# 1509. Minimum Difference Between Largest and Smallest Value in Three Moves

Medium Greedy Array Sorting

Leetcode Link

## Problem Description

You are given an array `nums` which consists of integers. The main task is to find out how to minimize the difference between the largest and smallest numbers in the array, after you're allowed to perform at most three modifications. Each modification lets you select one number from the array and change it to any value you wish (this could be any integer, not necessarily one that was already in the array).

Essentially, you want to make the array elements closer to each other in value while having at most three opportunities to adjust any of the elements. The goal is to return the smallest possible difference (also known as range) between the maximum and minimum values after doing these changes.

## Intuition

The intuition behind the solution is drawn from the understanding that the largest difference in values within the array is between the lowest and highest numbers. If the array has fewer than 5 elements, no modification is needed because you can at most delete all four elements to make them equal, resulting in a difference of 0.

For arrays with 5 or more elements, the strategy to minimize the range would be to either raise the value of the smallest numbers or lower the value of the largest numbers. Since we can make at most three moves, it leaves us with a few scenarios on which numbers to change:

1. Change the three smallest numbers: This would make the smallest number to be the one that was initially the fourth smallest.
2. Change the two smallest numbers and the largest number: This can potentially bring down the largest number and increase the smallest ones, possibly narrowing the range even further.
3. Change the smallest number and the two largest numbers: Similar rationale as above but with a different balance between how much you adjust the lowest and highest numbers.
4. Change the three largest numbers: The largest number become the one that was initially the fourth largest.

By sorting the array first, we can easily access the smallest and largest values. Trying out all four scenarios should give us the minimum possible difference since they encompass all possible ways we can use our three moves to minimize the range of the array.

The solution iterates through the sorted array and computes the difference for each scenario, always keeping track of the smallest difference found. Finally, the smallest difference computed signifies the least possible range achievable after three or fewer modifications.

## Solution Approach

The implementation of the solution makes use of basic concepts like sorting and iterating through arrays in Python.

Firstly, the array is sorted to organize the integers in increasing order. Sorting allows easy access to the smallest and largest values, which are the targets for potential changes. The `sort()` method is used for this purpose, which sorts the array in place.

After sorting, the algorithm checks the length of the array (stored in variable `n`). If the array has fewer than 5 elements ($n < 5$), it returns `0` immediately because, with three moves, we can create at least four equal values (which is the whole array if its length is less than 5), resulting in a minimum difference of zero.

If there are 5 or more elements, the algorithm considers four possible scenarios for amending the array using at most three moves:

- Change the three smallest numbers ($nums[3] - nums[0]$).
- Change the two smallest numbers and the largest number ($nums[n - 1] - nums[2]$).
- Change the smallest number and the two largest numbers ($nums[n - 2] - nums[1]$).
- Change the three largest numbers ($nums[n - 3] - nums[0]$).

These scenarios are checked within a loop that runs four times (since `range(4)` yields `0, 1, 2, 3`). Within the loop, the variable `l` represents the index of the small end and $n - 1 - r$, where $r = 3 - l$, represents the index of the large end of the array.

The difference between the selected elements for each scenario is calculated and compared using the `min()` function. The `ans` variable keeps track of the minimum difference found at any point in the loop.

At the end of the loop, `ans` holds the smallest possible difference between the largest and smallest values of `nums` after at most three moves. It is then returned as the result of the function.

```python
class Solution:
    def minDifference(self, nums: List[int]) -> int:
        n = len(nums)
        if n < 5:
            return 0
        nums.sort()
        ans = inf
        for l in range(4):
            r = 3 - l
            ans = min(ans, nums[n - 1 - r] - nums[l])
        return ans
```

In the given Python code, `inf` represents an infinitely large value. This initialization ensures that any difference calculated will be smaller than `inf`, and thus `ans` will be set properly after the first iteration of the loop.

Overall, the solution is straightforward but effective, combining sorting with simple arithmetic operations and a loop that leverages the problem constraints smartly.

### Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have the following array of integers:

```
nums = [1, 5, 10, 20, 30]
```

Firstly, this array is already sorted, but in practice, we would sort it to make it easier to find the smallest and largest numbers. Since we are allowed at most three modifications, and there are more than four elements in the array (5 in this case), we apply the following four scenarios to find the minimum range:

1. Change the three smallest numbers:
   - After changing, the new smallest number would be `nums[3]`, which is `20`.
   - The largest number would remain `nums[4]`, which is `30`.
   - The difference between the largest and smallest is $30 - 20 = 10$.
2. Change the two smallest numbers and the largest number:
   - After changes, the smallest number would be `nums[2]`, which is `10`.
   - The largest number would now be `nums[3]`, which is `20`.
   - The difference between the largest and smallest is $20 - 10 = 10$.
3. Change the smallest number and the two largest numbers:
   - The new smallest number would be `nums[1]`, which is `5`.
   - The new largest number would be `nums[2]`, which is `10`.
   - The difference is $10 - 5 = 5$.
4. Change the three largest numbers:
   - The smallest number would stay `nums[0]`, which is `1`.
   - The new largest number would be `nums[1]`, which is `5`.
   - The difference is $5 - 1 = 4$.

We then find the smallest of all these differences, which in this case is `4` from the last scenario. This is our answer: by changing the three largest numbers in the array, we minimize the difference to just 4.

The actual Python function would work as follows:

```
nums = [1, 5, 10, 20, 30]
n = len(nums) # Here, n = 5
if n < 5:
    return 0
else:
    nums.sort() # The array is already sorted but this step is necessary.
    ans = float('inf')
    for l in range(4): # We iterate four times to check every scenario.
        r = 3 - l
        ans = min(ans, nums[n - 1 - r] - nums[l]) # This finds the smallest range
    # The answer here would be 4 after the loop
    return ans
```

This approach efficiently considers the possibilities using the allowed number of modifications to find the minimal difference between the maximum and minimum elements of the array.

## Python Solution

```python
from typing import List

class Solution:
    def minDifference(self, nums: List[int]) -> int:
        # Determine the length of the input list
        num_len = len(nums)

        # If the list has less than 5 elements, return 0 as per problem statement,
        # because we can remove all elements to minimize difference to zero
        if num_len < 5:
            return 0

        # Sort the list to easily find the smallest difference
        nums.sort()

        # Initialize minimum difference to infinity, as we are looking for the minimum
        min_diff = float('inf')

        # We can remove 3 elements either from the beginning, the end, or both.
        # We loop through scenarios where we take from 0 to 3 elements from the start,
        # and respectively 3 to 0 elements from the end to balance out the total removed elements count.
        for left in range(4):
            right = 3 - left
            # Calculate the difference between the current left-most element we are considering
            # and the current right-most element we are considering
            current_diff = nums[num_len - 1 - right] - nums[left]
            # Update the minimum difference if the current difference is smaller
            min_diff = min(min_diff, current_diff)

        # Return the smallest difference we found
        return min_diff
```

## Java Solution

```java
class Solution {
    public int minDifference(int[] nums) {
        // Get the length of the nums array
        int length = nums.length;

        // If there are less than 5 elements, return 0 since we can remove all but 4 elements
        if (length < 5) {
            return 0;
        }

        // Sort the array to make it easier to find the minimum difference
        Arrays.sort(nums);

        // Initialize the minimum difference to a very large value
        long minDiff = Long.MAX_VALUE;

        // Loop through the array and consider removing 0 to 3 elements from the beginning
        // and the rest from the end to minimize the difference
        for (int leftRemoved = 0; leftRemoved <= 3; ++leftRemoved) {
            int right = 3 - left;
            // Calculate the difference between the selected elements
            long diff = nums[length - 1 - right] - nums[left];
            // Update the minimum difference if the current one is smaller
            minDiff = Math.min(minDiff, diff);
        }

        // Return the minimum difference as an integer
        return (int) minDiff;
    }
}
```

## C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for std::sort and std::min

class Solution {
public:
    // Function to find the minimum difference between the largest and smallest values
    // after at most 3 elements are removed from the array.
    int minDifference(std::vector<int>& nums) {
        int numElements = nums.size(); // Store the number of elements in nums

        // If there are fewer than 5 elements, return 0 since we can remove all but one element
        if (numElements < 5) {
            return 0;
        }

        // Sort the array in non-decreasing order
        std::sort(nums.begin(), nums.end());

        // Initialize the answer to a large number
        long long answer = 1LL << 60;

        // Try removing 0 to 3 elements from the start and from the end in such a way
        // that the total number of elements removed is 3 and find the minimum difference
        for (int leftRemoved = 0; leftRemoved <= 3; ++leftRemoved) {
            int rightRemoved = 3 - leftRemoved; // Ensure total of 3 elements are removed from both ends
            // Update the answer with the minimum of the current answer and the difference
            // between the current largest and smallest values
            answer = std::min(answer, static_cast<long long>(nums[numElements - 1 - rightRemoved] - nums[leftRemoved]));
        }

        // Return the final answer
        return static_cast<int>(answer);
    }
};
```

## Typescript Solution

```typescript
// Importing the 'sort' utility method from a library like Lodash could be helpful.
// In TypeScript, arrays have a built-in sort method, eliminating the need for a separate import.

// Function to find the minimum difference between the largest and smallest values
// after at most 3 elements are removed from the array.
function minDifference(nums: number[]): number {
    // Store the number of elements in nums.
    let numElements: number = nums.length;

    // If there are fewer than 5 elements, return 0 since we can remove all but one element.
    if (numElements < 5) {
        return 0;
    }

    // Sort the array in non-decreasing order (ascending).
    nums.sort((a, b) => a - b);

    // Initialize the answer to a large number.
    let answer: number = Number.MAX_SAFE_INTEGER;

    // Try removing 0 to 3 elements from the start and from the end in such a way
    // that the total number of elements removed is 3 and find the minimum difference.
    for (let leftRemoved = 0; leftRemoved <= 3; ++leftRemoved) {
        let rightRemoved: number = 3 - leftRemoved; // Ensure total of 3 elements are removed from both ends.

        // Update the answer with the minimum of the current answer and the difference
        // between the current largest and smallest values.
        answer = Math.min(answer, nums[numElements - 1 - rightRemoved] - nums[leftRemoved]);
    }

    // Return the final answer.
    return answer;
}

// Example usage:
// const result = minDifference([1,5,0,10,14]);
// console.log(result); // Output would be the minimum difference obtained.
```

## Time and Space Complexity

### Time Complexity

The time complexity of the algorithm is determined primarily by the sorting operation. The sort method in Python is implemented using Timsort, which has an average and worst-case complexity of $O(n \log n)$, where $n$ is the number of elements in the list.

The for loop iterates a constant 4 times, which does not depend on the size of the input, so it contributes an additional $O(1)$ to the time complexity.

Therefore, the total time complexity of the code is $O(n \log n)$ due to the sort operation, which is the dominant factor.

### Space Complexity

The space complexity is the amount of additional memory space required by the algorithm as the size of the input changes. In this case, the sorting operation can generally be done in-place with a space complexity of $O(1)$.

However, Python's sorting algorithm may require $O(n)$ space in the worst case because it can be a hybrid of merge sort, which requires additional space for merging. Since `nums` is sorted in-place, and no other data structures depend on the size of `n` are used, the space complexity is $O(1)$ in the best case and $O(n)$ in the worst case.

Therefore, the overall space complexity of the code is $O(n)$ in the worst case due to the potential additional space needed for sorting.