2592. Maximize Greatness of an Array

Medium Greedy Array Two Pointers Sorting

Problem Description

The problem provides an integer array `nums`, numerically indexed starting at 0. You are given the ability to rearrange the order of the elements in `nums` to form a new array `perm`. The `greatness` of an array is determined by the count of indices `i` (where `0 <= i < nums.length`) that satisfy the condition `perm[i] > nums[i]`. Your task is to figure out the arrangement of `nums` that results in the highest possible `greatness` value, and then return that maximum greatness value.

In simpler terms, you need to permute the original array in such a way that as many elements as possible in the new array are greater than the corresponding elements in the original array at the same indices.

Intuition

The intuitive approach to achieve maximum greatness after permuting the elements in the array is by using a greedy strategy. Greedy strategies typically involve making the best or optimal choice available at each step with the hope of finding the global optimum at the end.

First, we sort the original array `nums`. Sorting is a frequently used first step in many algorithms because it brings order to the elements, thereby making it easier to compare and organize them to meet a certain condition—in this case, to maximize greatness.

Once sorted, the strategy is to have a pointer `i` traverse the array `nums`. For each element `x` encountered during the traversal, one compares it with the element at the current index `i` of the sorted `nums`. If `x` is greater than `nums[i]`, it means this element contributes to the greatness and the pointer `i` is incremented to reflect this. The incremented `i` represents the tally of how many elements have so far contributed to the greatness.

The choice at each step is clear: take the smallest element still available (since the array is sorted) and compare it with the current element. If it qualifies, it gets counted towards greatness, and we move on to the next element.

In a way, each step is both isolated (involving a comparison between two specific elements) and cumulative (contributing to the overall count of greatness). This isolation allows each decision to be made independently. The cumulative nature of the count ensures that the count is reflective of the sum total of all right decisions made thus far.

By the end of the traversal, the pointer `i` reflects the maximum greatness, since it denotes the number of times we've successfully found an element in `nums` that can be placed at an index to satisfy the greatness condition `perm[i] > nums[i]`. We then return the value of the pointer `i` as the result.

Solution Approach

The solution adopts a rather straightforward approach that can be broken down into two main steps: sorting and then greedily counting the elements that can increase the array's greatness.

Here's the algorithm in greater detail:

1. First, the input array `nums` is sorted in ascending order. Sorting allows us to go through the elements in increasing order, which constructs the best scenario to increase greatness. After sorting, we can start from the smallest element in `nums` and try to satisfy the condition `perm[i] > nums[i]` incrementally as we proceed.
2. The next important step is to initialize a counter `i` to zero. This counter `i` will be used to traverse the sorted array `nums` and keep track of the count for the greatness.
3. We then iterate over the sorted array `nums`, at each step comparing the current element `x` against the element at index `i` of the sorted `nums`. If the condition `x > nums[i]` is true, it means placing `x` at index `i` in the permuted array `perm` would increase the `greatness` by 1, thus we increment the counter `i` by 1. Incrementing `i` effectively moves our 'checkpoint' forward in the array, to the next element that needs to be satisfied for increasing greatness.
4. The incrementing of `i` is dependent on the condition `x > nums[i]`, which is checked through `i += x > nums[i]`. This is a Pythonic way to increment `i` if `x` is greater than `nums[i]`, otherwise, `i` remains the same.
5. After the complete traversal of the array, counter `i` represents the maximum number of indices for which the condition `perm[i] > nums[i]` holds true. This maximum count is the desired maximum greatness, and the value of `i` is returned as the final result.

The reference solution approach clearly depends on the sorted nature of the array and the greedy strategy to increment the count only when the current element can contribute to the greatness. This one-pass solution is efficient as it makes a single comparison for each element in the array, yielding a time complexity of O(nlogn) due to sorting, and a space complexity of O(1) as it requires only a constant amount of additional memory space.

In summary, the solution makes use of:

- **Sorting (`nums.sort()`)**: To order the elements which is the precondition for our greedy strategy.
- **Greedy Iteration**: To compare and count the elements satisfying the greatness condition in a single forward pass through the sorted array.
- **Pointer Incrementing (`i += x > nums[i]`)**: To count the elements that are contributing towards the greatness.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we have the following integer array `nums`:

```
nums = [4, 3, 2, 1]
```

Here are the steps we would follow:

1. **Sort the array**: The first step is to sort `nums` in ascending order.

```
Sorted nums = [1, 2, 3, 4]
```

2. **Initialize the counter**: We set up a counter `i` to keep track of the number of times we can achieve `perm[i] > nums[i]`. Initially, `i = 0`.
3. **Greedy iteration and comparison**:
   - We compare `nums[0] = 1` with `x = nums[i = 0]` (since `i = 0` initially). Since `1` is not greater than itself, `i` remains unchanged.
   - Moving to the next element, we compare `nums[1] = 2` to `x = nums[i = 0] = 1`. Here, `2 > 1`, so this arrangement contributes to the greatness. Therefore, we increment `i` to `1`.
   - We then compare `nums[2] = 3` to `x = nums[i = 1] = 2`. Since `3 > 2`, we can increment `i` again to `2`.
   - Finally, we compare `nums[3] = 4` to `x = nums[i = 2] = 3`. `4 > 3` holds, so we increment `i` once more to `3`.
4. **Result**: At the end of this process, the counter `i` now equals `3`. That is the highest greatness we can achieve by rearranging the array.

In summary, we managed to find an arrangement of `nums` where three elements in the permutation array are greater than the corresponding elements in the original array. The maximum greatness of this example is hence `3`. This result is achieved through an ordered greedy approach that ensures an optimal arrangement leveraging Python's clean syntax for conciseness and efficiency.

Solution Implementation

Python

```python
from typing import List

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

        # Initialize the greatness counter to zero.
        greatness_counter = 0

        # Loop through each number in the sorted list.
        for number in numbers:
            # If the current number is greater than the number at the index
            # represented by the greatness_counter, increment the counter.
            # This implies we can increase our 'greatness' score.
            if number > numbers[greatness_counter]:
                greatness_counter += 1

        # Return the total greatness we can achieve.
        return greatness_counter
```

Java

```java
class Solution {

    // Method to determine the maximum level of greatness that can be achieved
    public int maximizeGreatness(int[] nums) {
        // Sort the array in non-decreasing order
        Arrays.sort(nums);

        // Initialize the count of greatness to 0
        int greatnessCount = 0;

        // Iterate over the sorted array
        for (int num : nums) {
            // If the current element is greater than the element at the current greatness count index
            if (num > nums[greatnessCount]) {
                // Increment the greatness count
                greatnessCount++;
            }
        }

        // Return the total count of greatness achieved
        return greatnessCount;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the number of times the current element is greater than the smallest element seen so far.
    int maximizeGreatness(vector<int>& nums) {
        // Sort the input vector in non-decreasing order.
        sort(nums.begin(), nums.end());

        // Variable to keep track of the current smallest element's index.
        int smallestIndex = 0;

        // Loop through the sorted vector to compute the result.
        for (int num : nums) {
            // Increase count if the current number is greater than the number at smallestIndex.
            smallestIndex += num > nums[smallestIndex];
        }

        // The result is the number of elements that were greater than their preceding elements.
        return smallestIndex;
    }
};
```

TypeScript

```typescript
function maximizeGreatness(nums: number[]): number {
    // Sort the array in non-decreasing order.
    nums.sort((a, b) => a - b);

    // Initialize the count of numbers that can contribute to 'greatness'
    let greatnessCount = 0;

    // Iterate through the sorted array.
    for (const currentNumber of nums) {
        // If the current number is greater than the smallest number in the set of 'greatness' numbers,
        // it can contribute, so we increment the count.
        if (currentNumber > nums[greatnessCount]) {
            greatnessCount += 1;
        }
    }

    // Return the final count of numbers that contribute to 'greatness'.
    return greatnessCount;
}
```


```python
from typing import List

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

        # Initialize the greatness counter to zero.
        greatness_counter = 0

        # Loop through each number in the sorted list.
        for number in numbers:
            # If the current number is greater than the number at the index
            # represented by the greatness_counter, increment the counter.
            # This implies we can increase our 'greatness' score.
            if number > numbers[greatness_counter]:
                greatness_counter += 1

        # Return the total greatness we can achieve.
        return greatness_counter
```


Time and Space Complexity

Time Complexity

The time complexity of the given function `maximizeGreatness` is primarily dominated by the `nums.sort()` operation. Sorting an array is typically done using the Timsort algorithm in Python, which has a time complexity of `O(n log n)`, where `n` is the length of the array `nums`. The for-loop that follows the sort operation has a time complexity of `O(n)` since it iterates over each element in the sorted list exactly once.

Therefore, when combining these two operations, the total time complexity remains `O(n log n)` as the sorting operation is the most significant factor.

Space Complexity

The space complexity of the sort operation in Python is `O(n)`, but this uses the internal buffer for sorting which does not count towards the additional space required by the algorithm. However, due to the recursive stack calls made during sorting, the worst-case space complexity is `O(log n)`. No additional data structures that are dependent on the size of the input array are used in the function, keeping the space complexity to `O(log n)` for this function.