Problem: Minimum Size Subarray Sum

Problem Statement:

- You are given an array of positive integers nums and a positive integer target. Your task is to find the minimal length of a contiguous subarray whose sum is greater than or equal to the given target. If no such subarray exists, return 0.

Input:

- **target:** A positive integer representing the minimum required sum.
- **nums:** An array of positive integers representing the elements of the subarray.

Output:

- Return an integer representing the length of the smallest subarray whose sum is greater than or equal to the target. If no valid subarray is found, return 0.

Example 1:

- **Input:** target = 7, nums = [2, 3, 1, 2, 4, 3]
- **Output:** 2
- **Explanation:** The subarray [4, 3] has a sum of 7, which meets the target. It has a minimal length of 2.

Example 2:

- **Input:** target = 4, nums = [1, 4, 4]
- **Output:** 1
- **Explanation:** The subarray [4] has a sum of 4, which meets the target. It has a minimal length of 1.

Example 3:

- **Input:** target = 11, nums = [1, 1, 1, 1, 1, 1, 1, 1]
- **Output:** 0
- **Explanation:** No subarray has a sum greater than or equal to 11, so the output is 0.

Constraints:

- The length of the nums array can range from 1 to 10^5.
- Each element in the nums array is a positive integer between 1 and 10^4.
- The target can be as large as 10^9.

Approach:

1. **Sliding Window Technique (O(n) Solution):**

- This approach is based on the sliding window technique, where we expand and shrink the window of elements to find the subarray with the minimal length that meets the target sum.

Sliding Window Concept:

- **Expanding the window:** We incrementally add elements to the window (subarray) by moving the end pointer to the right.

- **Shrinking the window:** Once the sum of the current window becomes greater than or equal to the target, we attempt to shrink the window by moving the start pointer to the right, removing elements from the left side of the window while still maintaining a valid sum.

Key Steps:

1. **Initialize Pointers and Variables:**

- ***start:*** This pointer will mark the beginning of the window.
- ***end:*** This pointer will traverse the array, expanding the window.
- ***sum_:*** Keeps track of the sum of the elements in the current window.
- ***min_length:*** Stores the minimal length of the valid subarray. This is initially set to infinity to indicate no valid subarray has been found.

2. **Traversing the Array:**

- Use the end pointer to iterate over the nums array, adding each element to sum_ (the sum of the current window).

3. **Shrinking the Window:**

- While the sum of the current window is greater than or equal to the target, attempt to shrink the window by moving the start pointer to the right. This process continues until the window no longer meets the sum condition (sum_ >= target).
- For every valid window (where sum_ >= target), update the min_length by calculating the difference between end and start pointers. This gives the current length of the subarray.

4. **Edge Case Handling:**

- If no valid subarray is found, return 0.
- Otherwise, return the smallest value stored in min_length.

Example Walkthrough:

- *For the input target = 7 and nums = [2, 3, 1, 2, 4, 3]:*
  - Start by initializing the start, sum_, and min_length.
  - Traverse the array using the end pointer, updating the sum_ as you go.
  - Expand the window until the sum_ is greater than or equal to 7.
  - Once the valid window is found, update min_length and attempt to shrink the window by incrementing the start pointer.

Time Complexity:

- The sliding window technique ensures that each element is processed at most twice (once when expanding the window and once when shrinking it). Therefore, the time complexity is O(n), where n is the number of elements in the nums array.

Space Complexity:

- The algorithm uses a constant amount of extra space, hence the space complexity is O(1).

Follow-Up:

- **O(n log n) Approach:**
  - After implementing the O(n) sliding window solution, the follow-up challenge is to develop a solution with O(n log n) time complexity. This can be done using binary search on the prefix sums of the array, but the sliding window solution is already optimal for most cases.

Conclusion:

- The sliding window approach is a highly efficient and intuitive method for solving the "Minimum Size Subarray Sum" problem. It effectively reduces the problem to a linear-time solution, making it feasible even for large inputs. If needed, the binary search technique can be explored to achieve an O(n log n) solution for further optimization.