Documentation

The problem asks us to return an array where each element indicates how many elements to its right are smaller than itself in a given array. This is a classic example of the need for an efficient solution due to the large input size constraints, which can be as large as 10^5. A brute force approach where we compare each element with every other element to the right would have a time complexity of $O(n^2)$, which is impractical for large arrays. Therefore, we need to adopt a more efficient strategy.

1. Problem Understanding

We are given an array of integers, and for each element, we need to determine how many elements to its right are smaller than the current element. The output is an array of the same length as the input, where each element at index ii holds the count of elements smaller than nums[i] that appear after it in the array. For example, if the input is [5, 2, 6, 1], the output would be [2, 1, 1, 0], because to the right of 5, there are two smaller elements (2 and 1), and to the right of 1, there are no smaller elements.

2. Challenges in the Problem

The main challenge in this problem is efficiently counting the smaller elements to the right of each number in the array. A direct approach that checks each element against all the others to its right would lead to excessive time complexity, $O(n^2)$, which becomes unmanageable when n is large (up to 10^5). Thus, the problem requires a more efficient approach that works in O(n log n), which leads us to consider advanced data structures like the Binary Indexed Tree (BIT) or the Fenwick Tree.

3. Coordinate Compression and Binary Indexed Tree (BIT)

To solve this problem efficiently, we use a combination of **Coordinate Compression** and a **Binary Indexed Tree (BIT)**. The coordinate compression step ensures that we can efficiently map the elements in the array to a smaller range, which is necessary for using a BIT. A BIT allows us to efficiently query how many numbers smaller than the current one have been encountered so far, and it also allows us to update our count as we process each element efficiently. By traversing the array from right to left, we can count how many smaller elements are encountered for each number while updating the BIT.

4. Coordinate Compression

Coordinate compression is a technique used to map a large range of values (in this case, integers ranging from -10^4 to 10^4) to a smaller range, making it feasible to work with these values in a Binary Indexed Tree. In this step, we first sort the unique elements of the array and then assign each element a new rank or index based on its position in the sorted array. This new index is used as the position for updating and querying the BIT. The main benefit of coordinate compression is that it ensures we can handle a large range of values efficiently by reducing the range to a manageable size.

5. Binary Indexed Tree (BIT)

A Binary Indexed Tree (BIT) is a data structure that provides efficient methods for updating an element and calculating prefix sums (or cumulative sums) in logarithmic time. It is particularly useful for problems that involve range queries and point updates, as it can handle both in O (log n) time. In the context of this problem, the BIT helps us efficiently count how many elements smaller than the current element have already been seen as we process the array from right to left. By querying the BIT, we can find how many numbers smaller than the current number have been processed so far, and by updating the BIT, we keep track of the elements we have already seen.

6. Traversing the Array

We traverse the array from right to left, which allows us to keep track of how many smaller numbers we've encountered to the right of each element. For each element, we first query the BIT to determine how many elements smaller than the current element have already been seen. This count is then stored in the result array. After querying, we update the BIT to include the current element, ensuring that subsequent queries will account for this element. By processing the array in reverse order, we ensure that we are always counting the number of smaller elements to the right of each element.

7. Time and Space Complexity

The time complexity of this solution is O (n log n). The most time-consuming operations are the sorting of the unique elements for coordinate compression (O (n log n) and the updates and queries on the Binary Indexed Tree (O(log n) for each of the n elements). Thus, the overall time complexity is dominated by O (n log n), which is efficient enough for the input size constraints. The space complexity is O(n)O(n) due to the storage required for the BIT, the coordinate compression map, and the result array. This approach ensures that the problem can be solved efficiently even for large input sizes.

Conclusion

This approach leverages a combination of coordinate compression and a Binary Indexed Tree to efficiently solve the problem of counting how many smaller elements exist to the right of each element in the array. By using the BIT for range queries and updates and applying coordinate compression to handle a large range of values, we can solve the problem in O (n log n) time, making it feasible for large input sizes. This method provides a practical and efficient solution to the problem, ensuring optimal performance even with the maximum constraints.