2448. Minimum Cost to Make Array Equal

Hard | Greedy | Array | Binary Search | Prefix Sum | Sorting | Leetcode Link

Problem Description

In this problem, we are given two arrays `nums` and `cost` of equal length `n`. The `nums` array holds the actual values of the elements we must manipulate, and the `cost` array indicates how much it costs to increase or decrease an element in the corresponding position of the `nums` array by `1`.

Our objective is to make all elements in the `nums` array equal to each other by either increasing or decreasing their values. The catch is, every operation has a cost associated with it, based on the `cost` array. We want to find the minimum total cost required to achieve this.

The main challenge is to figure out what value we should aim for all elements in the `nums` array to reach, and do this as cost-efficient as possible.

Intuition

The solution hinges on finding a target value in the `nums` array that minimizes the total cost. Intuitively, aiming for a value too high or too low will not be cost efficient, as it would require more operations for elements that are far from such extremes.

In fact, a good target value can be one of the current elements in the `nums` array, since aiming for a value not present may potentially increase the total cost compared to aiming for a value already there.

Sorting the array `nums` along with `cost` will help us approach this problem methodically. The sorted nature of `nums` allows us to consider the pivot elements (target values) one by one and calculate the cost required to make the rest of the elements equal to the pivot.

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

1. We zip and sort `nums` and `cost` together based on the `nums` values. This pairs each element of `nums` with its corresponding cost.
2. We prepare prefix sums for both `nums` multiplied by `cost` (`f`) and for `cost` alone (`g`) which will help us calculate the cost with better efficiency.
3. We iterate through each possible target value (which are now sorted) and calculate the total cost required for the rest of the array to reach this target value. This is done in two parts:
   - `l` is the total cost to decrease the left part of the array to the current target value.
   - `r` is the total cost to increase the right part of the array to the current target value.
4. The minimum of these calculated costs (`ans`) for each element being the target is the answer we are seeking. It represents the lowest possible cost to make all elements in `nums` equal.

The algorithm explores all elements as potential targets and uses cumulative costs to determine the most cost-effective target, hence ensuring we find the minimum total cost to equalize all elements in the given `nums` array.

Solution Approach

The implementation of the solution can be broken down into several key steps that leverage algorithms and data structures to find the minimum total cost efficiently:

1. **Sorting with Zip**: The `zip` function is used to combine `nums` and `cost` into a single list of tuples. This list is then sorted based on the `nums` values using Python's `sorted()` function. Sorting is crucial as it allows us to easily calculate the cost of changing every other element to match a particular element (the pivot).

2. **Prefix Sums**: Two types of prefix sums (`f` and `g`) are utilized.

   - `f` represents the sum of elements of `nums` each multiplied by its corresponding cost up to the current index. This is calculated as `f[i] = f[i - 1] + a * b`, where `a` is the value in `nums` and `b` is the value in `cost` for the current element.
   - `g` is the sum of the `cost` elements up to the current index. It is used to calculate the total cost of changing the left or right part of the array to match the current pivot.
3. **Dynamic Calculation of Costs**: For each potential target value (`a`), the total cost is computed in two halves: the cost to adjust elements to the left (`l`) and the cost to adjust elements to the right (`r`) of the current index.

   - The cost to the left is computed as `l = a * g[i - 1] - f[i - 1]`. Here, `a * g[i - 1]` estimates the total cost if all elements to the left were increased to `a`, and `f[i - 1]` subtracts the excess since we've already some elements at the desired value or higher.
   - The cost to the right is computed as `r = f[n] - f[i] - a * (g[n] - g[i])`. Here, `f[n] - f[i]` represents the total sum that would have been without decreasing any elements from `nums[i]` onwards, and `a * (g[n] - g[i])` subtracts the excess, similar to the left part calculation.
4. **Finding the Minimum**: The variable `ans` is initialized with the value `inf` (infinity) to ensure that any real calculated cost will be lower on the first comparison. As the loop iterates through each pivot, the algorithm computes the total cost for each pivot and updates `ans` to be the minimum between the current `ans` and the newly calculated total cost (`l + r`).

5. **Returning the Result**: After the loop, the `ans` value will hold the minimum cost found, and this value is returned as the solution.

This approach is efficient because it cleverly reduces what could be many variable operations into a simple range of sums and differences by leveraging the sorted nature of the array and mathematically sound calculations to find the minimum total cost.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with the following `nums` and `cost` arrays:

`nums = [3, 1, 2, 4] cost = [4, 2, 3, 1]`

Following the solution approach:

1. **Sorting with Zip**: Pairing and sorting based on `nums` values gives us: `sorted_pair = [(1, 2), (2, 3), (3, 4), (4, 1)]`

2. **Prefix Sums**: For `f` (cumulative sum of `nums` times `cost`), we compute: `f[0] = 1 * 2 = 2` `f[1] = f[0] + 2 * 3 = 8` `f[2] = f[1] + 3 * 4 = 20` `f[3] = f[2] + 4 * 1 = 24`

   For `g` (cumulative sum of `cost`), we get: `g[0] = 2` `g[1] = g[0] + 3 = 5` `g[2] = g[1] + 4 = 9` `g[3] = g[2] + 1 = 10`

3. **Dynamic Calculation of Costs**: We consider each element in `nums` as the target. For example:

   - Using `1` as the target, the cost to increase is `0` (since it's already the lowest): `l = 1 * g[0] - f[0] = 0` The cost to decrease `2`, `3`, and `4` to `1`: `r = f[3] - f[0] - 1 * (g[3] - g[0]) = 22 - 0 - 1 * (8) = 14` Total cost for target `1` is then `l + r = 0 + 14 = 14`.

   - Using `2` as the target, we compute `l` with elements to the left: `l = 2 * g[0] - f[0] = 2 * 2 - 2 = 2` Compute `r` for elements to the right: `r = f[3] - f[1] - 2 * (g[3] - g[1]) = 24 - 8 - 2 * (5) = 6` Total cost for target `2` is `l + r = 2 + 6 = 8`.

   - Similarly, we would compute for targets `3` and `4`.

4. **Finding the Minimum**: Initialise `ans = inf`. After computing `l + r` for all possible targets, we'd find `ans` is minimum for target `2` at a cost of `8`.

5. **Returning the Result**: The minimum cost found for equalizing all elements to `2` is `8`, which is thus the answer to this example problem.

This approach allows us to efficiently determine that the best target element in `nums` is `2`, and the minimum total cost required to make all elements in the `nums` array equal to `2` is `8`.

Python Solution

```python
class Solution:
    def min_cost(self, nums, costs):
        # Combine nums and costs into a list of tuples and sort them by 'nums'
        num_cost_pairs = sorted(zip(nums, costs))
        n = len(num_cost_pairs)

        # Prefix sums of costs multiplied by corresponding nums
        prefix_multiplied_costs = [0] * (n + 1)

        # Prefix sums of costs
        prefix_costs = [0] * (n + 1)

        # Calculate prefix sums
        for i in range(1, n + 1):
            num, cost = num_cost_pairs[i - 1]
            prefix_multiplied_costs[i] = prefix_multiplied_costs[i - 1] + num * cost
            prefix_costs[i] = prefix_costs[i - 1] + cost

        # Initialize the answer with infinity representing a very high value
        answer = float('inf')

        # Calculate the minimum cost
        for i in range(1, n + 1):
            # Choose the ith element as the 'pivot' number
            pivot = num_cost_pairs[i - 1][0]

            # Left part: calculate the total cost for numbers before the 'pivot' number
            left = pivot * prefix_costs[i - 1] - prefix_multiplied_costs[i - 1]

            # Right part: calculate the total cost for numbers after the 'pivot' number
            right = prefix_multiplied_costs[n] - prefix_multiplied_costs[i] - pivot * (prefix_costs[n] - prefix_costs[i])

            # Update the answer with the minimum sum of left and right parts
            answer = min(answer, left + right)

        # Return the minimum cost
        return answer

```

Java Solution

```java
import java.util.Arrays;

public class Solution {

    // Method to calculate the minimum cost of manipulating numbers
    public long minCost(int[] nums, int[] costs) {
        int length = nums.length; // Get the length of the array

        // Create a new two-dimensional array to hold numbers and their corresponding costs
        int[][] pairedArray = new int[length][2];
        for (int i = 0; i < length; ++i) {
            pairedArray[i] = new int[]{nums[i], costs[i]};
        }

        // Sort the paired array based on the numbers
        Arrays.sort(pairedArray, (firstPair, secondPair) -> firstPair[0] - secondPair[0]);

        // Initialize prefix sums array for numbers multiplied by their cost (f)
        // and another for the costs (g)
        long[] prefixSumProduct = new long[length + 1];
        long[] prefixSumCost = new long[length + 1];

        // Calculate prefix sums
        for (int i = 1; i <= length; ++i) {
            long number = pairedArray[i - 1][0];
            long cost = pairedArray[i - 1][1];
            prefixSumProduct[i] = prefixSumProduct[i - 1] + number * cost;
            prefixSumCost[i] = prefixSumCost[i - 1] + cost;
        }

        long minimumCost = Long.MAX_VALUE; // Variable to store the minimum cost

        // Calculate minimum total cost of all number manipulation operations
        for (int i = 1; i <= length; ++i) {
            long number = pairedArray[i - 1][0];
            long leftCost = number * prefixSumCost[i - 1] - prefixSumProduct[i - 1];
            long rightCost = prefixSumProduct[length] - prefixSumProduct[i]
                             - number * (prefixSumCost[length] - prefixSumCost[i]);
            minimumCost = Math.min(minimumCost, leftCost + rightCost);
        }

        return minimumCost; // Return the calculated minimum cost
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::sort

// Alias for long long type
using ll = long long;

class Solution {
public:
    // Function to find the minimum cost needed to make all elements equal
    long long minCost(vector<int>& nums, vector<int>& cost) {
        int n = nums.size(); // Get the size of the array
        // 'pairedArray' holds pairs of number and cost for easy sorting and accessing
        vector<pair<int, int>> pairedArray(n);

        // Pair each number with its cost
        for (int i = 0; i < n; ++i) {
            pairedArray[i] = {nums[i], cost[i]};
        }

        // Sort the paired array based on the number
        sort(pairedArray.begin(), pairedArray.end());

        // f[i] will store the total cost up to the i-th element
        vector<ll> prefixCostSum(n + 1);
        // g[i] will store the total sum of costs up to the i-th element
        vector<ll> prefixCountSum(n + 1);

        // Calculate the prefix sums for cost and count
        for (int i = 1; i <= n; ++i) {
            auto [number, cost] = pairedArray[i - 1];
            prefixCostSum[i] = prefixCostSum[i - 1] + static_cast<ll>(number) * cost;
            prefixCountSum[i] = prefixCountSum[i - 1] + cost;
        }

        // Initialize answer to a very high value
        ll answer = 1e18; // Using 1e18 as a representation of infinity

        // Calculate the minimal total cost to make all numbers equal
        for (int i = 1; i <= n; ++i) {
            auto [number, _] = pairedArray[i - 1];
            // Calculate the cost of making all numbers to the left equal to 'number'
            ll leftCost = static_cast<ll>(number) * prefixCountSum[i - 1] - prefixCostSum[i - 1];
            // Calculate the cost of making all numbers to the right equal to 'number'
            ll rightCost = prefixCostSum[n] - prefixCostSum[i] - static_cast<ll>(number) * (prefixCountSum[n] - prefixCountSum[i]);
            // Update the answer with the minimum cost found so far
            answer = min(answer, leftCost + rightCost);
        }

        return answer; // Return the minimal total cost
    }
};

```

Typescript Solution

```typescript
// Using an alias for readability
type Pair = [number, number];

// Function to find the minimum cost needed to make all elements equal
function minCost(nums: number[], costs: number[]): number {
    let n: number = nums.length; // Get the size of the array
    // 'pairedArray' holds pairs of number and cost for easy sorting and accessing
    let pairedArray: Pair[] = [];

    // Pair each number with its cost and fill the 'pairedArray'
    for (let i = 0; i < n; i++) {
        pairedArray.push([nums[i], costs[i]]);
    }

    // Sort the paired array based on the number
    pairedArray.sort((a, b) => a[0] - b[0]);

    // prefixCostSum stores the total cost up to the i-th element
    let prefixCostSum: number[] = new Array(n + 1).fill(0);
    // prefixCountSum stores the total sum of counts up to the i-th element
    let prefixCountSum: number[] = new Array(n + 1).fill(0);

    // Calculate the prefix sums for cost and count
    for (let i = 1; i <= n; i++) {
        let [number, cost] = pairedArray[i - 1];
        prefixCostSum[i] = prefixCostSum[i - 1] + number * cost;
        prefixCountSum[i] = prefixCountSum[i - 1] + cost;
    }

    // Initialize answer to a very high value
    let answer: number = Number.MAX_SAFE_INTEGER; // Using maximum safe integer value in JS

    // Calculate the minimal total cost to make all numbers equal
    for (let i = 1; i <= n; i++) {
        let [number, _] = pairedArray[i - 1];
        // Calculate the cost of making all numbers to the left equal to 'number'
        let leftCost: number = number * prefixCountSum[i - 1] - prefixCostSum[i - 1];
        // Calculate the cost of making all numbers to the right equal to 'number'
        let rightCost: number = prefixCostSum[n] - prefixCostSum[i] - number * (prefixCountSum[n] - prefixCountSum[i]);
        // Update the answer with the minimum cost found so far
        answer = Math.min(answer, leftCost + rightCost);
    }

    // Return the minimal total cost
    return answer;
}

// Example usage:
// const numsExample = [1,2,3];
// const costsExample = [10,10,10];
// const result = minCost(numsExample, costsExample);
// console.log(result); // Outputs the minimal cost to console

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be broken down into a few components:

1. Sorting the combined list of `nums` and `cost`: This is done using the `sorted()` function, which typically employs the Timsort algorithm, having a time complexity of `O(n log n)` where `n` is the length of the list to be sorted.

2. Populating the `f` and `g` arrays: The two arrays are filled by iterating over `arr` once, which has `n` elements. The operations within the loop are constant time, making this step take `O(n)` time.

3. Calculating the minimum cost `ans`: This involves iterating over each element in `arr` and performing constant time operations, thus taking `O(n)` time.

Overall, the time complexity is dominated by the sorting operation. Hence, the total time complexity of the code is `O(n log n)`.

Space Complexity

The space complexity can be attributed to the extra storage used by:

1. The `arr` list, which stores the sorted `nums` and `cost`, taking `O(n)` space.

2. The `f` and `g` arrays, each of which has `n + 1` elements, thus together taking `2(n + 1)` which is equivalent to `O(n)` space.

3. The `ans` variable, which is constant space `O(1)`.

Therefore, the total space complexity of the code is `O(n)`.