2875. Minimum Size Subarray in Infinite Array

Medium Array Hash Table Prefix Sum Sliding Window Leetcode Link

Problem Description

You are given an array called `nums`, which starts from the index `0`. There is also an integer called `target`. There's a hypothetical array called `infinite_nums`, which is made by continuously repeating the `nums` array infinitely. Your task is to find the shortest contiguous subarray within `infinite_nums` that adds up to the `target` value. The length of this subarray (the number of elements it includes) is what you need to return. If such a subarray can't be found, you must return `-1`.

For example, if `nums` is `[1,2,3]` and the `target` is `5`, you can use the subarray `[2,3]` from `nums` or `[3,1,1]` from `infinite_nums`, both sum up to `5` but the shortest length is `2` from the `[2,3]` subarray in `nums`.

Intuition

The intuition behind the solution is to use mathematical properties and an efficient data structure to avoid direct iteration through the infinite array, which would be impractical.

First, calculate the sum of all elements in `nums`, which is referred to as `s`. Compare `s` with the `target`. If the `target` is larger than `s`, then you can construct part of the required subarray using whole chunks of `nums`, each of which contributes `s` to the overall sum. This means you can repeatedly subtract `s` from `target` until `target` is less than `s`, and count how many times you've done this as part of the subarray length.

Then, the problem is reduced to two cases within the original `nums` array:

1. Find the shortest subarray that sums up exactly to `target`.
2. If it's not possible (because `target` is now less than `s` after subtraction), find two subarrays where one is at the start and one is at the end of `nums` such that their total sum equals `s - target`. This effectively simulates wrapping around due to the infinite nature of `infinite_nums`.

To achieve this efficiently, we use a prefix sum array and a hash table. The prefix sum array allows us to calculate the sum of any subarray quickly, and the hash table lets us lookup whether a needed sum to reach the `target` has been seen before as we iterate through `nums`. When we find such sums, we can calculate the length of the subarray that reaches the `target`.

With the above strategy, we ensure that we can get our answer without explicitly dealing with the infinite array, all while maintaining good time complexity.

Solution Approach

The given solution approach leverages a combination of prefix sums and hash tables to solve the problem with efficiency.

Here's a step-by-step breakdown of the implementation:

1. Compute the sum `s` of the entire `nums` array. Then check if `target` is greater than `s`. If yes, you can cover a significant portion of the `target` by using whole arrays of `nums`. Calculate `a = n * (target // s)`, which denotes the array length contributed by the complete `nums` blocks. After this, subtract `(target // s * s)` from `target`, reducing the problem to finding a shorter subarray that sums up to the new `target`. If the new `target` matches `s`, return `n` since it is the shortest subarray that can be formed by the whole array.

2. Initialize a hash table `pos` to keep track of the last position of every prefix sum encountered. Set the prefix sum for an empty array to `-1` to handle cases where the subarray starts at the beginning of `nums`.

3. Initialize a variable `pre` to hold the ongoing prefix sum as we iterate through the array and a variable `b` as `inf` (infinity) which will later hold the length of the shortest qualifying subarray found.

4. Iterate through the `nums` array, updating the `pre` (prefix sum) by adding each element `x` from `nums` to it.

5. Two main checks are performed for each iteration:

   - If the difference `t = pre - target` exists in `pos`, it means a subarray sums up to `target`. Update `b` to be the minimum of its current value or the distance from the current index to the index stored in `pos[t]`.
   - If the difference `t = pre - (s - target)` exists in `pos`, it implies that there's a contiguous subarray from the start and end of `nums` which together add up to `target`. Again, update `b` to be the minimum of its current value or `n - (i - pos[t])`.
6. After the loop, check if `b` has changed from `inf`. If it has, this means a valid subarray was found, and the solution returns `a + b`. Otherwise, return `-1` indicating that a subarray satisfying the conditions does not exist.

This approach uses a hash table to store prefix sums and their corresponding last index. This data structure combined with the prefix sum concept allows us to find contiguous subarrays that add up to a target effectively. By doing this in a single pass, the algorithm avoids the need for nested loops, resulting in an efficient solution.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose the given `nums` array is `[3, 1, 2]`, and the `target` is `6`. We want to find the shortest contiguous subarray within `infinite_nums` that sums up to `6`.

Following the solution approach:

1. First, we compute the sum `s` of the entire `nums` array, which in this case is `3 + 1 + 2 = 6`. Since `target` is equal to `s`, we can directly return the length of `nums`, which is `3`, as the shortest subarray because using the whole `nums` array once sums to the target. However, for sake of walkthrough, let's proceed as if we were to look for a subarray since often the `target` might not exactly match `s`.

2. Initialize a hash table `pos` with `-1` as the prefix sum for an empty array, and a variable `pre` for ongoing prefix sums starting at `0`. Also, initialize `b` as `inf` which indicates the shortest found subarray (to be minimized).

3. As we iterate, we build our prefix sum array on the fly, and update `pos` with the last position we encountered a specific prefix sum.

4. We start iterating `nums`:

   - At index `0`, `pre` becomes `3`. `pos` does not have `pre - target = 3 - 6 = -3`, so we continue.
   - At index `1`, `pre` becomes `4`. `pos` does not have `pre - target = 4 - 6 = -2`, so we continue.
   - At index `2`, `pre` becomes `6`. `pos` does have `pre - target = 6 - 6 = 0`, which is the prefix sum for an empty array, so we find that subarray `[3, 1, 2]` sums to `target`. We update `b` to current index (`2`) plus one minus `-1` (position of 0 in `pos`), which equals `3`.
5. After the iteration, `b` is no longer `inf`; it's `3`. So, we can directly return the length `3`, indicating that `[3, 1, 2]` is the shortest subarray that sums up to the `target`.

In this case, iteration reveals that using the entire array `nums` itself is the shortest subarray to reach the `target` of `6`. In scenarios where `target` is not equal to `s`, the method would identify the shortest contiguous subarray as per the steps described in the solution approach. If no such subarray exists, `-1` would be returned.

Python Solution

```python
from math import inf

class Solution:
    def minSizeSubarray(self, nums: List[int], target: int) -> int:
        total_sum = sum(nums)  # Calculate the total sum of all numbers in the array
        num_count = len(nums)  # Get the length of the array

        # Initialize an 'a' which is the number of complete array rotations required
        num_full_rotations = 0
        if target > total_sum:
            num_full_rotations = (target // total_sum) * num_count
            target -= (target // total_sum) * total_sum  # Update the target after the full rotations

        # If target equals total_sum, return the array length
        if target == total_sum:
            return num_count

        pos = {0: -1}  # Create a dictionary to store the prefix sum indices
        prefix_sum = 0  # Initialize the prefix sum
        min_length = inf  # Set initial min length to infinity

        # Traverse through the array
        for i, num in enumerate(nums):
            prefix_sum += num  # Update prefix sum

            # Check if there's a subarray that ends at index i with sum equals target
            if (t := prefix_sum - target) in pos:
                min_length = min(min_length, i - pos[t])

            # Check if there is a circular subarray that sums to target
            if (t := prefix_sum - (total_sum - target)) in pos:
                min_length = min(min_length, num_count - (i - pos[t]))

            # Store the latest index of this prefix sum
            pos[prefix_sum] = i

        # If min_length is still infinity, no subarray was found. Return -1.
        # Otherwise, return the answer which includes the rotations 'a' + the found subarray length 'b'
        return -1 if min_length == inf else num_full_rotations + min_length
```

Java Solution

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

class Solution {

    // Function to find the smallest subarray sum greater than or equal to the target
    public int minSizeSubarray(int[] nums, int target) {
        // Compute the sum of all elements in the array
        long totalSum = Arrays.stream(nums).sum();
        int length = nums.length;
        int additionalElements = 0;

        // If target is greater than the total sum, scale the number of times the whole array is needed
        if (target > totalSum) {
            additionalElements = length * (target / (int) totalSum);
            target -= target / totalSum * totalSum;
        }

        // If the target is now equal to the total sum, simply return the length of the array
        if (target == totalSum) {
            return length;
        }

        // Create a map to store the prefix sum and its corresponding index
        Map<Long, Integer> prefixSumToIndex = new HashMap<>();
        prefixSumToIndex.put(0L, -1); // Initialize with 0 sum at index -1
        long currentPrefixSum = 0;
        int minSize = Integer.MAX_VALUE; // Start with the maximum possible value

        // Iterate through the array to find the minimum size subarray
        for (int i = 0; i < length; i++) {
            currentPrefixSum += nums[i];

            // If the prefix sum indicating the end of subarray achieving the target is seen before
            if (prefixSumToIndex.containsKey(currentPrefixSum - target)) {
                minSize = Math.min(minSize, i - prefixSumToIndex.get(currentPrefixSum - target));
            }

            // Check if there's a complement subarray sum that together with the currentPrefixSum gives totalSum
            if (prefixSumToIndex.containsKey(currentPrefixSum - (totalSum - target))) {
                minSize = Math.min(minSize, length - (i - prefixSumToIndex.get(currentPrefixSum - (totalSum - target))));
            }

            // Update the map with the current prefix sum and index
            prefixSumToIndex.put(currentPrefixSum, i);
        }

        // If minSize is unchanged, no such subarray exists; return -1.
        // Otherwise, return the minimum size added by the number of additional elements needed
        return minSize == Integer.MAX_VALUE ? -1 : additionalElements + minSize;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    int minSizeSubarray(vector<int>& nums, int target) {
        // Calculate the total sum of the array
        long long totalSum = accumulate(nums.begin(), nums.end(), 0LL);
        int n = nums.size();

        // Calculate how many complete arrays are needed to reach close to the target
        int completeArraysCount = 0;
        if (target > totalSum) {
            completeArraysCount = n * (target / totalSum);
            target -= (target / totalSum) * totalSum;
        }

        // If target equals totalSum, a complete array is needed
        if (target == totalSum) {
            return n;
        }

        // Hash map to keep track of the prefix sum and its corresponding index
        unordered_map<int, int> prefixSumIndex{{0, -1}};

        long long prefixSum = 0; // Initialize prefix sum
        int minLength = 1 << 30; // Initialize the minimum length to a very large number

        // Iterate over the array to find the minimum subarray
        for (int i = 0; i < n; ++i) {
            prefixSum += nums[i]; // Update the prefix sum

            // Check if there is a subarray with sum equal to (prefixSum - target)
            if (prefixSumIndex.count(prefixSum - target)) {
                minLength = min(minLength, i - prefixSumIndex[prefixSum - target]);
            }

            // Check if there is a subarray that, when added to the current subarray, gives the complement to the target
            if (prefixSumIndex.count(prefixSum - (totalSum - target))) {
                minLength = min(minLength, n - (i - prefixSumIndex[prefixSum - (totalSum - target)]));
            }

            // Update the index for the current prefix sum
            prefixSumIndex[prefixSum] = i;
        }

        // If minLength has not been updated, return -1, as no valid subarray exists.
        // Otherwise, return the minimum length found plus the count of complete arrays needed.
        return minLength == 1 << 30 ? -1 : completeArraysCount + minLength;
    }
};
```

Typescript Solution

```typescript
// This function finds the minimal length of a contiguous subarray of which the sum is at least the target value.
// If there is no such subarray, the function returns -1.
function minSizeSubarray(nums: number[], target: number): number {
    // Calculate the sum of the entire array.
    const totalSum = nums.reduce((acc, num) => acc + num);

    // Initial preparations: calculation of repetitions of the full array sum to reach the target.
    let repeatedFullArrayCount = 0;
    if (target > totalSum) {
        // Calculate how many times we can fit the total sum into the target sum.
        repeatedFullArrayCount = Math.floor(target / totalSum);
        // Decrease the target by the amount already covered by the full array repetitions.
        target -= repeatedFullArrayCount * totalSum;
    }
    // If after the subtraction the target equals the total sum, we can return the array length.
    if (target === totalSum) {
        return nums.length;
    }

    // Map to store the prefix sum and its corresponding index.
    const prefixSums = new Map<number, number>();
    prefixSums.set(0, -1); // Initialize with prefix sum of 0 and index -1.

    // Variables for running prefix sum and the minimum size of the subarray found.
    let runningSum = 0;
    let minSubarraySize = Infinity;

    // Iterate over the array to find the minimum length subarray.
    for (let i = 0; i < nums.length; ++i) {
        runningSum += nums[i];

        // Check if the current prefix sum, minus the target, already exists.
        // If it does, we possibly found a smaller subarray.
        if (prefixSums.has(runningSum - target)) {
            const prevIndex = prefixSums.get(runningSum - target)!;
            minSubarraySize = Math.min(minSubarraySize, i - prevIndex);
        }

        // Similar check as above, accounting for the case where sum of nums minus target exists as a prefix sum.
        // This helps in covering scenarios where the sum cycles through the array.
        if (prefixSums.has(runningSum - (totalSum - target))) {
            const prevIndex = prefixSums.get(runningSum - (totalSum - target))!;
            minSubarraySize = Math.min(minSubarraySize, nums.length - (i - prevIndex));
        }

        // Update the prefix sum map with the current running sum and index.
        prefixSums.set(runningSum, i);
    }

    // If no subarray was found that adds up to the target, return -1.
    // Otherwise, return the count of full array repetitions plus the found subarray size.
    return minSubarraySize === Infinity ? -1 : repeatedFullArrayCount + minSubarraySize;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function is `O(n)` where `n` is the length of the array `nums`. This is because there is a single loop that goes through each element of `nums` exactly once. The operations within the loop have constant time complexity, such as the calculation of the running sum (`pre`), checking for the existence of a value in the hash table (`pos`), and updating the hash table. As a result, these constant-time operations do not change the overall linear time complexity.

Space Complexity

The space complexity of the function is also `O(n)`, which comes from the use of a hash table `pos` that keeps track of the indices of the prefix sums. In the worst case, if all prefix sums are unique, the hash table will have as many entries as there are elements in `nums`. Therefore, the space used by the hash table is directly proportional to the size of the input array, leading to linear space complexity.