1785. Minimum Elements to Add to Form a Given Sum

Medium Greedy Array

Leetcode Link

Problem Description

The problem provides an array of integers `nums`, where each integer satisfies the condition `abs(nums[i]) <= limit`. You are also given two integers, `limit` and `goal`. The task is to find the minimum number of elements that need to be added to the array so that the sum of the array equals the `goal` while still maintaining the condition that the absolute value of any element does not exceed `limit`.

An important thing to note is how absolute values work: `abs(x) = x` if `x >= 0`, and `abs(x) = -x` otherwise. This tells us that the elements we may add can be as large as `limit` but no larger.

Intuition

The intuition behind the solution is to find the difference `d` between the current sum of the array `sum(nums)` and the `goal`. This difference tells us what the deficit or surplus is relative to the `goal`. If the current sum is less than the `goal`, it means we have a deficit and we need to add elements to reach the `goal`. If it's greater, we need to subtract elements, but since we can only add elements, we think of what we can add to balance the excess (which mathematically is the same as subtracting to reach a lower `goal`).

After finding the difference `d`, we consider the maximum value we are allowed to add, which is `limit`. To minimize the number of elements we add, we should add elements of value `limit` until we reach or exceed the `goal`. However, since we might not reach the `goal` exactly, we might need to add one last element with a value smaller than `limit` to make the sum exactly equal the `goal`.

Mathematically, the minimum number of elements we need is the total difference divided by the `limit`, but since we're dealing with integers and we want to cover any remainder, we have to use the ceiling of the division, which is achieved by adding `limit - 1` to the difference before dividing. This is encapsulated in `(d + limit - 1) // limit`.

So the approach is:

1. Calculate the difference `d` between the current sum and the `goal`.
2. Divide `d` by `limit` using the ceiling of the division to account for any remainder (since we can only add whole elements and need to reach or exceed `goal`).
3. The result of this division gives the minimum number of elements needed to be added.

This is precisely what the implementation does in an efficient and concise manner.

Solution Approach

The implementation of the solution is straightforward and takes advantage of the mathematical foundation laid out in the intuition.

Here's the step-by-step breakdown of the algorithm using the provided Python code:

1. Calculate the difference `d` between the current sum of elements in `nums` and the `goal`. This is achieved using the `sum` function and `abs` to ensure the difference is positive regardless of whether the sum is below or above the goal:

```
d = abs(sum(nums) - goal)
```

The `abs` function is crucial here because it ensures that the difference is treated as a positive value, which aligns with our need to either add or subtract (handled as adding a negative value when the sum is above the goal) to reach the exact goal value.

2. Compute the minimum number of elements to add by dividing the difference `d` by `limit`. Since we have to deal with the possibility of having a non-zero remainder, we aim for the ceiling of the division by adding `limit - 1` before performing integer division:

```
(d + limit - 1) // limit
```

The `//` operator in Python indicates floor division, which would normally take the floor of the division result. However, the trick of adding `limit - 1` effectively changes the division result to a ceiling operation for positive numbers, ensuring that if there's a remainder, we count an additional element.

No specific data structures are used apart from the basics provided by Python, and the pattern applied here is purely mathematical. The algorithm's complexity is O(n) due to the summation of the elements in the array, but the actual calculation of the result is done in constant time, O(1). This makes the solution very efficient for inputs where `n`, the number of elements in `nums`, is not excessively large.

Example Walkthrough

Let's apply the solution approach to a small example for clarification. Suppose we have an array `nums = [1, 2, 3]`, a `limit` of `3`, and a `goal` of `10`. We want to find out how many minimum additional elements we need to add to `nums` so that the sum of the array equals `goal`, without adding any element with an absolute value greater than `limit`.

1. First, we calculate the current sum of the array: `sum(nums) = 1 + 2 + 3 = 6`.
2. We find the difference `d` between the current sum and the `goal`: `d = abs(6 - 10) = 4`, since the sum is below the goal.
3. Now, we determine the smallest number of elements of value up to the `limit` that need to be added to reach the goal. Since we want to add as few elements as possible, we will add elements with the maximum value allowed, which is `limit` (3 in this case).
4. We calculate this by dividing `d` by `limit` and using the ceiling of the division to get the minimum number of additional elements:
   - We add `limit - 1` to `d` before the division to account for any remainder: `d + limit - 1 = 4 + 3 - 1 = 6`.
   - Now, we perform the integer division by `limit`: `(6) // 3 = 2`.

So, we need to add at least 2 elements to reach the `goal`. The elements we can add are `[3, 1]`, where we're adding the maximum value `limit` (3) first and then topping it off with a final element (1) to reach the exact `goal`. After adding these numbers to our original array, `nums` now looks like `[1, 2, 3, 3, 1]` and the sum is `10`, which is equal to the `goal`.

Therefore, using the solution approach outlined, we have determined that the minimum number of elements we need to add to the `nums` array to reach the `goal` is `2`.

Python Solution

```python
from typing import List

class Solution:
    def minElements(self, nums: List[int], limit: int, goal: int) -> int:
        # Calculate the difference between the sum of the array and the goal
        difference = abs(sum(nums) - goal)

        # The number of elements required is the ceiling of 'difference / limit'
        # which is computed using (difference + limit - 1) // limit to avoid floating point division
        min_elements_needed = (difference + limit - 1) // limit

        return min_elements_needed
```

Java Solution

```java
class Solution {

    /**
     * Finds the minimum number of elements with value 'limit' that
     * must be added to the array to reach the 'goal' sum.
     *
     * @param nums  The array of integers.
     * @param limit The maximum value that could be added to or subtracted from the sum.
     * @param goal  The target sum.
     * @return The minimum number of elements needed to reach the goal.
     */
    public int minElements(int[] nums, int limit, int goal) {
        // Variable to store sum of the elements in the array.
        long sum = 0;

        // Loop to calculate the cumulative sum of the array elements.
        for (int number : nums) {
            sum += number;
        }

        // Calculate the difference between current sum and goal, using absolute value
        // because we can add or subtract elements to reach the goal.
        long difference = Math.abs(sum - goal);

        // Compute the minimum number of elements needed with value 'limit' to cover the difference.
        // The addition of 'limit - 1' is for upward rounding without using Math.ceil().
        return (int) ((difference + limit - 1) / limit);
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric> // For accumulate

class Solution {
public:
    // This function returns the minimum number of elements with the
    // value 'limit' that must be added to the array to reach the goal
    int minElements(vector<int>& nums, int limit, int goal) {
        // Calculate the current sum of the array using long long for large sums
        long long currentSum = accumulate(nums.begin(), nums.end(), 0LL);

        // Calculate the absolute difference needed to reach the goal
        long long differenceToGoal = abs(goal - currentSum);

        // Calculate the minimum number of elements needed by dividing the difference
        // by the limit and taking the ceiling of that value.
        // (The ceil is implicitly done by adding limit - 1 before division)
        // This is because we need to round up to make sure any remaining part of
        // the difference is covered, even if it's less than the limit value.
        int minElementsNeeded = (differenceToGoal + limit - 1) / limit;

        // Return the calculated number of minimum elements needed
        return minElementsNeeded;
    }
};
```

Typescript Solution

```typescript
/**
 * Calculates the minimum number of elements with the given limit to add to or subtract
 * from an array to achieve a specific goal sum.
 *
 * @param nums - The array of numbers representing the current elements.
 * @param limit - The limit of the absolute value of each element that can be added.
 * @param goal - The desired sum to be reached.
 * @returns The minimum number of elements required to reach the goal.
 */
function minElements(nums: number[], limit: number, goal: number): number {
    // Calculate the current sum of the array.
    const currentSum = nums.reduce((accumulator, value) => accumulator + value, 0);

    // Determine the absolute difference between current sum and goal.
    const differenceToGoal = Math.abs(goal - currentSum);

    // Calculate the minimum number of elements required to reach or surpass
    // the absolute difference by dividing by the limit and rounding up.
    // We subtract one before dividing to handle cases where the difference
    // is an exact multiple of the limit.
    const minimumElementsRequired = Math.ceil(differenceToGoal / limit);

    return minimumElementsRequired;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the `nums` array. This is because the code must sum up all elements in the array which takes `O(n)` time.

The space complexity of the code is `O(1)`, as the additional space used by the function does not depend on the input size and is limited to a fixed number of integer variables (`d` and the return value).