<https://leetcode.com/problems/minimum-operations-to-reduce-x-to-zero/description/>

You are given an integer array nums and an integer x. In one operation, you can either remove the leftmost or the rightmost element from the array nums and subtract its value from x. Note that this **modifies** the array for future operations.

Return **the minimum number of operations to reduce**x **to exactly** 0 **if it is possible, otherwise, return**-1.

**Example 1:**

**Input:** nums = [1,1,4,2,3], x = 5

**Output:** 2

**Explanation:** The optimal solution is to remove the last two elements to reduce x to zero.

**Example 2:**

**Input:** nums = [5,6,7,8,9], x = 4

**Output:** -1

**Example 3:**

**Input:** nums = [3,2,20,1,1,3], x = 10

**Output:** 5

**Explanation:** The optimal solution is to remove the last three elements and the first two elements (5 operations in total) to reduce x to zero.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^4

1 <= x <= 10^9

**Attempt 1: 2024-05-04**

**Wrong Solution 1: sum == target condition before while loop**

**Test case:**

**Input: nums = [3,2,20,1,1,3], x = 10, Output = -1, Expected = 5**

class Solution {

    public int minOperations(int[] nums, int x) {

        int n = nums.length;

        int sum = 0;

        for(int j = 0; j < n; j++) {

            sum += nums[j];

        }

        int target = sum - x;

        int maxLen = -1;

        int i = 0;

        sum = 0;

        for(int j = 0; j < n; j++) {

            sum += nums[j];

            if(sum == target) {

                maxLen = Math.max(maxLen, j - i + 1);

            }

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

                sum -= nums[i];

                i++;

            }

        }

        return maxLen < 0 ? -1 : n - maxLen;

    }

}

**Issue in the Wrong Solution**

Here’s the problematic part of the wrong code:

sum += nums[j];

if(sum == target) {

maxLen = Math.max(maxLen, j - i + 1);

}

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

sum -= nums[i];

i++;

}

**What Happens**

The if(sum == target) condition checks if the current subarray has the desired sum **before** **adjusting the window using the while loop**.

If the current sum is equal to target, the subarray length is updated. However, this check occurs **before reducing the window**, which can cause **valid subarrays to be missed when sum > target and then try to meet the target by forwarding left boundary (shrink the window) in certain iterations**.

Like the test case identify the issue here: **Input: nums = [3,2,20,1,1,3], x = 10, Output = -1, Expected = 5**

In correct solution, we have below order, when sum = 3 + 2 + 20 = 25 > 20 in 3rd iteration, it hits sum(=25) > target(=20) condition, we will try to shrink the window by moving forward left boundary, which removing 3 and 2 in sequence, and 25 - 3 - 2 = 20, hence sum(=20) <= target(=20) terminate the while loop, and following if condition also match, now we find a solution.

sum += nums[j];

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

sum -= nums[i];

i++;

}

if(sum == target) {

maxLen = Math.max(maxLen, j - i + 1);

}

In wrong solution, we have below order, sum(3 -> 5 -> 25) never match target(=20), so we cannot find a result till now, but till 3rd iteration, sum as 25, it hits sum(=25) > target(=20) condition, we will try to shrink the window by moving forward left boundary, which removing 3 and 2 in sequence, and 25 - 3 - 2 = 20, hence sum(=20) <= target(=20) terminate the while loop, **but there is NO following if condition to check when sum match target happening after while loop, it directly moves forward to next iteration, and sum change from 20 to 20 + 1 = 21... etc, the sum never hits back to 20 again**, so we could never find a solution.

sum += nums[j];

if(sum == target) {

maxLen = Math.max(maxLen, j - i + 1);

}

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

sum -= nums[i];

i++;

}

**Wrong Solution 2: sum == target condition inside while loop**

**Test case:**

**Input: nums = [1,1,4,2,3], x = 5, Output = 3, Expected = 2**

class Solution {

    public int minOperations(int[] nums, int x) {

        int n = nums.length;

        int sum = 0;

        for(int j = 0; j < n; j++) {

            sum += nums[j];

        }

        int target = sum - x;

        int maxLen = -1;

        int i = 0;

        sum = 0;

        for(int j = 0; j < n; j++) {

            sum += nums[j];

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

                sum -= nums[i];

                i++;

                if(sum == target) {

                    maxLen = Math.max(maxLen, j - i + 1);

                }

            }

        }

        return maxLen < 0 ? -1 : n - maxLen;

    }

}

**What Happens:**

The if(sum == target) check is **inside** the while loop that reduces the window size. This causes valid subarrays to be missed because the check is only triggered **while shrinking the window**.

When sum > target, the loop keeps shrinking the window and reduces sum, but the if(sum == target) check is skipped for valid subarrays that are identified **before** entering the loop.

Like the test case identify the issue here: **Input: nums = [1,1,4,2,3], x = 5, Output = 3, Expected = 2**

In correct solution, we have below order, when sum = 1 + 1 + 4 = 6 in 3rd iteration, it matches sum(=6) == target(=6) condition, we don't need to shrink the window by moving forward left boundary by now since sum > target condition not triggered, and following if condition directly match, now we find a solution as j - i + 1 = 2 - 0 + 1 = 3, maxLen = 3, in later process we will find sum = 6 + 2 = 8, it hits sum(=8) > target(=6) condition to trigger the while loop, in while loop shrink left boundary two times, then find another solution as j - i + 1 = 3 - 2 + 1 = 2, maxLen = 2 < previous maxLen = 3, we keep original solution as 3

sum += nums[j];

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

    sum -= nums[i];

    i++;

}

if(sum == target) {

    maxLen = Math.max(maxLen, j - i + 1);

}

In wrong solution, when sum = 1 + 1 + 4 = 6 in 3rd iteration, since sum not > target(=6), not trigger while loop, we will miss the solution as j - i + 1 = 2 - 0 + 1 = 3, maxLen = 3 scenario, and only able to find the later solution as j - i + 1 = 3 - 2 + 1 = 2, maxLen = 2, which like previously mentioned: **The if(sum == target) check is inside the while loop that reduces the window size. This causes valid subarrays to be missed because the check is only triggered while shrinking the window.**

sum += nums[j];

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

    sum -= nums[i];

    i++;

    if(sum == target) {

        maxLen = Math.max(maxLen, j - i + 1);

    }

}

**So be careful of the Sliding Window template usage: the later process step should out of while loop**

**Solution 1: Not fixed length Sliding Window (60 min)**

class Solution {

    public int minOperations(int[] nums, int x) {

        int n = nums.length;

        int sum = 0;

        for(int num : nums) {

            sum += num;

        }

        int target = sum - x;

        int curSum = 0;

        int maxLen = -1;

        int i = 0;

        for(int j = 0; j < n; j++) {

            curSum += nums[j];

            // i can equal to j which means use up all elements currently have, if only set as i < j, test out by below case:

            // Input: nums = [8828,9581,49,9818,9974,9869,9991,10000,10000,10000,9999,9993,9904,8819,1231,6309] and x = 134365

            // Output: -1

            // Expected: 16

            while(i <= j && curSum > target) {

                curSum -= nums[i];

                i++;

            }

            // check 'curSum == target' must after while loop, test out by below case:

            // Input: nums = [3,2,20,1,1,3] and x = 10

            // Output: -1

            // Expected: 5

            if(curSum == target) {

                maxLen = Math.max(maxLen, j - i + 1);

            }

        }

        return maxLen < 0 ? -1 : n - maxLen;

    }

}

Time Complexity

The provided code has a time complexity of O(n), where n is the length of the input list nums.

This is because the code uses a two-pointer approach (variables i and j) that iterates through

the list only once. The inner while loop adjusts the second pointer (denoted by j), but it does

not iterate more than n times across the entire for loop due to the nature of the two-pointer

strategy—each element is visited by each pointer at most once.

Space Complexity

The space complexity of the code is O(1). Outside of the input data, the algorithm uses only

a constant number of additional variables (x, ans, n, s, j, i, and v). The space required for

these variables does not scale with the size of the input list; hence, the space complexity

is constant.

**Refer to**

<https://leetcode.com/problems/minimum-operations-to-reduce-x-to-zero/solutions/936074/JavaPython-3-Sliding-window:-Longest-subarray-sum-to-the-target-sum(nums)-x./>

**Using sliding window to find the longest subarry that sums to sum(nums) - x.**

class Solution {

public int minOperations(int[] nums, int x) {

int target = -x;

// Calculate the negative target by adding all the numbers in 'nums' array

// Since we are looking for a subarray with the sum that equals the modified 'target'

for (int num : nums) {

target += num;

}

int n = nums.length;

int minimumOperations = Integer.MAX\_VALUE;

int sum = 0;

// Two pointers approach to find the subarray with the sum equals 'target'

for (int left = 0, right = 0; left < n; ++left) {

// Add the current element to 'sum'

sum += nums[left];

// If 'sum' exceeds 'target', shrink the window from the right

while (right <= left && sum > target) {

sum -= nums[right];

right++;

}

// If a subarray with sum equals 'target' is found, calculate the operations required

if (sum == target) {

minimumOperations = Math.min(minimumOperations, n - (left - right + 1));

}

}

// If no such subarray is found, return -1

// Otherwise, return the minimum number of operations

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

}

}

**Refer to**

<https://leetcode.com/problems/minimum-operations-to-reduce-x-to-zero/solutions/2136570/change-your-perspective-java-explanation/>

**Logic**

Initially, this question might look like a DP problem: pick either left or right recursively and collect the number of operations along the way. A brute force approach would result in an O(2^n) complexity which isn't great. Memoization would improve this but we'll exceed the memory limit as I've tested. Besides, we can do better.

By simply reframing the question, we can come up with an O(n) complexity solution. The difficulty in this question arises from deciding whether to choose the left or the right element to remove at any given point. However, while we're removing elements from the array, we're definitely going to be letting some elements remain as well. In particular, the middle elements.

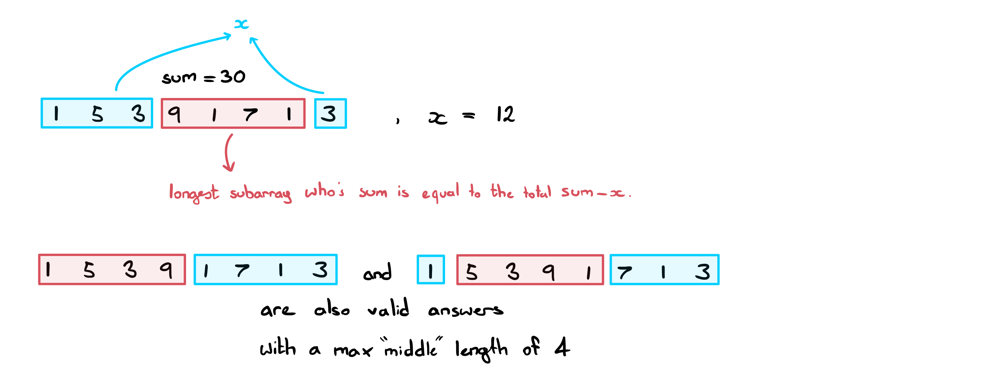
**Key observation:**

The number of elements removed equals n minus the number of elements that aren't removed.

Therefore, to find the minimum number of elements to remove, we can find the maximum number of elements to not remove!

So, instead of trying to find the minimum number of operations, why don't we focus on finding the longest subarray in the middle. One main thing to note is that our subarray should sum to sum - x (where sum is the sum of all elements in our array).

Why? because the middle elements are technically the ones we don't want. If the sum of the outer elements equals x, then we're looking for a middle sum of sum - x. If this doesn't quite make sense, pay attention to the below diagram with nums = [1,5,3,9,1,7,1,3], x = 12:



(Keep in mind that this approach only works due to the absence of negative numbers!)

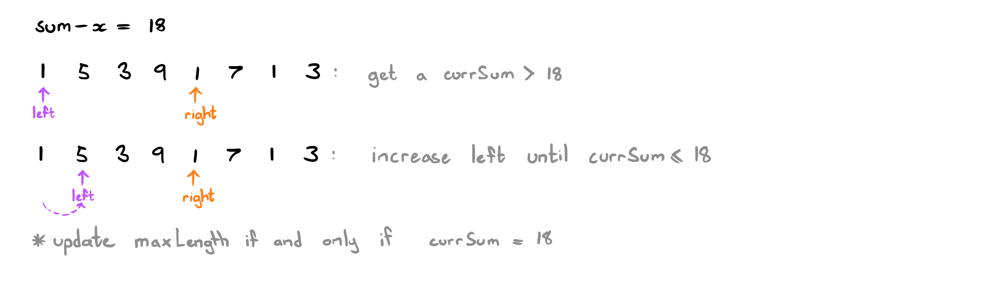
Finding our max middle subarray length:

We can find this using a two-pointer / sliding window approach. Start left and right both at index 0.

Increment right until the current window sum is > sum - x

Update our maxLength if and only if our current window sum == sum - x

Repeat the above steps until we reach the end of the array.



Awesome! Now we have everything we need to start coding.

**How should I approach this question in an interview?**

When you're presented this question, it might not be obvious that there exists a solution other than DP (at least, it wasn't obvious for me!). So, how would you know to approach the question through any other means? What saved me when doing this question was the simple realisation that removing edge values is the same as shrinking the array. Hence, there will always remain a middle subarray of length >= 0.

A habit that I've developed over time whenever I run into trouble coming up with a solution is to reframe the question. Often times when you're given a question in an interview, it's riddled with questionable requirements or red herrings. The interviewer is likely expecting you to extract out the core requirements from the question and truncate it down to as simple a question as you can manage.

It does take time and practice to spot the owl from the trees. That being said, here are some effective ways to ease yourself into the right mindset:

If a question is implicitly asking for A when the question is actually solved using B, draw examples and look for alternate patterns.

Pay attention to the constraints of the inputs/outputs (in this question, the values being strictly positive was a hint!)

Be creative! Even try rewording the question to something as different as you can think of while still maintaining the requirements of the question.

If all else fails, pay attention to the subtle hints your interviewer might be throwing your way. (By the way, the best way to get hints from your interviewer is to explain all your thoughts to them as you go. If they don't know where you're stuck, they can't help you).

**Code:**

If you have any questions, suggestions or improvements, feel free to let me know. Thanks for reading!

public int minOperations(int[] nums, int x) {

int sum = 0;

for (int num: nums) sum += num;

int maxLength = -1, currSum = 0;

for (int l=0, r=0; r<nums.length; r++) {

currSum += nums[r];

while (l <= r && currSum > sum - x) currSum -= nums[l++];

if (currSum == sum - x) maxLength = Math.max(maxLength, r-l+1);

}

return maxLength == -1 ? -1 : nums.length - maxLength;

}

**Refer to**

[L918.Maximum Sum Circular Subarray (Ref.L1658)](note://WEB1495ae8db0086f5ab30de99c6528274c)

[L1423.Maximum Points You Can Obtain from Cards (Ref.L1658)](note://WEBb70d72cb469c39161ffaeb2d08c64a44)