<https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k/description/>

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

**1.The difference between L862 and L209 is L209 only contains positive elements but L862 contains negative ones, we can apply Binary Search on L209 preSum solution since its monopoly increasing array but cannot do the same for L209 as preSum array in L862 not guarantee monopoly increasing**

**2.Use sliding window technique requires only non-negative elements in an array ? And how to handle negative elements in an array if still target O(n) time complexity ?**

**Refer to chatGPT**

**Q1: For leetcode 209 we can have a sliding window solution, but for its similar problem leetcode 862 (compare to leetcode 209 pure positive elements in array, leetcode 862 contains negative elements), can we also have a sliding window solution ?**

For LeetCode 862, finding a sliding window solution is more challenging than for LeetCode 209 due to the presence of negative elements, which makes it difficult to determine when the current subarray's sum meets the target. **In LeetCode 209, we can maintain a shrinking window because every element is positive, ensuring that adding elements increases the sum and removing them decreases it, allowing us to meet the target with a minimal length window. However, when negative elements are involved, this straightforward property no longer holds.**

**In LeetCode 862, we can still use a sliding window approach with the help of a monotonic deque.** Here’s the idea:

Use a prefix sum array where each entry prefix\_sum[i] is the sum of elements from the start to index i.

Maintain a deque that stores indices of prefix\_sum values in increasing order.

For each index i, check if there’s any j in the deque such that prefix\_sum[i] - prefix\_sum[j] >= target. If found, update the minimum length of the subarray.

Then, add i to the deque and ensure the deque remains increasing by removing indices where prefix\_sum[i] is less than or equal to prefix\_sum[deque[-1]].

This approach ensures that we efficiently find the shortest subarray with a sum of at least target. Unlike a pure sliding window, we leverage the deque to handle negative numbers by tracking feasible subarray start points based on prefix sums.

**Q2: Is that means only non-negative elements in an array is the pre-condition to use sliding window technique (not using Deque) on this array ?**

**Yes, having only non-negative elements in an array is generally a precondition for using a simple sliding window technique without additional data structures like a deque.** Here’s why:

**Guaranteed Sum Increase:** In an array with only non-negative elements, adding an element to the sliding window increases the sum, and removing an element decreases the sum. This predictable behavior makes it easy to adjust the window size by expanding (adding elements) or contracting (removing elements) until the sum condition is met.

**Monotonicity:** Non-negative elements ensure that the window sum either stays the same or grows as you add more elements. This monotonicity allows us to shrink the window from the left while moving the right pointer forward to find the minimal length subarray that meets the target sum.

**When an array contains negative elements, the sum of the current window can fluctuate in unpredictable ways as you expand or contract the window, breaking the monotonicity. In such cases, simply expanding or shrinking the window won’t reliably lead to the desired sum condition. This is why, in problems like LeetCode 862, a deque or similar data structure is needed to help keep track of feasible subarray boundaries, as the prefix sums and relative comparisons provide a workaround to this unpredictability.**

In summary, non-negative elements are generally required for a "pure" sliding window approach without a deque, as they ensure predictable sum adjustments and enable minimal window tracking directly.

**Refer to**

**Algorithm: how can't I use sliding window approach for this question?**

<https://stackoverflow.com/questions/66772752/algorithm-how-cant-i-use-sliding-window-approach-for-this-question>

**Q:** I encountered this question during an interview. It gives you an array and a threshold and the function should return the length of the shortest, non-empty, contiguous subarray of that array with sum at least past that threshold.

So if the array is [2,-1,2] and threshold is 3 then it should return 3 .

Here is my attempt written in JavaScript. I was taking a typical sliding window approach, where once the sum is bigger than threshold I will decrease the window size and I keep track of the window size during each iteration.

function(array, threshold) {

let minWindowSize = Infinity

let sum = 0

for (let start = 0, end = 0; end < array.length; end++) {

sum += array[end]

if(sum >= threshold) minWindowSize = Math.min(minWindowSize, end - start + 1)

while (sum > threshold) {

sum -= array[start++]

}

}

return minWindowSize === Infinity ? -1 : minWindowSize

};

However my solution is buggy for cases like

array = [17,85,93,-45,-21]

threshold = 150

I wanted to know what differentiates this question from a typical sliding window question and is there a way to implement a sliding window approach to solve this question? It seems pretty straightforward but it turned out to be a hard question on leetcode.

**A1:** As David indicates:

You could use a sliding window if all of the array elements were nonnegative.

The problem is that, with negative elements, it's possible for a subarray to be both shorter than and

have a greater sum than another subarray.

I don't know how to solve this problem with a sliding window.

The approach that I have in mind would be to loop over the prefix sums, inserting each into a segment tree

after searching the segment tree for the most recent sum that was at least threshold smaller.

**You can't use the sliding/stretching window technique when there are negative numbers in the array, because the sum doesn't grow monotonically with the window size.**

You can still solve this in O(n log n) time, though, using a technique that is usually used for the "sliding window minimum/maximum" problem.

First, transform your array into a prefix sum array, by replacing each element with the sum of that element and all the previous ones. Now your problem changes to "find the closest pair of elements with difference >= X" (assuming array[-1]==0).

As you iterate though the array, you need to find, for each i, the latest index j such that j < i and array[j] <= array[i]-x.

In order to do that quickly, first note that array[j] will always be less than all the following elements up to i, because otherwise there would be a closer element to choose.

So, as you iterate through the array, maintain a stack of the indexes of all elements that are smaller than all the later elements you've seen. This is easy and takes O(n) time overall -- after processing each i, you just pop all the indexes with >= values, and then push i.

Then for each i, you can do a binary search in this stack to find the latest index with a small enough value. The binary search works, because the values for index in the stack increase monotonically -- each element must be less than all the following ones.

With the **binary search**, to total time increases to O(n log n).

In JavaScript, it looks like this:

var shortestSubarray = function(A, K) {

//transform to prefix sum array

let sum=0;

const sums = A.map(val => {

sum+=val;

return sum;

});

const stack=[];

let bestlen = -1;

for(let i=0; i<A.length; ++i) {

const targetVal = sums[i]-K;

//binary search to find the shortest acceptable span

//we will find the position in the stack \*after\* the

//target value

let minpos=0;

let maxpos=stack.length;

while(minpos < maxpos) {

const testpos = Math.floor(minpos + (maxpos-minpos)/2);

if (sums[stack[testpos]]<=targetVal) {

//value is acceptable.

minpos=testpos+1;

} else {

//need a smaller value - go left

maxpos=testpos;

}

}

if (minpos > 0) {

//found a span

const spanlen = i - stack[minpos-1];

if (bestlen < 0 || spanlen < bestlen) {

bestlen = spanlen;

}

} else if (bestlen < 0 && targetVal>=0) {

// the whole prefix is a valid span

bestlen = i+1;

}

// Add i to the stack

while(stack.length && sums[stack[stack.length-1]] >= sums[i]) {

stack.pop();

}

stack.push(i);

}

return bestlen;

};

**A2**: The issue with your implementation lies in the fact that the sliding window approach doesn't handle cases where negative numbers can influence the sum in unpredictable ways. This problem can't be solved with a pure sliding window technique due to the presence of negative numbers in the array, which can cause the sum to decrease unexpectedly and disrupt the "shrink window" logic.

**Correct approach: Use a deque (monotonic queue) to solve this.**

**By using a deque, we can track indices of elements in a way that allows us to maintain a window where the sum of the subarray is at least k. This solution uses prefix sums, which help efficiently compute the sum of any subarray by subtracting two prefix sums.**

function shortestSubarray(nums, k) {

const n = nums.length;

let prefixSum = Array(n + 1).fill(0);

// Compute prefix sums

for (let i = 0; i < n; i++) {

prefixSum[i + 1] = prefixSum[i] + nums[i];

}

let deque = [];

let minLength = Infinity;

for (let i = 0; i <= n; i++) {

// Try to find the smallest subarray with sum >= k

while (deque.length > 0 && prefixSum[i] - prefixSum[deque[0]] >= k) {

minLength = Math.min(minLength, i - deque.shift());

}

// Maintain the deque in increasing order of prefix sums

while (deque.length > 0 && prefixSum[i] <= prefixSum[deque[deque.length - 1]]) {

deque.pop();

}

// Add the current index to the deque

deque.push(i);

}

return minLength === Infinity ? -1 : minLength;

}

**Explanation:**

**Prefix sum:** We calculate a running sum (prefixSum[i]) so that the sum of any subarray from i to j can be computed by prefixSum[j] - prefixSum[I].

**Deque:** This stores indices of the prefix sum in increasing order. It helps us efficiently find subarrays whose sums are greater than or equal to k.

**Shrinking the window:** Whenever the difference between the current prefix sum and the smallest prefix sum in the deque is greater than or equal to k, we shrink the window from the left by removing the leftmost index from the deque.

**Time complexity:** O(n): Every element is added and removed from the deque at most once, making the solution linear in time complexity.

**Attempt 1: 2024-11-01**

**Solution 1: Brute Force (10 min, TLE 85/98)**

class Solution {

    public int shortestSubarray(int[] nums, int k) {

        int n = nums.length;

        int minLen = n + 1;

        for(int i = 0; i < n; i++) {

            long sum = 0;

            for(int j = i; j < n; j++) {

                sum += nums[j];

                if(sum >= k) {

                    minLen = Math.min(minLen, j - i + 1);

                }

            }

        }

        return (int)(minLen == n + 1 ? -1 : minLen);

    }

}

Time Complexity: O(n^2)

Space Complexity: O(1)

**Solution 2: Presum + Brute Force (10 min, TLE 85/98)**

class Solution {

    public int shortestSubarray(int[] nums, int k) {

        int n = nums.length;

        int[] presum = new int[n + 1];

        for(int i = 0; i < n; i++) {

            presum[i + 1] = presum[i] + nums[i];

        }

        int minLen = n + 1;

        for(int i = 1; i <= n; i++) {

            for(int j = i; j <= n; j++) {

                if(presum[j] - presum[i - 1] >= k) {

                    minLen = Math.min(minLen, j - i + 1);

                }

            }

        }

        return (int)(minLen == n + 1 ? -1 : minLen);

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Presum + Binary Search (Not applicable: the reason is because binary search can only apply when the array is monotonic increasing or decreasing, but since negative elements in between, the presum array not monotonic increasing or decreasing, we should not apply binary search on it)**

**Solution 3: Presum + PriorityQueue (30 min)**

class Solution {

    public int shortestSubarray(int[] nums, int k) {

        int n = nums.length;

        long[] presum = new long[n + 1];

        for(int i = 0; i < n; i++) {

            presum[i + 1] = presum[i] + nums[i];

        }

        PriorityQueue<Integer> minPQ = new PriorityQueue<>((a, b) -> Long.compare(presum[a], presum[b]));

        int minLen = n + 1;

        for(int i = 0; i <= n; i++) {

            while(!minPQ.isEmpty() && presum[i] - presum[minPQ.peek()] >= k) {

                minLen = Math.min(minLen, i - minPQ.poll());

            }

            minPQ.offer(i);

        }

        return minLen == n + 1 ? -1 : minLen;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**Solution 4: Presum + Monotonic Increasing Deque (180 min)**

class Solution {

    public int shortestSubarray(int[] nums, int k) {

        int n = nums.length;

        long[] presum = new long[n + 1];

        for(int i = 0; i < n; i++) {

            presum[i + 1] = presum[i] + nums[i];

        }

        // Monotonic Increasing Deque

        Deque<Integer> deque = new ArrayDeque<>();

        int minLen = n + 1;

        for(int i = 0; i <= n; i++) {

            // If deque.peekFirst() exists in our deque, it means that before presum[i],

            // we didn't find a subarray whose sum at least K, presum[i] is the first

            // prefix sum that valid this condition.

            // In other words, nums[deque.peekFirst()] ~ nums[i - 1] is the shortest

            // subarray starting at nums[deque.peekFirst()] with sum at least K.

            // We have already find it for nums[deque.peekFirst()] and it can't be

            // shorter, so we can drop it from our deque.

            while(!deque.isEmpty() && presum[i] - presum[deque.peekFirst()] >= k) {

                minLen = Math.min(minLen, i - deque.removeFirst());

            }

            // If presum[i] <= presum[deque.peekLast()] and moreover we already know

            // that i > deque.peekLast(), it means that compared with presum[deque.peekLast()]

            // at index 'deque.peekLast()', presum[i] at index 'i' can help us make the

            // subarray length shorter and sum bigger. So no need to keep deque.peekLast()

            // in our deque.

            // In another word, we can maintain a monotonic increasing deque based

            // on presum's elements value.

            while(!deque.isEmpty() && presum[deque.peekLast()] >= presum[i]) {

                deque.removeLast();

            }

            deque.addLast(i);

        }

        return minLen == n + 1 ? -1 : minLen;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

**From n^2 -> nlogn -> n. Step-by-step.**

<https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k/solutions/1781875/from-n-2-nlogn-n-step-by-step/>

**Take #1:**

**n^2**

n^2 is the first one to come if you don't want to end up in the situation where you have no working solution:

var n = nums.length;

var minLen = Long.MAX\_VALUE;

for (int i = 0; i < n; i++) {

var sum = 0L;

for (int j = i; j >= 0; j--) {

sum += nums[j];

if (sum >= k) {

minLen = Math.min(minLen, i - j + 1);

}

}

}

return (int)(minLen == Long.MAX\_VALUE ? -1 : minLen);

**Optimizing to O(n). Trying...**

For such kind of problems the first thing comes to my mind [209. Minimum Size Subarray Sum](https://leetcode.com/problems/minimum-size-subarray-sum/) is the "sliding window technique". We do a lot of useless repeated recalculations:

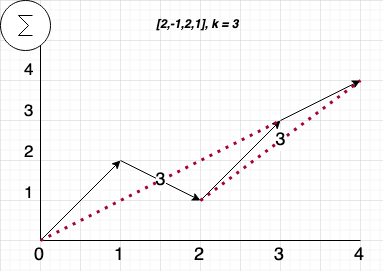
[0]+[1]+[2]+[3]

[1]+[2]+[3]

[2]+[3]

[3]

Let's look at the example [2,-1,2,1], k=3 and dry run it:



Expand right: +[0] 2, sum = 2

Expand right: +[1] -1, sum = 1

Expand right: +[2] 2, sum = 3 <-- the sliding window's invariant now 3 >= 3, minLen(3,∞)

Shrink left: -[0] -2, sum = 1

Expand right: +[3] 1, sum = 2

Here we can quickly notice that the classic variable-length sliding window approach will fail after the negative slope at [1,2]. We will get [0,3] as the shortest window instead of [2,4]. So the same approach used for 209. Minimum Size Subarray Sum won't work.

**Take #2**

**Improved n^2:**

We calculate sums in each iteration. Instead of wasting O(n) for a sum recalculation we can replace the repeated calculations with O(1) by using precalculated [prefix sums](https://www.dcc.fc.up.pt/~pribeiro/aulas/pc2021/material/prefixsums.html). This will not change O(n^2), but gives us more room for other ideas:

**Btw, in the case the prefix sum starts from 0, .... instead of nums[0].**

var prefixSum = new long[n + 1];

for (int i = 1; i <= n; i++) {

prefixSum[i] += prefixSum[i - 1] + nums[i - 1];

}

for (int i = 0; i <= n; i++) {

for (int j = i; j >= 0; j--) {

if (prefixSum[i] - prefixSum[j] >= k) {

minLen = Math.min(minLen, i - j);

}

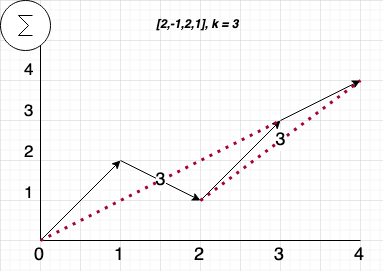
}

}

return (int)(minLen == Long.MAX\_VALUE ? -1 : minLen);

**Optimizing to O(nlog). Another approach**

Let's look at the same example again [2,-1,2,1], k=3:



We are looking for prefixSum[i] - prefixSum[j] >= k, where i > j and min(i - j). We can do it by iterating over [j,i] and ... we are back to square one - n^2.

How can we find prefixSum[i] - prefixSum[j] >= k in unordered array faster than n^2?

We can sort the sequence and turn our n^2 algo into nlogn (sort) + logn (lookup) = nlogn.

Here we can use the binary search algo or minimum heap.

Another important observation is that as there cannot be shorter sequence with the same starting point, we can remove "used" start points from next i iterations. For example: The sequence that starts @0 and ends @3. You cannot have shorter sequence >= 3 with start @0 and end > @3. Thus we can remove the smallest used starts from further lookups.

Dry run [2,-1,2,1], k=3: prefixSum[i] - k >= prefixSum[j]

Prefix sum: [0, 2, 1, 3, 4], k=3

Sort: [0, 1, 2, 3, 4], k=3

Iterating over the array o(n):

3-3 >= 0 = index of prefixSum 3 - index of prefixSum 0 = 3 - 0 = 3

and

4-3 >= 1 = index of prefixSum 4 - index of prefixSum 1 = 4 - 2 = 2

Or combine sort+iterating via minHeap:

Prefix sum: [0, 2, 1, 3, 4], k=3

i=0: [0]

i=1: [0,2] ==> 2 - 0 < k

i=2: [0,1,2] ==> 2 - 0 < k

i=3: [0,1,2,3] ==> 3 - 0 => k ==> minLen(index of prefixSum 3 - index of prefixSum 0) = 3 - 0 = 3

i=4: [1,2,3,4] ==> 4 - 1 => k ==> minLen(index of prefixSum 4 - index of prefixSum 1) = 4 - 2 = 2

var n = nums.length;

var minLen = Integer.MAX\_VALUE;

var prefixSum = new long[n + 1];

for (int i = 1; i <= n; i++) {

prefixSum[i] += prefixSum[i - 1] + nums[i - 1];

}

var minHeap = new PriorityQueue<Integer>((a,b) -> Long.compare(prefixSum[a], prefixSum[b]));

for (int i = 0; i <= n; i++) {

while (!minHeap.isEmpty() && prefixSum[i] - prefixSum[minHeap.peek()] >= k) {

minLen = Math.min(minLen, i - minHeap.remove());

}

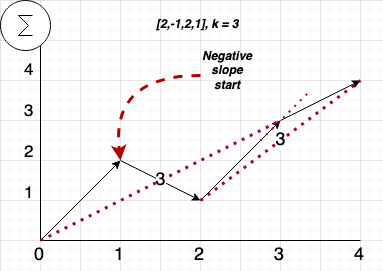
minHeap.add(i);

}

return minLen == Integer.MAX\_VALUE ? -1 : minLen;

**Optimizing O(nlogn) to O(n). Almost there...**

The same example [2,-1,2,1], k=3, and more **NOT OBVIOUS** observations:



When prefixSum[i] - prefixSum[j] <= 0 (e.g. [2] - [1] => 1 - 2 = -1) the [j,i] sequence length is increasing, but the sum is not. Thus sequences with negative prefixSum[i] - prefixSum[j] are not the shortest ones and can be ignored.

For example: [3]-[0] => 3 ==> 3-0=3. Ignore the negative slope start point [1]. The outcome has not changed. Still the same 2 sequences >= k: [0-3] and [2,4]. The only difference is that we saved time on processing negative slopes.

As a result of the observation we can ignore the negative slopes during calculations. And thus if we do not have the negative slopes, we have a non-decreasing sequence. Non-decreasing sequence does not need sorting and our algo gets rid of nlogn part.

Deque allows us to work with items from the both ends. We still remove "used" start points from the queue front and add new elemts to the end. Meanwhile also remove all the negative slope starts. This is a sliding window with monoqueue. In our case the monoqueue is the sliding window.

var n = nums.length;

var minLen = Integer.MAX\_VALUE;

var prefixSum = new long[n + 1];

for (int i = 1; i <= n; i++) {

prefixSum[i] += prefixSum[i - 1] + nums[i - 1];

}

var monoQueue = new ArrayDeque<Integer>();

for (int i = 0; i <= n; i++) {

// the same as in nlogn

while (!monoQueue.isEmpty() && prefixSum[i] - prefixSum[monoQueue.peekFirst()] >= k) {

minLen = Math.min(minLen, i - monoQueue.removeFirst());

}

// remove negative slope starts - make the sequence increasing only

while (!monoQueue.isEmpty() && prefixSum[i] - prefixSum[monoQueue.peekLast()] <= 0) {

monoQueue.removeLast();

}

monoQueue.addLast(i);

}

return minLen == Integer.MAX\_VALUE ? -1 : minLen;

**Refer to**

**[C++/Java/Python] O(N) Using Deque**

<https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k/solutions/143726/c-java-python-o-n-using-deque/>

**Prepare**

From @Sarmon:

"What makes this problem hard is that we have negative values.

If you haven't already done the problem with positive integers only,

I highly recommend solving it first"

Minimum Size Subarray Sum

**Explanation**

Calculate prefix sum B of list A.

B[j] - B[i] represents the sum of subarray A[i] ~ A[j-1]

Deque d will keep indexes of increasing B[i].

For every B[i], we will compare B[i] - B[d[0]] with K.

**Complexity:**

Every index will be pushed exactly once.

Every index will be popped at most once.

Time O(N)

Space O(N)

**How to think of such solutions?**

Basic idea, for array starting at every A[i], find the shortest one with sum at leat K.

In my solution, for B[i], find the smallest j that B[j] - B[i] >= K.

Keep this in mind for understanding two while loops.

**What is the purpose of first while loop?**

For the current prefix sum B[i], it covers all subarray ending at A[i-1].

We want to know if there is a subarray, which starts from an index, ends at A[i-1] and has at least sum K.

So we start to compare B[i] with the smallest prefix sum in our deque, which is B[D[0]], hoping that [i] - B[d[0]] >= K.

So if B[i] - B[d[0]] >= K, we can update our result res = min(res, i - d.popleft()).

The while loop helps compare one by one, until this condition isn't valid anymore.

**Why we pop left in the first while loop?**

This the most tricky part that improve my solution to get only O(N).

**D[0] exists in our deque, it means that before B[i], we didn't find a subarray whose sum at least K.**

**B[i] is the first prefix sum that valid this condition.**

**In other words, A[D[0]] ~ A[i-1] is the shortest subarray starting at A[D[0]] with sum at least K.**

**We have already find it for A[D[0]] and it can't be shorter, so we can drop it from our deque.**

**What is the purpose of second while loop?**

To keep B[D[i]] increasing in the deque.

**Why keep the deque increase?**

**If B[i] <= B[d.back()] and moreover we already know that i > d.back(), it means that compared with B[d.back()] at index 'd.back()',**

**B[i] at index 'i' can help us make the subarray length shorter and sum bigger. So no need to keep d.back() in our deque.**

**In another word, we can maintain a monotonic increasing deque based on B's elements value.**

public int shortestSubarray(int[] A, int K) {

int N = A.length, res = N + 1;

long[] B = new long[N + 1];

for (int i = 0; i < N; i++) B[i + 1] = B[i] + A[i];

Deque<Integer> d = new ArrayDeque<>();

for (int i = 0; i < N + 1; i++) {

while (d.size() > 0 && B[i] - B[d.getFirst()] >= K)

res = Math.min(res, i - d.pollFirst());

while (d.size() > 0 && B[i] <= B[d.getLast()])

d.pollLast();

d.addLast(i);

}

return res <= N ? res : -1;

}

**Refer to**

<https://algo.monster/liteproblems/862>

**Problem Description**

The problem asks us to find the length of the shortest contiguous subarray within an integer array nums such that the sum of its elements is at least a given integer k. If such a subarray does not exist, we are to return -1. The attention is on finding the minimum-length subarray that meets or exceeds the sum condition.

**Intuition**

To solve this efficiently, we utilize a monotonic queue and prefix sums technique. The intuition behind using prefix sums is that they allow us to quickly calculate the sum of any subarray in constant time. This makes the task of finding subarrays with a sum of at least k much faster, as opposed to calculating the sum over and over again for different subarrays.

A prefix sum array s is an array that holds the sum of all elements up to the current index. So for any index i, s[i] is the sum of nums[0] + nums[1] + ... + nums[i-1].

To understand the monotonic queue, which is a Double-Ended Queue (deque) in this case, let's look at its properties:

It is used to maintain a list of indices of prefix sums in increasing order.

When we add a new prefix sum, we remove all the larger sums at the end of the queue because the new sum and any future sums would always be better choices (smaller subarray) for finding a subarray of sum k.

We also continuously check if the current prefix sum minus the prefix sum at the start of the queue is at least k. If it is, we found a candidate subarray, and update ans with the subarray's length. Then we can pop that element off the queue since we've already considered this subarray and it won't be needed for future calculations.

In summary, the prefix sums help us quickly compute subarray sums, and the monotonic queue lets us store and traverse candidate subarray ranges efficiently, ensuring we always have the smallest length subarray that meets the sum condition, thus arriving at the optimal solution.

**Solution Approach**

The solution makes use of prefix sums and a monotonic queue, specifically a deque, to achieve an efficient algorithm to find the shortest subarray summing to at least k. Let's explore the steps involved:

**Prefix Sum Calculation:** We initiate the computation by creating a list called s that contains the cumulative sum of the nums list, by using the accumulate function with an initial parameter set to 0. This denotes that the first element of s is 0 and is a requirement to consider subarrays starting at index 0.

**Initialization:** A deque q is initialized to maintain the monotonic queue of indices. A variable ans is initialized to inf (infinity) which will later store the smallest length of a valid subarray.

**Iterate Over Prefix Sums:** We iterate over each value v in the prefix sums array s and its index i.

**Deque Front Comparison:** While there are elements in q and the current prefix sum v minus the prefix sum at q[0] (the front of the deque) is greater than or equal to k, we've found a subarray that meets the requirement. We then compute its length i - q.popleft() and update ans with the minimum of ans and this length. The popleft() operation removes this index from the deque as it is no longer needed.

**Deque Back Optimization:** Before appending the current index i to q, we pop indices from the back of the deque if their corresponding prefix sums are greater than or equal to v because these are not conducive to the smallest length requirement and will not be optimal candidates for future comparisons.

**Index Appending:** Append the current index i to q. This index represents the right boundary for the potential subarray sums evaluated in future iterations.

After the loop, two cases may arise:

If ans remains inf, it means no valid subarray summing to at least k was found, so we return -1.

Otherwise, we return the value stored in ans as it represents the length of the shortest subarray that fulfills the condition.

Overall, the algorithm smartly maintains a set of candidate indices for the start of the subarray in a deque, ensuring that only those that can potentially give a smaller length subarray are considered. The key here is to understand how the prefix sum helps us quickly calculate the sum of a subarray and how the monotonically increasing property of the queue ensures we always get the shortest length possible. The use of these data structures makes the solution capable of working in O(n) time complexity.

**Example Walkthrough**

Let's illustrate the solution approach with an example. Suppose we have the following array and target sum k:

nums = [2, 1, 5, 2, 3, 2]

k = 7

We want to find the length of the smallest contiguous subarray with a sum greater than or equal to 7. Here's how we would apply the solution approach:

Prefix Sum Calculation: We compute the prefix sum array s:

s = [0, 2, 3, 8, 10, 13, 15]

Notice that s[0] is 0 because we've added it artificially to account for subarrays starting at index 0.

Initialization: We create a deque q to maintain indices of prefix sums:

q = []

And initialize ans to inf:

ans = inf

Iterate Over Prefix Sums: We iterate over s, looking for subarrays that sum up to at least k.

Deque Front Comparison: As we proceed, when we reach s[3] = 8 (consider nums[0..2]), we find it is greater than or equal to k. The q is empty, so we just move on.

Deque Back Optimization: Before appending index 3 to q, we don't remove anything from q because it's still empty. So we append 3 into q.

Index Appending: Our q is now [3].

Continuing the iteration, we eventually come to s[5] = 13, which is the sum up to nums[0..4].

We have a non-empty q, and s[5] - s[q[0]] = 13 - 8 = 5, which is not greater than or equal to k. So we can't pop anything from the queue yet.

Once we reach s[6] = 15, we notice that 15 - 8 = 7 is exactly our k. We then calculate the subarray length 6 - q.popleft() = 6 - 3 = 3. Now the ans becomes 3, the smallest subarray [5, 2, 3] found so far.

Deque Back Optimization is also done each time before we append a new index, which keeps the indices in the deque monotonically increasing in sums.

After considering all elements in s, our ans is 3, as no smaller subarray summing to at least 7 is found. So we return 3 as the length of the shortest subarray. If ans was still infinity, we would return -1 indicating no such subarray was found.

**Solution Implementation**

class Solution {

// Function to find the length of the shortest subarray with a sum at least 'k'

public int shortestSubarray(int[] nums, int k) {

int n = nums.length; // Get the length of the input array

long[] prefixSums = new long[n + 1]; // Create an array to store prefix sums

// Calculate prefix sums

for (int i = 0; i < n; ++i) {

prefixSums[i + 1] = prefixSums[i] + nums[i];

}

// Initialize a deque to keep track of indices

Deque<Integer> indexDeque = new ArrayDeque<>();

int minLength = n + 1; // Initialize the minimum length to an impossible value (larger than the array itself)

// Iterate over the prefix sums

for (int i = 0; i <= n; ++i) {

// While the deque is not empty, check if the current sum minus the front value is >= k

while (!indexDeque.isEmpty() && prefixSums[i] - prefixSums[indexDeque.peek()] >= k) {

minLength = Math.min(minLength, i - indexDeque.poll()); // If true, update minLength

}

// While the deque is not empty, remove all indices from the back that have a prefix sum greater than or equal to the current sum

while (!indexDeque.isEmpty() && prefixSums[indexDeque.peekLast()] >= prefixSums[i]) {

indexDeque.pollLast();

}

// Add the current index to the deque

indexDeque.offer(i);

}

// If minLength is still greater than the length of the array, there is no valid subarray, return -1

return minLength > n ? -1 : minLength;

}

}

**Time and Space Complexity**

The given Python code is for finding the length of the shortest contiguous subarray whose sum is at least k. The code uses the concept of prefix sums and a monotonic queue to keep track of potential candidates for the shortest subarray.

**Time Complexity**

The time complexity of the code is O(N), where N is the length of the input array nums. This is because:

The prefix sum array s is computed using itertools.accumulate, which is a single pass through the array, thus taking O(N) time.

The deque q is maintained by iterating through each element of the array once. Each element is added and removed from the deque at most once. Since the operations of adding to and popping from a deque are O(1), the loop operations are also O(N) in total.

Inside the loop, q.popleft() and q.pop() are each called at most once per iteration, and as a result, each element in nums contributes at most O(1) to the time complexity.

**Space Complexity**

The space complexity of the code is O(N) as well:

The prefix sum array s requires O(N) space.

The deque q potentially stores indices from the entire array in the worst-case scenario. In the worst case, it could hold all indices in nums, also requiring O(N) space.

Apart from these two data structures, the code uses a constant amount of space for variables such as ans and v, which does not depend on the input size.

**Refer to**

[L1425.Constrained Subsequence Sum (Ref.L239,L739,L53,L862)](note://WEBa39329fbc7421070539b51823f6f484d)

[L209.P2.2.Minimum Size Subarray Sum (Ref.L862)](note://195893F25BDC45F2B63535FC5D9A4B26)