1887. Reduction Operations to Make the Array Elements Equal

Medium Array Sorting Leetcode Link

Problem Description

The goal of the given problem is to perform a series of operations on an integer array `nums` until all the elements in the array are equal. An operation consists of three steps:

1. Find the **largest** value in the array, denoted as `largest`. If there are multiple elements with the largest value, we select the one with the smallest index `i`.
2. Find the **next largest** value that is **strictly smaller** than `largest`, denoted as `nextLargest`.
3. Replace the element at index `i` with `nextLargest`.

The problem asks us to return the number of operations required to make all elements in the array equal.

Intuition

To solve this problem, a key insight is that sorting the array will make it easier to track the reductions needed to equalize all elements. After sorting:

1. The **largest** element will be at the end of the sorted array, and the **next largest** will be right before it.
2. Subsequent steps involve moving down the sorted array and reducing the largest remaining element to the next largest.

By maintaining a sorted array, we can avoid repeatedly searching for the largest and next largest elements, thus optimizing the process.

Here's the process of the solution approach:

- First, sort the array in non-decreasing order. This will ensure that each subsequent value from left to right will be greater than or equal to the previous one.
- Then, iterate through the sorted array from the second element onwards, comparing the current element with the previous one:
  - If they are the same, no operation is needed for this step, but we keep a count of how many times we would have had to reduce other elements to the current value.
  - If the current value is larger, it means an operation was needed to get from the previous value to this one. We increment the operation count (`cnt`) and add it to the total answer (`ans`) because we will need that number of operations for each element that needs to be reduced to this current value.

Each increment of `cnt` represents a step in which all larger elements need one more operation to reach equality, and by adding `cnt` to the answer every time, you account for the operations needed to reduce all larger elements to the current one. The final answer is the total count of operations needed.

Solution Approach

The given Python solution follows a straightforward approach, leveraging simple algorithms and data structure manipulation to arrive at the answer. Here's a breakdown of how the solution is implemented:

- **Algorithm**: The primary algorithm used here is sorting, which is an integral part of the solution. Python's built-in sorting is typically implemented as Timsort, which is efficient for the given task.

- **Data Structures**: The solution primarily works with the list data structure in Python, which is essentially an array.

- **Pattern Used**: The approach follows a pattern similar to counting, where for each unique value in the sorted array, we track how many operations are needed if we want to reduce the larger numbers to this number.

Let's examine the implementation step by step:

1. The `nums` list is sorted in non-decreasing order using `nums.sort()`.

2. A `for` loop with `enumerate` is set to iterate through the array (excluding the first element, as there's nothing to compare it to). Two variables are maintained:

   - `cnt`: This keeps count of how many different operations are performed. It starts at `0` because no operations are performed at the first element.
   - `ans`: This accumulates the total number of operations.
3. Inside the loop, each element `v` at index `i` (where `i` starts from `1` since we skipped the first element) is compared to its predecessor (`nums[i]`):

   - If `v` equals the previous element (`nums[i]`), it means that no new operation is needed for `v` to become equal to the previous element (as it's already equal).

   - If `v` is different (meaning it is larger since the array is sorted), then we found a new value that wasn't seen before. Therefore, we increment `cnt` by `1` since all occurrences of this new value would require an additional operation to be reduced to the previous smaller value.

4. After assessing each pair of elements, the value of `cnt` (which indicates the cumulative operations required to reduce the current and all previous larger values) is added to `ans`.

5. Finally, after the loop completes, `ans` holds the total number of operations required to make all elements equal and is returned as the result.

Here is the critical part of the code with added comments for clarity:

```python
class Solution:
    def reductionOperations(self, nums: List[int]) -> int:
        nums.sort()  # Sorting the array in non-decreasing order
        ans = cnt = 0  # Initialize counters to zero
        for i, v in enumerate(nums[1:]):  # Iterate through the array, skipping the first element
            if v != nums[i]:  # If current element is greater than the previous one (not equal)
                cnt += 1  # Increment the number of operations needed
            ans += cnt  # Add to total answer
        return ans  # Return the total
```

Notice that the `enumerate` function in the loop is used with the sublist `nums[1:]` which effectively shifts the indices of the enumerated items by one, meaning `nums[i]` actually refers to the element immediately preceding `v`.

To summarize, the use of sorting simplifies the identification of unique values that require operations, and the counting mechanism properly aggregates the steps needed to reach the desired equal state of the `nums` array.

Example Walkthrough

Let's walk through a small example using the solution approach described above. Consider the following array of integers:

```
nums = [5, 1, 3, 3, 5]
```

We want to perform operations until all the elements in this array are equal, following the given steps: sort the array, identify the largest and next largest elements, and replace occurrences of the largest element with the next largest until the array is homogenized.

Here is the breakdown of how we apply our algorithm to the example:

1. **Sort the array**:

   ```
   Sort the `nums` array: nums = [1, 3, 3, 5, 5]
   ```

   After sorting the array in non-decreasing order, we can easily identify which elements need to be replaced in each operation.

2. **Initial setup**:

   - Initialize our counters: `cnt = 0` and `ans = 0`.
   - Start iterating from the second element of `nums` (since we need to compare each element with its previous one).
3. **Iteration**:

   - Compare `3` with `1`. Since `3` is greater, we found a new value. So, `cnt` becomes `1` and `ans` becomes `1`.
   - Compare the second `3` with the first `3`. They are equal, no new operation is needed, `cnt` stays `1` and `ans` becomes `2`.
   - Compare `5` with `3`. `5` is greater, so `cnt` becomes `2` (indicating each `5` needs two operations to become a `3`) and `ans` becomes `4`.
   - Compare the second `5` with the first `5`. They are equal, so `cnt` stays `2` and `ans` becomes `6`.
4. **Final count**:

   - After the loop concludes, `ans = 6`, which represents the total number of operations needed to make all elements equal.

Through this walkthrough, we find that a total of 6 operations are required to make all elements of the array `[5, 1, 3, 3, 5]` equal. The sorted form, `[1, 3, 3, 5, 5]`, simplifies the identification of which elements need to be replaced, and our counting mechanism effectively calculates the necessary steps to achieve uniformity across the array.

Python Solution

```python
class Solution:
    def reductionOperations(self, nums: List[int]) -> int:
        # Sort the list of numbers in non-decreasing order.
        nums.sort()

        # Initialize the number of operations required to 0.
        operations_count = 0

        # This variable keeps track of the number of different elements encountered.
        different_elements_count = 0

        # Iterate through the sorted list of numbers, starting from the second element.
        for i in range(1, len(nums)):
            # Check if the current number is different from the previous one,
            # as only unique numbers will contribute to new operations.
            if nums[i] != nums[i - 1]:
                # If it's different, increment the count of different elements.
                different_elements_count += 1

            # Add the count of different elements to the total operations count.
            # This accounts for the operations required to reduce this number
            # to the next lower number in the list.
            operations_count += different_elements_count

        # Return the total count of reduction operations required.
        return operations_count

# Usage example:
# solution = Solution()
# result = solution.reductionOperations([5,1,3])
# print(result) # Output would be the number of operations required.

```

Java Solution

```java
class Solution {
    public int reductionOperations(int[] nums) {
        // Sort the array in non-decreasing order
        Arrays.sort(nums);

        // Initialize a variable to count the number of operations needed
        int operationsCount = 0;
        // Initialize a variable to count the distinct elements processed
        int distinctElementsCount = 0;

        // Iterate over the sorted array, starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            // Check if the current element is different from the previous one
            if (nums[i] != nums[i - 1]) {
                // Increment the distinct elements count since a new value has been encountered
                ++distinctElementsCount;
            }
            // Add the current distinct elements count to the total operations
            // This represents making the current element equal to the previous one
            operationsCount += distinctElementsCount;
        }

        // Return the total count of reduction operations required
        return operationsCount;
    }
}

```

C++ Solution

```cpp
// This solution utilizes sorting and then counts the steps required to reduce elements to make all equal.
class Solution {
public:
    int reductionOperations(vector<int>& nums) {
        // Sort the nums vector in non-decreasing order
        sort(nums.begin(), nums.end());

        // Initialize the answer and count variables
        int operations = 0; // Number of operations needed to make all elements equal
        int stepCounter = 0; // Number of steps needed to decrement to the next lower number

        // Iterate through the sorted vector starting from index 1
        for (int i = 1; i < nums.size(); ++i) {
            // If the current number is different from the one before it,
            // it means a new decrement step is needed.
            if (nums[i] != nums[i - 1]) {
                stepCounter++;
            }

            // Add the number of steps to the total operations
            operations += stepCounter;
        }

        // Return the total number of operations required
        return operations;
    }
};

```

Typescript Solution

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

    // Initialize a variable `result` to keep the count of operations.
    let result = 0;
    // Initialize a variable `countDistinct` to track the number of distinct elements encountered.
    let countDistinct = 0;

    // Iterate through the sorted array, starting from the second element.
    for (let i = 1; i < nums.length; ++i) {
        // If the current element is different from the previous one, increment the distinct count.
        if (nums[i] !== nums[i - 1]) {
            countDistinct++;
        }

        // Add the current count of distinct elements to the result.
        // This represents the number of operations needed for each element to reach the previous smaller element.
        result += countDistinct;
    }

    // Return the total number of operations needed to make all elements equal.
    return result;
}

```

Time and Space Complexity

Time Complexity

The given code has two main operations: sorting the `nums` array and iterating through the sorted array to calculate the `ans`.

- Sorting the array has a time complexity of `O(n log n)`, where `n` is the length of the `nums` list. This is because the Timsort algorithm, which is the sorting algorithm used by Python's `sort()` function, typically has this complexity.

- The for loop iterates through the array once, which gives a time complexity of `O(n)` for this part of the code.

Since the sorting operation is likely to dominate the overall time complexity, the final time complexity of the code is `O(n log n)`.

Space Complexity

The space complexity concerns the amount of extra space or temporary space that an algorithm uses.

- The `nums.sort()` operation sorts the list in-place, which means it does not require additional space proportional to the size of the input (`nums`). Thus, this part of the operation has a space complexity of `O(1)`.

- The variables `ans` and `cnt` use a constant amount of space, thus contributing `O(1)` to the space complexity.

- The enumeration in the for loop does not create a new list but rather an iterator over the sliced list (`nums[1:]`). The slice operation in Python creates a new list object, so this operation takes `O(n-1)` space, which simplifies to `O(n)`.

Therefore, the space complexity of the code is `O(n)` due to the list slicing operation.