Smallest Subarray with a given sum (easy)

We'll cover the following

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 eq
ual 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 eq
ual 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 equa
l to '8' are [3, 4, 1]
or [1, 1, 6].
```

Try it yourself#

Try solving this question here:

Java | Python3 | JS | C++

```python
import math


def smallest_subarray_with_given_sum(s, arr):
    winStart, cur_sum, small = 0, 0, math.inf
    for winEnd in range(len(arr)):
        cur_sum += arr[winEnd]
        # print(winEnd, cur_sum)
        while cur_sum >= s:
            cur_small = winEnd - winStart + 1
            # print(cur_small, winEnd, winStart)
            small = min(small, cur_small)
            cur_sum -= arr[winStart]
            winStart += 1
    return small if small != math.inf else 0
```

Show Results Show Console

1.26s

3 of 3 Tests Passed

Result	Input	Expected Output	Actual Output	Reaso

Result	Input	Expected Output	Actual Output	Reaso
✓	smallest_subarray_with_given_sum(7, [2, …	2	2	Succeed
✓	smallest_subarray_with_given_sum(7, [2, …	1	1	Succeed
✓	smallest_subarray_with_given_sum(8, [3, …	3	3	Succeed

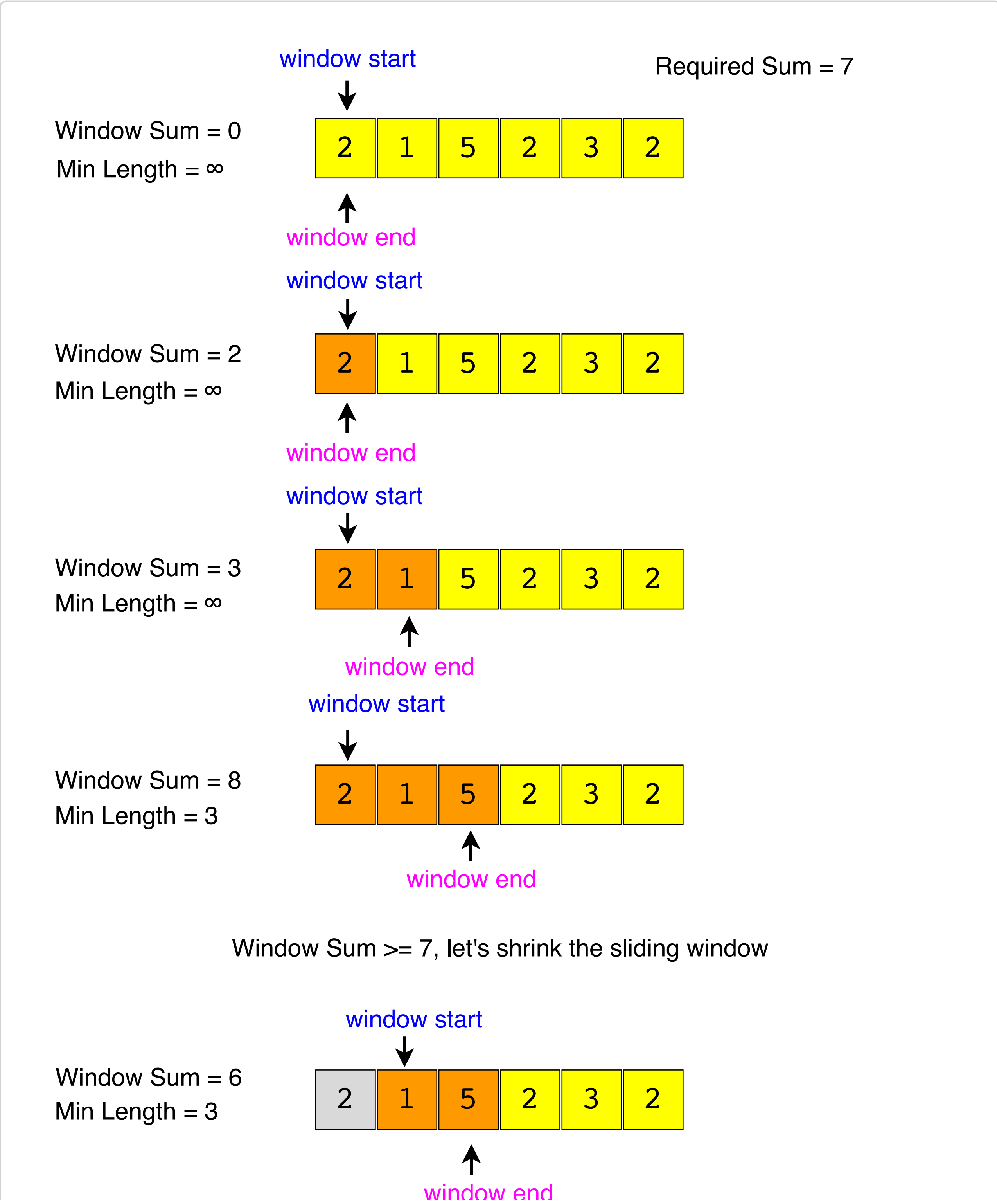
Solution#

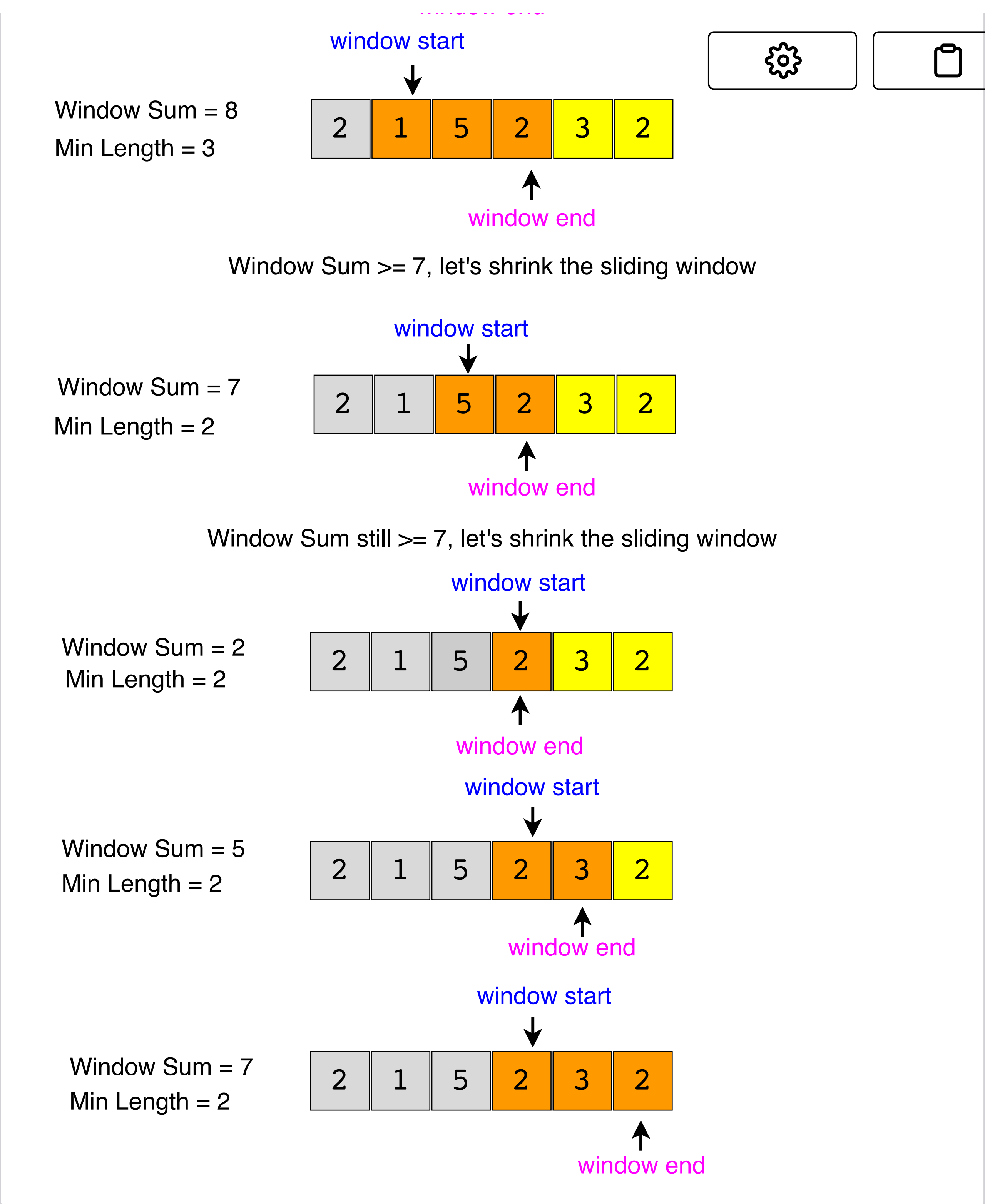
This problem follows the **Sliding Window** pattern, and we can use a similar strategy as discussed in Maximum Sum Subarray of Size K (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5177043027230720/). There is one difference though: in this problem, the sliding window size 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.'
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#

(/learn)

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
import math

```

```python

def smallest_subarray_with_given_sum(s, arr):
  window_sum = 0
  min_length = math.inf
  window_start = 0

  for window_end in range(0, len(arr)):
    window_sum += arr[window_end]  # add the next element
    # shrink the window as small as possible until the 'window_sum' is sma
    while window_sum >= s:
      min_length = min(min_length, window_end - window_start + 1)
      window_sum -= arr[window_start]
      window_start += 1
  if min_length == math.inf:
    return 0
  return min_length


def main():
  print("Smallest subarray length: " + str(smallest_subarray_with_given_su
  print("Smallest subarray length: " + str(smallest_subarray_with_given_su
  print("Smallest subarray length: " + str(smallest_subarray_with_given_su


main()

```

Output 0.75s

```
Smallest subarray length: 2
Smallest subarray length: 1
Smallest subarray length: 3
```

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

Maximum Sum Subarray of Size K (ea...

Next →

Longest Substring with maximum K D...

Mark as Completed

Report an Issue