560. Subarray Sum Equals K

Medium | Array | Hash Table | Prefix Sum

Problem Description

The problem at hand involves finding the total number of contiguous subarrays within a given array of integers (`nums`) that add up to a specified integer (`k`). A subarray is defined as a continuous, non-empty sequence of elements taken from the array.

To better understand the problem, let's consider an example:

If our input `nums` is `[1, 2, 1, 2, 1]` and `k` is `3`, we need to find all the subarrays where the elements sum to `3`. In this case, there are several such subarrays: `[1, 2]`, `[2, 1]`, `[1, 2]`, `[1, 2, 1] - ([2, 1] after removing the last element)`, hence the output would be `4`.

Intuition

The solution approach involves using a cumulative sum and hashmap (Counter) to keep track of the number of ways a certain sum can occur up to the current point in the array. The cumulative sum (`s`) represents the total sum of all elements up to the current index.

As we traverse the array, we attempt to find the number of times the complement of the current cumulative sum, given by the difference (`s - k`), has already appeared. This is because if we have seen a sum `s - k` before, then adding `k` to this sum would give us `s`, and thus, there exists a subarray ending at the current index which sums to `k`.

The reason a Counter is initialized with `{0: 1}` is because a cumulative sum of `0` occurs once by default before the start of the array (think of it as a dummy sum to start the process).

Given this setup, for each element in `nums`:

- We calculate the current cumulative sum (`s`) by adding the current element to it.
- We update the answer (`ans`) by adding the count of how many times `s - k` has occurred, because each occurrence represents a potential subarray that sums to `k`.
- Finally, we increment the count of the current cumulative sum in our hashmap, to be used for subsequent elements.

This algorithm effectively uses the prefix sum concept along with the hashmap to check for the existence of a required sum in constant time, leading to an efficient solution with a linear time complexity, O(n).

Solution Approach

The solution uses a hashmap (in Python, a `Counter`) to keep track of the cumulative sums at each index and how many times each sum occurs. It follows these steps:

1. Initialize the `Counter` with `{0: 1}` - This implies that there is one subarray (`[]`) that sums up to `0` before we start processing the array.
2. As we iterate over the array `nums`:
   - Update the running total `s` by adding the current element. This is done by `s += num`.
   - Determine if `s - k` has been seen before. The count of `s - k` in the `counter` tells us how many subarrays (ending right before the current index) have a sum that would complement the remainder of the current sum to reach `k`. This is done by `ans += counter[s - k]`.
   - Update the `counter` with the new cumulative sum. Increment the existing value of `counter[s]` by `1`, or set it to `1` if `s` is not yet a key in the counter. This is done by `counter[s] += 1`.
3. After the loop ends, the value held in `ans` is the total number of contiguous subarrays that add up to `k`.

The use of the running total (or cumulative sum) allows us to check for the existence of any subarray ending at the current index which sums to `k`, by checking if there is a sum of `s - k` earlier in the array. The `Counter` is essential here as it lets us track and retrieve the number of occurrences of each sum in constant time, which keeps the overall time complexity linear, O(n).

This method is efficient both in terms of time and space. There is no need for nested loops (which would result in O(n^2) time complexity), and we only need extra space for the hash table which stores at most `n` key-value pairs where `n` is the number of elements in `nums`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Consider the input `nums = [3, 4, -1, 2, 1]` and `k = 5`. We want to find all contiguous subarrays summing to `k`, which is `5` in this case.

Following the solution approach step by step:

1. We initialize our `Counter` with `{0: 1}` to account for the base case where a subarray can start from the beginning of the array.
2. Now, we iterate over the array `nums`. We'll keep track of our cumulative sum (`s`) and initialize `ans` to `0`.
   - Start with the first element: `s = 3`. No previous sum of `s - k = -2` exists, so don't update `ans`. Then, update the counter to `Counter = {0: 1, 3: 1}`.
   - Move to the second element: `s = 3 + 4 = 7`. We've seen `s - k = 2` before? No, so `ans` remains the same. Update the counter to `Counter = {0: 1, 3: 1, 7: 1}`.
   - Third element: `s = 7 - 1 = 6`. We check for `s - k = 1`. It does not exist. Update the counter to `Counter = {0: 1, 3: 1, 7: 1, 6: 1}`.
   - Fourth element: `s = 6 + 2 = 8`. Look for `s - k = 3`. We have seen `3` before, once. Increase `ans` by `1` as there is one subarray ending here that sums to `5`. Update the counter to `Counter = {0: 1, 3: 1, 7: 1, 6: 1, 8: 1}`.
   - Fifth element: `s = 8 + 1 = 9`. Check for `s - k = 4`. We have not seen `4` before, so `ans` remains `1`. Update the counter to `Counter = {0: 1, 3: 1, 7: 1, 6: 1, 8: 1, 9: 1}`.
3. After iterating through all elements, our `ans` value, which is `1`, represents the total number of contiguous subarrays that sum up to `5`. The subarray in this example is `[4, -1, 2]`.

By using a cumulative sum and a hashmap (`Counter`), we accurately and efficiently found all the contiguous subarrays that add up to a given sum with a single pass through the array. This method avoids the use of nested loops for a better time complexity of O(n).

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        # Initialize a counter to keep track of the cumulative sums encountered
        cumulative_sum_counter = Counter({0: 1})

        # 'count_subarrays' will store the total count of subarrays that sum up to 'k'
        count_subarrays = 0

        # 'cumulative_sum' holds the sum of numbers seen so far
        cumulative_sum = 0

        # Iterate through the list of numbers
        for num in nums:
            # Update the cumulative sum
            cumulative_sum += num

            # If there is a previous cumulative sum such that current_sum - k
            # is equal to that previous sum, then a subarray ending at the current
            # position would sum to 'k'
            count_subarrays += cumulative_sum_counter[cumulative_sum - k]

            # Increase the count of the current cumulative sum by 1 in the counter
            cumulative_sum_counter[cumulative_sum] += 1

        # Return the total number of subarrays that sum up to 'k'
        return count_subarrays
```

Java

```java
class Solution {

    public int subarraySum(int[] nums, int k) {
        // Map for storing the cumulative sum and its frequency.
        Map<Integer, Integer> sumFrequencyMap = new HashMap<>();

        // Initializing with zero sum having frequency one.
        sumFrequencyMap.put(0, 1);

        int totalCount = 0; // This will hold the number of subarrays that sum to k.
        int cumulativeSum = 0; // This holds the cumulative sum of elements.

        // Loop over all elements in the array.
        for (int num : nums) {
            // Add the current element to the cumulative sum.
            cumulativeSum += num;

            // If cumulativeSum - k exists in map, then there are some subarrays ending with num that sum to k.
            totalCount += sumFrequencyMap.getOrDefault(cumulativeSum - k, 0);

            // Increment the frequency of the current cumulative sum in the map.
            // If the cumulative sum isn't already in the map, DefaultValue (0) will be used first.
            sumFrequencyMap.put(cumulativeSum, sumFrequencyMap.getOrDefault(cumulativeSum, 0) + 1);
        }

        // Return the total count of subarrays that sum to k.
        return totalCount;
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // This function returns the number of subarrays that sum up to k.
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> prefixSumFrequency;
        prefixSumFrequency[0] = 1; // Base case: there's one way to have a sum of 0 (no elements).

        int answer = 0;  // Variable to store the number of subarrays that sum to k.
        int cumulativeSum = 0; // Variable to store the cumulative sum of elements.

        // Iterate through the array to calculate the cumulative sum and count subarrays.
        for (int num : nums) {
            cumulativeSum += num; // Update the cumulative sum.
            // If cumulativeSum - k exists in prefixSumFrequency, then a subarray ending at current
            // index has a sum of k. We add the count of those occurrences to answer.
            answer += prefixSumFrequency[cumulativeSum - k];
            // We then increment the count of cumulativeSum in our frequency map.
            prefixSumFrequency[cumulativeSum]++;
        }

        // Return the total count of subarrays that sum up to k.
        return answer;
    }
};
```

TypeScript

```typescript
function subarraySum(nums: number[], targetSum: number): number {
    let totalCount = 0;                  // This will hold the final count of subarrays
    let currentSum = 0;                  // This will store the cumulative sum of elements
    const sumFrequency = new Map();      // This map will store the frequency of sums encountered

    sumFrequency.set(0, 1);              // Initialize map with a zero sum having one occurrence

    // Loop through each number in the input array
    for (const num of nums) {
        currentSum += num;               // Add current number to the cumulative sum

        // If (currentSum - targetSum) is a sum we've seen before, it means there is a subarray
        // which adds up to the targetSum. We add to totalCount the number of times we've seen this sum.
        totalCount += sumFrequency.get(currentSum - targetSum) || 0;

        // Update the frequency of the currentSum in the map.
        // If currentSum is already present in the map, increment its count, otherwise initialize it to 1.
        sumFrequency.set(currentSum, (sumFrequency.get(currentSum) || 0) + 1);
    }

    return totalCount; // Return the total count of subarrays which sum up to targetSum
}
```


```python
from collections import Counter

class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        # Initialize a counter to keep track of the cumulative sums encountered
        cumulative_sum_counter = Counter({0: 1})

        # 'count_subarrays' will store the total count of subarrays that sum up to 'k'
        count_subarrays = 0

        # 'cumulative_sum' holds the sum of numbers seen so far
        cumulative_sum = 0

        # Iterate through the list of numbers
        for num in nums:
            # Update the cumulative sum
            cumulative_sum += num

            # If there is a previous cumulative sum such that current_sum - k
            # is equal to that previous sum, then a subarray ending at the current
            # position would sum to 'k'
            count_subarrays += cumulative_sum_counter[cumulative_sum - k]

            # Increase the count of the current cumulative sum by 1 in the counter
            cumulative_sum_counter[cumulative_sum] += 1

        # Return the total number of subarrays that sum up to 'k'
        return count_subarrays
```


Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the input list `nums`. This is because the code iterates through the list once, performing a constant number of operations for each element: calculating the cumulative sum `s`, checking and updating the `counter`, and incrementing `ans`.

The space complexity of the code is also `O(n)`, due to the `counter` object that can potentially store an entry for each unique cumulative sum `s`. In the worst case, this could be every subarray sum if all numbers in `nums` are distinct and add up to different sums, creating a new key for almost each `s`.