453. Minimum Moves to Equal Array Elements

Medium Array Math

Problem Description

The problem asks us to determine the minimum number of moves required to make all elements of an array equal. A move is defined as incrementing `n - 1` elements of the array by `1`, where `n` is the size of the array. This means that in each move, all elements except one will be increased by one unit.

Intuition

The key insight for solving this problem is to realize that incrementing `n - 1` elements by `1` is the same as decrementing 1 element by `1` with respect to the goal of making all elements equal. Imagine you have a set of numbers, and you want to make them all the same. Instead of adding 1 to all other numbers except the maximum (which is effectively trying to bring all numbers up to the level of the highest number), you can think of it as reducing the max number by 1 to reach the lower numbers. Therefore, the most efficient way to make all elements equal is to decrement the maximum number in the array until all the elements are equal to the minimum number in the array.

The number of moves it takes to make the maximum number equal to the minimum number is the difference between these two numbers. To generalize this, the number of moves required to make all numbers in the array equal to the minimum number is the sum of the differences between every number in the array and the minimum number.

The solution provided follows this philosophy. It calculates the sum of all numbers in the array and subtracts the minimum number times the length of the array. The subtraction term (`min(nums) * len(nums)`) represents the total value of the array if all elements were the minimum number. By subtracting this from the actual sum of the array (`sum(nums)`), we get the total number of increments needed to make all elements equal.

Solution Approach

The solution's approach involves understanding how simple math can translate into an algorithmic optimization. Instead of simulating each move — which would be inefficient for large arrays — we use a single pass calculation.

Algorithmically, we use a single loop provided by the `sum` function to calculate the sum of all elements in `nums`. We also use the built-in `min` function which also iterates over the array to find the minimum value. There are no complex data structures involved: the input is an integer array, and the output is a single integer value representing the minimum moves.

Here's a breakdown of the steps involved in the implementation of the solution:

- Calculate the sum of all numbers in the array with `sum(nums)` which iterates over `nums` once.
- Find the minimum number in the array with `min(nums)` which also iterates over `nums` once.
- Multiply the minimum element by the number of elements in the array with `min(nums) * len(nums)`. This gives the sum of elements if all of them were the minimum value.
- Subtract the sum of all elements as if they were the minimum value from the actual sum of the array elements to get the total number of increments needed.
- The result of the subtraction `sum(nums) - min(nums) * len(nums)` is the answer to the problem, which is returned by the solution function.

This approach uses the property of linearity in arithmetic operations — specifically, the distributive property allows us to condense the whole operation into a single expression. The formula is derived from the insight that bringing the maximum element down to the minimum is equivalent to making all elements equal by the least number of increments.

The algorithm runs in O(n) time complexity because it involves a full pass through the array twice (one for `sum` and one for `min`) and O(1) space complexity, as no additional space is used proportional to the input size (constant extra space is used for storing the sum, minimum, and result).

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have the following array of numbers: `nums = [1, 2, 3, 4]`.

To find the minimum number of moves to make all elements equal by incrementing `n - 1` elements (where `n` is the length of the array), we proceed with the following steps:

1. **Calculate the Sum**: First, we find the sum of all the numbers in the array. For our example, `sum(nums) = 1 + 2 + 3 + 4 = 10`.
2. **Find the Minimum Element**: Next, we determine the minimum element in the array. In this case, `min(nums) = 1`.
3. **Multiply Minimum Element by Array Length**: After that, we multiply the minimum number by the number of elements (`n`) in the array: `min(nums) * len(nums) = 1 * 4 = 4`.
4. **Find the Required Moves**: Finally, the total number of moves required is obtained by subtracting the product found in the previous step from the total sum: `sum(nums) - min(nums) * len(nums) = 10 - 4 = 6`.

Thus, the minimum number of moves required to make all elements equal in this example is `6`.

Repeat these steps with any array, and you'll get the minimum number of moves needed without incrementing each time, but rather by realizing incrementing `n - 1` is the inverse of decrementing `1`. This simple calculation saves time and computational resources.

Solution Implementation

Python

```python
class Solution:
    def minMoves(self, nums) -> int:
        # Calculate the sum of all numbers in the list
        total_sum = sum(nums)
        # Find the minimum value in the list
        min_value = min(nums)
        # Calculate the number of moves required to equalize the values
        # by subtracting the product of the minimum value and list length from the total sum
        moves = total_sum - min_value * len(nums)

        return moves
```

Java

```java
import java.util.Arrays; // Import Arrays utility to use its methods

class Solution {
    public int minMoves(int[] nums) {
        // Calculate the sum of all elements in the array
        int sum = Arrays.stream(nums).sum();

        // Find the minimum element in the array
        int min = Arrays.stream(nums).min().getAsInt();

        // The minimum number of moves is the sum of elements minus
        // the minimum element multiplied by the number of elements in the array.
        // This is because in each move you can increment n-1 elements (all but the max),
        // which is mathematically equivalent to decrementing the maximum element.
        // The target is making all elements equal to the minimum, hence the formula.
        int minMoves = sum - min * nums.length;

        return minMoves;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Include algorithm for using the min function

class Solution {
public:
    // Calculate the minimum number of moves to make all array elements equal
    // by incrementing n-1 elements by 1 each move
    int minMoves(vector<int>& nums) {
        int sum = 0; // Initialize sum of all elements in the array
        int minVal = INT_MAX; // Initialize the smallest value found in the array to the maximum possible integer value

        // Loop through all elements in the nums array
        for (int num : nums) {
            sum += num; // Add the current element's value to the sum
            minVal = min(minVal, num); // Update minVal if the current element's value is smaller
        }

        // The minimum number of moves is the total sum of array elements
        // minus the product of the array's size and the smallest element in the array
        return sum - minVal * nums.size();
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum number of moves required to make all array elements equal,
 * where a move is defined as incrementing n - 1 elements by 1.
 *
 * @param {number[]} nums - an array of integers.
 * @return {number} - the minimum number of moves.
 */
function minMoves(nums: number[]): number {
    // Initialize the minimum element to a large number.
    let minValue = Number.MAX_SAFE_INTEGER;
    // Initialize sum to store the total sum of the array elements.
    let sum = 0;

    // Iterate through the array to find the total sum and the minimum value.
    for (const num of nums) {
        sum += num;
        minValue = Math.min(minValue, num);
    }

    // The minimum number of moves to equalize all elements is
    // the total sum minus the product of the minimum value and the array's length.
    return sum - minValue * nums.length;
}
```

```python
class Solution:
    def minMoves(self, nums) -> int:
        # Calculate the sum of all numbers in the list
        total_sum = sum(nums)
        # Find the minimum value in the list
        min_value = min(nums)
        # Calculate the number of moves required to equalize the values
        # by subtracting the product of the minimum value and list length from the total sum
        moves = total_sum - min_value * len(nums)

        return moves
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)` where `n` is the length of the list `nums`. The reason for this is that the function performs two operations that depend linearly on the size of the input list:

1. The `sum(nums)` operation iterates through the list to calculate the sum of all its elements.
2. The `min(nums)` operation iterates through the list to find the minimum element.

Both these operations take `O(n)` time as each element of `nums` must be visited once.

Therefore, considering these two primary operations, the total time complexity remains `O(n)`.

Space Complexity

The space complexity of the code is `O(1)`. It uses a constant amount of extra space regardless of the size of the input list `nums`. The additional space is used for storing the sum of the elements, the minimum element, and the result of the expression `sum(nums) - min(nums) * len(nums)`. None of these require space that scales with the input size, hence the space complexity is constant.