2145. Count the Hidden Sequences

Medium Array **Prefix Sum**

Problem Description

array gives you the difference between each pair of consecutive elements in the hidden sequence. That is, for each i in the differences array, differences[i] = hidden[i + 1] - hidden[i].

You are presented with an array differences which is used to calculate the values of a hidden sequence hidden. The differences

The hidden sequence itself is not known, but it does have length n + 1, where n is the length of the differences array. The

problem also stipulates that any number in the hidden sequence must be within an inclusive range denoted by two given integers,

Your task is to find out how many different hidden sequences can be possibly made that adhere to the differences provided and also stay within the boundaries given by lower and upper. If no such sequences can be made, your answer should be 0. Intuition

lower and upper.

To solve this problem, we think about the constraints that the <u>lower</u> and upper bounds put on the possible values of the hidden

minimum and maximum values that occur if we start the sequence at zero. These minimum and maximum values are offset versions of what the real sequence could look like. Knowing the most extreme (minimum and maximum) values that our sequence reaches, we can calculate how many different starting points for the hidden sequence are possible that would keep the sequence within the bounds of lower and upper.

Essentially, we are sliding the window of the extreme values of our simulated sequence along the scale of lower to upper and

sequence. Since we know the differences between each pair of consecutive values, we can simulate the sequence to find the

checking for overlaps. Since we are given that the hidden sequence values must be in the range [lower, upper] (inclusive), we subtract the maximum value we found from upper and the result is the span of numbers that could potentially be the start of a hidden sequence. We also subtract the range span of the hidden sequence (max - min), so we get the number of sequences that can be formed. Adding 1

accounts for the inclusive boundaries. If the span is negative, that means there are no possible sequences, so we use max function to set the count to 0 in such cases. The one-liner calculation in the provided solution performs this sliding window computation to find the count of valid starting numbers, and thus, by extension, the count of valid sequences.

To implement the solution, we make use of a simple linear scan algorithm to iterate through the differences array and accumulate the changes to calculate the possible minimum and maximum values that the hidden sequence can take.

We start by initializing a variable num to 0, which represents the current value of the hidden sequence, assuming it starts at 0.

Alongside, we initialize two other variables, mi and mx, which stand for the minimum and the maximum values that we

update mi and mx with these two lines:

return max(0, upper - lower - (mx - mi) + 1)

that we do not return a negative number of possible sequences.

start at 0), and the minimum and maximum values we encounter.

possible sequences within the bounds [1, 6]:

for the maximum value encountered.

range_width = max_value - min_value

for diff in differences:

return num_of_arrays

current_sum = min_value = max_value = 0

Iterate through each difference in the array

max_value = max(max_value, current_sum)

#include <vector> // Include necessary header for using vectors

#include <algorithm> // Include necessary header for using min and max functions

Starting at 1: [1, -1, -2, 0, 1]

from typing import List

class Solution:

scan through the differences array once to arrive at the solution.

Solution Approach

encounter as we construct the hidden sequence. Both are initially set to 0. Then, we iterate over the differences array, and for each difference d, we add d to num. As we simulate the sequence, we

- mi = min(mi, num) mx = max(mx, num)If num is lesser than our current minimum, we update mi to reflect num, and similarly, if num is greater than our current maximum, we update mx to be num.
- After we finish iterating through all the elements in differences, we will have the most extreme values that the hidden sequence could attain if it started at 0. Now, we must consider the given bounds lower and upper.

```
The formula to calculate the number of valid starting points for the hidden sequence, and hence the number of possible
```

sequences, is:

are inclusive.

We subtract mx - mi from upper - lower to get the number of positions between lower and upper that could be the start of the hidden sequence, while still ensuring all of its values do not breach the bounds. We add 1 because both lower and upper

If the result of the subtraction is less than 0, it implies that there's no possible starting point that keeps the sequence within the bounds, therefore there are 0 possible sequences. We use the max function to handle this scenario, which helps to ensure

Example Walkthrough Let's consider the differences array as [-2, -1, 2, 1], with the bounds lower equal to 1, and upper equal to 6.

We start by initializing our variables num, mi, and mx to 0. These will keep track of our current sequence value (assuming we

This approach is efficient, with a time complexity of O(n) where n is the length of the differences array, since we only need to

We iterate through the differences array: ∘ For the first difference, -2, we update num to 0 - 2 = -2. We also update mi to -2 (since -2 is less than the old minimum 0) and mx remains as 0.

\circ For the third difference, 2, num becomes -3 + 2 = -1. mi remains as -3 and mx remains as 0. \circ For the last difference, 1, num is now -1 + 1 = 0. Again mi and mx remain as -3 and 0, respectively.

After iterating through the array, we have mi = -3 and mx = 0. Our hidden sequence, if starting at 0, would range in values between -3 and 0.

 \circ We first calculate the span of numbers between lower and upper: upper - lower which is 6 - 1 = 5. \circ Next, we find the span of our hidden sequence by computing mx - mi: 0 - (-3) = 3.

The final result is 3. Meaning we can have 3 different possible starting points for the hidden sequence that would keep the

sequence within the bounds given by lower and upper. The valid sequences would start at 1, 2, and 3, leading to the following

○ To find how many sequences can fit, we subtract the hidden sequence span from the bounds span and add 1: (5 - 3) + 1 = 3.

Starting at 2: [2, 0, -1, 1, 2] Starting at 3: [3, 1, 0, 2, 3]

def numberOfArrays(self, differences: List[int], lower: int, upper: int) -> int:

Initialize the variables: current_sum to track the running sum,

min_value to keep the minimum value encountered, and max_value

Calculate the width of the range spanned by the differences

Calculate the total number of distinct arrays that can be formed

within the given upper and lower bounds. Here we also include the

 \circ For the second difference, -1, num becomes -2 - 1 = -3. mi is updated to -3 and mx remains as 0.

We now compare the span of our possible hidden sequence values with the given bounds:

- Each starting value leads to a sequence that, when applying the differences, remains within the bounds of 1 to 6. Hence, there are 3 valid hidden sequences.
- Solution Implementation **Python**

Add the current difference to the running sum current_sum += diff # Update the minimum value if the new current_sum is lower min_value = min(min_value, current_sum) # Update the maximum value if the new current_sum is higher

```
# '+ 1' offset to account for inclusive bounds.
# If the resulting number is negative, we use max(0, ...) to default to 0.
# This represents cases where no valid arrays can be formulated.
num_of_arrays = max(0, (upper - lower) - range_width + 1)
```

Java

class Solution {

```
/**
    * Calculate the number of valid arrays that can be constructed with the given conditions.
    * @param differences An array of integers representing the difference between consecutive elements in the target array.
    * @param lower The lower bound for the elements of the target array.
    * @param upper The upper bound for the elements of the target array.
    * @return The number of valid arrays that can be constructed.
    public int numberOfArrays(int[] differences, int lower, int upper) {
       // Initialize running sum, minimum and maximum values observed while simulating the array creation
        long runningSum = 0;
        long minObserved = 0;
        long maxObserved = 0;
       // Iterate over the array of differences
        for (int difference : differences) {
           // Update the running sum with the current difference
            runningSum += difference;
           // Update the minimum observed sum, if necessary
           minObserved = Math.min(minObserved, runningSum);
           // Update the maximum observed sum, if necessary
           maxObserved = Math.max(maxObserved, runningSum);
       // Compute the number of possible starting values that satisfy the bounds
       int totalValidArrays = Math.max(0, (int) (upper - lower - (maxObserved - minObserved) + 1));
       // Return the computed total number of valid arrays
       return totalValidArrays;
C++
```

```
class Solution {
public:
    int numberOfArrays(vector<int>& differences, int lower, int upper) {
        long long runningSum = 0; // This will keep track of the accumulated sum of differences
        long long minSum = 0; // This will keep the minimum sum encountered
        long long maxSum = 0; // This will keep the maximum sum encountered
        // Iterate over the differences array
        for (int &difference : differences) {
            runningSum += difference; // Accumulate the sum of differences
            minSum = std::min(minSum, runningSum); // Update the minimum sum if necessary
            maxSum = std::max(maxSum, runningSum); // Update the maximum sum if necessary
        // Calculate the range of the final array values
        long long validRange = upper - lower - (maxSum - minSum) + 1;
        // If the range is negative, set it to zero
        return std::max(0LL, validRange);
};
TypeScript
// TypeScript does not have a standard header system like C++,
// so you don't 'include' modules. Instead, you import them if necessary.
// In this case, no import is needed since arrays and Math functions are built-in.
// A global variable to keep track of the accumulated sum of differences
let runningSum: number = 0;
// A global variable to keep track of the minimum sum encountered
let minSum: number = 0;
// A global variable to keep track of the maximum sum encountered
let maxSum: number = 0;
// Function to calculate the number of valid arrays from the given differences
// and the bounds provided by lower and upper limits
function numberOfArrays(differences: number[], lower: number, upper: number): number {
    // Reset the global variables for a new function call
    runningSum = 0;
```

```
maxSum = 0;
      // Iterate over the differences array
      for (let difference of differences) {
          // Accumulate the sum of differences
          runningSum += difference;
          // Update the minimum and maximum sums if necessary
          minSum = Math.min(minSum, runningSum);
          maxSum = Math.max(maxSum, runningSum);
      // Calculate the range of the final array values
      let validRange: number = upper - lower - (maxSum - minSum) + 1;
      // Return the number of valid arrays, ensuring the number is not negative
      return Math.max(0, validRange);
from typing import List
class Solution:
   def numberOfArrays(self, differences: List[int], lower: int, upper: int) -> int:
       # Initialize the variables: current_sum to track the running sum,
       # min_value to keep the minimum value encountered, and max_value
       # for the maximum value encountered.
        current_sum = min_value = max_value = 0
       # Iterate through each difference in the array
        for diff in differences:
            # Add the current difference to the running sum
            current_sum += diff
           # Update the minimum value if the new current_sum is lower
            min_value = min(min_value, current_sum)
            # Update the maximum value if the new current_sum is higher
            max_value = max(max_value, current_sum)
       # Calculate the width of the range spanned by the differences
        range_width = max_value - min_value
       # Calculate the total number of distinct arrays that can be formed
       # within the given upper and lower bounds. Here we also include the
       # '+ 1' offset to account for inclusive bounds.
       # If the resulting number is negative, we use max(0, ...) to default to 0.
       # This represents cases where no valid arrays can be formulated.
        num of arrays = max(0, (upper - lower) - range width + 1)
       return num_of_arrays
```

Time and Space Complexity

minSum = 0;

The given Python function number of Arrays calculates how many valid arrays can be generated from a list of differences within the given upper and lower bounds. Time Complexity: The time complexity of the numberOfArrays function is O(n), where n is the length of the differences list. This

minimum, maximum) at each step. Space Complexity: The space complexity of the function is 0(1) as the function uses a fixed number of integer variables (num, mi,

mx) and does not allocate any additional space that grows with the input size. The space used for the input differences list does not count towards the space complexity of the function itself as it is provided as input.

is because we iterate through each element of differences exactly once, performing constant-time operations (addition,