Number of Subarrays with Sum K

This study guide covers the problem of finding the number of subarrays within a given array that sum up to a target value *K*. A subarray is defined as any contiguous part of the array.

> Subarray: Any contiguous part of an array. For example, in the array [1, 2, 3, 4, 5], [2, 3, 4] is a subarray, but [1, 3, 5] is not.

Let's consider an example array: [1, 2, 3, -3, 1, 1, 1, 4, 2, -3] with $K = 3$. There are eight subarrays that sum to 3:

- [1, 2]
- [3]
- [1, 1, 1]
- [1, 2, 3, -3]
- [2, 3, -3, 1]
- [-3, 1, 1, 1, 4, 2, -3]
- [4, 2, -3]
- [-3, 1, 1, 1]

Brute Force Solution

The brute force approach involves generating all possible subarrays and checking if their sum equals *K*.

1. Generate All Subarrays: Use nested loops to define the start and end points of each subarray.
2. Calculate Sum: For each subarray, calculate the sum of its elements.
3. Count Matching Subarrays: If the sum equals *K*, increment the counter.

```
for i from 0 to n:
  for j from i to n - 1:
    sum = 0
    for k from i to j:
      sum = sum + array[k]
    if sum == K:
      counter++
```

- Time Complexity: O(n^3), due to the three nested loops.
- Space Complexity: O(1), as no extra space is used.

Better Solution

An improved solution reduces the time complexity by avoiding the repeated calculation of the subarray sum. Instead of using a third nested loop, we can maintain a running sum as we expand the subarray.

1. Outer Loops: Use nested loops to define the start and end points of each subarray.
2. Running Sum: Keep a running sum as you iterate through the inner loop. Instead of recalculating the sum each time, just add the next element.
3. Count Matching Subarrays: If the sum equals *K*, increment the counter.

```
for i from 0 to n:
  for j from i to n - 1:
    sum = 0
    sum = sum + array[j]
    if sum == K:
      counter++
```

- Time Complexity: O(n^2), due to the two nested loops.
- Space Complexity: O(1), as no extra space is used.

Optimal Solution Using Prefix Sum

The most efficient solution leverages the concept of prefix sum.

> Prefix Sum: The prefix sum at any index *i* in an array is the sum of all elements from the start of the array up to index *i*.

The main idea is to use the relationship between prefix sums to find subarrays with a sum of *K*. If the prefix sum up to a certain index is *X*, and we are looking for a subarray with a sum of *K*, then the sum of the remaining portion should be *X* - *K*.

1. Calculate Prefix Sum: Iterate through the array and calculate the prefix sum at each index.
2. Use a Hash Map: Store the prefix sums and their frequencies in a hash map.
3. Find Subarrays: For each prefix sum *X*, check if *X* - *K* exists in the hash map. If it does, it means there is a subarray with a sum of *K*.

$$PrefixSum[i] = PrefixSum[i-1] + arr[i]$$

$$IfPrefixSum[i] - Kexists, incrementcount$$

Here's a breakdown of the optimal solution:

Step	Description
1. Initialize Variables	- `count = 0`: Stores the number of subarrays with sum *K*. - `prefixSumMap = {0: 1}`: A hash map to store prefix sums and their frequencies.
2. Iterate Through the Array	For each element in the array, calculate the current prefix sum.
3. Update Prefix Sum	$$currentSum = currentSum + nums[i]$$
4. Check for Subarray with Sum K	Check if `currentSum - K` exists in the `prefixSumMap`. If it does, increment the `count` by the frequency of `currentSum - K`.
5. Update Prefix Sum Map	Update the frequency of the `currentSum` in the `prefixSumMap`.
6. Return Count	After iterating through the entire array, return the `count`.

- Time Complexity: O(n), as it requires a single pass through the array.
- Space Complexity: O(n), due to the use of a hash map to store prefix sums.

Subarray Sum Equals K

Understanding the Logic

When searching for a subarray with a sum equal to `K`, instead of directly looking for `K` within the array, we leverage prefix sums. The approach involves looking for prefix sums of `x - K`, where `x` is the total summation.

If there are two prefix sums with a value of `x - K`, it implies there are two subarrays ending at the current index with a sum of `K`. This reverse engineering works because prefix sums can be easily stored, unlike arbitrary subarray sums.

The core idea is that if the prefix sum up to a point is `s`, and we are looking for a subarray with sum `K`, we need to find how many times `s - K` has appeared as a prefix sum. The number of occurrences of `s - K` will be equal to the number of subarrays with sum `K` ending at the current index.

Data Structures Required

To efficiently implement this logic, we use a **hash map** to store prefix sums and their frequencies.

> **Hash Map**: A data structure that stores prefix sums and the number of times each sum has occurred.

- **Key**: Prefix sum value
- **Value**: Count of occurrences

We also maintain two variables:

- `prefixSum`: Keeps track of the current prefix sum.
- `count`: Keeps track of the number of subarrays that sum to `K`.

Algorithm Steps

1. Initialization:
    - Initialize the `prefixSum` to 0.
    - Initialize a hash map with an entry `0: 1`. This indicates that a prefix sum of 0 occurs once at the beginning.
    - Initialize `count` to 0.
2. Iteration:
    - Iterate through the array, updating the `prefixSum` as you go.
    - For each element, calculate the value `s - K`, where `s` is the current `prefixSum`.
    - Check if `s - K` exists in the hash map. If it does, increment `count` by the number of times `s - K` has occurred.
    - Update the frequency of the current `prefixSum` in the hash map.
3. Result:
    - After iterating through the entire array, the `count` variable will hold the number of subarrays with a sum equal to `K`.

Dry Run Example

Let's consider an array `[1, 2, 3, 3, -3, 1, 5, 6, 4, -3]` and `K = 3`.

Element	prefixSum	s - K	Hash Map	Count
	0	-3	{0: 1}	0
1	1	-2	{0: 1, 1: 1}	0
2	3	0	{0: 1, 1: 1, 3: 1}	1
3	6	3	{0: 1, 1: 1, 3: 1, 6: 1}	2
-3	3	0	{0: 1, 1: 1, 3: 2, 6: 1}	3
1	4	1	{0: 1, 1: 1, 3: 2, 6: 1, 4: 1}	4
5	9	6	{0: 1, 1: 1, 3: 2, 6: 1, 4: 1, 9: 1}	5
4	13	10	{0: 1, 1: 1, 3: 2, 6: 1, 4: 1, 9: 1, 13: 1}	5
-3	10	7	{0: 1, 1: 1, 3: 2, 6: 1, 4: 1, 9: 1, 13: 1, 10: 1}	5

Why Storing Zero is Important

Consider the array `[3, -3, 1, 1]`. Without storing zero in the hash map, you might miss some valid subarrays. Storing zero accounts for cases where the prefix sum itself equals `K`, indicating a subarray starting from the beginning of the array.

Prefix Sum and Hash Map Solution

This approach uses a **hash map** to store prefix sums and efficiently count subarrays with a sum of `k`.

```
1. Initialize a map to store prefix sums with initial value {0: 1}.
2. Initialize prefix sum as 0 and count as 0.
3. Iterate through the array:
    a. Update prefix sum by adding the current element.
    b. Look for the number of times (prefix sum - k) has appeared, as this inc
    c. Update the count by adding the frequency of (prefix sum - k) in the map
    d. Update the frequency of the current prefix sum in the map.
4. Return the count.
```

Here's an example illustrating the concept. To find subarrays summing to `k`, we look backwards to see how many initial prefix sums can be removed to achieve the target sum.

Code Implementation

The code demonstrates how to implement the prefix sum and hash map approach.

```python
def subarraySum(nums, k):
    prefix_sum_map = {0: 1}
    prefix_sum = 0
    count = 0

    for num in nums:
        prefix_sum += num
        if prefix_sum - k in prefix_sum_map:
            count += prefix_sum_map[prefix_sum - k]

        if prefix_sum in prefix_sum_map:
            prefix_sum_map[prefix_sum] += 1
        else:
            prefix_sum_map[prefix_sum] = 1

    return count
```

Time and Space Complexity Analysis

Category	Complexity
Time Complexity	$O(n)$
Space Complexity	$O(n)$

- Time Complexity: $O(n)$ because we linearly iterate through the array. On average, hash map operations take $O(1)$. In the worst case (collisions), it might degrade to $O(n)$ if a simple map is used.
- Space Complexity: $O(n)$ because, in the worst case, we might store `n` different prefix sums in the map.