```
"""
Given an array of positive integers nums and a positive integer target, return the minimal length of a
subarray
 whose sum is greater than or equal to target. If there is no such subarray, return 0 instead.

Example 1:

Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint
"""

# Brute force TC: O(N^2), SC: O(N) where N is length of nums
class Solution:
    def minSubArrayLen(self, target, nums):

        #Initialization
        numslength = len(nums) + 1
        prefixSum = [0 for _ in range(numslength)]

        # Calculate the prefix sum, helps in reusing, caching
        for idx in range(numslength-1):
            prefixSum[idx+1] = prefixSum[idx] + nums[idx]

        minLength = numslength

        for idx in range(numslength):
            for sIdx in range(idx+1, numslength):

                # If the condition is meet subarray sum >= target, try to update minLength
                if prefixSum[sIdx] - prefixSum[idx] >= target:
                    minLength = min(minLength, sIdx-idx)
                    break

        return minLength if minLength < numslength else 0

# Soln 2
# Optimized solution. TC: O(N), SC(1)
class Solution:
    def minSubArrayLen(self, target, nums):
```

```
        # Initialization
        windowStart = 0 # Left window
        minLength = len(nums) + 1 # To track result
        curSum = 0 # To track running sum

        for windowEnd, number in enumerate(nums): # idx, value in nums

            curSum += number # Add to running sum

            # As long as running sum >= target, update the minimal length and
shrinking the window
            while curSum >= target:

                # Current window length
                curLength = windowEnd - windowStart +1
                minLength = min(minLength, curLength)
                curSum -= nums[windowStart]
                windowStart += 1 # Shrink the window

        return minLength if minLength != len(nums) + 1 else 0
```

Question 2:

```
"""
Longest Repeating Character Replacement
You are given a string s and an integer k. You can choose any character of the
string and change it to any other uppercase English character. You can perform
this operation at most k times.
Return the length of the longest substring containing the same letter you can get
after performing the above operations.

Example 1:

Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.
"""

class Solution(object):
    def characterReplacement(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
```

```
        if not s:
            return 0

        # Initialization
        max_freq = float("-inf")
        windowStart = 0 # To track left window
        s_map = dict()

        for windowEnd, cur_char in enumerate(s): # idx, val in s

            # Update the number of occurrence of cur_char in s_map
            s_map[cur_char] = s_map.get(cur_char, 0)+1

            # Tries to update the max_freq: the char with current max occurence
            max_freq = max(max_freq, s_map[cur_char])

            # Length of the window so far
            window_length = windowEnd - windowStart+1

            # If the length of current window > max_freq + k, then we have go too
far, shrink the window
            if window_length > max_freq + k:
                to_del = s[windowStart]
                s_map[to_del] -= 1
                windowStart += 1

        # Return the window with the longest substring where we may (not
necessarily) perform at most k replacements
        return windowEnd - windowStart + 1
```

Question 3:

```
"""
Question: Find all Anagrams in a string
Given two strings s and p, return an array of all the start indices of p's
anagrams in s. You may return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different
word or phrase, typically using all the original letters exactly once.

Example 1:
Input: s = "cbaebabacd", p = "abc"
Output: [0,6]
Explanation:
```

```
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
"""

from collections import Counter
class Solution:
    def findAnagrams(self, string, pattern):

        if len(pattern) > len(string):
            return []

        # Utilizing a fixed size window: pattern length
        patternLength = len(pattern)
        stringLength = len(string)

        # Counter stores the pattern input with the number of occurence
        pattern_dict = Counter(pattern)
        string_dict = Counter() # To store occurence of string
        leftWindow = 0 # To track left window
        result = [] # result

        for windowEnd, strChar in enumerate(string):  # idx, value in string

            # Storing the current string character in string_dict
            string_dict[strChar] += 1

            # Found an anagram and store the left window (index), start indices
            if string_dict == pattern_dict:
                result.append(leftWindow)

            # Condition to remove from string_dict: windowEnd >= patternLength-1.
PatternLength - 1 because indices start from 0.
            if windowEnd >= patternLength - 1:
                leftChar = string[leftWindow]
                leftWindow +=1 #Tries to maintain the fixed size window
                string_dict[leftChar] -= 1
                if string_dict[leftChar] == 0:
                    del string_dict[leftChar]

        return result
```

Question 4:

```
"""
Question: Shortest subarray with sum at least k
```

```
Given an integer array nums and an integer k, return the length of the shortest
non-empty subarray of nums with a sum of at least k. If there is no such
subarray, return -1.
A subarray is a contiguous part of an array.
Example:
Input: nums = [1], k = 1
Output: 1
"""

from collections import deque
class Solution:
    def shortestSubarray(self, nums, k):

        # Prefix sum array to calculate the cummulative sume of nums input
        prefixSum = [0 for _ in range(len(nums)+1)]
        for idx in range(len(nums)):
            prefixSum[idx+1] = prefixSum[idx] + nums[idx]

        min_length = prefix_length = len(prefixSum)

        # Double ended queue for ease of removal from both sides
        # myQueue will store indexes of number that are strictly increasing
        myQueue = deque()

        for idx in range(prefix_length):

            # Only store indexes of number strictly increasing. <= because having
a zero also in nums extends (rather than shorten) the subarray size
            while myQueue and prefixSum[idx] - prefixSum[myQueue[-1]] <= 0:
                myQueue.pop()

            # Calculating the subarray length that meet the condition, subarray
with sum at least k, and removal from Queue
            while myQueue and prefixSum[idx] - prefixSum[myQueue[0]] >= k:
                lastIdx = myQueue.popleft()
                min_length = min(min_length, idx - lastIdx)

            myQueue.append(idx)

        # If total sum of numbers is less than k, and only (+)ve in nums,
returned -1
        return min_length if min_length != prefix_length else -1
```