1300. Sum of Mutated Array Closest to Target

Medium Array Binary Search Sorting Leetcode Link

Problem Description

Given an integer array `arr` and a target value `target`, the goal is to find an integer `value` such that when we replace all the integers in the array that are greater than `value` with `value` itself, the sum of the modified array is as close as possible to the `target`. In terms of absolute difference, we're looking to minimize the discrepancy between the sum of the modified array and the `target`. If there are multiple `value` that achieve the same minimum absolute difference, the problem asks us to return the smallest such integer. Also, it's important to note that the integer `value` we are looking for does not necessarily have to be an element already present in the array `arr`.

Intuition

The intuition behind the solution involves understanding that as we increase the `value` used to replace elements in the array, the sum of the array will increase in a step-wise fashion until it reaches a point where increasing `value` further won't change the sum since there won't be any larger elements left to replace. To find the integer `value` that brings the sum closest to the target, we can follow these steps:

1. **Sort the Array**: Sorting `arr` allows us to efficiently work through the array to find the point where changing greater elements to a certain `value` will affect the sum.

2. **Accumulate the Sum**: We use a prefix sum array `s` to keep a running total, which helps us quickly calculate the sum of the array when we hypothetically replace elements greater than the current `value`.

3. **Iterate and Simulate**: Iterate through potential values from `0` to the `max(arr)` and for each value, we simulate what the sum would be if all the elements larger than this `value` were replaced by it. We use binary search (`bisect_right`) to efficiently find the index where the elements start to be larger than the current `value` being considered.

4. **Compare Differences**: For each `value`, we compare the absolute difference between the target and the simulated sum. We keep track of the smallest difference and the associated `value`.

5. **Return the Best Answer**: Once we have considered all relevant values, we return the `ans` that corresponds to the smallest difference.

By iterating through the array and keeping track of the differences, we can find the best `value` that minimizes the absolute difference to the target. The choice of binary search for finding the right index is crucial for efficiency, and sorting the array beforehand makes such an approach possible.

Solution Approach

The solution approach implements the following steps in Python code, each relying on algorithms and data structures that ensure efficient execution:

1. **Sort the Array**: We begin by sorting the array `arr` in increasing order. This is done with `arr.sort()`, which uses an efficient sorting algorithm such as Timsort (Python's default sorting algorithm) that has a complexity of O(n log n) where n is the number of elements in `arr`.

2. **Accumulate the Sum**: We then create a prefix sum array `s` using the `accumulate` function with an initial value of 0 (`initial=0`). The prefix sum array holds the cumulative sum of elements, and by including `initial=0`, we ensure that `s[0]` is `0`, representing that no elements have been summed yet. This array allows us to quickly calculate the sum of all elements up to a certain point.

3. **Initialize Variables**: Two variables are initialized: `ans`, which will hold the final answer, and `diff`, which is set to infinity `inf` as a placeholder for the smallest difference found so far.

4. **Iteration and Value Simulation**: We iterate through possible values for replacement `value` in the range from `0` to `max(arr)+1` (inclusive). For each value, the code uses binary search with `bisect_right` to find the index `i` where elements in `arr` are greater than `value`. As `arr` is sorted, this tells us how many elements will be replaced with `value`.

5. **Calculating the Difference**: With the index `i`, the algorithm calculates the sum of the array up to `i` (the sum of elements not changed) and adds `(len(arr) - i) * value` (the sum if all larger elements are replaced by `value`). This gives us the simulated sum of the array after replacements. The difference `d` between the simulated sum and `target` is computed in absolute terms.

6. **Track the Best Value**: If the current difference `d` is smaller than the smallest difference found so far `diff`, we update `diff` to `d` and `ans` to the current value being considered. If two values result in the same `diff`, since we iterate in increasing order, `ans` will already hold the smaller value.

7. **Return the Result**: After the loop completes, the smallest `ans` is returned, representing the value that when used to replace elements in `arr` results in a sum closest to `target`.

The key to this solution is the combination of sorting, prefix sums, binary search, and careful tracking of the smallest absolute difference, which altogether yields an efficient approach to solving the problem.

Here is the solution code again for reference:

```python
class Solution:
    def findBestValue(self, arr: List[int], target: int) -> int:
        arr.sort()
        s = list(accumulate(arr, initial=0))
        ans, diff = 0, inf
        for value in range(max(arr) + 1):
            i = bisect_right(arr, value)
            d = abs(s[i] + (len(arr) - i) * value - target)
            if diff > d:
                diff = d
                ans = value
        return ans
```

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following input:

- arr: [4, 9, 3]
- target: 10

Now let's walk through the solution step by step:

1. **Sort the Array**:
   - After sorting, `arr` becomes [3, 4, 9].
2. **Accumulate the Sum**:
   - We create a prefix sum array `s` which would be `[0, 3, 7, 16]`. Here, `s[0]` is `0`, and other elements represent the sum up to that index in `arr`.
3. **Initialize Variables**:
   - We set `ans` to 0 and `diff` to infinity (`inf`) initially.
4. **Iteration and Value Simulation**:
   - We iterate `value` from `0` to `max(arr) + 1`, which is 10 in this case.
   - For each `value`, say `2`, we use binary search to find the first index `i` in `arr` where the element is greater than `2`.
   - In this case, for `value` equals `2`, `i` is `1` because `arr[1]` is the first element greater than `2`.
5. **Calculating the Difference**:
   - For `value` equals `2`, the sum of the array if all elements larger than `2` are replaced will be `s[i] + (len(arr) - i) * value`. Here, it will be `0 + (3 - 1) * 2` which equals `4`.
   - The difference `d` between this simulated sum (`4`) and the target (`10`) is `6`. We compare this to `diff` and since `6` is less than `inf`, we update `diff` to `6` and `ans` to `2`.
6. **Track the Best Value**:
   - We continue this process for each `value` up to `10`. For each `value`, we find a new `d` and update `ans` and `diff` accordingly if we find a smaller `d`.
7. **Return the Result**:
   - Once we go through all possible values, let's imagine we found the best `value` to be `3` which gives us the least `diff` of `1` (the sum of the array becomes `3 + 3 + 3 = 9` which is closest to the target `10`).
   - Thus, `ans` is set to `3`, and that will be the result returned.

Through the above steps, the `findBestValue` function would eventually return `3` as the smallest integer we can replace the elements greater than `value` with to make the sum of the array closest to the `target 10`.

Python Solution

```python
from bisect import bisect_right
from itertools import accumulate

class Solution:
    def findBestValue(self, arr, target):
        # Sort the array to enable binary search later
        arr.sort()

        # Precompute the prefix sums to quickly calculate total sum when needed
        prefix_sums = list(accumulate(arr, initial=0))

        # Initialize the answer and the smallest difference found so far
        best_value = 0
        smallest_difference = float('inf')  # Using 'inf' from the math module signifies infinity

        # Iterate over all possible values from 0 to the maximum in the array
        for value in range(max(arr) + 1):
            # Find the index of the first element greater than the current value
            index = bisect_right(arr, value)

            # Calculate the total sum if all elements from index onwards were changed to 'value'
            current_sum = prefix_sums[index] + (len(arr) - index) * value
            # Calculate the difference between the current sum and the target sum
            difference = abs(current_sum - target)

            # If the current difference is smaller than any we have seen so far
            if smallest_difference > difference:
                # Update the smallest difference and the best value we've found
                smallest_difference = difference
                best_value = value

        # Return the value that results in a sum closest to the target
        return best_value
```

Java Solution

```java
class Solution {
    public int findBestValue(int[] arr, int target) {
        // Sort the input array
        Arrays.sort(arr);

        int n = arr.length; // length of the array
        // Create a prefix sum array with an additional initial 0 for easier calculations
        int[] prefixSum = new int[n + 1];
        int maxValue = 0; // To keep track of the maximum value in the array

        // Calculate prefix sums and find the maximum value in the array
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + arr[i];
            maxValue = Math.max(maxValue, arr[i]);
        }

        // Initialize answer variables
        int bestValue = 0;
        int minDiff = Integer.MAX_VALUE;

        // Iterate over possible values, 0 to max value in the array
        for (int candidateValue = 0; candidateValue <= maxValue; ++candidateValue) {
            // Find the index where elements start to be greater than the candidateValue
            int index = binarySearch(arr, candidateValue);
            // Calculate the difference from the target for this candidateValue
            int difference = Math.abs(prefixSum[index] + (n - index) * candidateValue - target);

            // If this is a smaller difference, update the best value found so far
            if (minDiff > difference) {
                minDiff = difference;
                bestValue = candidateValue;
            }
        }
        return bestValue;
    }

    // Binary search to find the leftmost position where arr[i] > x
    private int binarySearch(int[] arr, int x) {
        int left = 0, right = arr.length;
        while (left < right) {
            // Middle index between left and right
            int mid = (left + right) >> 1;

            // Modify the search range based on the middle element
            if (arr[mid] > x) {
                right = mid; // Look in the left half
            } else {
                left = mid + 1; // Look in the right half
            }
        }
        return left;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to find the best value for the input array such that
    // the sum of the array after replacing values greater than the best value
    // is as close to the target as possible.
    int findBestValue(vector<int>& arr, int target) {
        // First, sort the array for binary search to work properly.
        sort(arr.begin(), arr.end());

        int n = arr.size(); // Store the size of the array for later use.

        // Create a prefix sum array where s[i] is the sum of arr[0] to arr[i-1].
        // It helps in calculating the sum efficiently.
        vector<int> prefixSum(n + 1, 0);
        int maxValue = 0; // Track the maximum value in the array.

        // Calculate the prefix sum and find the maximum value in the array.
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + arr[i];
            maxValue = max(maxValue, arr[i]);
        }

        // Initialize variables to track the answer and the smallest difference found so far.
        int answer = 0;
        int smallestDiff = INT_MAX;

        // Iterate through all possible values from 0 to maxValue to find the best value.
        for (int value = 0; value <= maxValue; ++value) {
            // Find the first element in the array that is greater than the current value
            // and calculate the total sum if we replace all larger elements with 'value'.
            int idx = upper_bound(arr.begin(), arr.end(), value) - arr.begin();
            int currentSum = prefixSum[idx] + (n - idx) * value;

            // Calculate the difference from the target sum.
            int diff = abs(currentSum - target);

            // If the current difference is smaller than the smallest found so far,
            // update the answer and the smallest difference.
            if (smallestDiff > diff) {
                smallestDiff = diff;
                answer = value;
            }
        }

        // Return the best value found.
        return answer;
    }
};
```

Typescript Solution

```typescript
function findBestValue(arr: number[], target: number): number {
    // Sort the array for efficient binary search.
    arr.sort((a, b) => a - b);

    const n: number = arr.length; // Store the length of the array.

    // Create a prefix sum array for efficient sum calculation.
    const prefixSum: number[] = new Array(n + 1).fill(0);
    let maxValue: number = 0; // To keep track of the maximum value in the array.

    // Calculate the prefix sum and find the maximum value.
    for (let i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + arr[i];
        maxValue = Math.max(maxValue, arr[i]);
    }

    // Initialize variables to track the closest sum and the smallest difference.
    let bestValue: number = 0;
    let smallestDifference: number = Number.MAX_SAFE_INTEGER;

    // Iterate over all possible values to find the one that gives us the closest sum to the target.
    for (let value = 0; value <= maxValue; value++) {
        // Use binary search to find the first element greater than the current value.
        const idx: number = binarySearch(arr, value);
        const currentSum: number = prefixSum[idx] + (n - idx) * value;

        const difference: number = Math.abs(currentSum - target);

        // If this difference is smaller than the smallest one found so far, record this value and difference.
        if (difference < smallestDifference) {
            smallestDifference = difference;
            bestValue = value;
        }
    }

    return bestValue;
}

// Helper function to perform binary search, finding the index of the first element greater than the value.
function binarySearch(arr: number[], value: number): number {
    let left = 0;
    let right = arr.length - 1;
    while (left <= right) {
        const mid = left + Math.floor((right - left) / 2);
        if (arr[mid] > value) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

Time and Space Complexity

The given Python code aims to find an integer value that, when used to replace larger elements in an array, results in the sum of the array being as close as possible to a given target. The time complexity and space complexity of the code are analyzed as follows:

Time Complexity

- The `arr.sort()` operation has a time complexity of `O(n log n)`, where `n` is the length of the array.
- The `list(accumulate(arr, initial=0))` operation constructs a prefix sum array in `O(n)` time.
- The for loop runs for a maximum of `max(arr) + 1` iterations. In the worst case, this is also `O(n)` since `max(arr)` is less than or equal to the sum of all `n` elements.
- Inside the loop, the `bisect_right` function is called, which operates in `O(log n)` time.
- The remaining operations within the loop (calculating the difference `d` and comparing/updating `diff` and `ans`) are all constant time operations, i.e., `O(1)`.

Combining these, the overall time complexity is dominated by the sorting operation and the for loop that includes the binary search. Therefore, the time complexity is `O(n log n) + O(n) * O(log n)`, which simplifies to `O(n log n)` because both terms have `log n` and the larger factor `n log n` dominates.

Space Complexity

- The additional space used by the algorithm includes the space for the prefix sum array `s` which is `O(n)` and a constant amount of space for variables like `ans`, `diff`, `i`, `d`, and `value`.

Therefore, the overall space complexity of the code is `O(n)` due to the use of the prefix sum array.