977. Squares of a Sorted Array

Easy Array Two Pointers Sorting Leetcode Link

Problem Description

The problem provides us with an array of integers `nums` that is sorted in non-decreasing order. Our task is to return an array containing the squares of each element from the `nums` array, and this resulting array of squares should also be sorted in non-decreasing order. For example, if the input is `[-4, -1, 0, 3, 10]`, after squaring each number, we get `[16, 1, 0, 9, 100]` and then we need to sort this array to get the final result `[0, 1, 9, 16, 100]`.

Intuition

Given that the input array is sorted in non-decreasing order, we realize that the smallest squares might come from the absolute values of negative numbers at the beginning of the array as well as the positive numbers at the end. The key insight is that the squares of the numbers will be largest either at the beginning or at the end of the array since squaring emphasizes larger magnitudes whether they are positive or negative. Therefore, we can use a two-pointer approach:

1. We create two pointers, `i` at the start of the array and `j` at the end.
2. We also create an array `res` of the same length as `nums` to store our results.
3. We iterate from the end of the `res` array backward, deciding whether the square of `nums[i]` or `nums[j]` should occupy the current position based on which one is larger. This ensures that the largest square is placed at the end of `res` array first.
4. We move pointer `i` to the right if `nums[i]` squared is greater than `nums[j]` squared since we have already placed the square of `nums[i]` in the result array.
5. Similarly, we move pointer `j` to the left if the square of `nums[j]` is greater to ensure we are always placing the next largest square.
6. We continue this process until all positions in `res` are filled with the squares of `nums` in non-decreasing order.
7. Finally, the `res` array is returned.

Solution Approach

The solution makes use of a two-pointer approach, an efficient algorithm when dealing with sorted arrays or sequences. The steps of the algorithm are implemented in the following way:

1. **Initialize pointers and an array**: A pointer `i` is initialized to the start of the array (`0`), a pointer `j` is initialized to the end of the array (`n - 1` where `n` is the length of the array), and an array `res` of the same length as `nums` is created to store the results.

2. **Iterate to fill result array**: A loop is used, where the index `k` starts from the end of the `res` array (`n - 1`) and decrements with each iteration. The `while` loop continues until `i` is greater than `j`, which means all elements have been considered.

3. **Compare and place squares**: During each loop iteration, the solution compares the squares of the values at index `i` and `j` (`nums[i] * nums[i]` vs `nums[j] * nums[j]`) to decide which one should be placed at the current index `k` of the `res` array. The larger square is placed at `res[k]`, and the corresponding pointer (`i` or `j`) is moved.

   - If `nums[i] * nums[i]` is greater than `nums[j] * nums[j]`, this means that the square of the number pointed to by `i` is currently the largest remaining square, so it's placed at `res[k]`, and `i` is incremented to move to the next element from the start.

   - Conversely, if `nums[j] * nums[j]` is greater than or equal to `nums[i] * nums[i]`, then `nums[j]` squared is placed at `res[k]`, and `j` is decremented to move to the next element from the end.

4. **Decrement `k`**: After each iteration, `k` is decremented to fill the next position in the `res` array, starting from the end and moving towards the start.

5. **Return the result**: Once the `while` loop is done, all elements have been squared and placed in the correct position, resulting in a sorted array of squares which is then returned.

This approach uses no additional data structures other than the `res` array to produce the final sorted array of squares. It is space-optimal, requiring O(n) additional space, and time-optimal with O(n) time complexity because it avoids the need to sort the squares after computation, which would take O(n log n) if a sorting method was used after squaring the elements.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. We will use the input array `nums = [-3, -2, 1, 4]`. Our goal is to compute the squares of each number and get a sorted array as a result. Here's how it works:

1. **Initialize pointers and an array**: We initialize `i` to `0`, `j` to `3` (since there are four elements, `n - 1 = 3`), and an array `res` with the length of `4` to store the results: `res = [0, 0, 0, 0]`.

2. **Iterate to fill result array**: We start a `while` loop with `k = 3`, which is the last index in the `res` array.

3. **Compare and place squares**:

   - First iteration: `nums[i]` is `-3`, `nums[j]` is `4`.
   - Squares: `nums[i] * nums[i] = 9`, `nums[j] * nums[j] = 16`.
   - Since `16` is greater than `9`, we place `16` at `res[k]`: `res = [0, 0, 0, 16]`.
   - Decrement `j` to `2` and `k` to `2`.
4. **Next iteration**:

   - Now `i = 0` (with `nums[i] = -3`), `j = 2` (with `nums[j] = 1`).
   - Squares: `nums[i] * nums[i] = 9`, `nums[j] * nums[j] = 1`.
   - `9` is greater than `1`, so we place `9` at `res[k]`: `res = [0, 0, 9, 16]`.
   - Increment `i` to `1` and decrement `k` to `1`.
5. **Next iteration**:

   - Now `i = 1` (with `nums[i] = -2`), `j = 2` (with `nums[j] = 1`).
   - Squares: `nums[i] * nums[i] = 4`, `nums[j] * nums[j] = 1`.
   - `4` is greater than `1`, so we place `4` at `res[k]`: `res = [0, 4, 9, 16]`.
   - Increment `i` to `2` and decrement `k` to `0`.
6. **Final iteration**:

   - Now `i = 2` (with `nums[i] = 1`), `j = 2` (with `nums[j] = 1`), and `k = 0`.
   - There's only one element left, so we square it and place it at `res[k]`: `nums[i] * nums[i] = 1`.
   - `res` becomes `[1, 4, 9, 16]`.
7. **Return the result**: At the end of the loop, we have the final sorted array of squares `res = [1, 4, 9, 16]`.

Following these steps, we have successfully transformed the `nums` array into a sorted array of squares without needing to sort them again after squaring. This approach efficiently uses the original sorted order to place the squares directly in the correct sorted positions.

Python Solution

```python
from typing import List

class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        # Get the length of the input array
        length = len(nums)

        # Initialize a result array of the same length as the input array
        result = [0] * length

        # Initialize pointers for the start and end of the input array,
        # and a pointer for the position to insert into the result array
        start_pointer, end_pointer, result_pointer = 0, length - 1, length - 1

        # Loop through the array from both ends towards the middle
        while start_pointer <= end_pointer:
            # Square the values at both pointers
            start_square = nums[start_pointer] ** 2
            end_square = nums[end_pointer] ** 2

            # Compare the squared values and add the larger one to the end of the result array
            if start_square > end_square:
                result[result_pointer] = start_square
                start_pointer += 1
            else:
                result[result_pointer] = end_square
                end_pointer -= 1

            # Move the result pointer to the next position
            result_pointer -= 1

        # Return the sorted square array
        return result

```

Java Solution

```java
class Solution {

    // Method that takes an array of integers as input and
    // returns a new array with the squares of each number sorted in non-decreasing order.
    public int[] sortedSquares(int[] nums) {
        int length = nums.length; // Store the length of the input array
        int[] sortedSquares = new int[length]; // Create a new array to hold the result

        // Initialize pointers for the start and end of the input array,
        // and a pointer 'k' for the position to insert into the result array, starting from the end.
        for (int start = 0, end = length - 1, k = length - 1; start <= end;) {
            // Calculate the square of the start and end elements
            int startSquare = nums[start] * nums[start];
            int endSquare = nums[end] * nums[end];

            // Compare the squares to decide which to place next in the result array
            if (startSquare > endSquare) {
                // If the start square is greater, place it in the next open position at 'k',
                // then increment the start pointer.
                sortedSquares[k--] = startSquare;
                ++start;
            } else {
                // If the end square is greater or equal, place it in the next open position at 'k',
                // then decrement the end pointer.
                sortedSquares[k--] = endSquare;
                --end;
            }
        }

        // Return the array with sorted squares
        return sortedSquares;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        int size = nums.size();
        vector<int> result(size); // This will store the final sorted squares of numbers

        // Use two pointers to iterate through the array from both ends
        int left = 0;         // Start pointer for the array
        int right = size - 1; // End pointer for the array
        int position = size - 1; // Position to insert squares in the result array from the end

        // While left pointer does not surpass the right pointer
        while (left <= right) {
            // Compare the square of the elements at the left and right pointer
            if (nums[left] * nums[left] > nums[right] * nums[right]) {
                // If the left square is larger, place it in the result array
                result[position] = nums[left] * nums[left];
                ++left; // Move the left pointer one step right
            } else {
                // If the right square is larger or equal, place it in the result array
                result[position] = nums[right] * nums[right];
                --right; // Move the right pointer one step left
            }
            --position; // Move the position pointer one step left
        }

        // Return the result which now contains the squares in non-decreasing order
        return result;
    }
};

```

Typescript Solution

```typescript
/**
 * Returns an array of the squares of each element in the input array, sorted in non-decreasing order.
 * @param {number[]} nums - The input array of integers.
 * @return {number[]} - The sorted array of squares of the input array.
 */
const sortedSquares = (nums: number[]): number[] => {
    // n is the length of the input array nums.
    const n: number = nums.length;

    // res is the resulting array of squares, initialized with the size of nums.
    const res: number[] = new Array(n);

    // Two pointers approach: start from the beginning (i) and the end (j) of the nums array.
    // The index k is used for the current position in the resulting array res.
    for (let i: number = 0, j: number = n - 1, k: number = n - 1; i <= j; ) {
        // Compare squares of current elements pointed by i and j.
        // The larger square is placed at the end of array res, at index k.
        if (nums[i] * nums[i] > nums[j] * nums[j]) {
            // Square of nums[i] is greater, so store it at index k in res and move i forward.
            res[k--] = nums[i] * nums[i];
            ++i;
        } else {
            // Square of nums[j] is greater or equal, so store it at index k in res and move j backward.
            res[k--] = nums[j] * nums[j];
            --j;
        }
    }

    // Return the sorted array of squares.
    return res;
};

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`. Each element in the `nums` array is accessed once during the while loop. Despite the fact that there are two pointers (i, j) moving towards each other from opposite ends of the array, each of them moves at most `n` steps. The loop ends when they meet or cross each other, ensuring that the total number of operations does not exceed the number of elements in the array.

Space Complexity

The space complexity of the code is `O(n)`. Additional space is allocated for the `res` array, which stores the result. This array is of the same length as the input array `nums`. No other additional data structures are used that grow with the size of the input, so the total space required is directly proportional to the input size `n`.