1658. Minimum Operations to Reduce X to Zero

Binary Search Prefix Sum Sliding Window

# **Problem Description**

Hash Table

<u>Array</u>

the array in each operation. By doing so, you decrease the value of x by the value of the element you removed. This process changes the array for the next operation since the element is no longer part of it. The challenge is to find the minimum number of operations needed to reduce x to exactly 0. It's a puzzle where you must choose

You are given an array of integers called nums and another integer x. Your goal is to remove either the first or last element from

which side to take numbers from at each step. However, the array has a fixed order, so you cannot rearrange it. If it's possible to reduce x to 0, you should return the minimum number of operations you need. If it's not possible to do so, then the answer is -1.

Intuition

The initial intuition might be to always choose the larger number between the first and last element of the array to subtract from

#### x. However, this greedy approach might not lead to the optimal solution. Instead, a different perspective can provide a more systematic method: instead of thinking in terms of our original target x, think about the rest of the numbers that will remain in the

Medium

array nums after all the operations. The key insight is to transform the problem into finding the longest subarray in nums that sums up to a new target, which is sum(nums) - x. This approach flips the original problem on its head—you're now looking for the subarray that remains rather than the numbers to reduce  $\times$  to zero. In the code, an inf (infinity) is initially set to represent an impossible situation. A two-pointer approach is then used to find the

longest subarray whose sum equals sum(nums) - x. As we iterate through the array with a variable i, indicating the right end of our subarray, a variable s (the running sum) is continuously updated. A second pointer j represents the potential left end of our subarray. We make sure to adjust j and s whenever our current running sum surpasses our new target, by subtracting values

while moving j to the right. When we find a subarray that sums to the new target, we calculate the number of operations it would take to remove the other elements by subtracting the length of this subarray from the total number of elements n. This operation count is kept if it is smaller than our current best answer in variable ans. Once the loop is finished, if ans remains as infinity, it means no suitable subarray was found, and we return -1. Otherwise, we

Solution Approach The solution follows a sliding window approach, which is a pattern used when we need to find a range or a subarray in an array that satisfies certain conditions. The idea behind a sliding window approach is to maintain a subset of items from the array as the

return the value of ans, which represents the minimum number of operations required to reduce x to exactly 0.

window and slide it to explore different subsets, according to conditions given in the problem.

In this specific problem, the target condition for the subarray is that its elements' sum should equal sum(nums) - x. The reason for this is that the sum of elements removed (to reduce x to 0) and the sum of elements that remain (the subarray) should equal

### Here is a step-by-step explanation of the code: Calculate the new target x as the sum of all elements in nums minus the original x. This represents the sum of the subarray

result.

x = 10

**Example Walkthrough** 

nums = [3, 2, 20, 1, 1, 3]

Start iterating over the array, beginning with i = 0:

 $\circ$  j is incremented to 2 (removing nums [1]), so the window is [20], and s = 20.

 $\circ$  First iteration (i=0): window is [3], s = 3.

Second iteration (i=1): window is [3, 2], s = 5.

• Third iteration (i=2): window is [3, 2, 20], s = 25.

provides us with an O(n) complexity for the problem.

def minOperations(self, nums: List[int], x: int) -> int:

for right index, value in enumerate(nums):

current sum -= nums[left\_index]

Solution Implementation

target = sum(nums) - x

current\_sum = left\_index = 0

current\_sum += value

left\_index += 1

if current sum == target:

return −1 if answer == inf else answer

int minimumOperations = Integer.MAX\_VALUE;

// Add the current element to 'sum'

if (!prefixSumIndex.count(sum)) {

// Check if (sum - target) has been observed before

int startIdx = prefixSumIndex[sum - target];

// Calculate the target sum by subtracting x from the total sum of nums

let target = nums.reduce((acc, current) => acc + current, 0) - x;

if (prefixSumIndex.count(sum - target)) {

function minOperations(nums: number[], x: number): number {

prefixSumIndex[sum] = i;

sum -= nums[right];

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

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

**Python** 

from math import inf

class Solution:

we want to find.

minimized as we find valid subarrays.

subtracting nums[j] from s).

the sum of the original array.

Start iterating through the array with index i representing the end boundary of the sliding window. Add nums[i] to the sum S.

Set n to the length of nums, s to 0 as the current sum, and j to 0 as the starting index of our sliding window.

Initialize ans with the value inf to represent a very high number of operations (assumed to be impossible) that will be

While the current sum s exceeds the target x, adjust the window by moving the start j to the right (increasing j and

When a subarray sum equals the target, update ans with the minimum of its current value and n - (i - j + 1). This

calculation gives us the number of operations because i - j + 1 is the length of our subarray, and subtracting this from the

At the end, if no valid subarray is found (meaning ans is still inf), return -1. Otherwise, return the value of ans as the final

- total length n gives the number of operations needed to achieve the original x. Continue the steps 4-6 until all elements have been visited.
- The use of a sliding window ensures that we check every possible subarray in an efficient manner, only making a single pass through the array, which gives us a time complexity of O(n).

Let's take a small example to illustrate the solution approach. Consider the following array nums and integer x:

The sum of nums is 3 + 2 + 20 + 1 + 1 + 3 = 30, so our new target is sum(nums) - x which is 30 - 10 = 20.

Now we apply the sliding window approach: Initialize ans with a value representing infinity, s (current sum) to 0, j (window start index) to 0, and n (the length of nums)

#### At this point, our sum s exceeds our target 20. We need to adjust the window: $\circ$ j is incremented to 1 (removing nums [0] from sum), so the window is [2, 20], and s = 22.

to 6.

Continue the algorithm with the rest of nums, checking for a longer subarray that sums to 20, but no such longer subarray will

needed to remove the other elements: n - (i - j + 1) which is 6 - (2 - 2 + 1) = 5.

be found. The window would shrink and grow as the sum oscillates relative to our target.

and 2) and the last three elements (1, 1, and 3) while keeping the subarray [20], which adds up exactly to our target of 20. Hence, the minimum number of operations required to reduce  $\mathbf{x}$  to 0 in this example is 5. The sliding window approach makes this process efficient by ensuring that we only need to scan through the array once, which

We have found a subarray that sums up to our target (20), which is [20]. We update ans with the number of operations

At the end of the process, we will find that the smallest ans was 5, which corresponds to removing the first two elements (3

# Initialize variables: set answer to infinity to represent initially impossible scenario # `n` holds the length of the nums list; `current\_sum` and `left\_index` for the sliding window answer = infn = len(nums)

# Since we're looking for the minimum operations, we subtract the length of current subarray from total length

# Calculate new target which is the sum of elements that remain after removing the elements summing to x.

# Use a sliding window approach to find if there's a continuous subarray summing to `target`

# Expand the window by adding the current value to the `current\_sum`

while left index <= right index and current\_sum > target:

# Update the answer with the minimum number of operations

# If `current sum` matches `target`, we found a valid subarray

answer = min(answer, n - (right\_index - left\_index + 1))

# Shrink the window from the left as long as `current sum` exceeds `target`

# If answer has not been updated, return -1, otherwise return the minimum operations

Java class Solution { public int minOperations(int[] nums, int x) {

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

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

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

// Record the first occurrence of this sum. This prevents overwriting previous indices.

// Calculate operations as the difference between the current ending index and

// the starting index of the subarray, adjusting for whole array length.

minOperations = std::min(minOperations, n - (i - startIdx));

// If minOperations hasn't changed, return -1 to indicate no solution found

return minOperations == std::numeric\_limits<int>::max() ? -1 : minOperations;

```
// 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 target = -x;

int n = nums.length;

sum += nums[left];

right++;

int sum = 0;

```
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;
C++
#include <vector>
#include <unordered map>
#include <numeric>
#include <algorithm>
class Solution {
public:
    // Calculate the minimum number of operations to reduce the sum of the array
    // by 'x' by either removing elements from the start or end.
    int minOperations(std::vector<int>& nums. int x) {
        // Compute the new target which is the desired sum of the subarray we want to find
        int target = std::accumulate(nums.begin(), nums.end(), 0) - x;
       // If target is negative, no solution is possible as we cannot create a negative sum.
        if (target < 0) {
            return -1;
        // Special case: if target is 0, it means we want to remove the whole array
        if (target == 0) {
            return nums.size();
        // Use a hashmap to store the cumulative sum at each index
        std::unordered map<int, int> prefixSumIndex;
        prefixSumIndex[0] = -1; // Initialize hashmap with base case
        int n = nums.size(); // Size of the input array
        // Initialize the answer with a large number, signifying that we haven't found a solution yet
        int minOperations = std::numeric limits<int>::max();
        for (int i = 0, sum = 0; i < n; ++i) {
            sum += nums[i]; // Cumulative sum up to the current index
```

```
let minimumOps = Infinity; // Initialize with a large number representing infinity
// Initialize two pointers for the sliding window and a sum to hold the window's sum
```

**}**;

**TypeScript** 

const length = nums.length;

```
let windowStart = 0:
    let currentSum = 0;
   // Loop over the array to find the maximum-sized subarray that adds up to target
    for (let windowEnd = 0; windowEnd < length; ++windowEnd) {</pre>
       // Add the current element to windowSum
        currentSum += nums[windowEnd];
       // Shrink the window from the start if the currentSum exceeds the target
       while (windowStart <= windowEnd && currentSum > target) {
            currentSum -= nums[windowStart++];
       // If the currentSum equals the target, update minimumOps if the found window is better
       if (currentSum == target) {
           minimumOps = Math.min(minimumOps, length - (windowEnd - windowStart + 1));
   // Return minimum operations required, or -1 if the target sum isn't achievable
   return minimumOps == Infinity ? -1 : minimumOps;
from math import inf
class Solution:
   def minOperations(self, nums: List[int], x: int) -> int:
       # Calculate new target which is the sum of elements that remain after removing the elements summing to x.
       target = sum(nums) - x
       # Initialize variables: set answer to infinity to represent initially impossible scenario
       # `n` holds the length of the nums list; `current_sum` and `left_index` for the sliding window
       answer = inf
       n = len(nums)
       current_sum = left_index = 0
       # Use a sliding window approach to find if there's a continuous subarray summing to `target`
       for right index, value in enumerate(nums):
           # Expand the window by adding the current value to the `current_sum`
           current_sum += value
           # Shrink the window from the left as long as `current sum` exceeds `target`
           while left index <= right index and current_sum > target:
                current sum -= nums[left_index]
                left_index += 1
```

#### answer = min(answer, n - (right\_index - left\_index + 1)) # If answer has not been updated, return -1, otherwise return the minimum operations return -1 if answer == inf else answer

if current sum == target:

# If `current sum` matches `target`, we found a valid subarray

# Update the answer with the minimum number of operations

## **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

Time and Space Complexity

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** 

two-pointer approach (variables i and j) that iterates through the list only once. The inner while loop adjusts the second

# Since we're looking for the minimum operations, we subtract the length of current subarray from total length

The space complexity of the code is 0(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.