910. Smallest Range II

Medium Greedy Array Math Sorting

Problem Description

In this problem, you are given an integer array called `nums` and an integer `k`. Your task is to modify each element in the array by either adding `k` to it or subtracting `k` from it. After modifying each element in this manner, the `score` of the array is defined as the difference between the maximum and minimum elements in the modified array. The objective is to determine the minimum `score` that can be achieved by any combination of adding or subtracting `k` to/from the elements of `nums`.

Here is an example to illustrate:

Suppose `nums = [1, 3, 6]` and `k = 3`. You could transform `nums` as follows:

- Adding `k` to the first element: `[1 + 3, 3, 6] = [4, 3, 6]`
- Subtracting `k` from the second element: `[4, 3 - 3, 6] = [4, 0, 6]`
- Subtracting `k` from the third element: `[4, 0, 6 - 3] = [4, 0, 3]`

The score is then `max(4, 0, 3) - min(4, 0, 3) = 4 - 0 = 4`.

The problem is asking you to find the minimum score possible, that is, the smallest difference between the highest and lowest number in the array after each element has been increased or decreased by `k`.

Intuition

The intuition behind the solution involves recognizing that sorting the array can help in minimizing the score. By sorting, you can ensure that the operation that you perform (addition or subtraction) will not increase the range unnecessarily.

Here's why sorting helps:

- After sorting `nums`, the smallest and largest elements are `nums[0]` and `nums[-1]`, respectively. The initial score is `nums[-1] - nums[0]`.
- Consider a pivot point at index `i` in the sorted array. Everything to the left of `i` could be increased by `k`, and everything to the right of `i` (including element at `i`) could be decreased by `k`. This creates two "blocks" within the array: one with increased values, and one with decreased values.
- The new minimum possible value is the minimum of `nums[0] + k` (left-most element of the increased block) and `nums[i] - k` (left-most element of the decreased block).
- The new maximum possible value is the maximum of `nums[i - 1] + k` (right-most element of the increased block) and `nums[-1] - k` (right-most element of the decreased block).
- By iterating through all possible pivot points (from `1` to `len(nums) - 1`), you can find the minimum score by comparing this with the previously calculated minimum `ans`.

The key observation here is that by sorting and choosing an appropriate pivot, one can minimize the difference between the maximum and minimum values affected by the addition or subtraction of `k`, leading directly to a solution that iterates through possible pivot points to find the minimum score.

Solution Approach

The solution follows a simple yet effective approach leveraging sorting and iteration.

Algorithm:

1. Sort the `nums` array. This will help us easily identify the smallest and largest elements and ensures that we do not increase the range unnecessarily.
2. Initialize the variable `ans` with the initial score, which is the difference between the last element and the first element of the sorted array (`nums[-1] - nums[0]`).
3. Iterate through the array starting from index `1` up to `len(nums) - 1`. The reason we start at `1` is because we are considering the pivot point where the array is divided into two parts: one that will get the addition of `k` and the other the subtraction. There is no point in considering index `0` for this pivot as the sorted array's first element cannot be the start of the subtraction section.
4. For each index `i`, calculate the minimum and maximum values as follows:
   - Calculate the new minimum value `mi` as the minimum between `nums[0] + k` and `nums[i] - k`. The `nums[0] + k` represents the smallest possible value after the increment and `nums[i] - k` represents the smallest value for the section of the array where `k` is subtracted.
   - Calculate the new maximum value `mx` as the maximum between `nums[i - 1] + k` and `nums[-1] - k`. The `nums[i - 1] + k` is the largest value of the incremented section, and `nums[-1] - k` is the largest value for the decremented section.
5. Update the score (`ans`) to be the minimum value between the current `ans` and the difference `mx - mi`. This ensures that with each iteration, we are considering the lowest possible range after applying our operation at each pivot.
6. Once the loop completes, `ans` contains the minimum score achievable after performing the add or subtract operation at each index, based on the given `k`.

Data Structures:

- The sorted array itself is the primary data structure used. No additional data structures are required.

Patterns:

- The use of sorting to establish a predictable order of elements, hence allowing for a methodical approach to finding the solution.
- Iteration to explore possible optimal solutions by checking pivot points in the array.
- Decision-making to update the candidate solution (`ans`) based on comparisons calculated within the loop.

By following these steps, the algorithm ensures that we calculate the minimum range after adding or subtracting `k` from each element in the most efficient manner. It elegantly handles the problem by transforming it into a series of operations where we only need to look at the edges of the two blocks created by the pivot point.

Example Walkthrough

Let's go through the solution approach with a small example to better illustrate the algorithm.

Suppose we have the following `nums` array and integer `k`:

```
nums = [4, 7, 1]
k = 5
```

According to the problem, we have to either add `k` to or subtract `k` from each element in the array to achieve the smallest possible score (the difference between the maximum and minimum elements of the array). Let's follow the solution approach:

1. **Sort the array**:

   ```
   nums.sort() -> nums = [1, 4, 7]
   ```

2. **Initialize the score (`ans`)**:

   ```
   ans = nums[-1] - nums[0] -> ans = 7 - 1 -> ans = 6
   ```

3. **Iterate through the array starting from index `1`**:

   For each index `i`, we will calculate the potential new minimum (`mi`) and maximum (`mx`) values after either adding or subtracting `k`:

   - At index `i = 1` (element `4`):

     ```
     mi = min(nums[0] + k, nums[i] - k) -> mi = min(1 + 5, 4 - 5) -> mi = min(6, -1) -> mi = -1
     mx = max(nums[i - 1] + k, nums[-1] - k) -> mx = max(1 + 5, 7 - 5) -> mx = max(6, 2) -> mx = 6
     ```

     Now, we update the `ans` if `mx - mi` is smaller than the current `ans`:

     ```
     ans = min(ans, mx - mi) -> ans = min(6, 6 - (-1)) -> ans = min(6, 7) -> ans = 6
     ```

     (No change in `ans` as the `score` is still 6 which is not better than the previous score)

   - At index `i = 2` (element `7`):

     ```
     mi = min(nums[0] + k, nums[i] - k) -> mi = min(1 + 5, 7 - 5) -> mi = min(6, 2) -> mi = 2
     mx = max(nums[i - 1] + k, nums[-1] - k) -> mx = max(4 + 5, 7 - 5) -> mx = max(9, 2) -> mx = 9
     ```

     Now, update the `ans`:

     ```
     ans = min(ans, mx - mi) -> ans = min(6, 9 - 2) -> ans = min(6, 7) -> ans = 6
     ```

     Again, the `score` is 7, which does not improve the `ans`.

4. **Complete the loop**: The algorithm has finished checking all possible pivot points in the array.
5. **Result**: The `ans` calculated is `6`, which means:

   ```
   The minimum score achievable after adding or subtracting `k` from each element in nums is 6.
   ```

In this example, the modifications to the elements are unnecessary, as the initial score is already minimal. The algorithm efficiently identifies this by analyzing the potential effects of adding and subtracting `k` at each pivot point in the sorted array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def smallestRangeII(self, nums: List[int], k: int) -> int:
        # First, sort the numbers to organize them in ascending order.
        nums.sort()

        # The initial range would be the max value minus the min value.
        smallest_range = nums[-1] - nums[0]

        # Loop through the sorted numbers, starting from the second element,
        # to find the minimum possible range.
        for i in range(1, len(nums)):
            # Calculate the possible minimum by adding k to the smallest value
            # and subtracting k from the current value.
            possible_min = min(nums[0] + k, nums[i] - k)

            # Calculate the possible maximum by adding k to the previous value
            # and subtracting k from the maximum value.
            possible_max = max(nums[i - 1] + k, nums[-1] - k)

            # Update the smallest range if a smaller one is found.
            smallest_range = min(smallest_range, possible_max - possible_min)

        # Finally, return the smallest range after considering all elements.
        return smallest_range
```

Java

```java
class Solution {
    public int smallestRangeII(int[] nums, int k) {
        // First, sort the input array to deal with numbers in a sorted order.
        Arrays.sort(nums);
        // Get the length of the array
        int n = nums.length;
        // Calculate the initial range from the first and last element of the sorted array.
        int minRange = nums[n - 1] - nums[0];

        // Iterate through the array starting from the second element
        for (int i = 1; i < n; ++i) {
            // Calculate the minimum possible value after adding or subtracting k to the current or first element
            int currentMin = Math.min(nums[0] + k, nums[i] - k);
            // Calculate the maximum possible value after adding or subtracting k to the previous or last element
            int currentMax = Math.max(nums[i - 1] + k, nums[n - 1] - k);
            // Find the smallest range by comparing the current computed range and the previously stored minimum range
            minRange = Math.min(minRange, currentMax - currentMin);
        }
        // Return the minimum range found
        return minRange;
    }
}
```

C++

```cpp
class Solution {
public:
    // Defines the function which will find the smallest range after modification
    int smallestRangeII(vector<int>& nums, int k) {
        // Initially, sort the array in non-decreasing order
        sort(nums.begin(), nums.end());

        int numsSize = nums.size(); // Store the size of the nums vector
        // Compute the initial range between the largest and smallest numbers
        int minRange = nums[numsSize - 1] - nums[0];

        // Iterate through the sorted numbers starting from the second element
        for (int i = 1; i < numsSize; ++i) {
            // Determine the new minimum by comparing the increased smallest number
            // and the decreased current number
            int newMin = min(nums[0] + k, nums[i] - k);
            // Determine the new maximum by comparing the increased previous number
            // and the decreased largest number
            int newMax = max(nums[i - 1] + k, nums[numsSize - 1] - k);
            // Update the minRange with the minimum of the current and new ranges
            minRange = min(minRange, newMax - newMin);
        }

        // Return the smallest range found
        return minRange;
    }
};
```

TypeScript

```typescript
// Array `nums` holds the numbers and `k` is the allowed modification range
let nums: number[];
let k: number;

// Sorts the array in non-decreasing order
const sortArray = (a: number, b: number) => a - b;

// Finds the smallest range after modification
function smallestRangeII(nums: number[], k: number): number {
    // Sort the array in non-decreasing order
    nums.sort(sortArray);

    // Store the size of the nums array
    const numsSize: number = nums.length;

    // Compute the initial range between the largest and smallest numbers
    let minRange: number = nums[numsSize - 1] - nums[0];

    // Iterate through the sorted numbers starting from the second element
    for (let i = 1; i < numsSize; ++i) {
        // Determine the new minimum by comparing the increased smallest number and the decreased current number
        const newMin: number = Math.min(nums[0] + k, nums[i] - k);

        // Determine the new maximum by comparing the increased previous number and the decreased largest number
        const newMax: number = Math.max(nums[i - 1] + k, nums[numsSize - 1] - k);

        // Update the minRange with the minimum of the current and new ranges
        minRange = Math.min(minRange, newMax - newMin);
    }

    // Return the smallest range found
    return minRange;
}

// Example usage:
// nums = [1, 3, 6];
// k = 3;
// console.log(smallestRangeII(nums, k)); // Output would be the result of the smallestRangeII function
```

```python
from typing import List

class Solution:
    def smallestRangeII(self, nums: List[int], k: int) -> int:
        # First, sort the numbers to organize them in ascending order.
        nums.sort()

        # The initial range would be the max value minus the min value.
        smallest_range = nums[-1] - nums[0]

        # Loop through the sorted numbers, starting from the second element,
        # to find the minimum possible range.
        for i in range(1, len(nums)):
            # Calculate the possible minimum by adding k to the smallest value
            # and subtracting k from the current value.
            possible_min = min(nums[0] + k, nums[i] - k)

            # Calculate the possible maximum by adding k to the previous value
            # and subtracting k from the maximum value.
            possible_max = max(nums[i - 1] + k, nums[-1] - k)

            # Update the smallest range if a smaller one is found.
            smallest_range = min(smallest_range, possible_max - possible_min)

        # Finally, return the smallest range after considering all elements.
        return smallest_range
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code depends on the sorting algorithm and the for loop.

1. `nums.sort()` → The sort operation typically has a time complexity of `O(n log n)` where `n` is the number of elements in the list `nums`.
2. The for loop → Iterates through the sorted list once, accounting for a time complexity of `O(n)`.

Combining both, the time complexity remains dominated by the sorting step, and therefore, the overall time complexity is `O(n log n)`.

Space Complexity

The space complexity of the provided code is mainly due to the sorted list.

1. `nums.sort()` → The sort operation in Python is usually done in-place, which means the space complexity is `O(1)`.
2. No additional data structures are used that are dependent on the number of elements in the list, and therefore, no extra space is utilized that is proportional to the input size.

Thus, the space complexity of the code is `O(1)`.