2567. Minimum Score by Changing Two Elements

Medium Greedy Array Sorting

Leetcode Link

Problem Description

We are given an array of integers called `nums`, where the indices start at 0. We need to find the score of the array, which is defined as the sum of its high score and low score. The high score is the maximum difference in value between any two distinct elements in the array, and the low score is the minimum difference in value between any two distinct elements in the array. In mathematical terms, for two indices i and j where `0 <= i < j < len(nums)`, high score is `max(|nums[i] - nums[j]|)` and low score is `min(|nums[i] - nums[j]|)`.

However, we have the possibility to change the value of at most two elements in the array `nums` to reduce the score to a minimum. Our task is to determine the minimum possible score after these changes, while recognizing that we can only perform at most two value modifications.

Note: `|x|` in the description represents the absolute value of x.

Intuition

To minimize the score of an array, we need to look at both ends of the sorted array since the high score depends on the largest difference (which will be between the smallest and the largest elements) and the low score depends on the smallest non-zero difference (which will be between any two consecutive elements). Since we can change at most two elements, the ideal candidates for change are toward the ends of the sorted array to have the most significant impact on the high score.

Firstly, we sort the array to make it easier to find the candidates for the elements that we could potentially change to minimize the score. Once the array is sorted, we know that the high score comes from the difference between the first and the last element in the sorted array, and thus changing either of these will have the most significant impact on reducing the high score. Similarly, the low score will be affected if we change elements that are adjacent to each other and result in the smallest difference.

Hence, we need to determine the minimum score by changing at most two elements, considering:

1. Changing the last element and/or the second to last element may minimize the high score.
2. Changing the first element and/or the second element may minimize the low score.

Our solution checks the three possible scenarios where we can achieve the minimum score by:

- Changing the first and third elements, which affects both the high and low scores.
- Changing the second and last but one elements, which also affects both the high and low scores.
- Changing the first and last but two elements, yet again affecting both scores.

The `minimizeSum` function sorts `nums` and then returns the minimum score by evaluating the three listed score differences. We don't need to evaluate the difference between the first and second or last and second to last elements for the low score because, in the sorted array, changing either the first or the last two elements would yield the smallest possible low score regardless of the specific values of the adjacent elements.

Solution Approach

The implementation of the solution involves using a simple but effective algorithm that utilizes sorting and minimal computation. Here's how it works:

1. **Sorting**: First, the solution involves sorting the array `nums`. This is crucial as it allows us to easily access the values that will give us the highest and lowest scores. In sorted order, the smallest difference (low score) is between consecutive elements, and the largest difference (high score) is between the first and the last elements.
2. **Computing Differences**: After sorting, the score is determined by the difference between specific elements of the array. We look at three possible pairs of elements whose values we would change to minimize the overall score:
   - The difference between the last and the third element (`nums[-1] - nums[2]`): This represents changing the first two elements to values close to the third element, which would potentially minimize both the high and low scores.
   - The difference between the second to last and the second element (`nums[-2] - nums[1]`): This represents changing the last and the first element to values near the second element, affecting both ends of the sorted array.
   - The difference between the third to last and the first element (`nums[-3] - nums[0]`): This represents changing the last two elements to values close to the first element, again potentially minimizing the high and low scores.
3. **Choosing Minimum Difference**: The solution then evaluates these three possible scores and returns the smallest one using the `min` function. The smallest value would be the minimum score achievable after changing at most two elements of the `nums` array to bring them closer to the other elements, thereby minimizing the difference between them.
4. **Returning Result**: Ultimately, the function `minimizeSum` returns the smallest of these three calculated differences, which corresponds to the minimum possible score for the array `nums` after changing at most two elements.

Here is the actual Python code snippet for the solution approach:

```python
class Solution:
    def minimizeSum(self, nums: List[int]) -> int:
        nums.sort()
        return min(nums[-1] - nums[2], nums[-2] - nums[1], nums[-3] - nums[0])
```

The use of sorting makes the algorithm efficient, and the straightforward computation of the three cases ensures that we consider all possible ways to reduce the overall score with up to two changes. The use of the built-in `sort` method and `min` function make the solution concise and elegant, relying on Python's rich standard library to handle the necessary computations efficiently.

Example Walkthrough

Let's assume we have an array `nums` with the following elements: `[4, 1, 2, 9]`. Here is how we can apply the solution approach step by step:

1. **Sorting**: We start by sorting the array `nums`. After sorting, our array becomes: `[1, 2, 4, 9]`.
2. **Computing Differences**: With the array sorted, we compute the differences for the three scenarios mentioned in the solution approach:
   - The difference between the last (9) and the third (4) element is `9 - 4 = 5`. This is the score if we modify the first two elements (1 and 2) to values close to the third element (4).
   - The difference between the second to last (4) and the second (2) element is `4 - 2 = 2`. This is the score if we modify the first (1) and the last (9) element to values close to the second element (2).
   - The difference between the third to last (2) and the first (1) element is `2 - 1 = 1`. This is the score if we modify the last two elements (4 and 9) to values close to the first element (1).
3. **Choosing Minimum Difference**: Out of the differences calculated (5, 2, and 1), the smallest one is 1. This represents the scenario where we change the last two elements to values close to the first element, potentially minimizing both the high and low scores.
4. **Returning Result**: The function `minimizeSum` would take our array and ultimately return `1`, which is the minimum possible score after changing at most two elements in the array.

The code that would implement this solution for our sample array is as follows:

```python
def minimizeSum(nums):
    nums.sort()
    return min(nums[-1] - nums[2], nums[-2] - nums[1], nums[-3] - nums[0])

# Example usage:
nums = [4, 1, 2, 9]
print(minimizeSum(nums)) # Prints 1, which is the minimum possible score
```

In this example, modifying the values at the end of the array has the greatest impact on reducing the score, so the optimization strategy is to adjust these elements to minimize the array's high and low scores as much as possible within the constraints of changing at most two elements.

Python Solution

```python
from typing import List

class Solution:
    def minimize_sum(self, nums: List[int]) -> int:
        # Sort the list of numbers in non-decreasing order.
        nums.sort()

        # Calculate the minimum possible sum by considering the absolute difference
        # between either the largest and third largest, the second largest and
        # second smallest, or the third largest and the smallest number.
        # This works since the list is sorted.
        min_sum = min(
            nums[-1] - nums[2],  # Difference between the largest and third-largest number.
            nums[-2] - nums[1],  # Difference between the second-largest and second-smallest number.
            nums[-3] - nums[0]   # Difference between the third-largest and smallest number.
        )

        # Return the minimum sum calculated.
        return min_sum

# Example usage:
# solution = Solution()
# result = solution.minimize_sum([1, 2, 3, 4, 5])
# print(result)  # This would output the result of the minimize_sum function.
```

Java Solution

```java
class Solution {
    public int minimizeSum(int[] nums) {
        // Sort the array to establish a non-decreasing order
        Arrays.sort(nums);
        int length = nums.length;

        // Calculate the three possible differences based on the given formula
        // These differences seem to represent some form of calculation after sorting
        // a corresponds to the difference between the last and third element from the start
        int a = nums[length - 1] - nums[2];
        // b corresponds to the difference between the second-to-last and second element
        int b = nums[length - 2] - nums[1];
        // c corresponds to the difference between the third-to-last and first element
        int c = nums[length - 3] - nums[0];

        // Return the minimum value among a, b, and c
        // The goal is to find the smallest of these three differences
        return Math.min(a, Math.min(b, c));
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Required for std::sort and std::min
using namespace std;

class Solution {
public:
    // Function to calculate the minimized sum
    // Takes a vector of integers as the input
    int minimizeSum(vector<int>& nums) {
        // Sort the input array in non-decreasing order
        sort(nums.begin(), nums.end());

        // Calculate the size of the array once to avoid multiple calculations
        int n = nums.size();

        // To minimize the sum, we look at the three smallest values
        // and the three largest values after sorting the array.
        // We are considering three possibilities:
        //   1. nums[n - 1] (last element) - nums[2] (third element)
        //   2. nums[n - 2] (second to last element) - nums[1] (second element)
        //   3. nums[n - 3] (third to last element) - nums[0] (first element)
        // We return the minimum difference among these three
        return min({
            nums[n - 1] - nums[2],
            nums[n - 2] - nums[1],
            nums[n - 3] - nums[0]
        });
    }
};
```

Typescript Solution

```typescript
// Function to minimize the sum difference between elements of an array
// by removing any three elements.
function minimizeSum(nums: number[]): number {
    // First, sort the array in ascending order.
    nums.sort((a, b) => a - b);

    // Calculate the number of elements in the array.
    const n = nums.length;

    // Calculate the minimum difference after removing three elements.
    // We consider three cases:
    // - Removing the three largest elements.
    // - Removing the two largest elements and the smallest element.
    // - Removing the largest element and the two smallest elements.
    // The minimum difference is found by taking the smallest difference from these cases.
    // It's based on the fact that, to minimize the sum difference, we should remove
    // elements from the ends of the sorted array.
    return Math.min(
        // Case 1: Remove the three largest elements.
        nums[n - 1] - nums[3],
        // Case 2: Remove the two largest and the smallest element.
        nums[n - 2] - nums[2],
        // Case 3: Remove the largest and the two smallest elements.
        nums[n - 3] - nums[1]
    );
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the sorting of the list `nums`. The sort operation has a time complexity of `O(n log n)` where `n` is the number of elements in the list. Since the successive operations after the sorting take constant time, the overall time complexity of the function is `O(n log n)`.

Space Complexity

The space complexity of the code is `O(1)`. This is because the sorting is done in-place (assuming the sort method in Python is implemented in-place), and only a constant amount of extra space is used for variables in the calculations after sorting.