462. Minimum Moves to Equal Array Elements II

Medium · Array · Math · Sorting

Problem Description

The problem is to find the smallest number of moves to make all elements in an integer array `nums` identical. Each move consists of either incrementing or decrementing any element by `1`. The goal is to figure out the minimum total number of such increments or decrements across the entire array to reach the state where every element is the same.

Understanding the problem through an example makes it easier. If `nums = [1, 2, 3]`, we could transform all elements to `2` (the middle element) by increasing `1` by `1` and decreasing `3` by `1`. This would take a total of `2` moves. If we chose any number other than `2`, it would take more than `2` moves, which demonstrates that the median of the array provides the target value.

The problem specifies that the solution needs to fit within a 32-bit integer, which means we should take care to avoid integer overflows in our calculations.

Intuition

To minimize the number of moves, we need to choose a value that is in some sense central to the array since this reduces the total distance that other elements need to be moved. Mathematically, this is achieved by choosing the median of the array. The median is the central value that separates the higher half from the lower half of the data set. When all elements are moved to the median, the sum of the distances (the absolute differences) is minimized.

Here's the thinking process to arrive at the solution:

- First, sort the array `nums`. Sorting brings the elements in a sequence where the median can be easily identified.
- Next, find the median of the array, which will be the target value all elements should match. The median is located at the central index, which can be found by dividing the length of the array by two. For an even number of elements, any value between the two middle elements will work.
- Calculate and return the sum of absolute differences between each element in the array and the median. The absolute difference ensures we count all moves, whether they're increments or decrements.

The reason this approach is efficient and correct is due to the properties of the median. It ensures the total distance for all elements to reach equality is as small as possible, which translates to the minimum number of moves.

Solution Approach

The implementation of the solution follows the intuition and requires understanding of sorting algorithms and efficient calculation of the central tendency (median) of the list.

Here's the step-by-step implementation walkthrough:

1. **Sorting the List**: The initial step involves sorting the list of numbers. This can be done through any efficient sorting algorithm. Python's built-in `sort` function is typically implemented as Timsort, which has a time complexity of O(n log n).

```
nums.sort()
```

2. **Finding the Median**: After sorting, the median is the middle element of the sorted list if it has an odd length. Otherwise, it is any value between the two middle elements if the list has an even length. Since we are interested in the number of moves, we can select either of the middle elements as our target for an even-length array. In the code, the median is found by taking the element at index `len(nums) >> 1`. The `>>` is a right shift bitwise operator which in this context is equivalent to integer division by 2.

```
k = nums[len(nums) >> 1]
```

3. **Calculating the Moves**: The total number of moves is the sum of the absolute differences between each element in the sorted list and the median. The `abs` function is used for obtaining the absolute value. This part takes advantage of the fact that the median minimizes these differences.

```
return sum(abs(v - k) for v in nums)
```

- The **data structure** used in this solution is the given list `nums`.
- The **algorithm** used includes a sorting technique to order the elements, followed by a linear scan to calculate the total moves.
- The **pattern** utilized here is finding a value to minimize the sum of distances which is a classical optimization strategy.

The reason why these particular choices were made in the approach is because sorting the list first simplifies the determination of the median. Once the list is sorted, elements below the median are all smaller and those above are all bigger. Thus, when each element is moved to the median, the total number of moves is minimized. Since we are interested in the magnitude of the changes rather than the direction, we use absolute values.

Example Walkthrough

Let's consider the array `nums = [1, 5, 2, 4]`.

1. **Sorting the List**: Initially, we sort the array `nums`. After sorting, it looks like this: `[1, 2, 4, 5]`.

```
nums.sort()  # results in nums being [1, 2, 4, 5]
```

2. **Finding the Median**: Since the array has an even number of elements (4 elements), any value between the two middle elements can be used. Here, we can choose either `2` or `4`. For simplicity, and following the approach, we'll select the lower index, so our target is `2`.

```
k = nums[len(nums) >> 1]  # results in k being 2
```

3. **Calculating the Moves**: We calculate the sum of absolute differences between each element and `k`, our median value, which in this case is `2`.

```
moves = sum(abs(v - k) for v in nums)  # moves is abs(1-2) + abs(2-2) + abs(4-2) + abs(5-2)
```

   This gives us `1 + 0 + 2 + 3 = 6` moves. Thus, to make all elements in the array identical, we need a minimum of `6` moves in this case:

   - Element at index 0: `1` must be incremented by `1` to become `2` (1 move)
   - Element at index 1: `2` is already `2`, so no moves needed (0 move)
   - Element at index 2: `4` must be decremented by `2` to become `2` (2 moves)
   - Element at index 3: `5` must be decremented by `3` to become `2` (3 moves)

To summarize, our optimal solution involves incrementing and decrementing elements to turn `[1, 2, 4, 5]` into `[2, 2, 2, 2]` with a minimum of `6` moves.

Solution Implementation

Python

```python
class Solution:
    def minMoves2(self, numbers: List[int]) -> int:
        # Sort the list of numbers to find the median more easily
        numbers.sort()

        # Calculate the median by taking the middle element after sorting
        # Note: 'len(numbers) >> 1' is the bitwise right shift operation, equivalent to 'len(numbers) // 2'
        median = numbers[len(numbers) // 2]

        # Calculate the minimum number of moves required by summing up the absolute differences
        # between each number and the median
        # The absolute difference represents how far each number is from the median,
        # summing them up gives the minimum moves to equalize the numbers
        total_moves = sum(abs(number - median) for number in numbers)

        # Return the total number of moves
        return total_moves
```

Java

```java
class Solution {
    public int minMoves2(int[] nums) {
        // Sort the input array
        Arrays.sort(nums);

        // Find the median of the array which will be our target number
        // Using bitwise right shift for finding the mid element (equivalent to dividing by 2)
        int median = nums[nums.length >> 1];

        // Initialize the number of moves required to bring all elements to the median
        int moves = 0;

        // Calculate the total number of moves required
        // by summing up the distance of each element from the median
        for (int num : nums) {
            moves += Math.abs(num - median);
        }

        // Return the total number of moves
        return moves;
    }
}
```

C++

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

class Solution {
public:
    // Function to find the minimum number of moves required to make all the array elements equal,
    // where a move is incrementing or decrementing an element by 1.
    int minMoves2(std::vector<int>& nums) {
        // First, we sort the input array.
        std::sort(nums.begin(), nums.end());

        // Find the median of the array. This will be the target value for all elements.
        // Since we're making all values equal to this target,
        // it minimizes the total number of moves required.
        int median = nums[nums.size() / 2];

        // Initialize the answer variable to accumulate the total moves needed.
        int totalMoves = 0;

        // Iterate over the array, adding the absolute difference between
        // each element and the median to our total moves.
        // This gives us the total moves required to make each element equal to the median.
        for (int value : nums) {
            totalMoves += std::abs(value - median);
        }

        // Return the total number of moves to make all elements equal.
        return totalMoves;
    }
};
```

TypeScript

```typescript
/**
 * This function calculates the minimum number of moves required to
 * make all array elements equal. A single move is incrementing or
 * decrementing an array element by 1.
 *
 * @param {number[]} numbers - The array of integers.
 * @return {number} - The minimum number of moves to make all elements equal.
 */
function minMoves2(numbers: number[]): number {
    // Sort the array in ascending order so that we can find the median.
    numbers.sort((a, b) => a - b);

    // Find the median of the array, which is the middle element after the sort,
    // or the average of two middle elements if the array has an even number of elements.
    // This median will be the target value for all elements.
    const median = numbers[numbers.length >> 1]; // '>> 1' is equivalent to dividing by 2 but faster.

    // Reduce the array, accumulating the total moves needed by adding the absolute
    // differences between each element and the median.
    return numbers.reduce((totalMoves, currentValue) => totalMoves + Math.abs(currentValue - median), 0);
}
```


```python
class Solution:
    def minMoves2(self, numbers: List[int]) -> int:
        # Sort the list of numbers to find the median more easily
        numbers.sort()

        # Calculate the median by taking the middle element after sorting
        # Note: 'len(numbers) >> 1' is the bitwise right shift operation, equivalent to 'len(numbers) // 2'
        median = numbers[len(numbers) // 2]

        # Calculate the minimum number of moves required by summing up the absolute differences
        # between each number and the median
        # The absolute difference represents how far each number is from the median,
        # summing them up gives the minimum moves to equalize the numbers
        total_moves = sum(abs(number - median) for number in numbers)

        # Return the total number of moves
        return total_moves
```


Time and Space Complexity

Time Complexity

The provided Python code involves sorting an array and then iterating through the sorted array once.

1. The `nums.sort()` function has a time complexity of `O(n log n)`, where `n` is the length of the list `nums`. This is because the sorting algorithm used by Python's sort function (Timsort) has this time complexity for sorting an average list.
2. The list comprehension `sum(abs(v - k) for v in nums)` iterates through the sorted list exactly once to compute the absolute differences and sum them up, which has a time complexity of `O(n)`.

When combining both steps, the dominant factor is the sorting step, so the overall time complexity is `O(n log n)`.

Space Complexity

The space complexity of the code is determined by the additional space used beyond the input size.

1. The `sort()` method sorts the list in place and does not require additional space proportional to the size of the input list, so the space complexity for this step is `O(1)`.
2. The computation of the sum using a generator expression also does not require space proportional to the input size since it computes the sum on the fly.

Therefore, the total space complexity of the provided solution is `O(1)`.