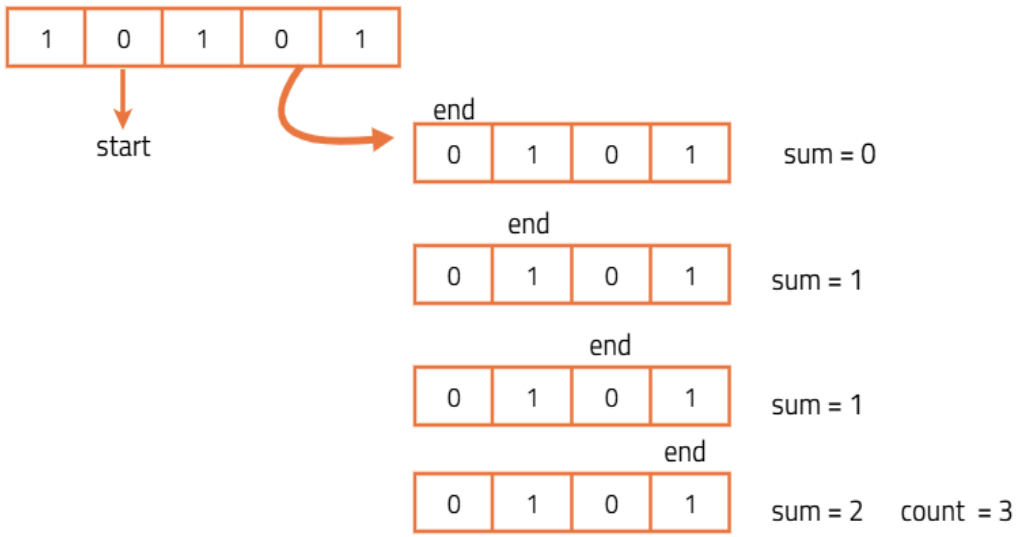
Problem Statement: You are given a binary array nums (containing only 0s and 1s) and an integer goal. Return the number of non-empty subarrays of nums that sum to goal. A subarray is a contiguous part of the array.

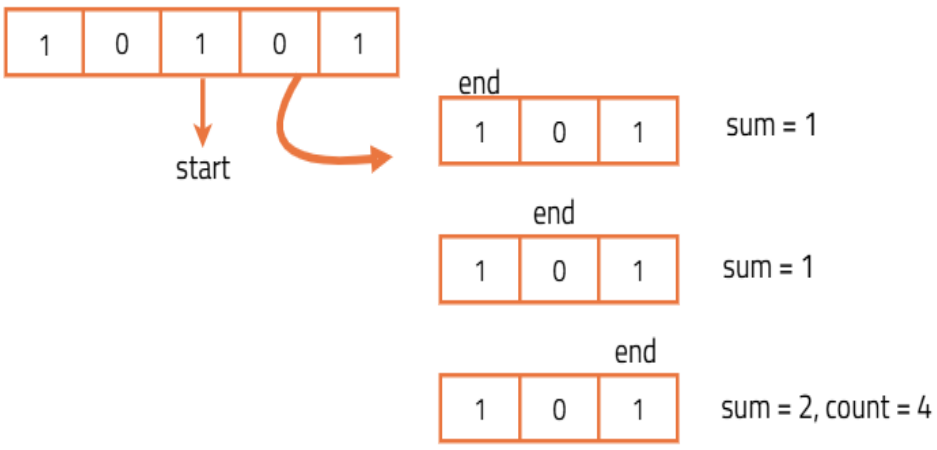
Examples

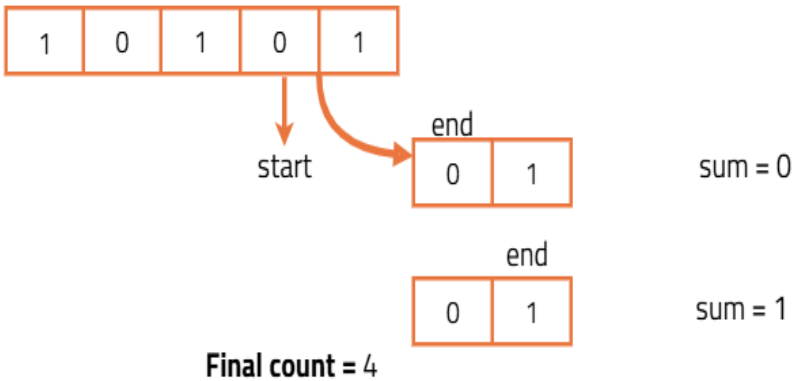
Brute force Approach
Algorithm
In the brute-force approach, we can compute the sum of all possible subarrays using nested loops and check whether the sum is equal to the target. This method is simple to implement and helpful to understand the core idea but is inefficient for large inputs.

- Initialize a counter to keep track of valid subarrays.
- Loop through all possible starting points of the subarrays .
- For each starting point i, initialize a sum variable to 0.
- Loop through all possible ending points of the subarrays
- Add the current element to the sum.
- If the sum equals the goal, increment the count.
- After checking all subarrays, return the count as the result.









Final count = 4

Code

```
import java.util.*;
     class Solution {
    // Function to count number of subarrays with sum equal to goal
    public int numSubarraysWithSum(int[] nums, int goal) {
        // Variable to store the final count of valid subarrays
        int count = 0;

        // Outer loop to fix the starting index of subarray
        for (int start = 0; start < nums.length; start++) {
            // Variable to store sum of current subarray
            int sum = 0;

            // Inner loop to fix the ending index of subarray
            for (int end = start; end < nums.length; end++) {
                // Add the current element to sum
                sum += nums[end];

                // If subarray sum equals goal, increment count
                if (sum == goal) {
                    count++;
                }
            }
        }

        // Return the total count of valid subarrays
        return count;
    }
}

// Driver code
class Main {
    public static void main(String[] args) {
        Solution obj = new Solution();
        int[] nums = {1, 0, 1, 0, 1};
        int goal = 2;
        // Output : 4
        System.out.println(obj.numSubarraysWithSum(nums, goal));
    }
}
```

Complexity Analysis

Time Complexity: O(n^2), where n is the length of the array.We are using two nested loops to explore all possible subarrays. Each subarray takes O(1) time to compute the sum cumulatively, so overall O(n^2) pairs are checked.

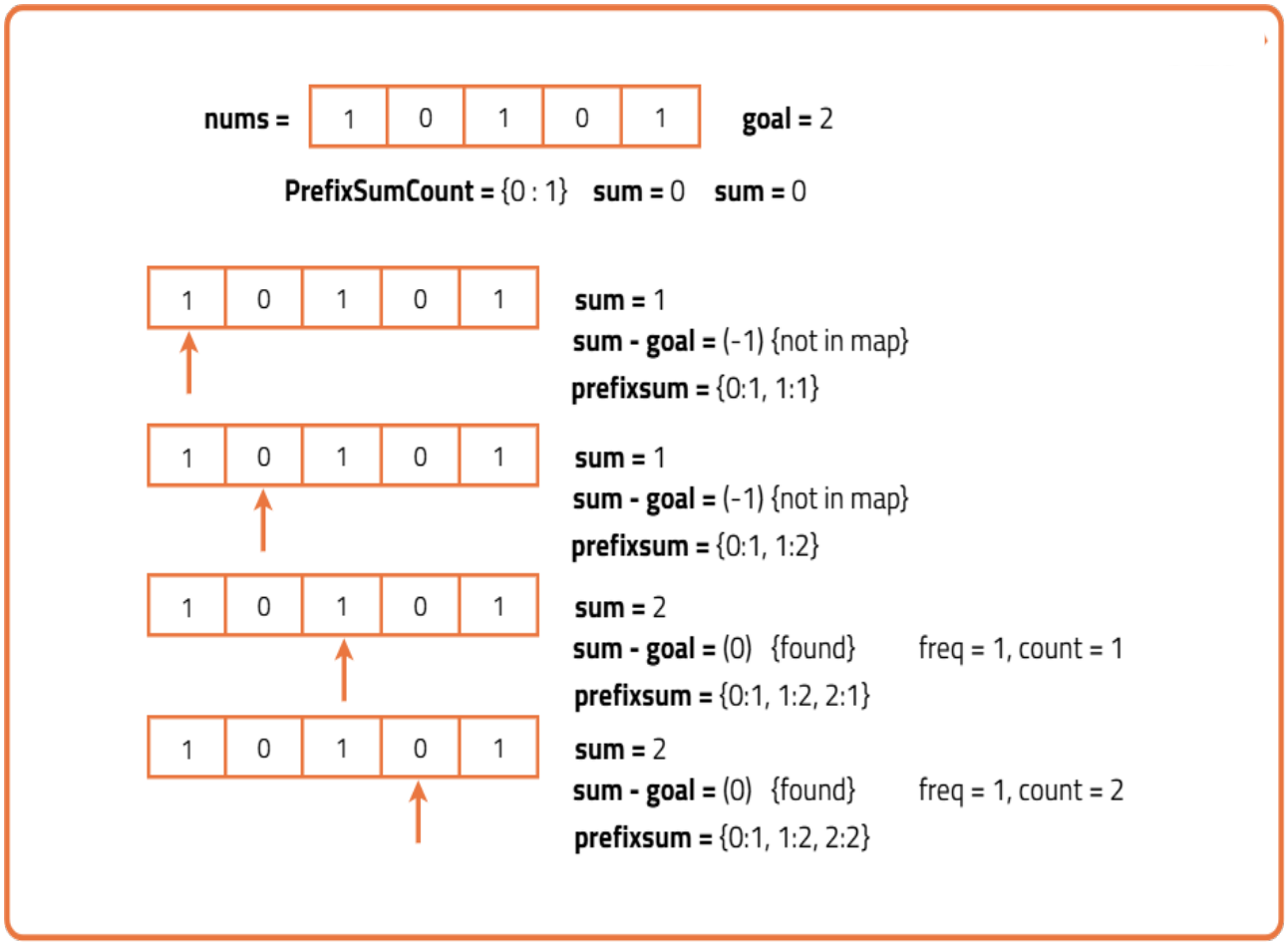
Space Complexity: O(1), constant space .We only use integer variables to store counts and intermediate sums.

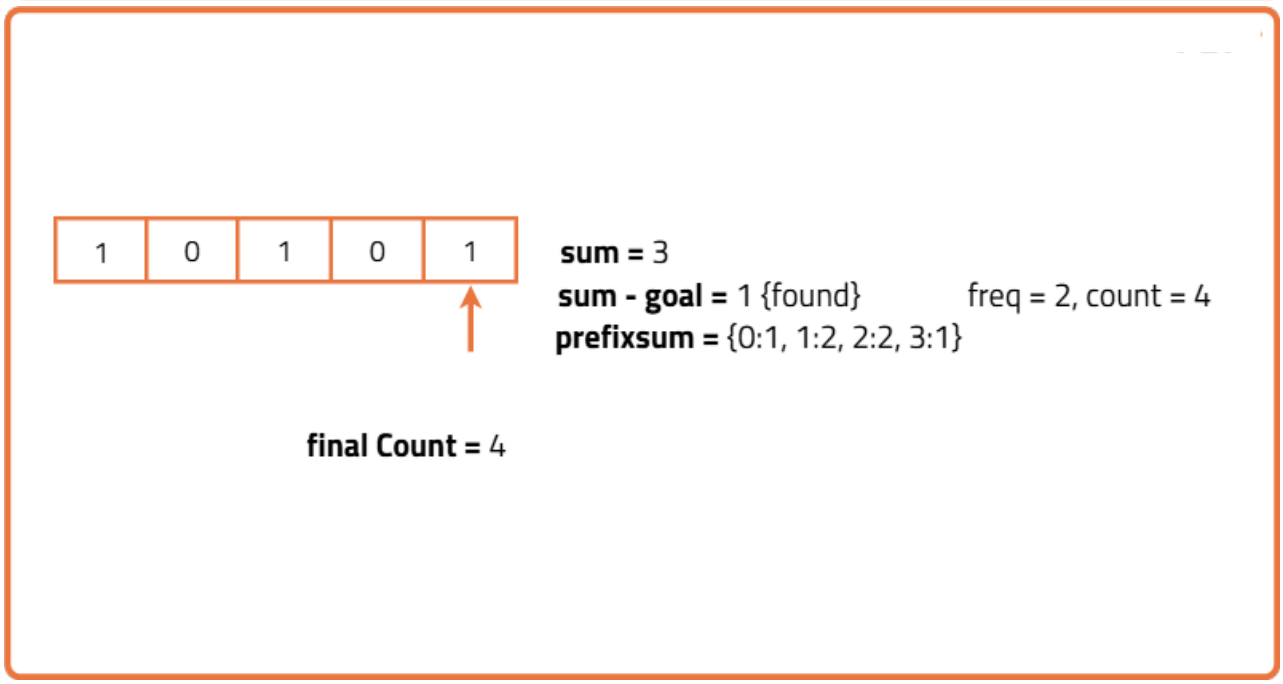
Better Approach

Algorithm

Instead of iterating over all subarrays, we can use the prefix sum technique, we can keep track of how many times a specific prefix sum has occurred. If at index i, the prefix sum is currSum, and we know how many times currSum - goal has occurred before, then each of those occurrences can form a valid subarray ending at index i. This avoids recomputation and eliminates the need to check all possible subarrays explicitly.

- Initialize a hash map to store frequency of prefix sums seen so far.
- Initialize variables for the current prefix sum and the count of valid subarrays.
- Iterate over the input array:
    - At each step, update the running prefix sum.
    - Check if `(prefix_sum - goal)` exists in the map; if yes, add its frequency to the result.
    - Update the prefix sum count in the hash map.





Code

```
import java.util.*;

class Solution {
    // Function to count number of subarrays with sum equal to goal
    public int numSubarraysWithSum(int[] nums, int goal) {
        // Map to store prefix sum frequencies
        Map<Integer, Integer> prefixSumCount = new HashMap<>();

        // Initialize count and sum
        int count = 0, sum = 0;

        // Add base case: prefix sum 0 appears once
        prefixSumCount.put(0, 1);

        // Iterate through array
        for (int num : nums) {
            // Add current number to sum
            sum += num;

            // If (sum - goal) exists, add its count to result
            if (prefixSumCount.containsKey(sum - goal)) {
                count += prefixSumCount.get(sum - goal);
            }

            // Update prefix sum count
            prefixSumCount.put(sum, prefixSumCount.getOrDefault(sum, 0) + 1);
        }

        // Return final count
        return count;
    }
}

// Driver code
class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] nums = {1, 0, 1, 0, 1};
        int goal = 2;
        System.out.println(sol.numSubarraysWithSum(nums, goal));
    }
}
```

Complexity Analysis

Time Complexity: O(n), where n is the length of the input array . Each element is visited exactly once during the single-pass traversal.

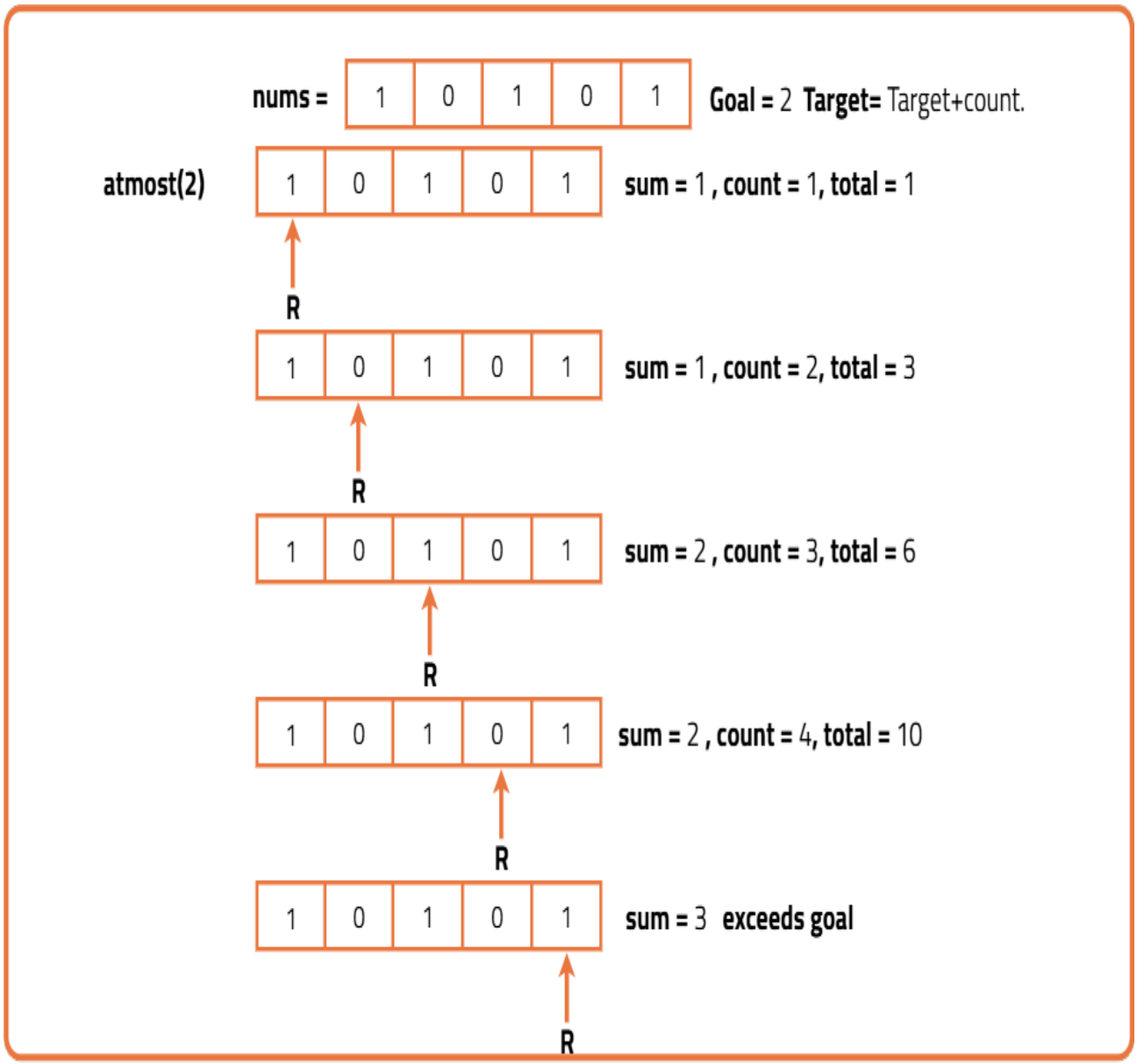
Space Complexity: O(n), where n is the length of the input array . In the worst case, all cumulative sums are distinct, so the hash map can store up to n unique keys. Thus, the space required grows linearly with the input size.

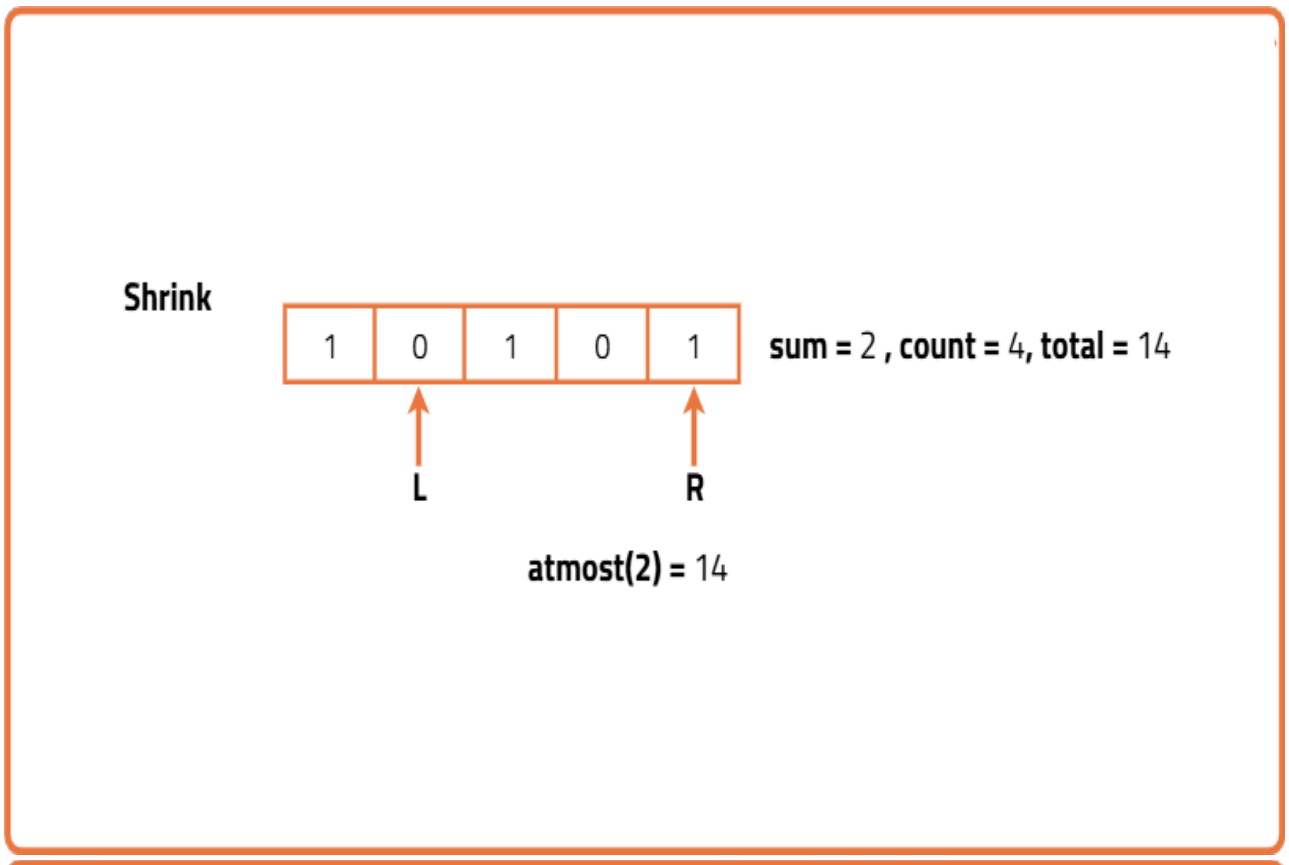
Optimal Approach
Algorithm
To count the number of subarrays with sum exactly equal to goal, a clever strategy is to reframe the problem: We count the number of subarrays whose sum is at most goal, and subtract from it the number of subarrays whose sum is at most goal- 1.

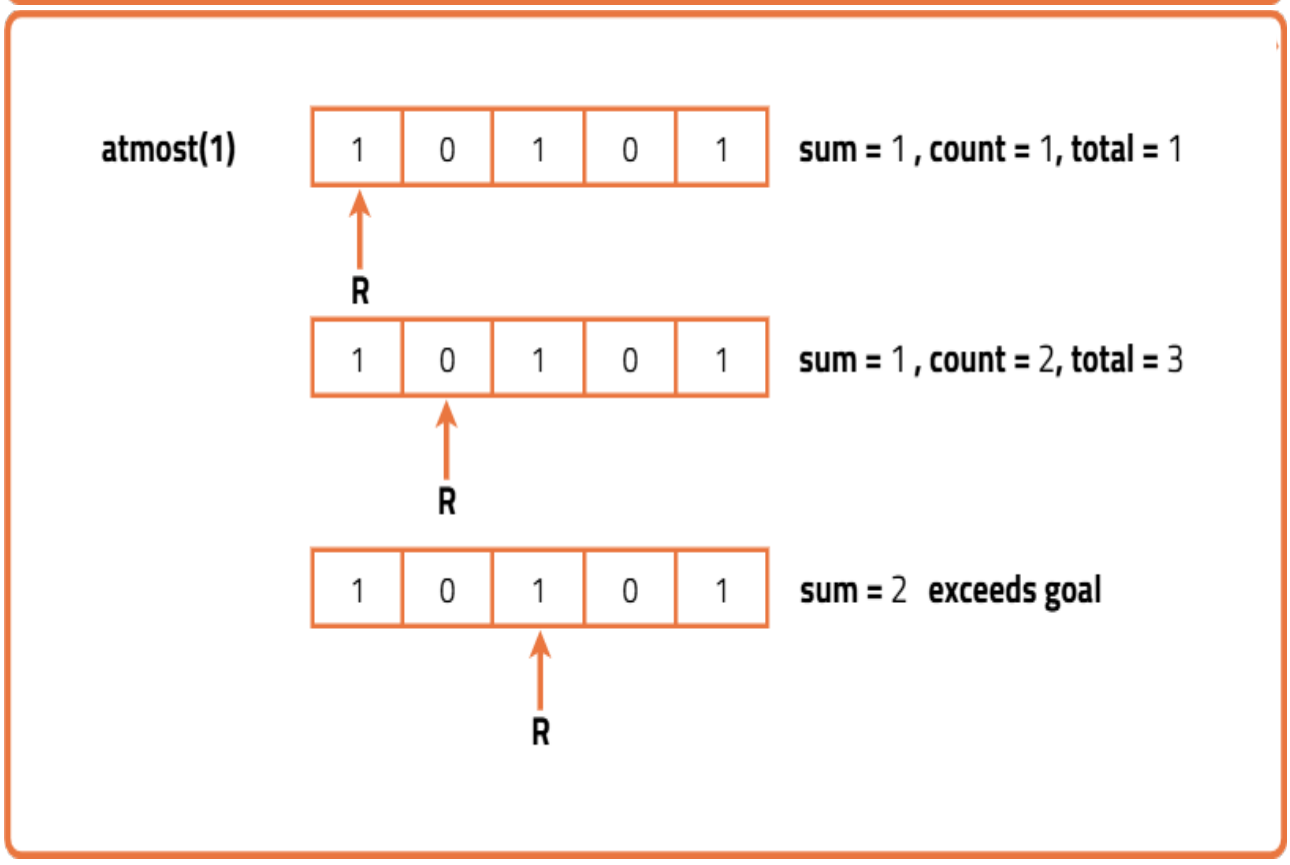
This works because: The subarrays with sum exactly goal are the ones included in atMost(goal) but not in atMost(goal - 1) and This is valid for non-negative elements.

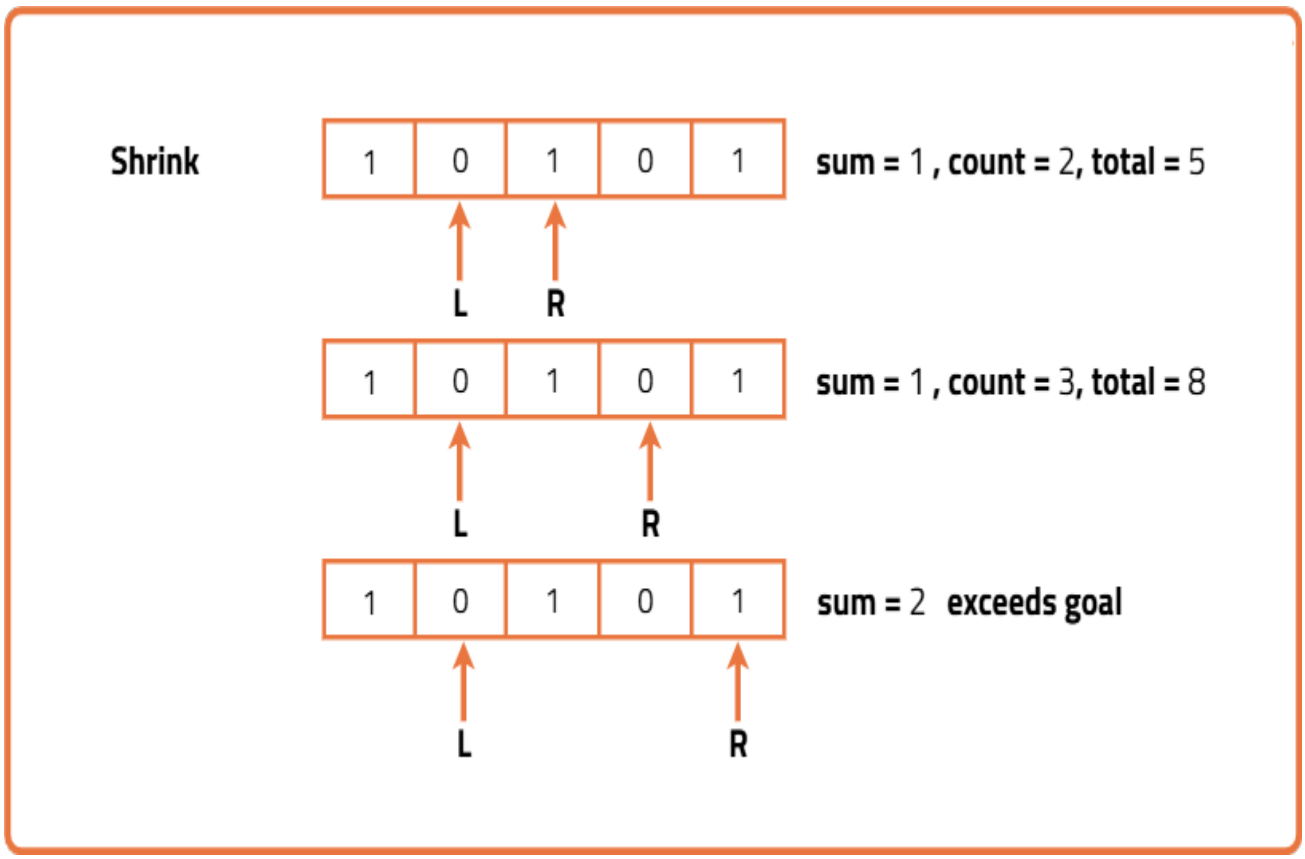
Why is this more efficient?
Instead of recomputing subarray sums from scratch , we maintain a sliding window that expands and contracts based on the current sum.This gives us linear time performance by moving each pointer at most once. This method only works when goal ≥ 1, because the atMost(goal - 1) calculation is invalid for goal = 0 (negative index/window not possible).

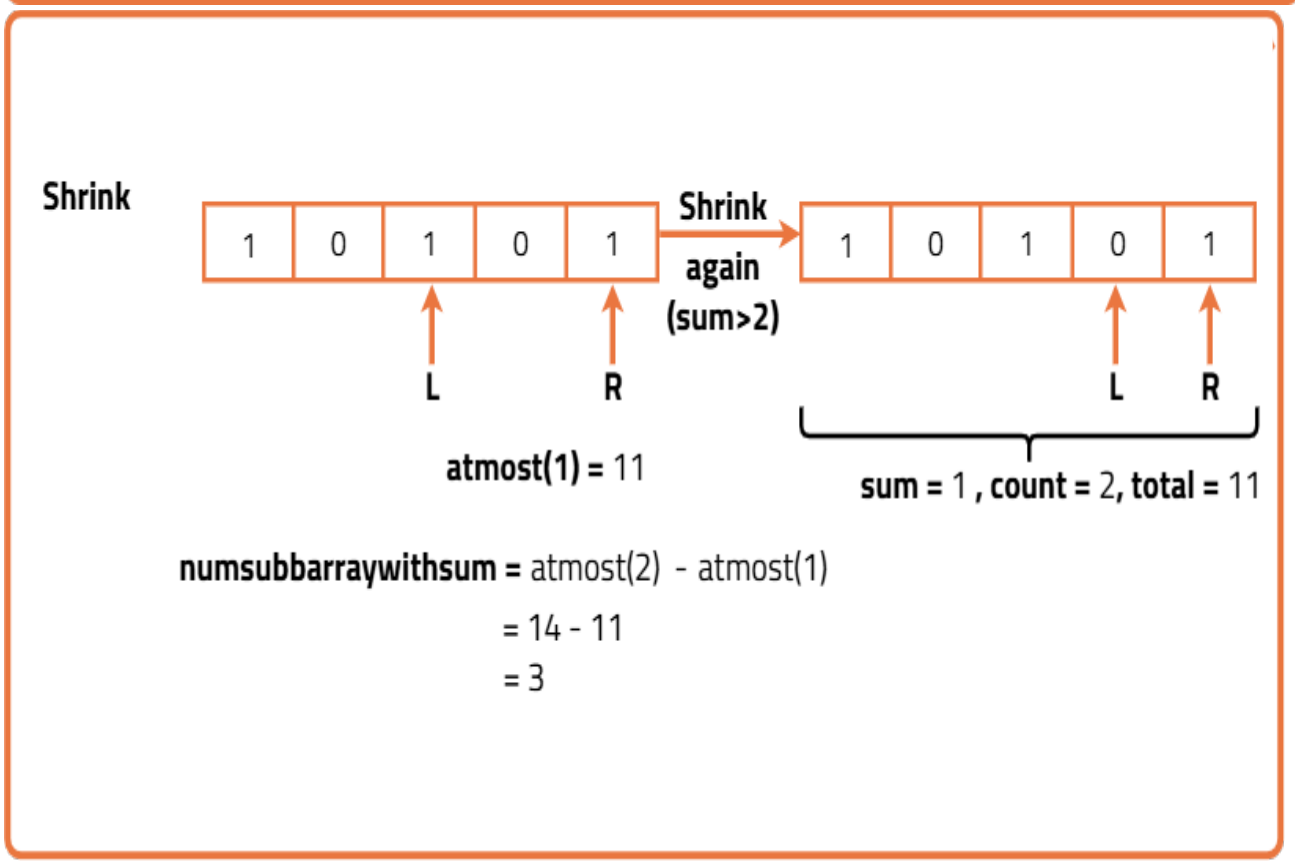
- Define a helper function to calculate the number of subarrays with sum at most a given value
- Initialize a sliding window with two pointers (left and right)
- Iterate through the array, expanding the right pointer and adding to current sum
- If the current sum exceeds the target, shrink the window from the left until it's valid
- At each step, add the size of the current valid window to the count
- To get the final answer, compute: atMost(goal) - atMost(goal - 1)




Shrink
1 0 1 0 1
sum = 2 , count = 4, total = 14
L
R
atmost(2) = 14


atmost(1)
1 0 1 0 1
sum = 1 , count = 1, total = 1
R
1 0 1 0 1
sum = 1 , count = 2, total = 3
R
1 0 1 0 1
sum = 2 exceeds goal
R





numsubbarraywithsum = atmost(2) - atmost(1)

= 14 - 11

= 3

Code

```
import java.util.*;

class Solution {
    // Function to calculate number of subarrays with sum exactly equal to goal
    public int numSubarraysWithSum(int[] nums, int goal) {
        // Return difference between atMost(goal) and atMost(goal - 1)
        return atMost(nums, goal) - atMost(nums, goal - 1);
    }

    // Helper method to count subarrays with sum at most k
    private int atMost(int[] nums, int k) {
        // No valid subarray for negative sum
        if (k < 0) return 0;

        int left = 0;
        int sum = 0;
        int count = 0;

        // Traverse array using right pointer
        for (int right = 0; right < nums.length; right++) {
            // Add current element to sum
            sum += nums[right];

            // Shrink window if sum exceeds k
            while (sum > k) {
                sum -= nums[left];
                left++;
            }

            // Add number of valid subarrays ending at right
            count += (right - left + 1);
        }

        return count;
    }
}

// Driver class
class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] nums = {1, 0, 1, 0, 1};
        int goal = 2;
        System.out.println(sol.numSubarraysWithSum(nums, goal)); // Output: 4
    }
}
```

Complexity Analysis

Time Complexity: O(n), where n is the size of the input array.This is because the algorithm uses the sliding window technique twice (in the two calls to atMost). Each `atMost` function runs in linear time , the left and right pointers only move forward, never backward, so the total number of operations is at most 2n.

Space Complexity: O(1), constant extra space.Only a few integer variables are used .