360. Sort Transformed Array

Medium Array Math Two Pointers Sorting Leetcode Link

Problem Description

We're given an array `nums` which is sorted in ascending order, and three integers `a`, `b`, and `c`. These three integers are coefficients of a quadratic function `f(x) = ax^2 + bx + c`. Our task is to apply this function to each element in the array and then return the array with its elements sorted after the transformation. This transformed and sorted array needs to adhere to the standard non-decreasing order.

Intuition

The key to solving this problem lies in understanding how a quadratic function behaves. Depending on the value of the leading coefficient `a`, the graph of the function is either a parabola opening upwards (`a > 0`) or downwards (`a < 0`). When `a > 0`, the function's values are minimized at the vertex, and as you move away from the vertex, the values increase. When `a < 0`, the scenario is flipped; the values are maximized at the vertex, and as you move away, they decrease.

This understanding informs us that the transformed array will have its smallest or largest values at the ends if `a` is nonzero. If `a` is equal to 0, the function is linear, and the transformed array will maintain the same order as the input array, scaled and shifted by `b` and `c`.

The solution leverages this by comparing the transformed values of the start `i` and end `j` of the array `nums`, filling in the result array `res` from the start or the end, depending on the sign of `a`. With each comparison, it picks the extreme (either smallest or largest) of the transformed values and then increments or decrements the pointers accordingly.

For `a < 0`, we fill the `res` array from the beginning because we find that the smallest values will occur at the ends of the original array. Conversely, for `a > 0`, we fill the `res` array from the end because we'll encounter the largest values at the ends of the original array due to the "U" shape of the function.

The solution continues this process of comparison and selection until all elements in the input array have been transformed and placed in the result array in sorted order.

Solution Approach

The solution to this problem uses a two-pointer technique and knowledge of the properties of quadratic functions. The two-pointer technique is an algorithmic pattern where two pointers are used to iterate through the data structure—typically an array or string—usually from the beginning and end, to converge at some condition.

Here's a step-by-step breakdown based on the given Solution class and its method:

1. **Function Definition - `f(x)`:** The method `sortTransformedArray` includes a nested function `f(x)` which applies the quadratic transformation to a passed value `x`. Applying `f(x)` to any number will give us the result of the quadratic function for that number, using the formula `f(x) = ax^2 + bx + c`.

2. **Initialize Pointers and Result Array:**
   - Two pointers `i` (starting at `0`) and `j` (starting at `n - 1`) are used, where `n` is the length of the original array `nums`.
   - The pointer `k` is initialized differently based on the sign of `a`:
     - If `a` is negative (`a < 0`), `k` starts from `0` to fill in the result array from the start.
     - If `a` is positive (`a >= 0`), `k` starts from `n - 1` to fill in the result array from the end.
3. **Comparing and Filling the Result Array:**
   - We iterate while `i` is less than or equal to `j`.
   - For each iteration, we calculate `f(nums[i])` and `f(nums[j])`.
   - We compare the transformed values, `v1` and `v2`, and select the appropriate one based on the sign of `a`:
     - For `a < 0`, if `v1` is less than or equal to `v2`, `v1` is selected to fill the current position in the result array `res[k]`, and `i` is incremented. Else, `v2` is selected, and `j` is decremented. Pointer `k` is also incremented because we fill the `res` from the start.
     - For `a >= 0`, if `v1` is greater than or equal to `v2`, `v1` is selected to fill `res[k]`, and `i` is incremented. Else, `v2` is selected, and `j` is decremented. Here, `k` is decremented because we fill the `res` from the end.
4. **Return the Transformed and Sorted Array:**
   - After the loop completes, every position in the `res` array has been filled correctly.
   - The `res` array is then returned. It contains the elements of `nums` transformed by `f(x)` in sorted order.

The choice of whether to fill the result array from the start or end and the comparison logic within the loop all hinge on the behavior of the quadratic function. The two-pointer technique ensures that we traverse the array exactly once, giving us a time-efficient solution with `O(n)` complexity, where `n` is the number of elements in `nums`.

Example Walkthrough

Let's assume we have the following array and coefficients for our quadratic function:

- nums: `[1, 2, 3, 4, 5]`
- a: `-1`
- b: `0`
- c: `0`

The quadratic function is `f(x) = -x^2 + 0x + 0`, which simplifies to `f(x) = -x^2`.

Step-by-Step Walkthrough:

1. **Apply the Function `f(x)`:** To predict how the final array will look after applying `f(x)` to each element, we note that with `a < 0`, the function value is maximized at the vertex, and it decreases as we move away from the vertex. Since our `nums` array is sorted, the smallest transformed values will be at the ends, and the largest at the center.

2. **Initialize Pointers and Result Array:**
   - We set pointers `i = 0` and `j = 4` since `nums` has 5 elements.
   - Since `a` is negative (-1 in our case), we initialize the pointer `k` to fill the `res` array from the start. So `k = 0`.
3. **Comparing and Filling the Result Array:**
   - We calculate `f(nums[i])` which is `f(1) = -1^2 = -1`, and `f(nums[j])` which is `f(5) = -5^2 = -25`.
   - Since `a < 0`, we compare and find that `f(nums[i]) > f(nums[j])`, so we place `-1` in `res[0]`, and increment `i` to 1 and `k` to 1.
   - We continue this process, comparing the transformed values at each end and always choosing the larger one, filling `res` sequentially as we go along:
     - Compare `f(nums[i = 1]) = -2^2 = -4` and `f(nums[j = 4]) = -25`, put `-4` in `res[1]`, increment `i` to 2 and `k` to 2.
     - Compare `f(nums[i = 2]) = -3^2 = -9` and `f(nums[j = 4]) = -25`, put `-9` in `res[2]`, increment `i` to 3 and `k` to 3.
     - Compare `f(nums[i = 3]) = -4^2 = -16` and `f(nums[j = 4]) = -25`, put `-16` in `res[3]`, increment `i` to 4 and `k` to 4.
     - Now `i` equals `j`, we have one element left: `f(nums[j = 4]) = -25`, we put `-25` in `res[4]`.
4. **Return the Transformed and Sorted Array:**
   - After iterating through the entire array, our result array `res` is fully populated with the transformed elements in sorted order: `[-25, -16, -9, -4, -1]`.
   - We return the `res` array.

By following the logic laid out in the solution approach and considering the properties of the quadratic function, we've successfully transformed the `nums` array and returned it sorted in non-decreasing order as required.

Python Solution

```python
from typing import List

class Solution:
    def sort_transformed_array(
        self, nums: List[int], a: int, b: int, c: int
    ) -> List[int]:
        # Function to calculate the transformed value based on input x
        def quadratic(x):
            return a * x ** 2 + b * x + c

        # length of the input nums list
        n = len(nums)

        # Initialize pointers:
        # 'left' to start of array, 'right' to end of array
        # 'index' to either start or end based on sign of a
        left, right = 0, n - 1
        index = 0 if a < 0 else n - 1

        # Initialize the result array with zeros
        result = [0] * n

        # Iterate through the array until left exceeds right
        while left <= right:
            # Calculate the transformed values for both ends
            left_val = quadratic(nums[left])
            right_val = quadratic(nums[right])

            # If 'a' is negative, parabola opens downward.
            # Smaller values are closer to the ends of the array.
            if a < 0:
                if left_val <= right_val:
                    result[index] = left_val
                    left += 1
                else:
                    result[index] = right_val
                    right -= 1
                index += 1
            else:
                # If 'a' is non-negative, parabola opens upward.
                # Larger values are closer to the ends of the array.
                if left_val >= right_val:
                    result[index] = left_val
                    left += 1
                else:
                    result[index] = right_val
                    right -= 1
                index -= 1

        # Return the sorted transformed array
        return result

```

Java Solution

```java
class Solution {

    // This method sorts a transformed array generated by applying a quadratic function to an input array.
    public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
        int length = nums.length;

        // Two pointers initialized to the start and end of the array respectively.
        int startIndex = 0, endIndex = length - 1;

        // Determine the sorting direction based on the leading coefficient 'a'.
        // If 'a' is negative, sort ascending, otherwise sort descending.
        int sortIndex = a < 0 ? 0 : length - 1;

        // Create an array to store the result.
        int[] result = new int[length];

        // Use a while loop to process elements from both ends of the input array.
        while (startIndex <= endIndex) {

            // Apply the quadratic function to the elements at the start and end indices.
            int transformedStart = applyQuadraticFunction(a, b, c, nums[startIndex]);
            int transformedEnd = applyQuadraticFunction(a, b, c, nums[endIndex]);

            // If 'a' is negative, we fill the result array starting from the beginning.
            if (a < 0) {
                if (transformedStart <= transformedEnd) {
                    result[sortIndex] = transformedStart;
                    startIndex++;
                } else {
                    result[sortIndex] = transformedEnd;
                    endIndex--;
                }
                sortIndex++;

            // If 'a' is positive, we fill the result array starting from the end.
            } else {
                if (transformedStart >= transformedEnd) {
                    result[sortIndex] = transformedStart;
                    startIndex++;
                } else {
                    result[sortIndex] = transformedEnd;
                    endIndex--;
                }
                sortIndex--;
            }
        }

        // Return the sorted transformed array.
        return result;
    }

    // This helper method applies a quadratic function to the input value x.
    private int applyQuadraticFunction(int a, int b, int c, int x) {
        return a * x * x + b * x + c;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {
        int n = nums.size(); // Size of the input vector nums

        // Initialize two pointers for the start and end of the vector, and k for the position to fill in the result array
        int start = 0, end = n - 1;
        // If 'a' is positive, fill the result from the end; otherwise, from the start
        int k = a >= 0 ? n - 1 : 0;

        vector<int> result(n); // Initialize the result vector with the same size as nums
        while (start <= end) {
            // Apply the quadratic function to the current elements at the start and end pointers
            int transformedStart = quadratic(a, b, c, nums[start]);
            int transformedEnd = quadratic(a, b, c, nums[end]);

            if (a >= 0) {
                // For a positive 'a', larger values will be on the ends of the resulting array
                if (transformedStart >= transformedEnd) {
                    result[k--] = transformedStart; // Assign and then decrement k
                    start++; // Move start pointer to the right
                } else {
                    result[k--] = transformedEnd; // Assign and then decrement k
                    end--; // Move end pointer to the left
                }
            } else {
                // For a non-positive 'a', smaller values will be at the start of the resulting array
                if (transformedStart <= transformedEnd) {
                    result[k++] = transformedStart; // Assign and then increment k
                    start++; // Move start pointer to the right
                } else {
                    result[k++] = transformedEnd; // Assign and then increment k
                    end--; // Move end pointer to the left
                }
            }
        }
        return result; // Return the sorted and transformed array
    }

    // Helper function to apply the quadratic formula
    int quadratic(int a, int b, int c, int x) {
        return a * x * x + b * x + c; // Calculate ax^2 + bx + c
    }
};

```

Typescript Solution

```typescript
// Function to transform the array using the quadratic equation ax^2 + bx + c
function sortTransformedArray(nums: number[], a: number, b: number, c: number): number[] {
    const n: number = nums.length; // Size of the input array nums

    // Initialize two pointers for the start and end of the array, and index for the position to fill in the result array
    let start: number = 0, end: number = n - 1;
    // If 'a' is positive, fill the result from the end; otherwise, from the start
    let index: number = a >= 0 ? n - 1 : 0;

    const result: number[] = new Array(n); // Initialize the result array with the same size as nums
    while (start <= end) {
        // Apply the quadratic function to the current elements at the start and end pointers
        const transformedStart: number = quadratic(a, b, c, nums[start]);
        const transformedEnd: number = quadratic(a, b, c, nums[end]);

        if (a >= 0) {
            // For a positive 'a', larger values will be on the ends of the resulting array
            if (transformedStart >= transformedEnd) {
                result[index--] = transformedStart; // Assign and then decrement index
                start++; // Move start pointer to the right
            } else {
                result[index--] = transformedEnd; // Assign and then decrement index
                end--; // Move end pointer to the left
            }
        } else {
            // For a non-positive 'a', smaller values will be at the start of the resulting array
            if (transformedStart <= transformedEnd) {
                result[index++] = transformedStart; // Assign and then increment index
                start++; // Move start pointer to the right
            } else {
                result[index++] = transformedEnd; // Assign and then increment index
                end--; // Move end pointer to the left
            }
        }
    }
    return result; // Return the sorted and transformed array
}

// Function to evaluate the quadratic equation ax^2 + bx + c
function quadratic(a: number, b: number, c: number, x: number): number {
    return a * x * x + b * x + c; // Calculate ax^2 + bx + c
}

```

Time and Space Complexity

The given code has a time complexity of `O(n)`, where `n` is the number of elements in the provided `nums` list. This is because it processes each element in the list exactly once. Despite having a while loop that iterates while `i <= j`, no element is ever processed more than once due to the pointers `i` and `j` moving towards the center from the ends. The function `f(x)` is called twice per iteration, but it executes in constant time `O(1)`, thereby not affecting the overall linear time complexity.

The space complexity of the solution is also `O(n)`, as it creates a new list `res` of size `n` to store the transformed and sorted values. The rest of the variables `i`, `j`, `k`, `v1`, and `v2` use constant space, so the main contributing factor to the space complexity is the `res` array.