Smallest Subarray with a given sum (easy)

We'll cover the following

- Problem Statement
- Try it yourself
- Solution
- Code
  - Time Complexity
  - Space Complexity

Problem Statement

Given an array of positive numbers and a positive number 'S', find the length of the **smallest contiguous subarray whose sum is greater than or equal to 'S'**. Return 0, if no such subarray exists.

Example 1:

```
Input: [2, 1, 5, 2, 3, 2], S=7
Output: 2
Explanation: The smallest subarray with a sum great than or equal t
o '7' is [5, 2].
```

Example 2:

```
Input: [2, 1, 5, 2, 8], S=7
Output: 1
Explanation: The smallest subarray with a sum greater than or equa
l to '7' is [8].
```

Example 3:

```
Input: [3, 4, 1, 1, 6], S=8
Output: 3
Explanation: Smallest subarrays with a sum greater than or equal t
o '8' are [3, 4, 1] or [1, 1, 6].
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

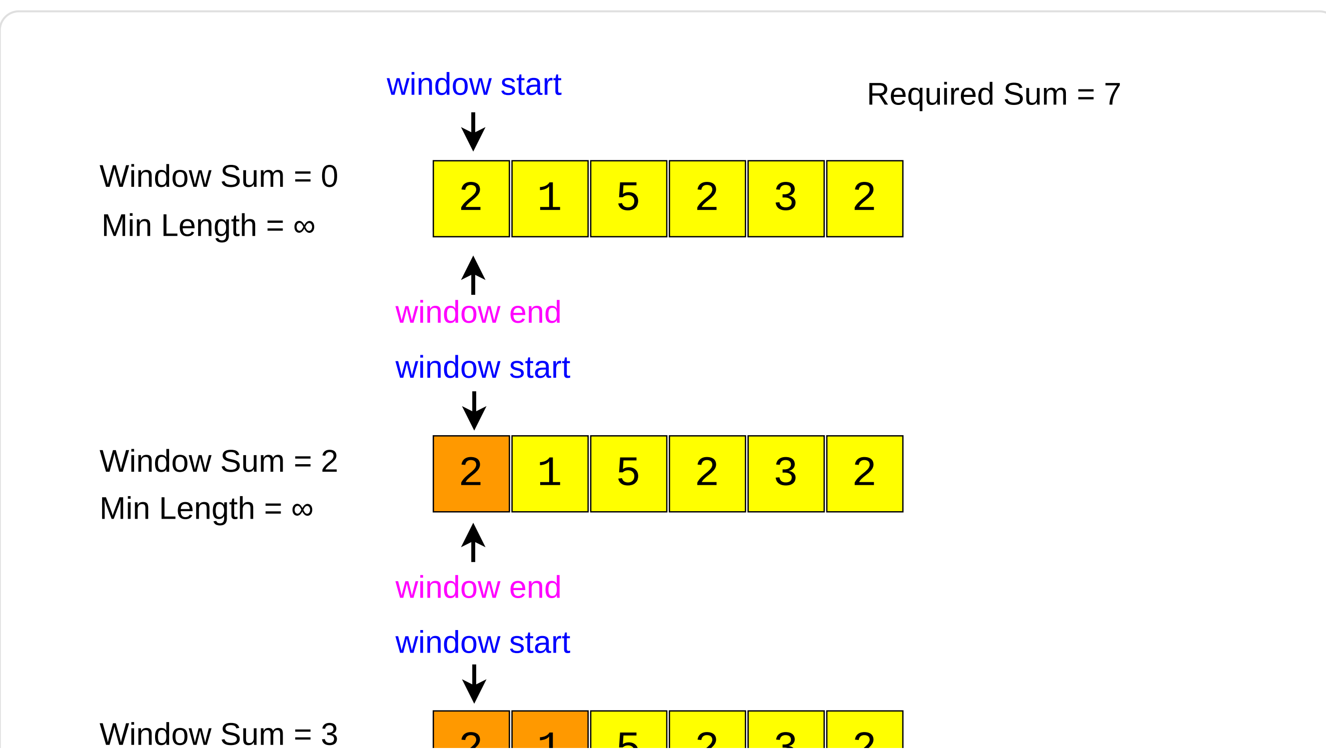
```java
class MinSizeSubArraySum {
  public static int findMinSubArray(int S, int[] arr) {
    // TODO: Write your code here
    return -1;
  }
}
```

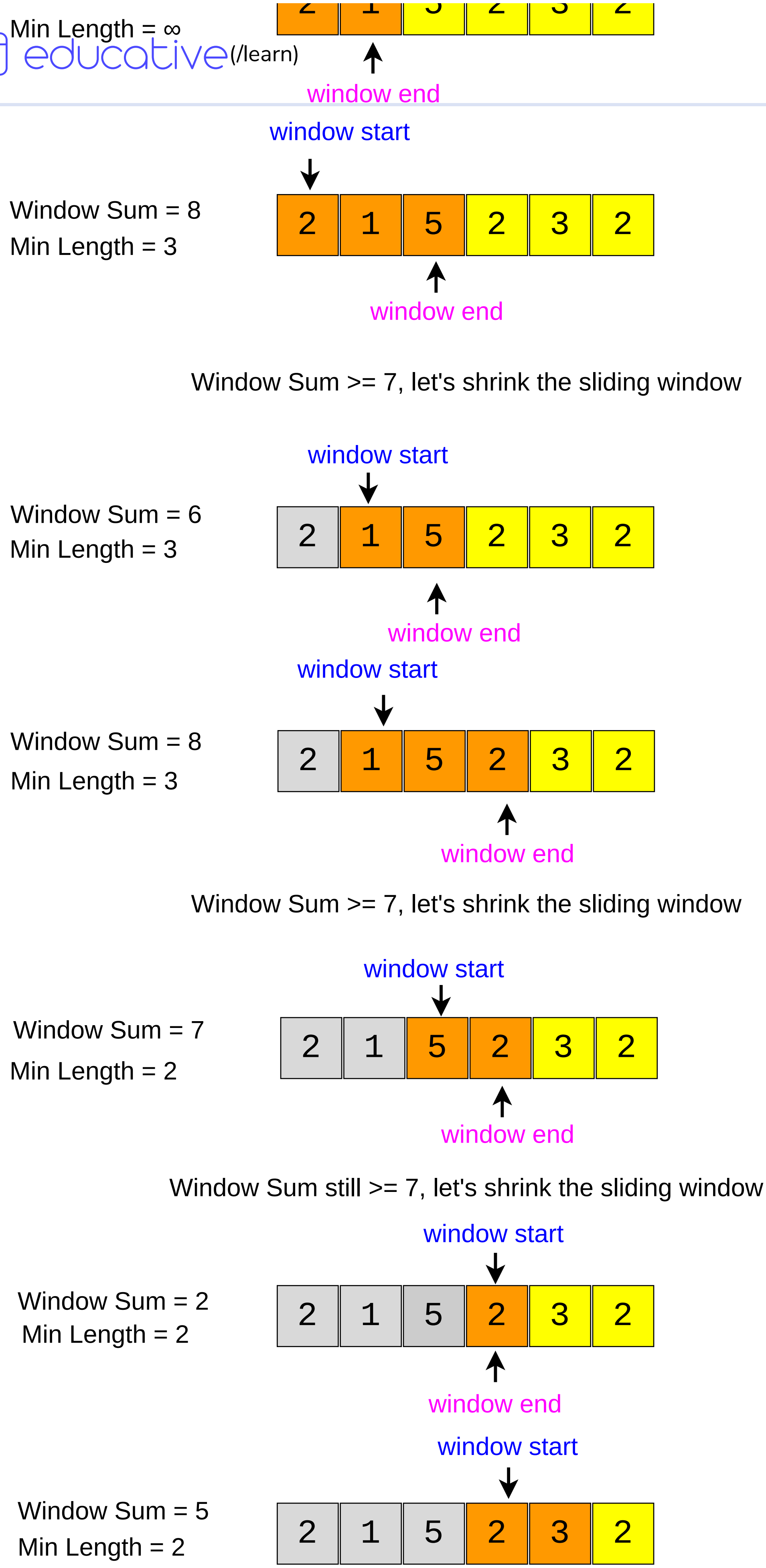
Solution

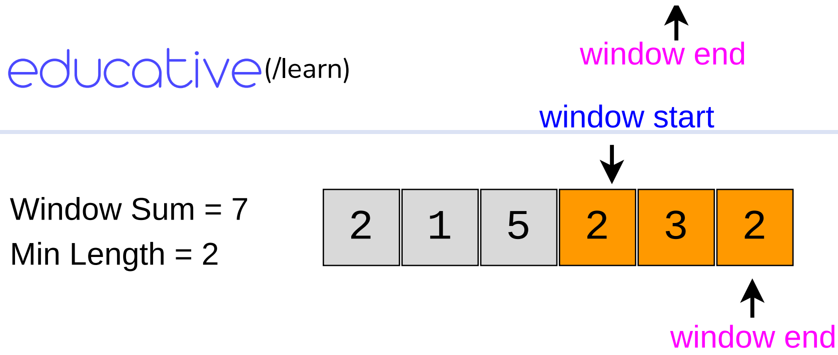
This problem follows the **Sliding Window** pattern and we can use a similar strategy as discussed in Maximum Sum Subarray of Size K (https://www.educative.io/collection/page/5668639101419520/567146485435596
There is one difference though: in this problem, the size of the sliding window is not fixed. Here is how we will solve this problem:

1. First, we will add-up elements from the beginning of the array until their sum becomes greater than or equal to 'S'.
2. These elements will constitute our sliding window. We are asked to find the smallest such window having a sum greater than or equal to 'S'. We will remember the length of this window as the smallest window so far.
3. After this, we will keep adding one element in the sliding window (i.e. slide the window ahead), in a stepwise fashion.
4. In each step, we will also try to shrink the window from the beginning. We will shrink the window until the window's sum is smaller than 'S' again. This is needed as we intend to find the smallest window. This shrinking will also happen in multiple steps; in each step we will do two things:
    - Check if the current window length is the smallest so far, and if so, remember its length.
    - Subtract the first element of the window from the running sum to shrink the sliding window.

Here is the visual representation of this algorithm for the Example-1




Min Length = ∞
window end
window start
Window Sum = 8
Min Length = 3
window end
Window Sum >= 7, let's shrink the sliding window
window start
Window Sum = 6
Min Length = 3
window end
window start
Window Sum = 8
Min Length = 3
window end
Window Sum >= 7, let's shrink the sliding window
window start
Window Sum = 7
Min Length = 2
window end
Window Sum still >= 7, let's shrink the sliding window
window start
Window Sum = 2
Min Length = 2
window end
window start
Window Sum = 5
Min Length = 2



Code

Here is what our algorithm will look:

Java | Python3 | C++ | JS

```
class MinSizeSubArraySum {
  public static int findMinSubArray(int S, int[] arr) {
    int windowSum = 0, minLength = Integer.MAX_VALUE;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {
      windowSum += arr[windowEnd]; // add the next element
      // shrink the window as small as possible until the 'windowSum' is smaller than
      while (windowSum >= S) {
        minLength = Math.min(minLength, windowEnd - windowStart + 1);
        windowSum -= arr[windowStart]; // subtract the element going out
        windowStart++; // slide the window ahead
      }
    }

    return minLength == Integer.MAX_VALUE ? 0 : minLength;
  }

  public static void main(String[] args) {
    int result = MinSizeSubArraySum.findMinSubArray(7, new int[] { 2, 1, 5, 2, 3, 2 ]
    System.out.println("Smallest subarray length: " + result);
    result = MinSizeSubArraySum.findMinSubArray(7, new int[] { 2, 1, 5, 2, 8 });
    System.out.println("Smallest subarray length: " + result);
    result = MinSizeSubArraySum.findMinSubArray(8, new int[] { 3, 4, 1, 1, 6 });
    System.out.println("Smallest subarray length: " + result);
  }
}
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$. The outer `for` loop runs for all elements and the inner `while` loop processes each element only once, therefore the time complexity of the algorithm will be $O(N + N)$ which is asymptotically equivalent to $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

← Back (/courses/grokking-the-coding-interview/JPKr0kqLGNP)
Maximum Sum Subarray of Size K (ea...

Completed
Next (/courses/grokking-the-coding-interview/YQQwQMWI
Longest Substring with K Distinct Ch...