2602. Minimum Operations to Make All Array Elements Equal

Medium | Array | Binary Search | Prefix Sum | Sorting | Leetcode Link

Problem Description

In this problem, we are given an array of positive integers called `nums` and a separate array of positive integers called `queries`. For each query `queries[i]`, the goal is to transform all elements in `nums` such that they are all equal to the value of `queries[i]`. To do this, we can increase or decrease any element in the `nums` array by 1. The objective is to find the minimum number of these increase or decrease operations required to achieve the goal for each query.

It is important to mention that these operations are independent for each query; that is, after performing the operations required by one query, the `nums` array returns to its original state before applying the next query.

Our task is to return an array `answer` where each element `answer[i]` is the minimum number of operations needed to make all elements of `nums` equal to `queries[i]`.

Intuition

The straightforward approach of repeatedly increasing or decreasing each number in `nums` until it matches `queries[i]` would be too slow, especially for large arrays.

To optimize this, we can think about the problem in two parts:

1. We need to reduce the larger numbers down to `queries[i]`.
2. We need to increase the smaller numbers up to `queries[i]`.

Each of these parts needs a different sort of operations. If the numbers are sorted, we can quickly find the point in the array where the numbers shift from being less than to greater than `queries[i]`. This helps us to separate the two groups conveniently.

Once we have the numbers sorted, we can use a prefix sum array (`s` in the solution code) to quickly calculate the total sum of numbers up to any point. The prefix sum array keeps a running total, which allows us to compute the sum of any sub-array in constant time, by subtracting two prefix sums.

With the help of binary search, we can then find the exact point of separation between numbers we need to increase and those we need to decrease. The binary search efficiently finds the position in a sorted array where a given element could be inserted to maintain the order (less than or equal / greater than `queries[i]` in our case).

Using the point of separation and prefix sums, we can calculate the number of operations needed for both increasing the smaller numbers and decreasing the larger ones. The sum of these two gives us the minimum number of operations needed for the `nums` array to be all equal to `queries[i]`.

Finally, we repeat this process for each query and compile the results into an answer array.

Solution Approach

The solution employs a combination of sorting, prefix summation, and binary search to efficiently find the minimum number of operations required to make all elements of `nums` equal to each query value.

Here is a detailed breakdown of the solution approach:

1. **Sorting `nums`:** We start by sorting the array `nums`. Sorting is helpful because it allows us to apply binary search later in the process and also makes it easy to separate elements into those that need to be increased vs. those that need to be decreased to match the query value.
2. **Prefix Sum Array (`s`):** After sorting `nums`, we compute the prefix sum array `s`. The prefix sum is essentially a cumulative sum up to the current index, and we initialize it with a zero to account for the hypothetical prefix sum before the first element. The formula used to obtain the prefix sum array is `s[i] = s[i-1] + nums[i-1]`. This approach simplifies calculating the sum of subarrays later on.
3. **Binary Search:** For each query `x`, we need to find the split point where we increase the numbers below `x` and decrease the numbers above `x`. We use the `bisect_left` function from the `bisect` module, which is an implementation of the binary search algorithm, to find this index quickly. We find two indices in this step - the index of the first element greater than `x` and the index of the first element equal to or greater than `x`.
4. **Calculating the Number of Operations:**
   - For the elements greater than `x`, we subtract the sum of these elements given by `s[-1] - s[i]` from the total reduction required `(len(nums) - i) * x` to find the total decrease operations needed `t`.
   - For elements less than `x`, we multiply `x * i` to find the total increase needed and subtract the sum of these elements `s[i]` to find the total increase operations needed.
   - The sum of these two values gives us the total minimum operations required for current query `x`.
5. **Build the Result Array (`ans`):** We append the total minimum operations found for each query to the result array `ans`.

Combining these steps and repeating them for each value in `queries`, we arrive at the final result of the minimum number of operations required to equalize the `nums` array to each of the query values. The use of binary search and prefix sums ensures that the operations are performed efficiently, managing to keep the computational complexity within acceptable limits even for large arrays.

Example Walkthrough

Let's illustrate the solution approach using an example. Given an array `nums = [1, 3, 4, 9]` and a `queries` array containing a single element `[6]`, follow the steps below to find the minimum number of operations required to make all elements in `nums` equal to `6`.

1. **Sorting `nums`:** The array `nums` is already sorted: `[1, 3, 4, 9]`.
2. **Prefix Sum Array (`s`):** We compute the prefix sum array as follows:
   - Initialize by setting `s[0] = 0`
   - For `i=1`, add `s[0] + nums[0]` → `s[1] = 0 + 1 = 1`
   - For `i=2`, add `s[1] + nums[1]` → `s[2] = 1 + 3 = 4`
   - For `i=3`, add `s[2] + nums[2]` → `s[3] = 4 + 4 = 8`
   - For `i=4`, add `s[3] + nums[3]` → `s[4] = 8 + 9 = 17`

   The prefix sum array now is `s = [0, 1, 4, 8, 17]`.
3. **Binary Search:** We apply binary search to find the split point for query `6`. Using `bisect_left`, we find that:
   - The index `i` for the first element greater than `6` is `3` (`nums[3] = 9`).
   - The index `j` for the first element equal to or greater than `6` is also `3`.
4. **Calculating the Number of Operations:**
   - For elements greater than `6` (`nums[3]`), calculate the reduction operations: The sum of elements bigger than `6` is `s[-1] - s[i] = 17 - 8 = 9`. Total reduction needed `(len(nums) - i) * 6 = (4 - 3) * 6 = 6`. Total decrease operations `t = 6 - 9 = -3` (since we are decreasing, this number should be positive, so we take absolute: `3`).
   - For elements less than `6` (from `nums[0]` to `nums[2]`), calculate the increase operations: Total increase needed is `6 * i = 6 * 3 = 18`. Sum of these elements is `s[i] = 8`. Total increase operations needed `t = 18 - 8 = 10`.

   Consequently, the total minimum operations for `6` is the sum of increase and decrease operations: `3 + 10 = 13`.
5. **Build the Result Array (`ans`):** Append `13` to the result array `ans`. Since there is only one query value, `ans = [13]`.

By following these steps, the final answer we get is that 13 operations are necessary to make all elements in the array `nums` equal to the single query value `6`.

Python Solution

```python
from bisect import bisect_left
from itertools import accumulate

class Solution:
    def minOperations(self, nums, queries):
        # Sort the input array to allow binary search operations
        nums.sort()

        # Accumulate the sum of elements in the array, starting from 0
        prefix_sum = list(accumulate(nums, initial=0))

        # Initialize the list to store results of each query
        answer = []

        # Evaluate each query in the queries list
        for x in queries:
            # Find the index of the smallest number in nums that is greater than x
            index = bisect_left(nums, x + 1)
            # Calculate the total operations needed for the larger portion of nums
            total_ops = prefix_sum[-1] - prefix_sum[index] - ((len(nums) - index) * x)

            # Find the index of the smallest number in nums that is greater than or equal to x
            index = bisect_left(nums, x)
            # Add the operations needed for the smaller portion of nums
            total_ops += x * index - prefix_sum[index]

            # Append the total number of operations to answer
            answer.append(total_ops)

        # Return the final result after processing all queries
        return answer

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    public List<Long> minOperations(int[] nums, int[] queries) {
        // Sort the array of numbers in non-decreasing order
        Arrays.sort(nums);

        int n = nums.length; // Number of elements in nums
        // Create a prefix sum array with an extra space for ease of calculations
        long[] prefixSum = new long[n + 1];

        // Calculate the prefix sum
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        // Initialize the answer list
        List<Long> ans = new ArrayList<>();

        // Process each query
        for (int x : queries) {
            // Search for the first element in nums that is greater than or equal to x + 1
            int index = binarySearch(nums, x + 1);
            // Calculate the total sum needed to reduce elements larger or equal to x + 1 to x
            long totalOperations = prefixSum[n] - prefixSum[index] - 1L * (n - index) * x;

            // Search for the first element in nums that is greater than or equal to x
            index = binarySearch(nums, x);
            // Add the total sum needed to increase elements smaller than x to x
            totalOperations += 1L * x * index - prefixSum[index];

            // Add the calculated operations for the current query to the answer list
            ans.add(totalOperations);
        }

        return ans; // Return the answer list
    }

    // Helper method to perform binary search
    private int binarySearch(int[] nums, int x) {
        int left = 0;
        int right = nums.length;
        // Binary search to find the index of the first number greater or equal to x
        while (left < right) {
            int mid = (left + right) >> 1; // Middle position
            if (nums[mid] >= x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left; // Return the first index where nums[index] >= x
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find the minimum operations needed for each query
    vector<long long> minOperations(vector<int>& nums, vector<int>& queries) {
        // Sort the input numbers array for binary search
        sort(nums.begin(), nums.end());

        // Calculate the size of the nums array
        int numsSize = nums.size();

        // Prefix sum array initialized with an extra element for ease of calculations
        vector<long long> prefixSums(numsSize + 1, 0);

        // Compute the prefix sums for efficient range sum queries
        for (int i = 0; i < numsSize; ++i) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // Vector to store the answer for each query
        vector<long long> answer;

        // Iterate over all queries to compute the minimum operations
        for (auto& queryValue : queries) {
            // Find the first element greater than the query value
            int upperIndex = lower_bound(nums.begin(), nums.end(), queryValue + 1) - nums.begin();

            // Calculate the operations needed for transforming numbers greater than the query value
            long long operationsForGreater = prefixSums[numsSize] - prefixSums[upperIndex] - 1LL * (numsSize - upperIndex) * queryV

            // Find the first element that is not less than the query value
            int lowerIndex = lower_bound(nums.begin(), nums.end(), queryValue) - nums.begin();

            // Calculate the operations needed for transforming numbers less than or equal to the query value
            long long operationsForLessOrEqual = 1LL * queryValue * lowerIndex - prefixSums[lowerIndex];

            // The total operations is the sum of operations for both ranges
            long long totalOperations = operationsForGreater + operationsForLessOrEqual;

            // Append the total operations to the answer vector
            answer.push_back(totalOperations);
        }

        // Return the final answer vector
        return answer;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the minimum operations required for the queries
function minOperations(nums: number[], queries: number[]): number[] {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);

    const length = nums.length;

    // Precompute the prefix sums of the sorted array
    const prefixSums: number[] = new Array(length + 1).fill(0);
    for (let i = 0; i < length; ++i) {
        prefixSums[i + 1] = prefixSums[i] + nums[i];
    }

    // Binary search function to find the first index where value is >= x
    const binarySearch = (x: number): number => {
        let left = 0;
        let right = length;
        while (left < right) {
            const mid = (left + right) >> 1; // Equivalent to Math.floor((left + right) / 2)
            if (nums[mid] >= x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    const answers: number[] = [];

    // Process each query
    for (const query of queries) {
        // Find the first number greater than query
        const greaterIndex = binarySearch(query + 1);

        // Calculate the total of subtracting query from elements greater than query
        let total = prefixSums[length] - prefixSums[greaterIndex] - (length - greaterIndex) * query;

        // Find the first number greater than or equal to query
        const equalOrGreaterIndex = binarySearch(query);

        // Calculate the total of adding query to elements less than query
        total += query * equalOrGreaterIndex - prefixSums[equalOrGreaterIndex];

        // Add the result to the answer array
        answers.push(total);
    }

    return answers;
}

```

Time and Space Complexity

The given algorithm involves sorting the array `nums` and performing binary searches for each query in `queries`.

Time Complexity

1. Sorting the `nums` array has a time complexity of `O(n \log n)` where `n` is the length of the `nums` array.
2. The `accumulate` function is linear, contributing `O(n)` to the time complexity.
3. For each query, a binary search is performed twice using `bisect_left`, which has a time complexity of `O(\log n)`. Since there are `q` queries, the total complexity for this part is `O(q \log n)`.

The total time complexity combines these contributions, resulting in `O(n \log n) + O(n) + O(q \log n)`. Since `O(n \log n)` dominates `O(n)`, and the number of queries `q` could vary independently of `n`, the overall time complexity is `O((n + q) \log n)`.

However, since the reference answer only specifies `O(n \log n)`, it implies that `n` is the dominant term, and `q` is expected to be not significantly larger than `n`.

Space Complexity

1. The sorted array is a modification in place, so it does not add to the space complexity.
2. The `s` variable is a list storing the cumulative sum, contributing `O(n)` to the space complexity.
3. A constant amount of extra space is used for variables `i`, `t`, and the iterative variables, which does not depend on the size of the input.

Thus, the overall space complexity is `O(n)`.