930. Binary Subarrays With Sum

Medium | Array | Hash Table | Prefix Sum | Sliding Window

Problem Description

Given a binary array, which consists of only 0's and 1's, and a target integer `goal`, the task is to find the number of subarrays whose sum equals `goal`. A subarray is defined as any continuous sequence of the array. The binary nature of the array means any sum of its subarray will also be an integer, making the problem about finding subarray sums that match the given integer.

Intuition

The solution employs a two-pointer approach that creates a [sliding window] of variable size over the array. Two pointers, `i1` and `i2`, are moved along the array to adjust the window size. Pointer `j` is used to extend the window by moving right. The variables `s1` and `s2` are used to track the sums of numbers within the window.

- We use two sliding windows: one to find the number of subarrays with sum just over `goal` (`s1` and `i1`), and another to find the number of subarrays with sum equal to or just over `goal` (`s2` and `i2`).
- As we move the main pointer `j` through the array, we increase `s1` and `s2` by the new element included in the window `nums[j]`.
- Next, we need to shrink the windows if necessary. If `s1` is greater than `goal`, we move `i1` to reduce the window and the sum. Similarly, if `s2` is greater or equal than `goal`, we move `i2`.
- The difference between `i2` and `i1` gives us the number of new subarrays that have a sum equal to `goal` considering the new element `nums[j]`.
- We increment `ans` by this difference since `i2 - i1` represents the number of valid subarrays ending at `j` with sum exactly `goal`.
- We repeat this process for every element of the array by incrementing `j`, thus exploring every potential subarray.

Why does this work? The two sliding windows track the lower and upper bounds of sums around our `goal`. By taking the difference of the counts, we effectively count only those subarrays whose sum is exactly `goal`. Since we consider each subarray ending at various `j` positions, we can ensure that all possible subarrays are counted.

With every increment of `j`, we essentially say, "Given all the subarrays ending at `j`, count how many have a sum of `goal`." The sliding of `i1` and `i2` keeps the window's sum around `goal` and accurately maintains the count.

This approach is efficient since it requires only a single pass through the array, leading to a time complexity of O(n), where n is the length of the input array `nums`.

Solution Approach

The solution to the subarray sum problem implements a variation of the two-pointer technique which helps to optimize the search for subarrays that add up to the `goal`. Here's a step-by-step explanation:

1. Initialize two pairs of pointers `i1`, `i2` and their corresponding sums `s1`, `s2` to `0`. These will be used to manage two sliding windows. Also, initialize a pointer `j` to extend the window and `ans` to accumulate the number of valid subarrays found.
2. Iterate over the array with the main pointer `j`. For each element `nums[j]` being considered:
   - Add `nums[j]` to both `s1` and `s2`, which represents attempting to add the current element to our current subarray sums.
3. If `s1` exceeds the `goal`, shrink the first window:
   - Subtract `nums[i1]` from `s1` and increment `i1` to reduce the window from the left, doing this until `s1` is no longer greater than `goal`.
4. Similarly, if `s2` is at least the `goal`, shrink the second window:
   - Subtract `nums[i2]` from `s2` and increment `i2` to reduce the window from the left, doing this until `s2` is smaller than `goal`.
5. After adjusting both windows, calculate the number of new subarrays found:
   - The difference `i2 - i1` tells us how many valid subarrays end at `j` with the desired sum `goal` because those will be the subarrays contained within the window tracked by `i2` but not yet by `i1`.
   - Add this difference to `ans` which tallies our result.
6. Repeat steps 2 through 5 as you move the pointer `j` to the right until you've processed every element in the array.
7. Once the iteration is complete, `ans` holds the final count of subarrays whose sum is equal to `goal`.

This code efficiently finds all subarrays with the desired sum in linear time, utilizing `O(1)` extra space, excluding the input and output. The key ingredients of the solution are the two-pointer technique and a single pass, avoiding unnecessary recomputation.

No complex data structures are needed because the pointers and sums effectively manage the windows of potential subarrays. The algorithm iteratively adjusts these windows to find all valid subarrays in the most optimized manner.

Example Walkthrough

Let's consider the binary array `nums = [1, 0, 1, 0, 1]` and the target integer `goal = 2`. We need to find the number of subarrays with the sum equal to `goal`.

1. We initialize pointers and sums: `i1 = i2 = 0`, `s1 = s2 = 0`, and `ans = 0`.
2. Start iterating with pointer `j` from left to right. The main goal is to add `nums[j]` to both `s1` and `s2` and then adjust the windows with pointers `i1` and `i2`.

Using the array provided, here's how the solution progresses:

- For `j = 0` (the first element is `1`):
  - `s1 = s2 = 1`. No window adjustment needed since neither `s1` nor `s2` is over the `goal` yet.
- For `j = 1` (the second element is `0`):
  - `s1 = s2 = 1`. Still no adjustment as we are not over the `goal`.
- For `j = 2` (the third element is `1`):
  - `s1 = s2 = 2`. Since `s2 >= goal`, we increment `i2`. Now, `i2 = 1` and `s2 = 1` (`s1` remains `2` because we have to exceed `goal` to adjust `i1`).
  - `i2 - i1 = 1`. We found one valid subarray [1, 0, 1] and increment `ans` by 1.
- For `j = 3` (the fourth element is `0`):
  - `s1 = 2` and `s2 = 1`. No adjustment required.
- For `j = 4` (the fifth element is `1`):
  - `s1 = s2 = 2`. Both sums are equal to the `goal`.
  - Since `s1 == goal`, we increment `i1`. Now, `i1 = 1` and `s1 = 1`.
  - Similarly, `i2` also increments because `s2 >= goal`. Now, `i2 = 3` and `s2 = 0`.
  - `i2 - i1 = 2`. The two new valid subarrays are [1, 0, 1] and [0, 1] ending at `j=4`. So we increment `ans` by 2.

After iterating through the array, we've found `ans = 3` valid subarrays ([1, 0, 1], [1, 0, 1], and [0, 1]) where the sum matches the `goal`.

It's important to understand that the same subarray may be counted at different stages depending on the `j` position. This method ensures that every unique subarray is considered without double counting, and the result is achieved with a single traversal.

Solution Implementation

Python

```python
class Solution:
    def numSubarraysWithSum(self, nums: List[int], goal: int) -> int:
        left1 = left2 = sum1 = sum2 = idx = total_subarrays = 0
        array_length = len(nums)

        # Iterate over the array to find subarrays with sum equal to goal
        while idx < array_length:
            # Increase running sums with the current number
            sum1 += nums[idx]
            sum2 += nums[idx]

            # Decrease sum1 until it's no more than goal by moving left1 pointer right
            while left1 <= idx and sum1 > goal:
                sum1 -= nums[left1]
                left1 += 1

            # Decrease sum2 until it's just less than goal by moving left2 pointer right
            while left2 <= idx and sum2 >= goal:
                sum2 -= nums[left2]
                left2 += 1

            # Add the number of new subarrays found to the total
            total_subarrays += left2 - left1

            # Move to the next element
            idx += 1

        return total_subarrays
```

Java

```java
class Solution {
    // Method to count the number of subarrays with a sum equal to the given goal.
    public int numSubarraysWithSum(int[] nums, int goal) {
        int left1 = 0, left2 = 0, sum1 = 0, sum2 = 0, right = 0, result = 0;
        int n = nums.length;

        // Iterate over the elements of the array using 'right' as the right end of the subarray.
        while (right < n) {
            // Increase sums by the current element.
            sum1 += nums[right];
            sum2 += nums[right];

            // Shrink the window from the left (left1) until the sum (sum1) is not greater than the goal.
            while (left1 <= right && sum1 > goal) {
                sum1 -= nums[left1++];
            }

            // Shrink the window from the left (left2) until the sum (sum2) is not greater than or equal to the goal.
            while (left2 <= right && sum2 >= goal) {
                sum2 -= nums[left2++];
            }

            // The window between left2 and left1 contains all the starting points for subarrays ending at 'right'
            // with sums that are exactly equal to the goal.
            result += left2 - left1;

            // Move to the next element.
            ++right;
        }

        // Return total count of subarrays with a sum equal to the goal.
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    int numSubarraysWithSum(vector<int>& nums, int goal) {
        int startIndexForStrictlyGreater = 0; // Start index for subarrays with sum strictly greater than goal
        int startIndexForAtLeastGoal = 0; // Start index for subarrays with sum at least as much as goal
        int sumForStrictlyGreater = 0; // Current sum for the subarrays which is strictly greater than goal
        int sumForAtLeastGoal = 0; // Current sum for the subarrays which is at least as much as goal
        int endIndex = 0; // Current end index of the subarray
        int countSubarrays = 0; // Count of subarrays with sum exactly equals to goal
        int numSize = nums.size(); // Size of the input array

        // Iterate over each element in nums to find subarrays with sum equal to goal
        while (endIndex < numSize) {
            sumForStrictlyGreater += nums[endIndex]; // Increment sum by current element for strictly greater sum
            sumForAtLeastGoal += nums[endIndex]; // Increment sum by current element for at least goal sum

            // Move startIndexForStrictlyGreater till the sum is strictly greater than the goal
            while (startIndexForStrictlyGreater <= endIndex && sumForStrictlyGreater > goal) {
                sumForStrictlyGreater -= nums[startIndexForStrictlyGreater++];
            }

            // Move startIndexForAtLeastGoal till the sum is at least as much as the goal
            while (startIndexForAtLeastGoal <= endIndex && sumForAtLeastGoal >= goal) {
                sumForAtLeastGoal -= nums[startIndexForAtLeastGoal++];
            }

            // The number of subarrays which sum up to the goal equals to the difference of indices
            // This gives us the count of all the subarrays between the two starts
            countSubarrays += startIndexForAtLeastGoal - startIndexForStrictlyGreater;

            // Move to the next element
            ++endIndex;
        }
        return countSubarrays; // Return the total count of subarrays with sum equal to goal
    }
};
```

TypeScript

```typescript
/**
 * Counts the number of subarrays with a sum equal to the given goal.
 *
 * @param {number[]} nums - The array of numbers to search within.
 * @param {number} goal - The target sum for the subarrays.
 * @return {number} The number of subarrays whose sum equals the goal.
 */
const numSubarraysWithSum = (nums: number[], goal: number): number => {
    let leftIndexForStrict = 0, // Left index for the subarray where the sum is strictly more than the goal.
        leftIndexForLoose = 0, // Left index for the subarray where the sum is at least the goal.
        strictSum = 0, // Sum of the current subarray for the strict condition.
        looseSum = 0, // Sum of the current subarray for the loose condition.
        rightIndex = 0, // Right index of the subarray currently being considered.
        count = 0; // The count of valid subarrays.
    const length = nums.length;

    // Traverse through the array using the right index.
    while (rightIndex < length) {
        // Add the current element to both the strict and loose sum.
        strictSum += nums[rightIndex];
        looseSum += nums[rightIndex];

        // If the strict sum is greater than the goal, move the left index to reduce the sum.
        while (leftIndexForStrict <= rightIndex && strictSum > goal) {
            strictSum -= nums[leftIndexForStrict++];
        }

        // If the loose sum is at least the goal, move the left index to find the next valid subarray start.
        while (leftIndexForLoose <= rightIndex && looseSum >= goal) {
            looseSum -= nums[leftIndexForLoose++];
        }

        // The difference between leftIndexForLoose and leftIndexForStrict gives the count of subarrays where the sum equals the
        count += leftIndexForLoose - leftIndexForStrict;

        // Move to the next element in the array.
        ++rightIndex;
    }

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

// This function can now be called with TypeScript's type checking.
```

```python
class Solution:
    def numSubarraysWithSum(self, nums: List[int], goal: int) -> int:
        left1 = left2 = sum1 = sum2 = idx = total_subarrays = 0
        array_length = len(nums)

        # Iterate over the array to find subarrays with sum equal to goal
        while idx < array_length:
            # Increase running sums with the current number
            sum1 += nums[idx]
            sum2 += nums[idx]

            # Decrease sum1 until it's no more than goal by moving left1 pointer right
            while left1 <= idx and sum1 > goal:
                sum1 -= nums[left1]
                left1 += 1

            # Decrease sum2 until it's just less than goal by moving left2 pointer right
            while left2 <= idx and sum2 >= goal:
                sum2 -= nums[left2]
                left2 += 1

            # Add the number of new subarrays found to the total
            total_subarrays += left2 - left1

            # Move to the next element
            idx += 1

        return total_subarrays
```

Time and Space Complexity

The code provided uses two pointers technique to keep track of subarrays with sums equal to or just above the goal. The time complexity of the code is `O(n)` because the while loop runs for each element in the array (`n` elements), and inside the loop, each pointer (i1 and i2) moves forward (never backwards), meaning each element is processed once.

The space complexity of the code is `O(1)` as the space used does not grow with the input size `n`. The variables `i1`, `i2`, `s1`, `s2`, `j`, `ans`, and `n` use a constant amount of space.