209. Minimum Size Subarray Sum

Medium | Array | Binary Search | Prefix Sum | Sliding Window

Problem Description

The problem is essentially about finding the shortest continuous sequence of elements within an array of positive integers such that the sum of that sequence is at least as large as a given target value. To clarify, a subarray is defined as a contiguous part of an array. The task is to find the length of the smallest subarray that meets or exceeds the target sum. If no such subarray exists, the function should return zero.

For example, if the input array is `[2, 3, 1, 2, 4, 3]` and the target is 7, the subarray `[4, 3]` has the smallest length that sums up to 7 or more, so the answer would be the length of this subarray, which is 2.

Intuition

To solve this problem efficiently, we need to avoid checking all possible subarrays one by one since doing so would result in a very slow solution when dealing with a large array. The intuitive insight here is that we can do much better by using a "sliding window" approach. This approach involves maintaining a window that expands and contracts as we iterate through the array to find the smallest window that satisfies the condition.

Here's the thought process behind the sliding window solution:

1. We start with two pointers, both at the beginning of the array. These pointers represent the margins of our current window.
2. We move the end pointer to the right, adding numbers to our current window's sum.
3. As soon as the window's sum becomes equal to or greater than the target, we attempt to shrink the window from the left to find smaller valid windows that still meet the sum criterion.
4. Each successful window gives us a potential answer (its size), we keep track of the minimum size found.
5. We continue this process until the end pointer reaches the end of the array, and there are no more subarrays to check.
6. If the minimum length of the window is never updated from the initial setting that is larger than the array length, it means no valid subarray has been found, and we should return 0.

By using this approach, we can ensure that we only traverse the array once, giving us an efficient solution with a time complexity of O(n), where n is the length of the input array.

Solution Approach

The provided solution uses the **Sliding Window** pattern to solve the problem efficiently. This approach is useful when you need to find a subarray that meets certain criteria, and the problem can be solved in linear time without the need to check every possible subarray individually.

Here's how the sliding window algorithm is implemented in the solution:

1. Initialize two pointers, `j` at `0` to represent the start of the window and `i` which will move through the array.
2. Maintain a running sum, `s`, of the values within the current window which starts at `0`.
3. Iterate over the array using `i` and continuously add the value of `nums[i]` to `s`.
4. Inside the loop, use a `while` loop to check if the current sum `s` is greater than or equal to the target. If it is, attempt to shrink the window from the left by:
   - Updating the minimum length of the valid window if necessary using `ans = min(ans, i - j + 1)`.
   - Subtracting `nums[j]` from the sum `s` since the left end of the window is moving to the right.
   - Incrementing `j` to actually move the window's start to the right.
5. Once the end of the array is reached and there are no more elements to add to the sum, check if `ans` was updated or not. If `ans` is still greater than the length of the array `n`, it means no valid subarray was found, so we return `0`.
6. If `ans` was updated during the process (meaning a valid subarray was found), return the value of `ans` which holds the length of the smallest subarray with a sum of at least `target`.

The use of two pointers to create a window that slides over the array allows this algorithm to run in `O(n)` time, making it very efficient for this type of problem.

Please note that the Reference Solution Approach mentions another method which is using **PreSum & Binary Search** but the provided code doesn't implement this method. The **PreSum & Binary Search** method involves creating an array of prefix sums and then using binary search to find the smallest valid subarray for each element. This is a bit more complex and generally not as efficient as the **Sliding Window** method used here which requires only `O(n)` time and `O(1)` extra space.

Example Walkthrough

Let's use a small example to illustrate the sliding window approach described in the solution. Suppose we have the array `[1, 2, 3, 4, 5]` and our target sum is 11.

We want to find the smallest subarray whose sum is at least 11. We'll follow the steps of the sliding window algorithm:

1. Initialize the pointers `i` and `j` both to `0`, and the running sum `s` also to `0`.
2. Start iterating over the array with `i`. Our window size is currently `0`.

First iteration (`i = 0`):

- Add `nums[i]` to `s`. Now, `s` is `1`.
- It's less than the target (11), so we move on to the next number.

Second iteration (`i = 1`):

- Now, `s` is `1 + 2 = 3`.
- Still less than the target.

Third iteration (`i = 2`):

- `s` becomes `1 + 2 + 3 = 6`.
- Again, we continue since `s` is less than our target sum.

Fourth iteration (`i = 3`):

- `s` is now `1 + 2 + 3 + 4 = 10`.
- It is still below the target sum of 11.

Fifth iteration (`i = 4`):

- `s` is now `1 + 2 + 3 + 4 + 5 = 15`, which is greater than our target of 11. We've now found a subarray `[1, 2, 3, 4, 5]` with the sum greater than or equal to the target.
- Now we try to shrink the window from the left to see if there is a smaller subarray that still meets or exceeds the target sum.
- Before moving `j`, we update the answer `ans` to the current window size, which is `i - j + 1 = 5 - 0 + 1 = 5`.

Now we enter the while loop to shrink the window since `s >= target`:

- We reduce `s` by `nums[j]`. `s` becomes `15 - 1 = 14`, and we increment `j` to `1`. The window is now `[2, 3, 4, 5]`.
- `s` is `14`, which is still greater than 11, so we repeat the procedure.
- `s` becomes `14 - 2 = 12` after removing the next element and `j` goes to `2`. The window is `[3, 4, 5]`.
- With `s` at `12`, it's still greater than 11, continue to shrink.
- We subtract `3` to get `s = 12 - 3 = 9` and move `j` to `3`. Now `s` is less than 11, so we stop shrinking the window.

Our smallest subarray that meets the requirement so far is `[3, 4, 5]` with a length of `3`. Since we've already reached the end of the array, we're done iterating, and we can return the answer, which is `3`.

This example successfully demonstrated the sliding window technique where we expanded the window until we exceeded the target sum, then shrank the window from the left to find the smallest subarray that still meets the sum condition. The array `[3, 4, 5]` is the smallest subarray with a sum greater than or equal to the target 11, so the result is the length of this subarray, which is `3`.

Solution Implementation

Python

```python
class Solution:
    def minSubarrayLength(self, target: int, nums: List[int]) -> int:
        # Initialize the length of the array
        length_of_nums = len(nums)
        # Set an initial answer value to a large number (beyond possible maximum)
        min_length = length_of_nums + 1
        # Initialize the sum of the current subarray and the start index j
        sum_of_subarray = start_index = 0
        # Loop over the elements in the array with their indices
        for end_index, value in enumerate(nums):
            # Add the current number to the sum of the current subarray
            sum_of_subarray += value
            # Shrink the subarray from the left (increase the start index)
            # until the sum is no longer greater or equal to the target
            while start_index < length_of_nums and sum_of_subarray >= target:
                # Update the minimum length if a shorter subarray is found
                min_length = min(min_length, end_index - start_index + 1)
                # Subtract the element at start_index from sum as we are excluding it from the subarray
                sum_of_subarray -= nums[start_index]
                start_index += 1
        # If min_length is updated, return it; otherwise, no such subarray exists and return 0
        return min_length if min_length <= length_of_nums else 0
```

Java

```java
class Solution {

    // This method finds the minimum length of a subarray that sums to at least the given target.
    public int minSubArrayLen(int target, int[] nums) {
        int n = nums.length; // The length of the input array.
        long sum = 0; // The sum of the current subarray.
        int minLength = n + 1; // Initialize minLength with max possible value plus one for comparison.

        // Two pointers method: i is the end-pointer, j is the start-pointer of the sliding window.
        for (int end = 0, start = 0; end < n; ++end) {
            sum += nums[end]; // Increment the sum by the current element value.

            // Shrink the window from the left until the sum is smaller than the target.
            // This finds the smallest window that ends at position 'end'.
            while (start < n && sum >= target) {
                minLength = Math.min(minLength, end - start + 1); // Update minLength if a smaller length is found.
                sum -= nums[start++]; // Decrease the sum by the start-value and increment start-pointer to shrink the window.
            }
        }

        // If minLength is updated (smaller than n + 1), we found a valid subarray.
        // Otherwise, return 0 as a subarray meeting the conditions is not found.
        return minLength <= n ? minLength : 0;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum length subarray sum
    // that is greater than or equal to the given target.
    int minSubArrayLen(int target, vector<int>& nums) {
        int n = nums.size();              // Size of the input array
        long long sum = 0;                // Long long to avoid overflow when summing up
        int ans = n + 1;                  // Initialize the answer to the max possible length+1 (invalid case scenario)

        // Two pointers, 'i' is the end of the current subarray
        // 'j' is the start of the current subarray
        for (int i = 0, j = 0; i < n; ++i) {
            sum += nums[i];               // Increase the sum by the current element

            // While sum is not smaller than the target and start pointer 'j' has not reached the end
            while (j < n && sum >= target) {
                ans = min(ans, i - j + 1); // Update the answer with the new minimum length
                sum -= nums[j++];          // Subtract the first element of the subarray and move 'j' right
            }
        }

        // If 'ans' didn't change, no valid subarray was found, return 0
        // Otherwise, return the length of the shortest subarray
        return ans == n + 1 ? 0 : ans;
    }
};
```

TypeScript

```typescript
function minSubArrayLen(target: number, nums: number[]): number {
    // Length of the input array
    const length = nums.length;
    // Initialize the sum of the current subarray
    let currentSum = 0;
    // Set an initial minimum length for the subarray to an impossible maximum (length+1)
    let minLength = length + 1;

    // Set the starting points for the sliding window
    for (let start = 0, end = 0; start < length; ++start) {
        // Add the current number to the current sum
        currentSum += nums[start];

        // While the current sum is equal or above the target,
        // adjust the window to find the minimum length
        while (currentSum >= target) {
            // Calculate the length of the current subarray
            // and update the minLength if it's smaller than the existing minLength
            minLength = Math.min(minLength, start - end + 1);

            // Subtract the first number of the current subarray
            // and move the window forward
            currentSum -= nums[end++];
        }
    }

    // If minLength was not updated, it means no valid subarray was found, so return 0
    // Otherwise, return the minLength found
    return minLength === length + 1 ? 0 : minLength;
}
```

```python
class Solution:
    def minSubarrayLength(self, target: int, nums: List[int]) -> int:
        # Initialize the length of the array
        length_of_nums = len(nums)
        # Set an initial answer value to a large number (beyond possible maximum)
        min_length = length_of_nums + 1
        # Initialize the sum of the current subarray and the start index j
        sum_of_subarray = start_index = 0
        # Loop over the elements in the array with their indices
        for end_index, value in enumerate(nums):
            # Add the current number to the sum of the current subarray
            sum_of_subarray += value
            # Shrink the subarray from the left (increase the start index)
            # until the sum is no longer greater or equal to the target
            while start_index < length_of_nums and sum_of_subarray >= target:
                # Update the minimum length if a shorter subarray is found
                min_length = min(min_length, end_index - start_index + 1)
                # Subtract the element at start_index from sum as we are excluding it from the subarray
                sum_of_subarray -= nums[start_index]
                start_index += 1
        # If min_length is updated, return it; otherwise, no such subarray exists and return 0
        return min_length if min_length <= length_of_nums else 0
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. This is because there are two pointers `i` and `j`, both of which travel across the list at most once. The inner `while` loop only increases `j` and decreases the sum `s` until the sum is less than the `target`, but `j` can never be increased more than `n` times throughout the execution of the algorithm. Therefore, each element is processed at most twice, once when it is added to `s` and once when it is subtracted, leading to a linear time complexity.

The space complexity of the code is `O(1)`, which means it requires a constant amount of additional space. This is because the algorithm only uses a fixed number of single-value variables (`n`, `ans`, `s`, `j`, `i`, `x`) and does not utilize any data structures that grow with the size of the input.