209. Minimum Size Subarray Sum

https://leetcode.com/problems/minimum-size-subarray-sum/description/

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a **contiguous subarray** $[nums_l, nums_{l+1}, ..., nums_{r-1}, nums_r]$ of which the 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.
```

Example 2:

```
Input: target = 4, nums = [1,4,4]
Output: 1
```

Example 3:

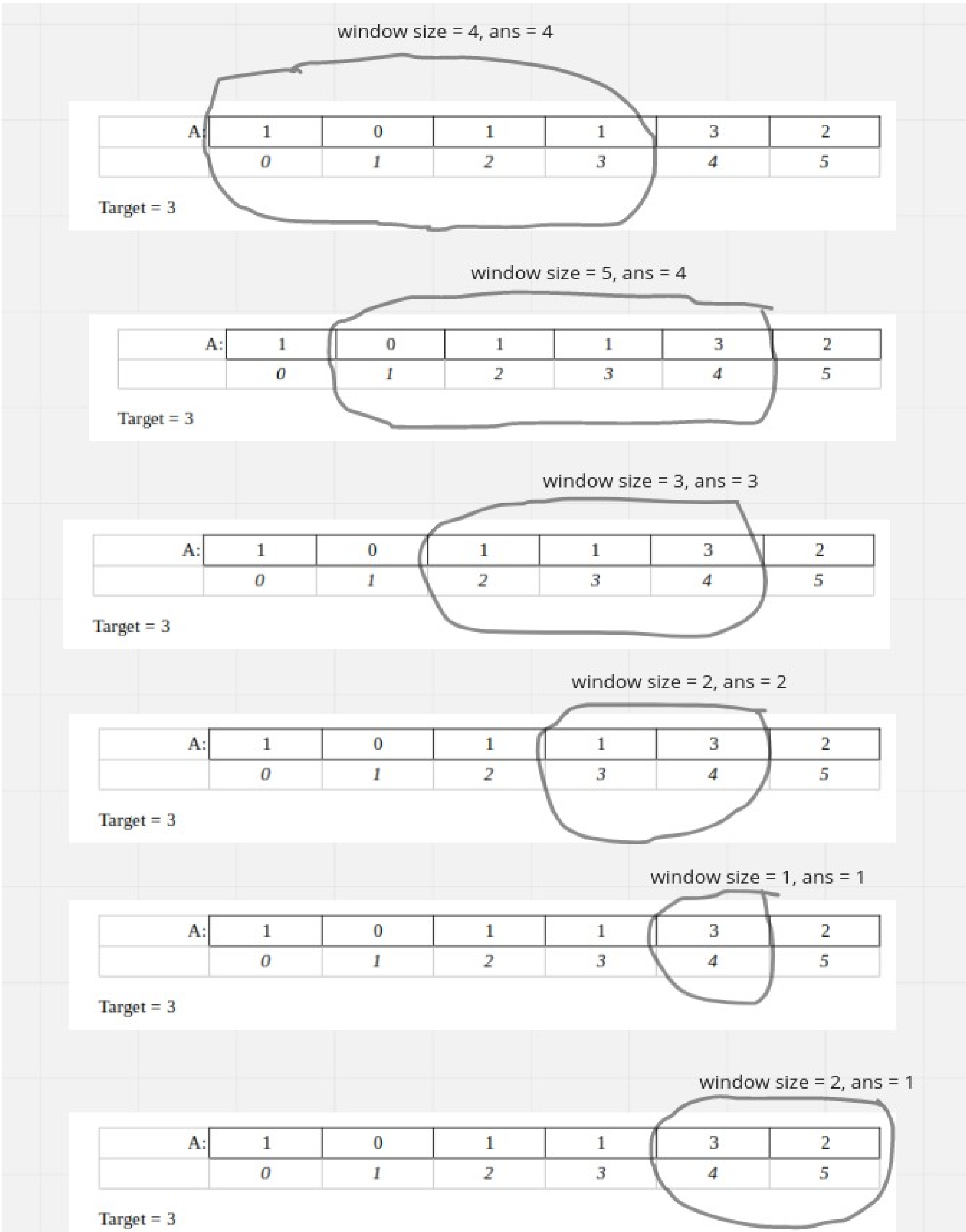
```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```

Constraints:

- $1 <= target <= 10^9$
- $1 <= nums.length <= 10^5$
- $1 <= nums[i] <= 10^4$

Follow up: If you have figured out the `O(n)` solution, try coding another solution of which the time complexity is `O(n log(n))`.

Example


window size = 4, ans = 4
A: 1 0 1 1 3 2
0 1 2 3 4 5
Target = 3
window size = 5, ans = 4
A: 1 0 1 1 3 2
0 1 2 3 4 5
Target = 3
window size = 3, ans = 3
A: 1 0 1 1 3 2
0 1 2 3 4 5
Target = 3
window size = 2, ans = 2
A: 1 0 1 1 3 2
0 1 2 3 4 5
Target = 3
window size = 1, ans = 1
A: 1 0 1 1 3 2
0 1 2 3 4 5
Target = 3
window size = 2, ans = 1
A: 1 0 1 1 3 2
0 1 2 3 4 5
Target = 3

Brute force approach

Test all combinations.

Time complexity: $O(n^2)$

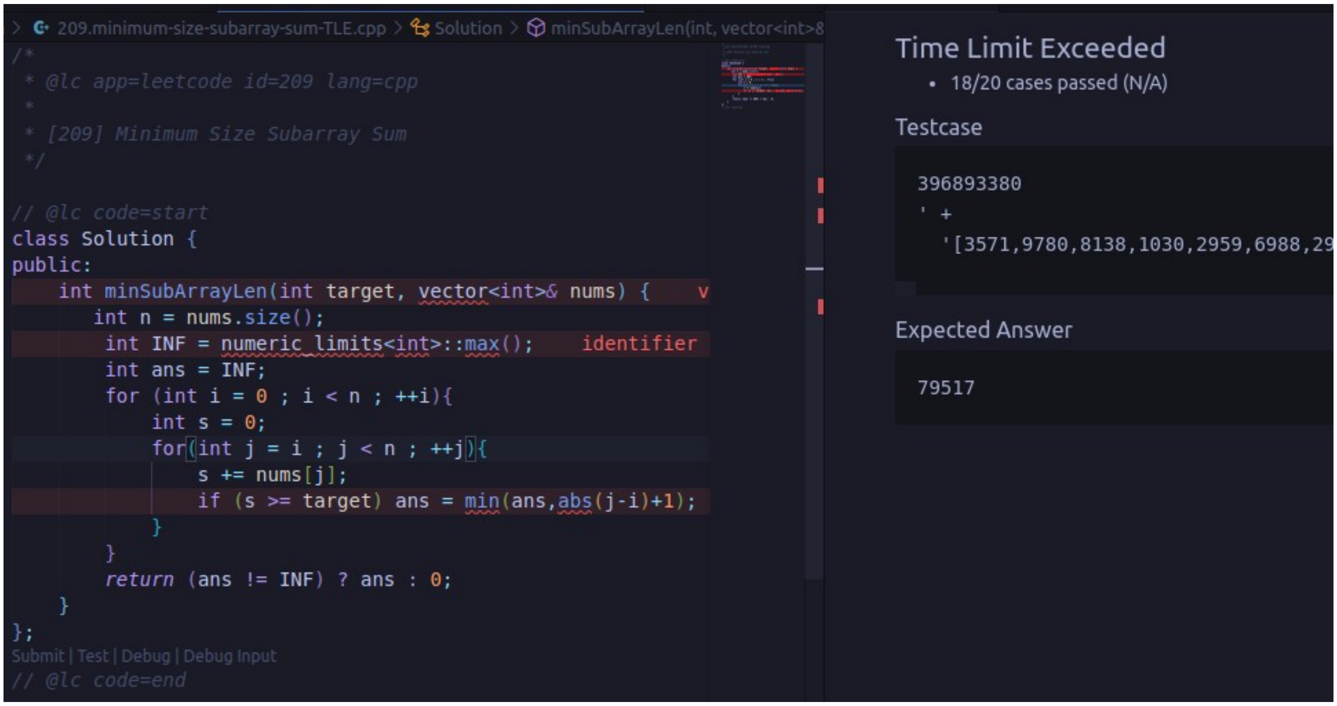
Space complexity: $O(1)$

pseudo-code

```
ans ← +∞

n ← nums.length

for i ∈ [0,n[ {

  s ← 0

  for j ∈ [i,n[{

    s ← s + nums[j]

  }

  if s >= target {

    ans ← min(ans,|j-i|+1)

  }

}

if (ans == +∞) return 0

else return ans
```

C++

```cpp
int minSubArrayLen(int target, vector<int>& nums) {
       int n = nums.size();
        int INF = numeric_limits<int>::max();
        int ans = INF;
        for (int i = 0 ; i < n ; ++i){
            int s = 0;
            for(int j = i ; j < n ; ++j){
                s += nums[j];
                if (s >= target) ans = min(ans,abs(j-i)+1);
            }
        }
        return (ans != INF) ? ans : 0;
}
```


209.minimum-size-subarray-sum-TLE.cpp > Solution > minSubArrayLen(int, vector<int>&
/*
 * @lc app=leetcode id=209 lang=cpp
 *
 * [209] Minimum Size Subarray Sum
 */
// @lc code=start
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        int n = nums.size();
        int INF = numeric_limits<int>::max();
        int ans = INF;
        for (int i = 0 ; i < n ; ++i){
            int s = 0;
            for(int j = i ; j < n ; ++j){
                s += nums[j];
                if (s >= target) ans = min(ans,abs(j-i)+1);
            }
        }
        return (ans != INF) ? ans : 0;
    }
};
Submit | Test | Debug | Debug Input
// @lc code=end
identifier
Time Limit Exceeded
18/20 cases passed (N/A)
Testcase
396893380
' +
'[3571,9780,8138,1030,2959,6988,29
Expected Answer
79517

Using a prefix sum array + a deque

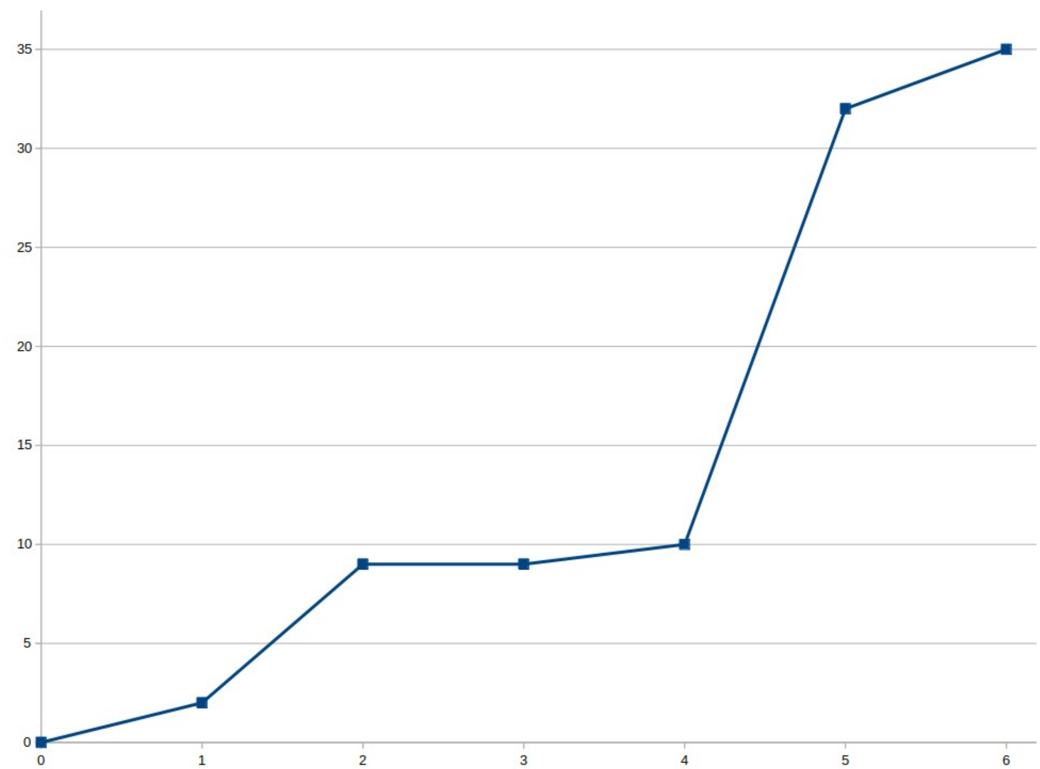
Prefix sum array

$$PRE[i] = A[0] + A[1] + \ldots + A[i-1]$$

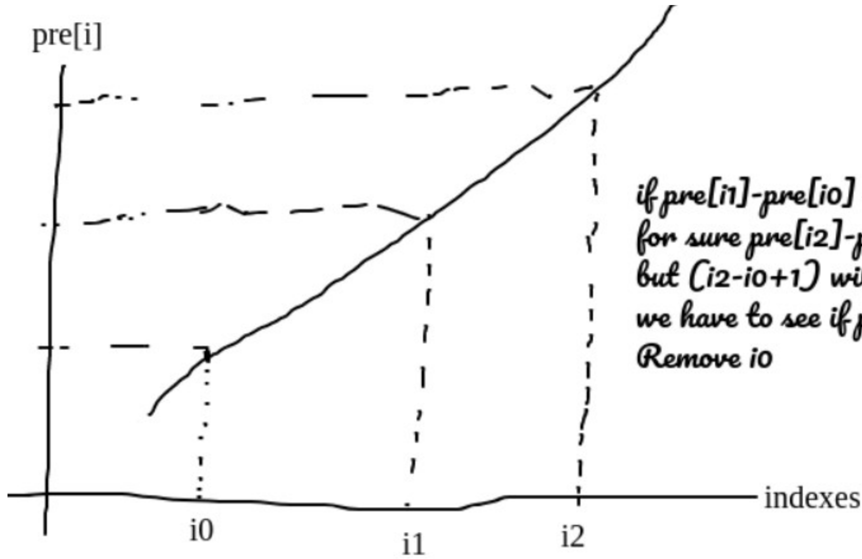
example:

A:	2	7	0	1	22	3
	0	*1*	*2*	*3*	*4*	*5*

PRE:	0	2	9	9	10	32	35
	0	*1*	*2*	*3*	*4*	*5*	*6*



All the values in original A are positive, then for sure we have a monotonic ascending order in the prefix sum array PRE .



if pre[i1]-pre[i0] >= target, then
for sure pre[i2]-pre[i0] >= target
but (i2-i0+1) will not be the shortest length,
we have to see if pre[i2]-pre[i1] is >= target or not.
Remove i0

To do that, we can use a deque.

Deque

https://cplusplus.com/reference/deque/deque/

class template

<deque>

std::deque

```
template < class T, class Alloc = allocator<T> > class deque;
```

Double ended queue

deque (usually pronounced like *"deck"*) is an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Specific libraries may implement *deques* in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes *undefined behavior*.

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists and forward lists.

Container properties

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Dynamic array
Generally implemented as a dynamic array, it allows direct access to any element in the sequence and provides relatively fast addition/removal of elements at the beginning or the end of the sequence.

Allocator-aware
The container uses an allocator object to dynamically handle its storage needs.

To resolve this problem with a deque, we have to satisfy a main condition, which is the **monotonically increasing order of the sum must be maintained**, by:

- insertion of the new sum value at the end in monotonically ascending order.
- Reducing of the sum value from the begin to minimize window(subarray) size.

The deck will store the indexes of prefix sum array in ascending order of the sum.

Example:

A:	1	0	1	1	3	2
	0	*1*	*2*	*3*	*4*	*5*

Target = 3

PRE:	0	1	1	2	3	6	8
	0	*1*	*2*	*3*	*4*	*5*	*6*

Target = 3

<table>
<tr><th>ans</th><th>i</th><th>DQ ≠ Ø and PRE[i] - PRE[DQ[0]] >= target</th><th>DQ</th></tr>
<tr><td>+∞</td><td></td><td></td><td>Ø</td></tr>
<tr><td></td><td>0</td><td>DQ ≠ Ø and …..</td><td></td></tr>
<tr><td></td><td></td><td></td><td>{0}</td></tr>
<tr><td></td><td>1</td><td>DQ ≠ Ø and PRE[1] – PRE[0] = 1 – 0 = 1 >= 3</td><td></td></tr>
<tr><td></td><td></td><td></td><td>{0,1}</td></tr>
<tr><td></td><td>2</td><td>DQ ≠ Ø and PRE[2] – PRE[0] = 1 – 0 = 1 >= 3</td><td></td></tr>
<tr><td></td><td></td><td></td><td>{0,1,2}</td></tr>
<tr><td></td><td>3</td><td>DQ ≠ Ø and PRE[3] – PRE[0] = 2 – 0 = 2 >= 3</td><td></td></tr>
<tr><td></td><td></td><td></td><td>{0,1,2,3}</td></tr>
<tr><td>min(+∞,4-0) = 4</td><td>4</td><td>DQ ≠ Ø and PRE[4] – PRE[0] = 3 – 0 = 3 >= 3</td><td>{1,2,3}</td></tr>
<tr><td></td><td></td><td><table><tr><td>A:</td><td>1</td><td>0</td><td>1</td><td>1</td><td>3</td><td>2</td></tr><tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td><td></td></tr>
<tr><td></td><td></td><td>DQ ≠ Ø and PRE[4] – PRE[1] = 3 – 1 = 2 >= 3</td><td></td></tr>
<tr><td></td><td></td><td></td><td>{1,2,3,4}</td></tr>
</table>

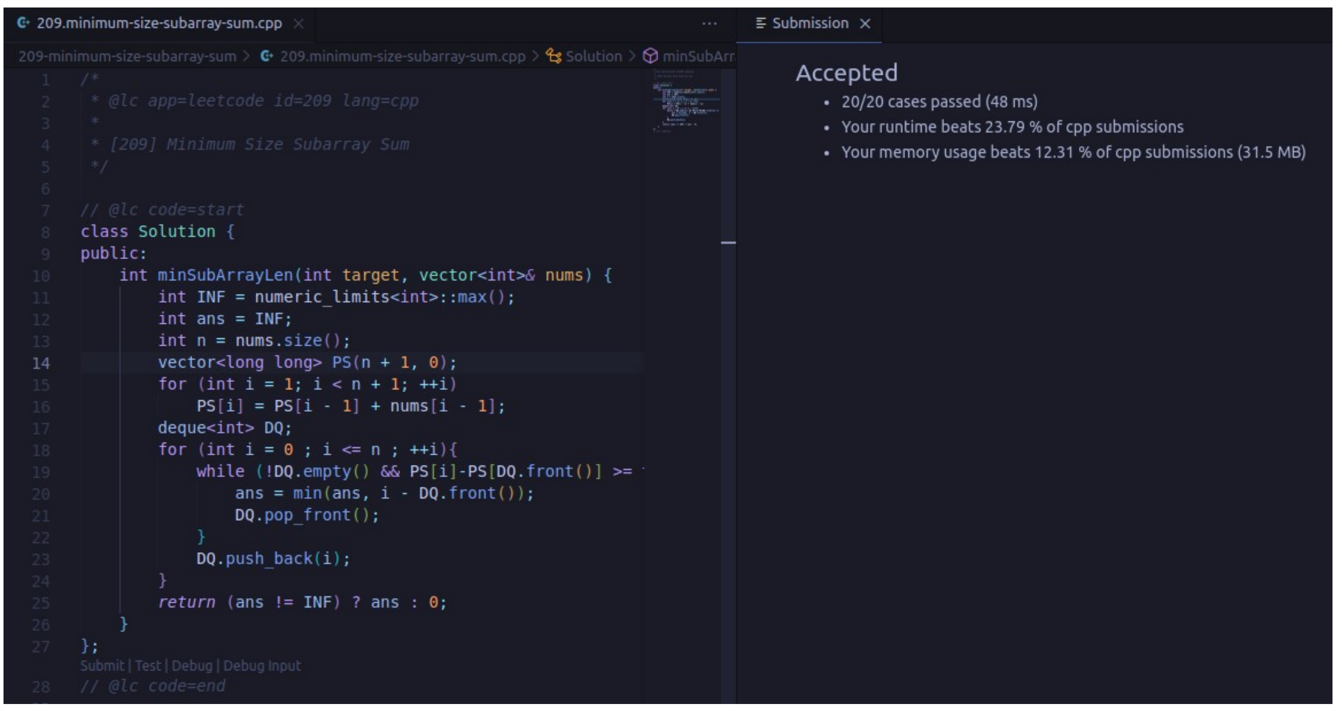
ans	i	DQ ≠ Ø and PRE[i] - PRE[DQ[0]] >= target	DQ
min(4,5-1) = 4	5	DQ ≠ Ø and PRE[5] – PRE[1] = 6 – 1 = 5 >= 3	{2,3,4}
		A: 1 **0** **1** **1** **3** 2 0 1 2 3 4 5	
min(4,5-2) = 3		DQ ≠ Ø and PRE[5] – PRE[2] = 6 – 1 = 5 >= 3	{3,4}
		A: 1 0 **1** **1** **3** 2 0 1 2 3 4 5	
min(4,5-3) = 2		DQ ≠ Ø and PRE[5] – PRE[3] = 6 – 2 = 4 >= 3	{4}
		A: 1 0 1 **1** **3** 2 *0 1 2 3 4 5*	
min(4,5-4) = 1		DQ ≠ Ø and PRE[5] – PRE[4] = 6 – 3 = 3 >= 3	Ø
		A: 1 0 1 1 **3** 2 *0 1 2 3 4 5*	
		DQ ≠ Ø and …….	
			{5}
	6	DQ ≠ Ø and PRE[6] – PRE[5] = 8 – 6 = 2 >= 3	
			{5,6}
Final ans = 1			

pseudo-code

```
ans ← +∞
n ← nums.length
Initialize PRE of size n+1 to zeros
for i ∈ [0,n[ {
  PRE[i+1] ← PRE[i] + A[i]
}
for i ∈ [0,n+1[ {
  while DQ ≠ Ø and PRE[i] - PRE[DQ[0]] >= target {
    ans ← min(ans,i-DQ[0])
    remove DQ[0]
  }
  push i to DQ
}
if (ans == +∞) return 0
else return ans
```

C++

```cpp
int minSubArrayLen(int target, vector<int>& nums) {
        int INF = numeric_limits<int>::max();
        int ans = INF;
        int n = nums.size();
        vector<long long> PS(n + 1, 0);
        for (int i = 1; i < n + 1; ++i)
            PS[i] = PS[i - 1] + nums[i - 1];
        deque<int> DQ;
        for (int i = 0 ; i <= n ; ++i){
            while (!DQ.empty() && PS[i]-PS[DQ.front()] >= target){
                ans = min(ans, i - DQ.front());
                DQ.pop_front();
            }
            DQ.push_back(i);
        }
        return (ans != INF) ? ans : 0;
}
```



Time complexity: $Amortized\, O(n)$

Space complexity: $O(2N)$

window sliding technique using two pointers(variables)

Idea

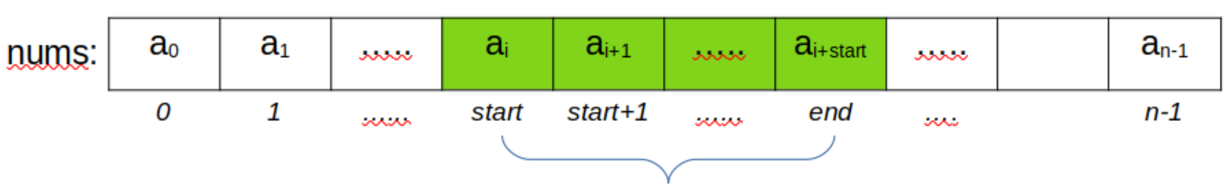
The idea behind it is to maintain two pointers: $start$ and end, moving them in a smart way to avoid examining all possible values $0 \leq end \leq n-1$ and $0 \leq start \leq end$ (to avoid brute force).

1. Incrementing the end pointer while the sum of current subarray (defined by current values of $start$ and end) is smaller than the target.

2. Once we satisfy our condition $\sum_{i=start}^{end} nums[i] \geq target$ we keep incrementing the $start$ pointer until we violate it (until $\sum_{i=start}^{end} nums[i] < target$).

3. Once we violate the condition we keep incrementing the **end** pointer until the condition is satisfied again and so on.

The reason why we stop incrementing $start$ when we violate the condition is that we are sure we will not satisfy it again if we keep incrementing $start$. In other words, if the sum of the current subarray from $start$ to end is smaller than the target then the sum from $start+1$ to end is neccessarily smaller than the target. (positive values)



$$We\ have\ all\ a_i > 0, (0 \leq i \leq n-1), so, if \sum_{i=start}^{end} nums[i] < target, then \sum_{i=start+1}^{end} nums[i] < target$$

$$if \sum_{i=start}^{end} nums[i] \geq target, \sum_{i=start+x}^{end} nums[i], could\ be\ still \geq target, (1 \leq x \leq end),$$

$$the\ shortest\ length\ of\ subarray\ which\ \mathrm{sum} \geq target\ is\ \mathbf{min}(\mathbf{previous\_length}, \boldsymbol{end} - \boldsymbol{start} + \mathbf{1})$$

pseudo-code

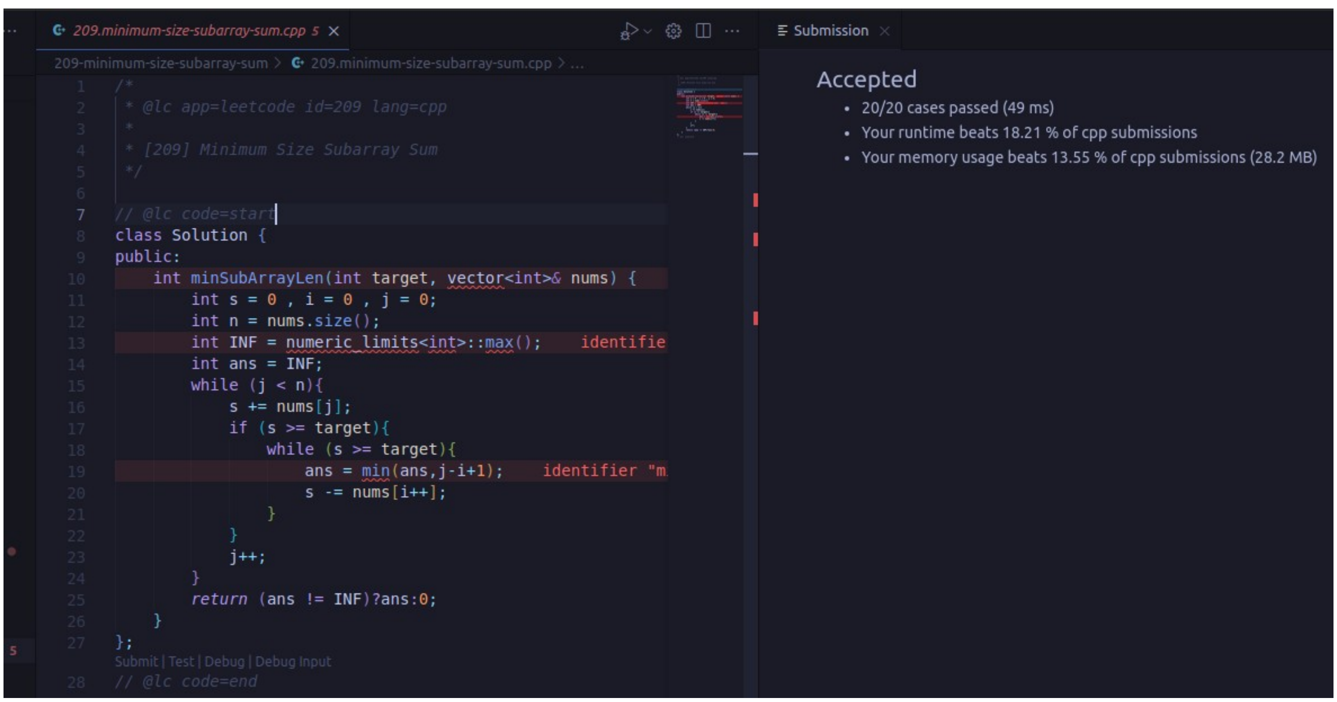
```
start ← 0, end ← 0 , s ←  0
ans ← +∞
n ← nums.length
while end < n {
  s ← s +  nums[end]
  if s >= target {
    while s >= target {
      ans ←  min(ans,end-start+1)
      s ← s - nums[start]
      start ← start + 1
    }
  }
  end ← end + 1
}
if (ans == +∞) return 0
else return ans
```

Time complexity: $\boldsymbol{Amortized\,O(n)}$

Space complexity: $\boldsymbol{O(1)}$

C++

```cpp
int minSubArrayLen(int target, vector<int>& nums) {
        int s = 0 , i = 0 , j = 0;
        int n = nums.size();
        int INF = numeric_limits<int>::max();
        int ans = INF;
        while (j < n){
            s += nums[j];
            if (s >= target){
                while (s >= target){
                    ans = min(ans,j-i+1);
                    s -= nums[i++];
                }
            }
            j++;
        }
        return (ans != INF)?ans:0;
    }
```



862. 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 1:

```
Input: nums = [1], k = 1
Output: 1
```

Example 2:

```
Input: nums = [1,2], k = 4
Output: -1
```

Example 3:

```
Input: nums = [2,-1,2], k = 3
Output: 3
```

Constraints:

- $1 <= nums.length <= 10^5$
- $-10^5 <= nums[i] <= 10^5$
- $1 <= k <= 10^9$

In #209, we compute the shortest length of subarray with sum at least target.

We used two approaches to resolve it:

- deque,
- sliding window technique.

These two approaches works because the original array contains only positive values, so we have a monotonically increasing sum.

However, in #862, the original array could contain **negative values**, that's why **we can not have a monotonically increasing sum**.

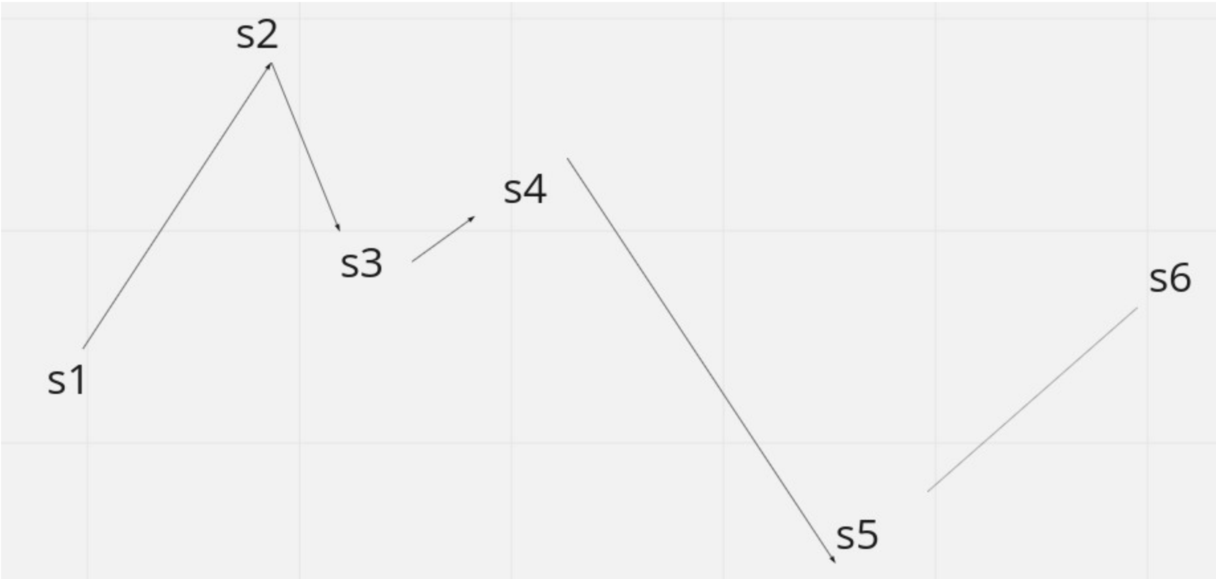
To resolve #862, we must maintain a monotonically increasing sum.

Can we maintain a monotonically increasing sum with the two pointers sliding window technique?

The answer is no, because pre-calculated sums are not saved.

We can maintain a monotonically increasing sum with a deque.

How to maintain a monotonically increasing sum?



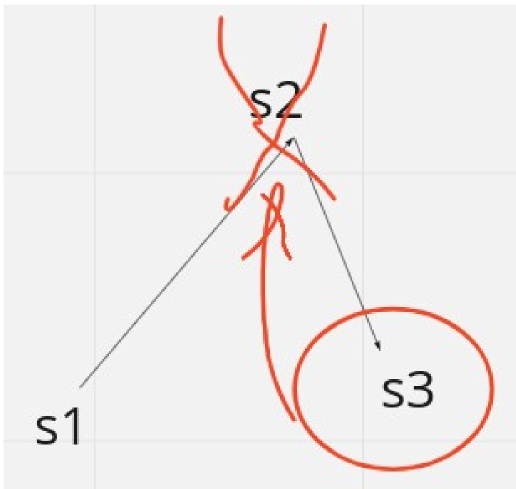
Objectif: push sums in the deque in monotonically increasing sum.

- At the beginning the deque is empty: DQ = Ø
- we push the first sum

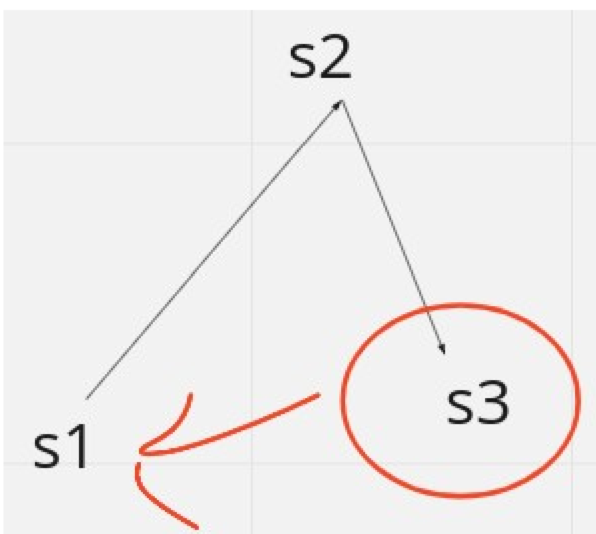
DQ = {s1}

- at s2, we compare it with the last element of the deque, if the current sum is less than the sum in last element of the deck, then remove it.
  DQ = {s1,s2}

- s3 < s2: pop s2
  DQ = {s1}

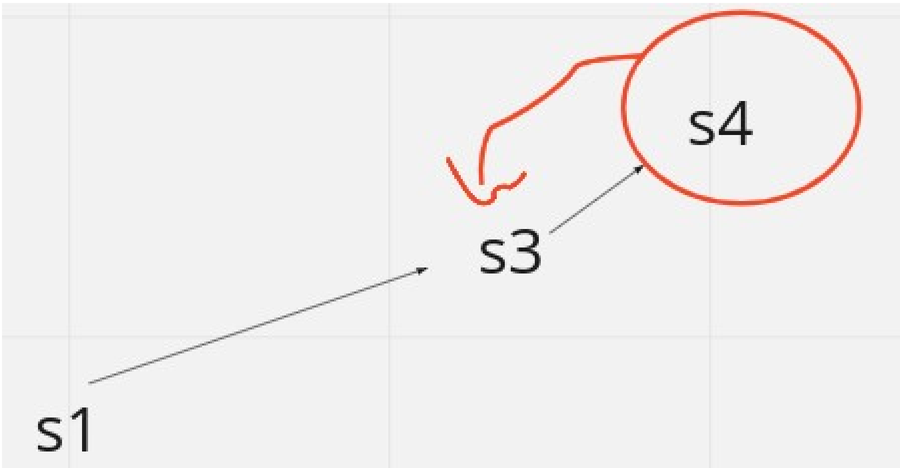


s3 > s1: keep s1

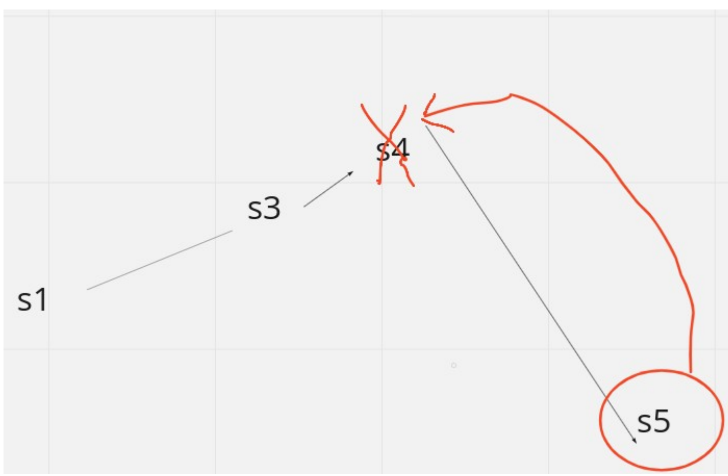


push s3: DQ = {s1,s3}

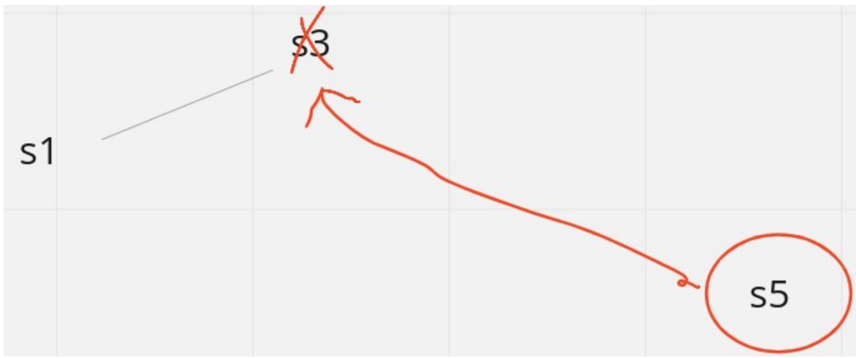
- s4 > s3: keep s3, push s4
  DQ = {s1,s3,s4}



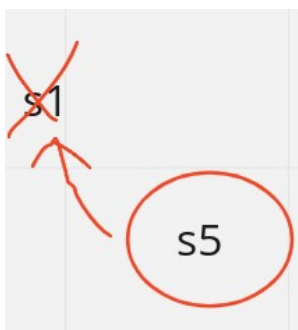
- s5 < s4: remove s4
  DQ = {s1,s3}



s5 < s3: remove s3
DQ = {s1}

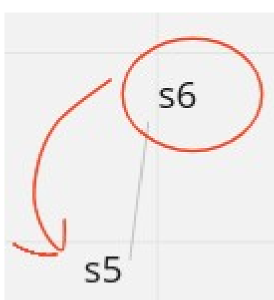


s5 < s1: remove s1



push s5: DQ = {s5}

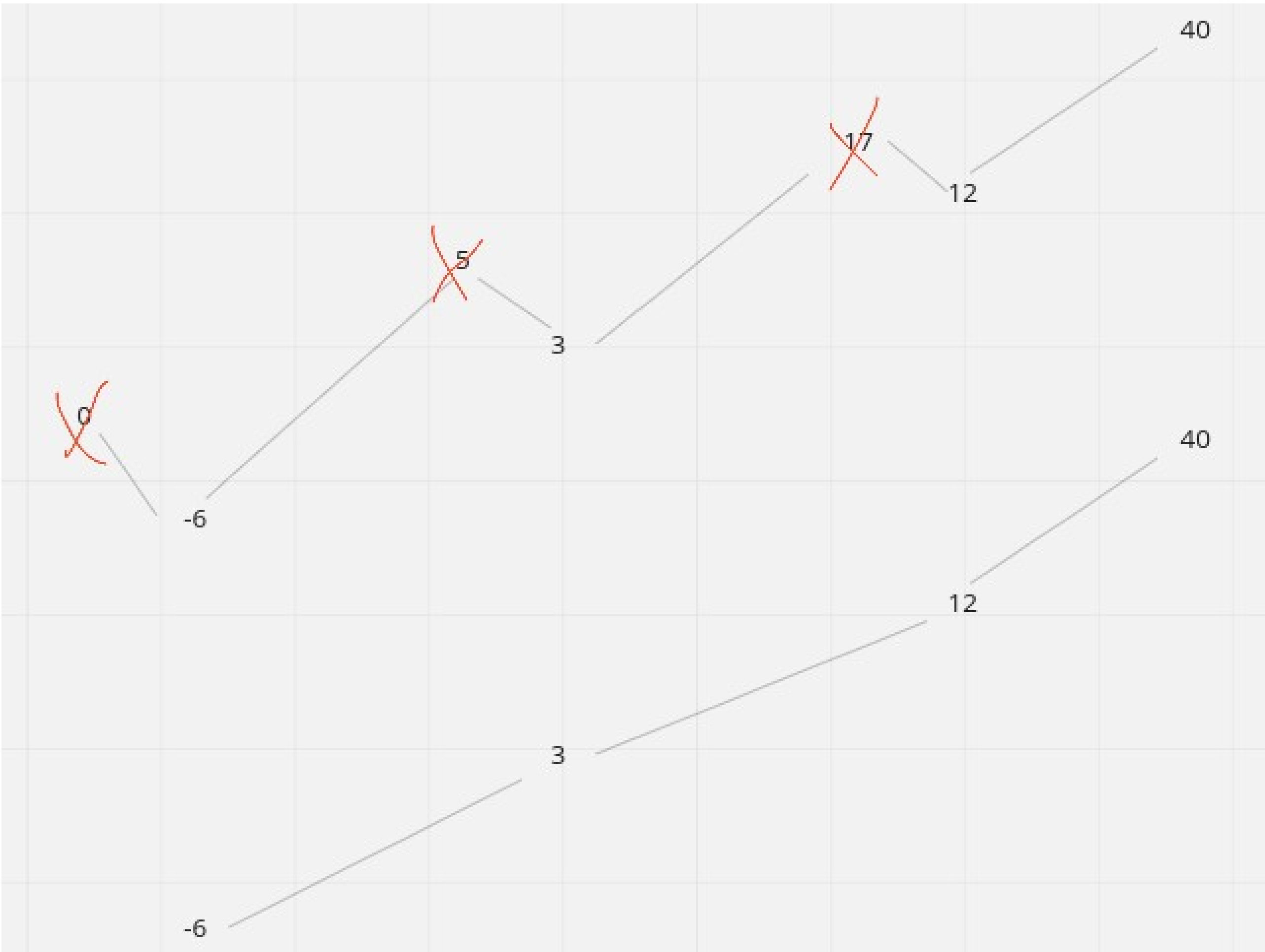
- s6 > s5: keep s5, push s6:
  DQ = {s5,s6}



Examples:

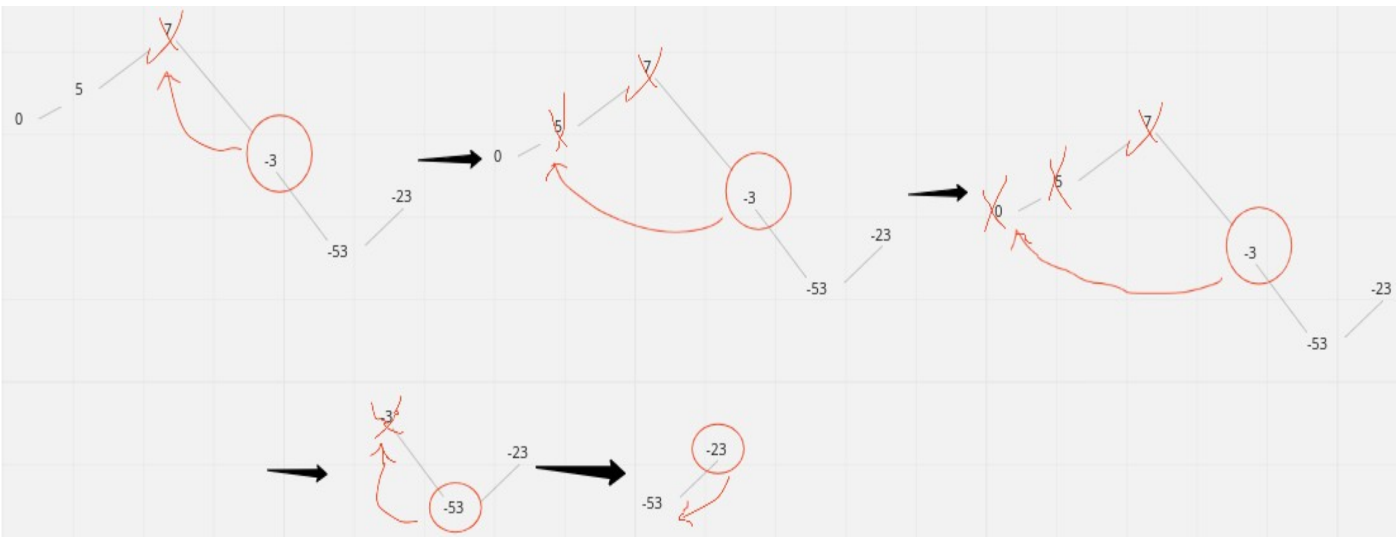
A:	-6	11	-2	14	-5	28
	0	*1*	*2*	*3*	*4*	*5*

PRE:	0	-6	5	3	17	12	40
	0	*1*	*2*	*3*	*4*	*5*	*6*



A:	5	2	-10	-50	30
	0	*1*	*2*	*3*	*4*

PRE:	0	5	7	-3	-53	-23
	0	1	2	3	4	5



Pseudo code of the maintaining monotonically increasing sum part

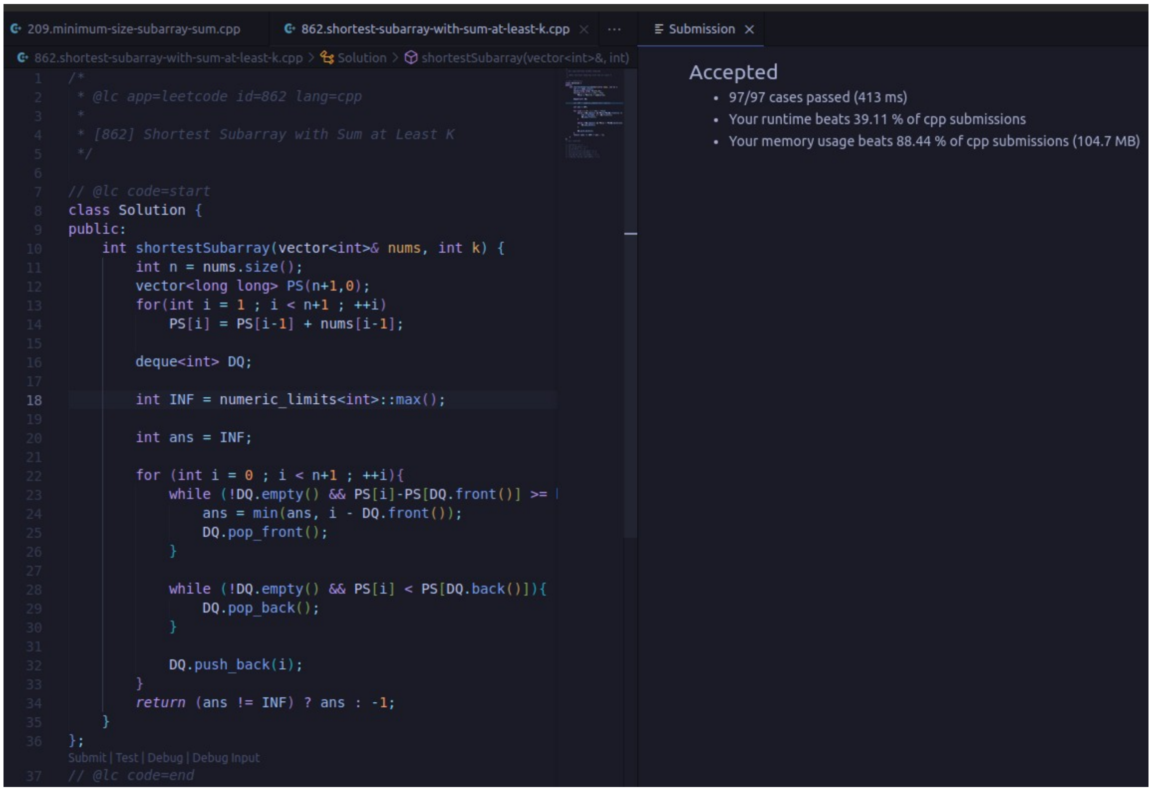
```
while DQ ≠ Ø and PRE[i] < DQ.last_element {
  remove the last element of the deque
}
```

Pseudo code

```
ans ← +∞
n ← nums.length
for i ∈ [0,n[ {
  while DQ ≠ Ø and PRE[i] - PRE[DQ.first_element] >= k {
    ans ← min(ans,i-index_of_first_element)
    remove the first element of the deque
  }
  while DQ ≠ Ø and PRE[i] < DQ.last_element {
    remove the last element of the deque
  }
  push i in DQ
}
if (ans == +∞) return 0
else return ans
```

C++

```
int minSubArrayLen(int target, vector<int>& nums) {

        int n = nums.size();
        vector<long long> PS(n+1,0);
        for(int i = 1 ; i < n+1 ; ++i)
            PS[i] = PS[i-1] + nums[i-1];

        deque<int> DQ;

        int INF = numeric_limits<int>::max();

        int ans = INF;

        for (int i = 0 ; i < n+1 ; ++i){
            while (!DQ.empty() && PS[i]-PS[DQ.front()] >= k){
                ans = min(ans, i - DQ.front());
                DQ.pop_front();
            }

            while (!DQ.empty() && PS[i] < PS[DQ.back()]){
                DQ.pop_back();
            }

            DQ.push_back(i);
        }
        return (ans != INF) ? ans : -1;
}
```



```
209.minimum-size-subarray-sum.cpp    862.shortest-subarray-with-sum-at-least-k.cpp    Submission
862.shortest-subarray-with-sum-at-least-k.cpp > Solution > shortestSubarray(vector<int>&, int)
/*
 * @lc app=leetcode id=862 lang=cpp
 *
 * [862] Shortest Subarray with Sum at Least K
 */

// @lc code=start
class Solution {
public:
    int shortestSubarray(vector<int>& nums, int k) {
        int n = nums.size();
        vector<long long> PS(n+1,0);
        for(int i = 1 ; i < n+1 ; ++i)
            PS[i] = PS[i-1] + nums[i-1];

        deque<int> DQ;

        int INF = numeric_limits<int>::max();

        int ans = INF;

        for (int i = 0 ; i < n+1 ; ++i){
            while (!DQ.empty() && PS[i]-PS[DQ.front()] >=
                ans = min(ans, i - DQ.front());
                DQ.pop_front();
            }

            while (!DQ.empty() && PS[i] < PS[DQ.back()]){
                DQ.pop_back();
            }

            DQ.push_back(i);
        }
        return (ans != INF) ? ans : -1;
    }
};
Submit | Test | Debug | Debug Input
// @lc code=end
```

Accepted
- 97/97 cases passed (413 ms)
- Your runtime beats 39.11 % of cpp submissions
- Your memory usage beats 88.44 % of cpp submissions (104.7 MB)

Time complexity: *Amortized* $O(n)$

Space complexity: $O(2N)$

References:
https://www.youtube.com/watch?v=S6Xg-0uaODc
https://www.youtube.com/watch?v=K0NgGYEAkA4
https://www.youtube.com/watch?v=101kTeuKQ2M