

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 greater than or equal to '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 equal 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 to '8' are [3, 4, 1] or [1, 1, 6].
```

Try it yourself

Try solving this question here:

Java

Python3

JS

C++

```
1 def smallest_subarray_with_given_sum(s, arr):
2     # TODO: Write your code here
3     return -1
4
```

Test

Save

Reset

Solution

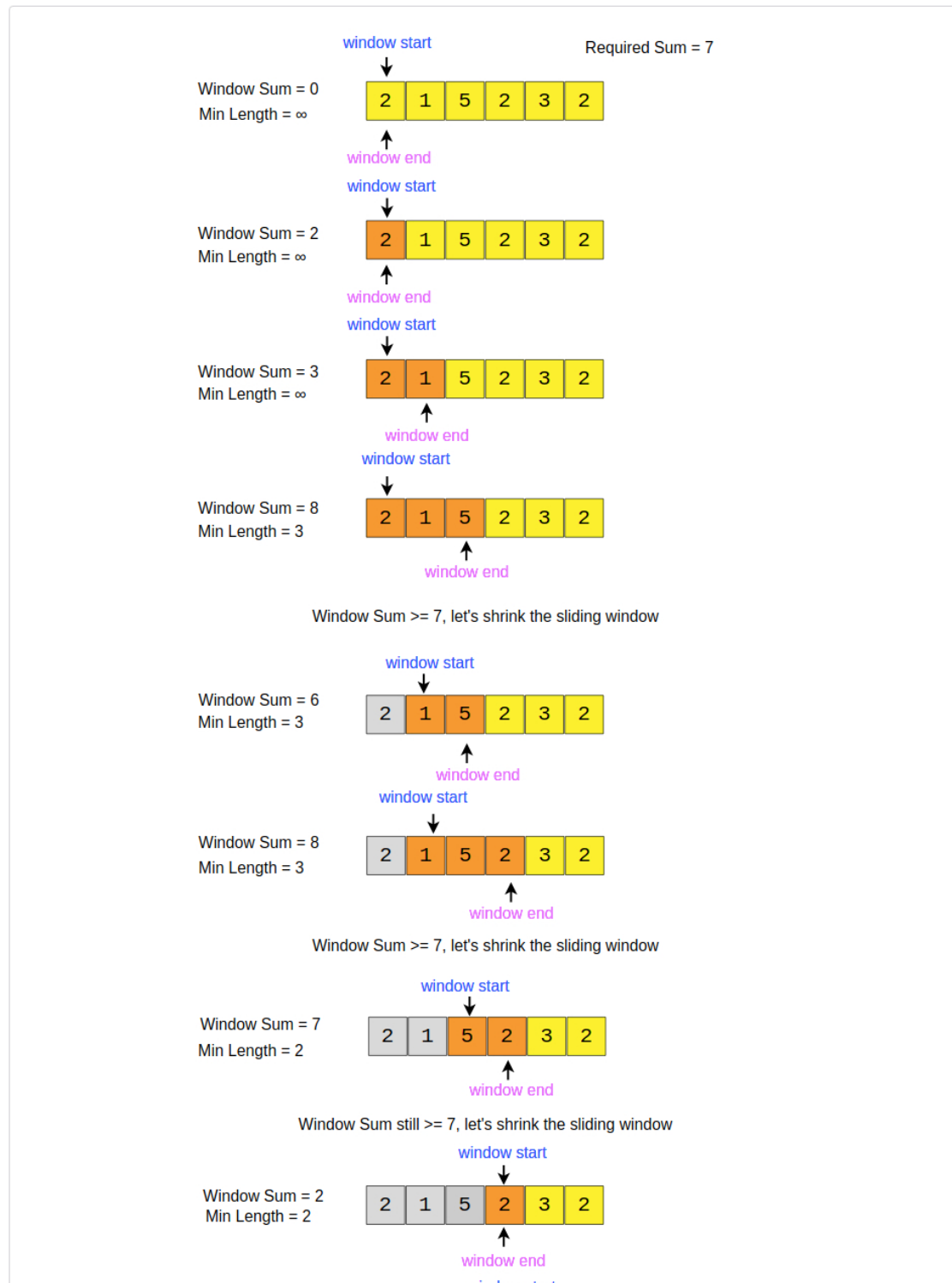
This problem follows the **Sliding Window** pattern, and we can use a similar strategy as discussed in [Maximum Sum Subarray of Size K](#). There is one difference though: in this problem, the sliding window size is not fixed. Here is how we will solve this problem:

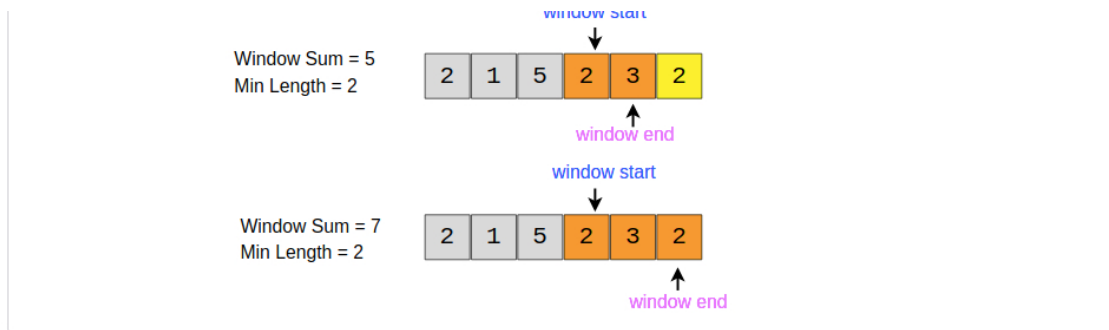
1. First, we will add-up elements from the beginning of the array until their sum becomes greater than or equal to 'S'.

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





Code

Here is what our algorithm will look:

Java

Python3

C++

JS

```

1 import math
2
3
4 def smallest_subarray_with_given_sum(s, arr):
5     window_sum = 0
6     min_length = math.inf
7     window_start = 0
8
9     for window_end in range(0, len(arr)):
10         window_sum += arr[window_end] # add the next element
11         # shrink the window as small as possible until the 'window_sum' is smaller than 's'
12         while window_sum >= s:
13             min_length = min(min_length, window_end - window_start + 1)
14             window_sum -= arr[window_start]
15             window_start += 1
16         if min_length == math.inf:
17             return 0
18         return min_length
19
20
21 def main():
22     print("Smallest subarray length: " + str(smallest_subarray_with_given_sum(7, [2, 1, 5, 2, 3, 2])))
23     print("Smallest subarray length: " + str(smallest_subarray_with_given_sum(7, [2, 1, 5, 2, 8])))
24     print("Smallest subarray length: " + str(smallest_subarray_with_given_sum(8, [3, 4, 1, 1, 6])))
25
26
27 main()
28

```

Run

Save

Reset

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

Next →

Maximum Sum Subarray of Size K (ea...

Longest Substring with K Distinct Cha...

✓ Completed

Report an Issue Ask a Question

