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

Given an array of positive integers nums and a positive integer target, return the minimal length of a **contiguous subarray** [numsl, numsl+1, ..., numsr-1, numsr] 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 <= 109

1 <= nums.length <= 105

1 <= nums[i] <= 104

 **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)).

**Solution 1: Brute Force (2022-09-07, 10min)**

class Solution {

    public int minSubArrayLen(int target, int[] nums) {

        int minLen = Integer.MAX\_VALUE;

        int len = nums.length;

        int i = 0;

        int sum = 0;

        for(int j = 0; j < len; j++) {

            sum += nums[j];

            // Keep shrink left end

            while(sum >= target) {

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

                sum -= nums[i];

                i++;

            }

        }

        return minLen == Integer.MAX\_VALUE ? 0 : minLen;

    }

}

Space Complexity: O(1)

Time Complexity: O(n)

<https://leetcode.com/problems/minimum-size-subarray-sum/discuss/59078/Accepted-clean-Java-O(n)-solution-(two-pointers)/924580>

For those trying to figure out how is it O(n):

Here we have defined 2 index i & j,

In case of O(n^2) for each outer loop, inner loop runs some n or m number of times to make it O(nm), that means, as soon as the outer loop finishes one iteration, inner loop resets itself.

In case of O(n2), as in this case, we are not resetting the inner inner variable i, it's just incrementing each time. It is like 2 loops one after another and both runs n number of time.

**Follow up: Time complexity O(n log(n)) solution, Prefix Sum + Binary Search**

**Solution 2: Prefix Sum + Binary Search (2022-09-09, 360min, too long for understanding binary search template)**

class Solution {

    public int minSubArrayLen(int s, int[] nums) {

        int len = nums.length;

        int[] preSum = new int[len + 1];

        //preSum[0] = 0;

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

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

        }

        // 根据模板第一种情况，给定一个升序排列的数组，我们将preSum中满足 x ≥ target 的

        // 第一个元素定义为「下界」在加入无法找到就返回-1这种情况以后类型更接近模板第三种情况：

        // 查找指定值第一次出现的位置，修正为查找满足 x ≥ target 的第一个元素，如果因为下标

        // 越界或仍在界内却依然有 x < target, 则判定为不存在，返回-1。

        // e.g

        // nums = {2,3,1,2,4,3}, s = 7

        // preSum = {0,2,5,6,8,12,15}

        // i = 0, target = 7, binarySearch(0,6,7,preSum) -> low\_bound = 4, minLen = 4(low\_bound - i = 4 - 0 = 4, mapping nums subarray = {2,3,1,2})

        // i = 1, target = 9, binarySearch(1,6,9,preSum) -> low\_bound = 5, minLen = 4(low\_bound - i = 5 - 1 = 4, mapping nums subarray = {3,1,2,4})

        // i = 2, target = 12, binarySearch(2,6,12,preSum) -> low\_bound = 5, minLen = 3(low\_bound - i = 5 - 2 = 3, mapping nums subarray = {1,2,4})

        // i = 3, target = 13, binarySearch(3,6,13,preSum) -> low\_bound = 6, minLen = 3(low\_bound - i = 6 - 3 = 3, mapping nums subarray = {2,4,3})

        // i = 4, target = 15, binarySearch(4,6,15,preSum) -> low\_bound = 6, minLen = 2(low\_bound - i = 6 - 4 = 2, mapping nums subarray = {4,3}) ==> answer

        // i = 5, target = 19, binarySearch(5,6,19,preSum) -> low\_bound = NA, minLen = NA

        int minLen = Integer.MAX\_VALUE;

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

            int target = preSum[i] + s;

            int low\_bound = binarySearch(i, len, target, preSum);

            if(low\_bound != -1) {

                minLen = Math.min(minLen, low\_bound - i);

            }

        }

        return minLen == Integer.MAX\_VALUE ? 0 : minLen;

    }

// Binary Search template refer to

    // https://imageslr.com/2020/03/15/binary-search.html

    private int binarySearch(int lo, int hi, int target, int[] preSum) {

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(preSum[mid] >= target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Important: In case we could not find target in preSum return -1

        // Test out when i = 5, target = 19, binarySearch(5,6,19,preSum),

        // 'lo' increase from 5 to 7 and index out of boundary, no valid

        // range between 'lo' and 'hi' (actually 'lo' is already over

        // 'hi') and still not able to find preSum[mid] >= target.

        // we can ignore the second condition as 'preSum[lo] < target' since

        // first condition as 'lo >= preSum.length' automatically means 'lo'

        // go through all indexes and no 'preSum[lo] < target' found

        // if(lo >= preSum.length || preSum[lo] < target)

        if(lo >= preSum.length) {

            return -1;

        }

        return lo;

    }

}

Space Complexity: O(n)

Time Complexity: O(nlogn)

**Note: Why we can use binary search ?**

<https://leetcode.com/problems/minimum-size-subarray-sum/discuss/59103/Two-AC-solutions-in-Java-with-time-complexity-of-N-and-NLogN-with-explanation>

As to NLogN solution, logN immediately reminds you of binary search. In this case, you cannot sort as the current order actually matters. How does one get an ordered array then?

**Since all elements are positive, the cumulative sum must be strictly increasing. Then, a subarray sum can expressed as the difference between two cumulative sum. Hence, given a start index for the cumulative sum array, the other end index can be searched using binary search.**

**Solution 3: Not fixed length Sliding Window (10 min)**

class Solution {

    public int minSubArrayLen(int target, int[] nums) {

        int n = nums.length;

        int sum = 0;

        int minLen = n + 1;

        // Not fixed length sliding window:

        // i is the start-pointer, j is the end-pointer of the sliding window.

        int i = 0;

        for(int j = 0; j < n; j++) {

            sum += nums[j];

            // Shrink the window from the left ('i') until the sum is smaller than the target.

            // This finds the smallest window that ends at position 'j'.

            while(sum >= target && i < n) {

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

                sum -= nums[i];

                i++;

            }

        }

        // If minLen is updated (smaller than n + 1), we found a valid subarray.

        // Otherwise, return 0 as a subarray meeting the conditions is not found.

        return minLen <= n ? minLen : 0;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**The difference between L209 and L862 is L862 contains negative elements but L209 only contains positive elements**

**Refer to**

<https://leetcode.com/problems/minimum-size-subarray-sum/solutions/433123/java-c-python-sliding-window/>

**Intuition**

[L862.Shortest Subarray with Sum at Least K (Ref.L209,L1425)](note://WEBfed451877e2413f5a50e9fddac1fc127)

Actually I did this first, the same prolem but have negatives, I suggest solving this prolem first then take 862 as a follow-up.

**Explanation**

The result is initialized as res = n + 1.

One pass, remove the value from sum s by doing s -= A[j].

**If s <= 0, it means the total sum of A[i] + ... + A[j] >= sum that we want, then we update the res = min(res, j - i + 1).**

Finally we return the result res

public int minSubArrayLen(int s, int[] A) {

int i = 0, n = A.length, res = n + 1;

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

s -= A[j];

while (s <= 0) {

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

s += A[i++];

}

}

return res % (n + 1);

}

**Refer to**

<https://algo.monster/liteproblems/209>

**Problem Description**

The problem is essentially about finding the shortest continuous sequence of elements within an array of positive integers such that the sum of that sequence is at least as large as a given target value. To clarify, a subarray is defined as a contiguous part of an array. The task is to find the length of the smallest subarray that meets or exceeds the target sum. If no such subarray exists, the function should return zero.

For example, if the input array is [2, 3, 1, 2, 4, 3] and the target is 7, the subarray [4, 3] has the smallest length that sums up to 7 or more, so the answer would be the length of this subarray, which is 2.

**Intuition**

To solve this problem efficiently, we need to avoid checking all possible subarrays one by one since doing so would result in a very slow solution when dealing with a large array. The intuitive insight here is that we can do much better by using a "[sliding window](https://algo.monster/problems/sliding_window_maximum)" approach. This approach involves maintaining a window that expands and contracts as we iterate through the array to find the smallest window that satisfies the condition.

Here's the thought process behind the [sliding window](https://algo.monster/problems/sliding_window_maximum) solution:

We start with two pointers, both at the beginning of the array. These pointers represent the margins of our current window.

We move the end pointer to the right, adding numbers to our current window's sum.

As soon as the window's sum becomes equal to or greater than the target, we attempt to shrink the window from the left to find smaller valid windows that still meet the sum criterion.

Each successful window gives us a potential answer (its size), we keep track of the minimum size found.

We continue this process until the end pointer reaches the end of the array, and there are no more subarrays to check.

If the minimum length of the window is never updated from the initial setting that is larger than the array length, it means no valid subarray has been found, and we should return 0.

By using this approach, we can ensure that we only traverse the array once, giving us an efficient solution with a time complexity of O(n), where n is the length of the input array.

**Solution Approach**

The provided solution uses the **Sliding Window** pattern to solve the problem efficiently. This approach is useful when you need to find a subarray that meets certain criteria, and the problem can be solved in linear time without the need to check every possible subarray individually.

Here's how the [sliding window](https://algo.monster/problems/sliding_window_maximum) algorithm is implemented in the solution:

Initialize two pointers, j at 0 to represent the start of the window and i which will move through the array.

Maintain a running sum, s, of the values within the current window which starts at 0.

Iterate over the array using i and continuously add the value of nums[i] to s.

Inside the loop, use a while loop to check if the current sum s is greater than or equal to the target. If it is, attempt to shrink the window from the left by:

Updating the minimum length of the valid window if necessary using ans = min(ans, i - j + 1).

Subtracting nums[j] from the sum s since the left end of the window is moving to the right.

Incrementing j to actually move the window's start to the right.

Once the end of the array is reached and there are no more elements to add to the sum, check if ans was updated or not. If ans is still greater than the length of the array n, it means no valid subarray was found, so we return 0.

If ans was updated during the process (meaning a valid subarray was found), return the value of ans which holds the length of the smallest subarray with a sum of at least target.

The use of two pointers to create a window that slides over the array allows this algorithm to run in O(n) time, making it very efficient for this type of problem.

Please note that the Reference Solution Approach mentions another method which is using **PreSum & Binary Search** but the provided code doesn't implement this method. The **PreSum & Binary Search** method involves creating an array of prefix sums and then using binary search to find the smallest valid subarray for each element. This is a bit more complex and generally not as efficient as the **Sliding Window** method used here which requires only O(n) time and O(1) extra space.

**Example Walkthrough**

Let's use a small example to illustrate the sliding window approach described in the solution. Suppose we have the array [1, 2, 3, 4, 5] and our target sum is 11.

We want to find the smallest subarray whose sum is at least 11. We'll follow the steps of the sliding window algorithm:

Initialize the pointers i and j both to 0, and the running sum s also to 0.

Start iterating over the array with i. Our window size is currently 0.

First iteration (i = 0):

Add nums[i] to s. Now, s is 1.

It's less than the target (11), so we move on to the next number.

Second iteration (i = 1):

Now, s is 1 + 2 = 3.

Still less than the target.

Third iteration (i = 2):

s becomes 1 + 2 + 3 = 6.

Again, we continue since s is less than our target sum.

Fourth iteration (i = 3):

s is now 1 + 2 + 3 + 4 = 10.

It is still below the target sum of 11.

Fifth iteration (i = 4):

s is now 1 + 2 + 3 + 4 + 5 = 15, which is greater than our target of 11. We've now found a subarray [1, 2, 3, 4, 5] with the sum greater than or equal to the target.

Now we try to shrink the window from the left to see if there is a smaller subarray that still meets or exceeds the target sum.

Before moving j, we update the answer ans to the current window size, which is i - j + 1 = 5 - 0 + 1 = 5.

Now we enter the while loop to shrink the window since s >= target:

We reduce s by nums[j]. s becomes 15 - 1 = 14, and we increment j to 1. The window is now [2, 3, 4, 5].

s is 14, which is still greater than 11, so we repeat the procedure.

s becomes 14 - 2 = 12 after removing the next element and j goes to 2. The window is [3, 4, 5].

With s at 12, it's still greater than 11, continue to shrink.

We subtract 3 to get s = 12 - 3 = 9 and move j to 3. Now s is less than 11, so we stop shrinking the window.

Our smallest subarray that meets the requirement so far is [3, 4, 5] with a length of 3. Since we've already reached the end of the array, we're done iterating, and we can return the answer, which is 3.

This example successfully demonstrated the sliding window technique where we expanded the window until we exceeded the target sum, then shrank the window from the left to find the smallest subarray that still meets the sum condition. The array [3, 4, 5] is the smallest subarray with a sum greater than or equal to the target 11, so the result is the length of this subarray, which is 3.

**Solution Implementation**

class Solution {

// This method finds the minimum length of a subarray that sums to at least the given target.

public int minSubArrayLen(int target, int[] nums) {

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

long sum = 0; // The sum of the current subarray.

int minLength = n + 1; // Initialize minLength with max possible value plus one for comparison.

// Two pointers method: i is the end-pointer, j is the start-pointer of the sliding window.

for (int end = 0, start = 0; end < n; ++end) {

sum += nums[end]; // Increment the sum by the current element value.

// Shrink the window from the left until the sum is smaller than the target.

// This finds the smallest window that ends at position 'end'.

while (start < n && sum >= target) {

minLength = Math.min(minLength, end - start + 1); // Update minLength if a smaller length is found.

sum -= nums[start++]; // Decrease the sum by the start-value and increment start-pointer to shrink the window.

}

}

// If minLength is updated (smaller than n + 1), we found a valid subarray.

// Otherwise, return 0 as a subarray meeting the conditions is not found.

return minLength <= n ? minLength : 0;

}

}

**Time and Space Complexity**

The time complexity of the code is O(n), where n is the length of the input list nums. This is because there are two pointers i and j, both of which travel across the list at most once. The inner while loop only increases j and decreases the sum s until the sum is less than the target, but j can never be increased more than n times throughout the execution of the algorithm. Therefore, each element is processed at most twice, once when it is added to s and once when it is subtracted, leading to a linear time complexity.

The space complexity of the code is O(1), which means it requires a constant amount of additional space. This is because the algorithm only uses a fixed number of single-value variables (n, ans, s, j, i, x) and does not utilize any data structures that grow with the size of the input.

**Refer to**

[L862.Shortest Subarray with Sum at Least K (Ref.L209,L1425)](note://WEBfed451877e2413f5a50e9fddac1fc127)

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)