<https://leetcode.com/problems/minimum-operations-to-make-all-array-elements-equal/description/>

You are given an array nums consisting of positive integers.

You are also given an integer array queries of size m. For the ith query, you want to make all of the elements of nums equal to queries[i]. You can perform the following operation on the array **any** number of times:

**Increase** or **decrease** an element of the array by 1.

Return *an array*answer*of size*m*where*answer[i]*is the****minimum****number of operations to make all elements of*nums*equal to*queries[i].

**Note** that after each query the array is reset to its original state.

**Example 1:**

**Input:** nums = [3,1,6,8], queries = [1,5]

**Output:** [14,10]

**Explanation:** For the first query we can do the following operations:

- Decrease nums[0] 2 times, so that nums = [1,1,6,8].

- Decrease nums[2] 5 times, so that nums = [1,1,1,8].

- Decrease nums[3] 7 times, so that nums = [1,1,1,1].

So the total number of operations for the first query is 2 + 5 + 7 = 14.

For the second query we can do the following operations:

- Increase nums[0] 2 times, so that nums = [5,1,6,8].

- Increase nums[1] 4 times, so that nums = [5,5,6,8].

- Decrease nums[2] 1 time, so that nums = [5,5,5,8].

- Decrease nums[3] 3 times, so that nums = [5,5,5,5].

So the total number of operations for the second query is 2 + 4 + 1 + 3 = 10.

**Example 2:**

**Input:** nums = [2,9,6,3], queries = [10]

**Output:** [20]

**Explanation:** We can increase each value in the array to 10. The total number of operations will be 8 + 1 + 4 + 7 = 20.

**Constraints:**

n == nums.length

m == queries.length

1 <= n, m <= 10^5

1 <= nums[i], queries[i] <= 10^9

**Attempt 1: 2025-04-22**

**Solution 1: Binary Search + Math + Prefix Sum (30 min)**

class Solution {

    public List<Long> minOperations(int[] nums, int[] queries) {

        Arrays.sort(nums);

        int n = nums.length;

        long[] preSum = new long[n + 1];

        for(int i = 1; i <= n; i++) {

            preSum[i] = preSum[i - 1] + nums[i - 1];

        }

        List<Long> result = new ArrayList<>();

        for(int q : queries) {

            int i = binarySearch(nums, q);

            // preSum[n] - preSum[i] is sum of numbers greater than or equal to query[i]

            // preSum[i] is sum of numbers smaller than query[i]

            // query[i] \* i - preSum[i] is increments required

            // preSum[n] - preSum[i] - query[i] \* (n - i) is decrements required

            // Total = query[i] \* i - preSum[i] + preSum[n] - preSum[i] - query[i] \* (n - i)

            // Can be simplified to query[i] \* (2 \* i - n) + preSum[n] - 2 \* preSum[i]

            result.add(1L \* q \* (2 \* i - n) + preSum[n] - 2 \* preSum[i]);

        }

        return result;

    }

    private int binarySearch(int[] nums, int target) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] >= target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // If not found index situation return insertion point

        // but this point is still 'lo', no need separate logic

        //if(lo == nums.length || nums[lo] != target) {

        //    return lo;

        //}

        return lo;

    }

}

Time Complexity: O((n + m) \* log(n))

Space Complexity: O(n)

**Refer to**

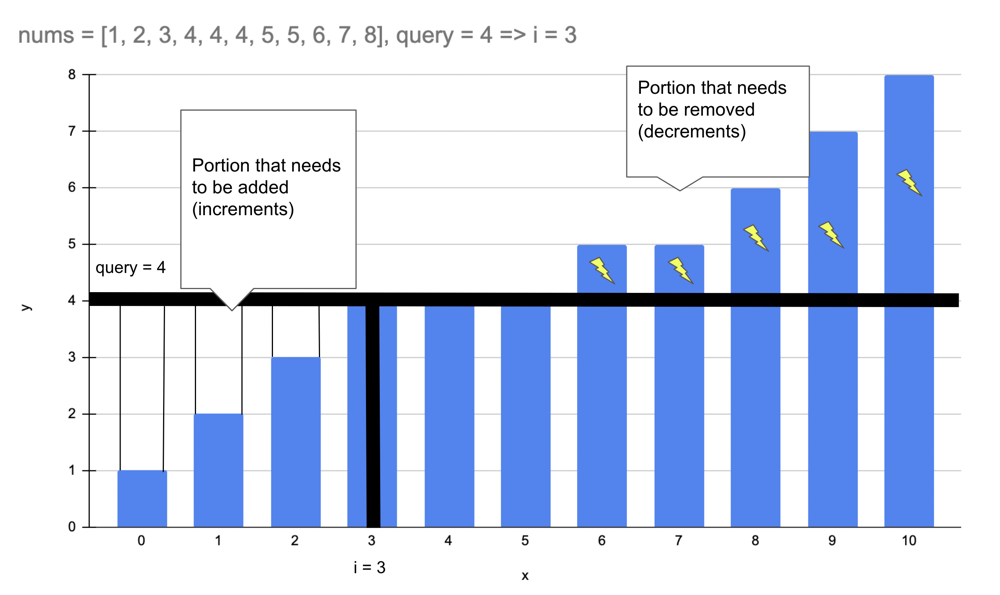
<https://leetcode.com/problems/minimum-operations-to-make-all-array-elements-equal/solutions/3341928/c-java-python3-prefix-sums-binary-search/?page=1&search=2602>

**Intuition**

If there are j numbers in nums that are smaller than query[i], you need to find query[i] \* j - sum(j numbers smaller than query[i]) to find increments required in nums

If there are k numbers in nums that are greater than query[i], you need to find sum(k numbers larger than query[i]) - query[i] \* k to find decrements required in nums

Sum of above two values is ans[i]



**Approach**

To find smaller numbers than query[i] we can sort the array and use binary search

**Binary search over sorted nums to find index of query[i]**

Then use prefix sums to find sum of number in smaller and larger segments

prefix[n] - prefix[i] is sum of numbers greater than or equal to query[i]

prefix[i] is sum of numbers smaller than query[i]

query[i] \* i - prefix[i] is increments required

prefix[n] - prefix[i] - query[i] \* (n - i) is decrements required

Total = query[i] \* i - prefix[i] + prefix[n] - prefix[i] - query[i] \* (n - i)

Can be simplified to query[i] \* (2 \* i - n) + prefix[n] - 2 \* prefix[i]

**Complexity**

Time complexity: O((n + m) \* log(n))

Space complexity: O(n)

**Code**

public List<Long> minOperations(int[] nums, int[] queries) {

Arrays.sort(nums);

List<Long> ans = new ArrayList<>();

int n = nums.length;

long[] prefix = new long[n + 1];

for (int i = 1; i <= n; i++)

prefix[i] = prefix[i - 1] + nums[i - 1];

for (int x: queries) {

int i = bisect\_left(nums, x);

ans.add(1L \* x \* (2 \* i - n) + prefix[n] - 2 \* prefix[i]);

}

return ans;

}

private int bisect\_left(int[] nums, int x) {

int lo = 0, hi = nums.length;

while (lo < hi) {

int mid = lo + (hi - lo) / 2;

if (nums[mid] < x) lo = mid + 1;

else hi = mid;

}

return lo;

}

**Java using Arrays.binarySearch**:

public List<Long> minOperations(int[] nums, int[] queries) {

Arrays.sort(nums);

List<Long> ans = new ArrayList<>();

int n = nums.length;

long[] prefix = new long[n + 1];

for (int i = 1; i <= n; i++)

prefix[i] = prefix[i - 1] + nums[i - 1];

for (int x: queries) {

int i = Arrays.binarySearch(nums, x);

if (i < 0) i = -(i + 1); // convert negative index to insertion point

ans.add(1L \* x \* (2 \* i - n) + prefix[n] - 2 \* prefix[i]);

}

return ans;

}

*Good to know*: In Java, Arrays.binarySearch returns a negative index when the target element is not found in the array. The negative index returned is equal to -(insertion point) - 1, where the insertion point is the index at which the target element would be inserted into the array to maintain the sorted order.

For example, suppose we have an array arr = {1, 3, 5, 7, 9} and we want to search for the element 4. The Arrays.binarySearch(arr, 4) method would return -3. To convert this to the insertion point, we take the absolute value of the negative index and subtract 1: insertion point = |-(-3)| - 1 = 2.

Therefore, the insertion point for the element 4 in the sorted array arr would be at index 2, which is the position where we would need to insert 4 to maintain the sorted order.

**More binary search problems:**

[Minimum Time to Repair Cars](https://leetcode.com/problems/minimum-time-to-repair-cars/solutions/3311975/c-java-python3-binary-search-1-liner/)

[Minimum Time to Complete Trips](https://leetcode.com/problems/minimum-time-to-complete-trips/solutions/1802415/python3-java-c-binary-search-1-liner/)

[Kth Missing Positive Number](https://leetcode.com/problems/kth-missing-positive-number/solutions/3262163/c-java-python3-1-line-ologn/)

[Maximum Tastiness of Candy Basket](https://leetcode.com/problems/maximum-tastiness-of-candy-basket/solutions/2947983/c-java-python-binary-search-and-sorting/)

[Number of Flowers in Full Bloom](https://leetcode.com/problems/number-of-flowers-in-full-bloom/solutions/1976804/python3-2-lines-sort-and-binary-search/)

**Refer to Deepseek**

Here's a Java solution for LeetCode 2602, which calculates the minimum operations to make all array elements equal for each query, using sorting, prefix sums, and binary search for efficiency:

**Approach**

Sort the Array: Sorting helps in efficiently partitioning the array into elements that need to be increased or decreased for each query.

Prefix Sum Array: Compute a prefix sum array to quickly calculate the sum of any subarray, which is essential for determining the total operations needed.

Binary Search: For each query, use binary search to find the split point where elements transition from being less than the query value to greater than or equal to it. This allows us to separate the elements that need to be increased from those that need to be decreased.

Calculate Operations:

For elements less than the query value, compute the total operations needed to increase them to the query value.

For elements greater than the query value, compute the total operations needed to decrease them to the query value.

Sum these operations to get the total for each query.

**Solution Code**

class Solution {

public List<Long> minOperations(int[] nums, int[] queries) {

Arrays.sort(nums);

int n = nums.length;

long[] prefixSum = new long[n + 1];

for (int i = 0; i < n; ++i) {

prefixSum[i + 1] = prefixSum[i] + nums[i];

}

List<Long> ans = new ArrayList<>();

for (int x : queries) {

int i = binarySearch(nums, x + 1);

long t = prefixSum[n] - prefixSum[i] - (long) (n - i) \* x;

i = binarySearch(nums, x);

t += (long) x \* i - prefixSum[i];

ans.add(t);

}

return ans;

}

private int binarySearch(int[] nums, int x) {

int l = 0, r = nums.length;

while (l < r) {

int mid = (l + r) >> 1;

if (nums[mid] >= x) {

r = mid;

} else {

l = mid + 1;

}

}

return l;

}

}

**Explanation**

Sorting: The array nums is sorted to facilitate binary search and prefix sum calculations.

Prefix Sum Array: The prefixSum array stores cumulative sums, allowing O(1) access to the sum of any subarray.

Binary Search: The binarySearch method finds the first index where the element is greater than or equal to the query value, which helps in partitioning the array.

Operations Calculation:

For elements greater than the query value, the operations are calculated as the sum of these elements minus the product of the query value and the count of these elements.

For elements less than the query value, the operations are calculated as the product of the query value and the count of these elements minus the sum of these elements.

The total operations for each query are the sum of these two values.

This approach ensures that the solution is efficient with a time complexity of O(n log n) due to sorting and O(m log n) for processing each query, where n is the number of elements in nums and m is the number of queries. The space complexity is O(n) for storing the prefix sum array.

**I have a correct solution below, but i have a doubt, binary search is based on 0-indexed sorted 'nums' array, but 'preSum' array is padding preSum[0] as 0, which means the first 'number' in 'nums' record at preSum[1], how these two arrays sync up the index system ?**

Great question! Let me explain how the indexing works between the sorted nums array (0-indexed) and the preSum array (1-indexed with padding).

**Indexing System Sync**

nums Array (0-indexed):

nums[0] = first element

nums[n-1] = last element

preSum Array (1-indexed with padding):

preSum[0] = 0 (padding)

preSum[1] = sum of first 1 element (nums[0])

preSum[2] = sum of first 2 elements (nums[0] + nums[1])

...

preSum[n] = sum of all elements

**How They Sync in Binary Search**

When binarySearch(nums, q) returns index i:

This i is the 0-based insertion point in the sorted nums array.

It represents the count of elements less than q (since nums[i] >= q).

Key Insight:

preSum[i] = sum of the first i elements in nums (which are all < q).

preSum[n] - preSum[i] = sum of elements from nums[i] to nums[n-1] (which are all >= q).

**Example Walkthrough**

Let nums = [1, 3, 5, 7] (sorted) and query q = 4:

Binary Search:

Returns i = 2 (insertion point for 4 in nums).

Interpretation: 2 elements (1, 3) are < 4.

Prefix Sum Access:

preSum = [0, 1, 4, 9, 16]

preSum[i] = preSum[2] = 4 (sum of 1 + 3).

preSum[n] - preSum[i] = 16 - 4 = 12 (sum of 5 + 7).

Operations Calculation:

Increments: q \* i - preSum[i] = 4\*2 - 4 = 4

Decrements: preSum[n] - preSum[i] - q\*(n-i) = 12 - 4\*2 = 4

Total: 4 + 4 = 8

**Why This Works**

The i from binary search (0-based in nums) directly aligns with the 1-based preSum because:

preSum[i] sums nums[0..i-1] (exactly the elements < q).

preSum[n] - preSum[i] sums nums[i..n-1] (elements >= q).

**Edge Case Handling**

If i = 0: All elements are >= q → only decrements needed.

If i = n: All elements are < q → only increments needed.

**Summary**

The 0-based i from binary search seamlessly maps to the 1-based preSum because:

preSum[i] = sum of the first i elements in nums (which are < q).

The padding (preSum[0] = 0) ensures correct calculations when i = 0.

This design elegantly handles all cases while maintaining O(log n) per query efficiency.

**Refer to**

[L462.Minimum Moves to Equal Array Elements II (Ref.L453)](note://WEBf26377967abd88cf360cde7f55549d27)

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)