1413. Minimum Value to Get Positive Step by Step Sum

Easy Array Prefix Sum Leetcode Link

Problem Description

In this problem, we are given an array of integers called `nums`. We are to imagine that we start with an initial positive value referred to as `startValue`. In a series of iterations, we keep adding the `startValue` to the sum of the elements in the array `nums`, one element at a time, starting from the first element of the array and moving to the right.

The core challenge is to find the smallest positive value for `startValue` such that, when we keep adding it to the sums in each iteration, the resulting sum after each addition never drops below 1. This is essential to ensure that the cumulative sum is positive at all times.

Intuition

To arrive at the solution, we need to think backwards from the requirement that the cumulative sum should never be less than 1. This implies that at every step of adding the array elements to `startValue`, the resultant sum must be greater than or equal to 1.

If we process the array and keep a running sum, this sum will fluctuate up and down as we add positive and negative numbers. Our goal is to find out how negative this running sum can get, because that tells us how much `startValue` we need initially to offset that negative dip and stay positive overall.

The intuition is to keep two trackers: `s`, which is the running sum, and `t`, which holds the smallest value that `s` has reached. As we iterate over `nums`, we update `s` to be the sum so far, and `t` becomes the minimum of its current value and `s`. The reason for this is that `t` helps us find the lowest point our cumulative sum reaches.

Once we finish going through the array, we check the value of `t`. If `t` is negative, it means that the running sum went below 1 at some point, and therefore, we must set `startValue` to be at least `1 - t` to compensate for that dip and ensure the sum never falls below 1. If `t` is 0 or positive, we don't need any additional positive `startValue`, so the answer is simply 1 (since `startValue` must be positive).

Thus, the final answer is the maximum of 1 and `1 - t` to ensure we always return a positive `startValue`.

Solution Approach

The implementation of the solution follows a simple algorithm that leverages the concept of a running sum and a minimum tracker. We do not need any complex data structures as the calculations can be done with simple variables that hold integral values.

Here is a step-by-step explanation of how the algorithm in the reference solution works:

1. Initialize two variables, `s` and `t`. `s` will keep track of the running sum and is initially set to 0. `t` is our minimum tracker that records the minimum value that `s` attains while iterating through the array; it is initially set to Python's `inf` (infinity) to ensure it starts higher than any possible sum.
2. Iterate over the array `nums`. For each element `num` in `nums`:
   - Update the running sum `s` by adding the current element `num` to it: `s += num`.
   - Update the tracker `t` to be the minimum of its current value and the updated running sum `s`: `t = min(t, s)`.
3. After iterating through the entire array, we have the lowest point our running sum has reached stored in `t`. We use `t` to calculate the minimum positive `startValue` by taking the maximum between 1 and `1 - t`. If `t` is negative or zero, `1 - t` will ensure that `startValue` is enough to keep the sum positive. If `t` is already positive, we just need a `startValue` of 1.

The mathematical formula used to calculate the required `startValue` is:

`startValue = max(1, 1 - t)`

The algorithm is efficient with a time complexity of O(n), where n is the length of `nums`, since it requires a single pass through the array. The space complexity is O(1) since we only use a fixed amount of extra space regardless of the size of the input array.

The implementation does not make use of any specific patterns or advanced algorithms; it is a straightforward linear scan with constant updates to two tracking variables, capitalizing on the simple but important observation that the lowest point in the running sum dictates the initial positive value required to maintain a positive cumulative sum.

Example Walkthrough

Let's illustrate the solution approach using a simple example. Suppose we have an array `nums` with the integers: `[-3, 2, -3, 4, 2]`. We need to determine the minimum positive `startValue` such that when added to the cumulative sum of the array, the sum never falls below 1.

1. We initialize our two variables `s` and `t`. At the beginning, `s` is 0 and `t` is set to infinity.

   ```
   s = 0, t = inf
   ```

2. We iterate over the array `nums`. Let's walk through each step:

   - First element (`-3`): `s` becomes `0 + (-3) = -3`. We update `t` to be the minimum of `inf` and `-3`, which is `-3`.

     ```
     s = -3, t = -3
     ```

   - Second element (`2`): `s` becomes `-3 + 2 = -1`. `t` is updated to the minimum of `-3` and `-1`, so it stays at `-3`.

     ```
     s = -1, t = -3
     ```

   - Third element (`-3`): `s` is `-1 + (-3) = -4`. `t` becomes the minimum of `-3` and `-4`, which is `-4`.

     ```
     s = -4, t = -4
     ```

   - Fourth element (`4`): `s` is `-4 + 4 = 0`. `t` remains `-4` as it is the minimum value reached.

     ```
     s = 0, t = -4
     ```

   - Fifth element (`2`): `s` is `0 + 2 = 2`. There's no change to `t` since `s` has not gone below `-4`.

     ```
     s = 2, t = -4
     ```

3. After iterating through the array, the lowest value of the running sum `t` is `-4`. To ensure that the cumulative sum never falls below 1, we need to start with `1 - (-4) = 5` as our `startValue`.

   ```
   startValue = max(1, 1 - (-4)) = max(1, 5) = 5
   ```

Therefore, given the array `nums` of `[-3, 2, -3, 4, 2]`, the smallest positive `startValue` so that the cumulative sum never drops below 1 is 5.

Python Solution

```python
class Solution:
    def minStartValue(self, nums: List[int]) -> int:
        # Initialize the sum and the minimum prefix sum.
        # `current_sum` denotes the running sum of the numbers.
        # `min_prefix_sum` is the minimum of all prefix sums encountered.
        current_sum, min_prefix_sum = 0, float('inf')

        # Iterate through each number in the list of numbers.
        for num in nums:
            # Update the running sum with the current number.
            current_sum += num

            # Update the minimum prefix sum if the current running sum is less than the recorded minimum.
            min_prefix_sum = min(min_prefix_sum, current_sum)

        # Calculate the minimum start value where the prefix sum never drops below 1.
        # The minimum start value is the difference between 1 and the minimum prefix sum recorded,
        # ensuring the starting value makes the smallest prefix sum equal to 1 at least.
        # If the minimum prefix sum is zero or positive, the start value is at least 1.
        start_value = max(1, 1 - min_prefix_sum)

        return start_value

```

Java Solution

```java
class Solution {

    // Function to find the minimum start value for a given array
    public int minStartValue(int[] nums) {
        // Initialize sum to accumulate the values in the array
        int sum = 0;

        // Initialize minSum to keep track of the minimum sum encountered
        int minSum = Integer.MAX_VALUE;

        // Iterate through each element in the array
        for (int num : nums) {
            // Update the running sum by adding the current element
            sum += num;

            // Update minSum if the current sum is less than the previous minSum
            minSum = Math.min(minSum, sum);
        }

        // Calculate the minimum start value.
        // If minSum is negative or zero, the minimum start value must be at least 1 - minSum
        // to ensure the running sum never drops below 1.
        // Otherwise, if minSum is positive, the minimum start value is 1.
        return Math.max(1, 1 - minSum);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <climits>

class Solution {
public:
    // Function to calculate the minimum start value to ensure the step-wise sum is always positive
    int minStartValue(vector<int>& nums) {
        int currentSum = 0;   // To hold the running sum of elements
        int minSum = INT_MAX; // Initialized to the maximum possible value of int as a starting point to track the minimum sum encour

        // Iterate through each number in the vector
        for (int num : nums) {
            currentSum += num;         // Add the current number to the running sum
            minSum = min(minSum, currentSum); // Update minSum with the smallest sum seen so far
        }

        // Calculate the minimum starting value.
        // If minSum is negative or zero, the minimum starting value will be 1 minus the smallest sum to make the starting sum positi
        // If minSum is already positive, the minimum starting value should be 1.
        return max(1, 1 - minSum);
    }
};

```

Typescript Solution

```typescript
// Function to determine the minimum starting value needed
// for the cumulative sum of the array to always be positive
function minStartValue(nums: number[]): number {
    let cumulativeSum = 0;     // Initialize the cumulative sum of array elements
    let minCumulativeSum = 0;  // The minimum cumulative sum found

    // Iterate over each number in the array
    for (const num of nums) {
        cumulativeSum += num;  // Update the cumulative sum with the current number
        // Update the minimum cumulative sum if the current cumulative is lower
        minCumulativeSum = Math.min(minCumulativeSum, cumulativeSum);
    }

    // Calculate and return the minimum starting value that ensures the cumulative
    // sum never drops below 1. If the minimum cumulative sum is less than 0, we
    // offset it by adding 1. Otherwise, if the minimum cumulative sum is non-negative,
    // the minimum starting value is 1.
    return Math.max(1, 1 - minCumulativeSum);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by the for loop that iterates over each element in the input list `nums`. The operations inside the loop (addition, min function, and assignment) are constant time operations that do not depend on the size of the input list. Therefore, the time complexity is `O(n)`, where `n` is the length of `nums`.

Space Complexity

The space complexity of the function is `O(1)`, which means it uses a constant amount of additional space regardless of the input size. It only requires a fixed number of variables (`s` and `t`) to store the running sum and the minimum sum found, and these do not scale with the input size.