945. Minimum Increment to Make Array Unique

Medium Greedy Array Counting Sorting

Leetcode Link

Problem Description

You're handed an array of integers named `nums`. The task is to make each number in the array unique by performing a certain operation: for any element at index `i`, where `0 <= i < nums.length`, you can increase the value of `nums[i]` by `1`. The goal is to find out the minimum number of such increments needed to ensure that all elements in the array are unique. It is guaranteed that the final answer will be small enough to fit within a 32-bit integer.

Intuition

The intuitive solution involves:

- First, sort the array. This ensures that we can process numbers in a sequence and efficiently deal with duplicates.
- Then, iterate through the array starting from the second element, comparing each element with the one before it.
- If the current element is less than or equal to the previous element, it means we have a duplicate or a possibility of a non-unique value.
- To make the current value unique, we need to increment it to be just one more than the previous value. The difference plus one `d = nums[i - 1] - nums[i] + 1` gives the exact number of increments needed for that element.
- We update the current element to its new unique value and add the number of increments `d` to a running total, which will eventually be the answer.
- Repeat this process for all elements in the array. By the end, the running total gives us the minimum number of moves necessary to achieve uniqueness for all elements in `nums`.

Solution Approach

The solution utilizes a simple but effective algorithm which requires minimal data structures – the primary one being the ability to sort the given array. Here's the approach outlined step by step:

1. **Sorting the Array**: Start by sorting the `nums` array. This allows us to process the array in ascending order and handle any duplicates or clusters of the same number effectively.
2. **Initialization**: A variable, `ans`, is initialized to `0`. This variable will keep track of the total number of increments we perform across the entire array.
3. **Iterate Through the Array**: Iterate through the sorted `nums` starting from the index `1` to the end of the array. We compare each element with the previous one to check for duplicates or the sequence being out of order.
4. **Processing Each Element**:
   - For each element at index `i`, check if it's less than or equal to the element at index `i - 1`.
   - If it is, we need to increment it to make it unique. We calculate the `difference + 1` by `nums[i - 1] - nums[i] + 1`, which tells us how much we need to increment the current element to not only make it unique but also ensure it's greater than the previous element.
   - Add this difference to the `nums[i]` to update the value of the current element, making it unique.
   - Also, add this difference to the `ans` variable, which accumulates the total increments made.
5. **Returning the Answer**: After the loop terminates, `ans` holds the total number of increments needed to make all elements in the array unique. Return `ans`.

This solution is quite efficient with a time complexity of `O(n log n)` due to the sort operation. The following loop has a complexity of `O(n)`, but since sorting takes precedence in terms of complexity analysis, it doesn't change the overall time complexity.

The Python code implementation following this algorithm ensures that with minimal space overhead, and in a reasonably optimal time, we arrive at the least number of moves needed to make every element in the array `nums` unique.

Example Walkthrough

Given an array of integers `nums = [3, 2, 1, 2]`, our task is to make each number in the array unique with the least number of increments.

Following the solution approach, let's walk through the process:

1. **Sorting the Array**:
   - We start by sorting `nums`. After sorting, the array becomes `nums = [1, 2, 2, 3]`.
2. **Initialization**:
   - Next, we initialize the answer variable `ans` to `0`. This variable will count the total increments.
3. **Iterate Through the Array**:
   - We then iterate through the sorted `nums` beginning from the index `1`.
4. **Processing Each Element**:
   - At index `1`, `nums[0] = 1` and `nums[1] = 2`. Since `nums[1]` is greater than `nums[0]`, no increment is needed.
   - At index `2`, we have a duplicate since `nums[2] = 2` and `nums[1]` was also `2`. We need to increment `nums[2]` by `1` to make it unique. So `nums[2]` becomes `3` and `ans` is incremented by `1` (total increments so far: `1`).
   - However, now at index `3`, `nums[3] = 3` which is equal to `nums[2]` after the previous increment. So, we need to increment `nums[3]` until it is unique. The next unique value would be `4`, which means we need to increment `nums[3]` by `1`. Now `nums[3]` becomes `4`, and we add `1` to `ans` making the total increments `2`.
5. **Returning the Answer**:
   - After processing each element, the modified array is `nums = [1, 2, 3, 4]`, and `ans = 2`. Therefore, the minimum number of increments needed to ensure all elements are unique is `2`.

The array modification steps are summarized as follows:

- Initial Array: `[3, 2, 1, 2]`
- After Sorting: `[1, 2, 2, 3]`
- Make `nums[2]` unique: `[1, 2, 3, 3]` (`ans = 1`)
- Make `nums[3]` unique: `[1, 2, 3, 4]` (`ans = 2`)

Return `ans`, which is `2`. If we apply the same approach to any array using the described algorithm, we will determine the minimum increments necessary to make all elements unique.

Python Solution

```python
class Solution:
    def minIncrementForUnique(self, nums: List[int]) -> int:
        # Sort the input list to ensure duplicate or smaller values follow larger values.
        nums.sort()

        # Initialize the answer to count the minimum increment required.
        increments_needed = 0

        # Iterate through the list starting from the second element.
        for i in range(1, len(nums)):
            # If the current number is less than or equal to the previous number,
            # it's not unique or needs to be incremented to be unique.
            if nums[i] <= nums[i - 1]:
                # Calculate the difference needed to make the current number
                # greater than the previous number by one.
                diff = nums[i - 1] - nums[i] + 1

                # Increment the current number by the calculated difference.
                nums[i] += diff

                # Add the difference to the increments needed.
                increments_needed += diff

        # Return the total number of increments needed to make all numbers unique.
        return increments_needed

```

Java Solution

```java
class Solution {
    public int minIncrementForUnique(int[] nums) {
        // Sort the input array to make it easier to deal with duplicates
        Arrays.sort(nums);

        // Initialize a variable to keep track of the number of increments needed
        int increments = 0;

        // Start iterating from the second element (i = 1) since we compare with the previous one
        for (int i = 1; i < nums.length; ++i) {
            // If the current element is less than or equal to the previous one, it's not unique
            if (nums[i] <= nums[i - 1]) {
                // Calculate the difference needed to make the current element unique
                int difference = nums[i - 1] - nums[i] + 1;

                // Increment the current element by the needed difference
                nums[i] += difference;

                // Accumulate the total increments needed
                increments += difference;
            }
        }

        // Return the total number of increments needed for the array to have all unique elements
        return increments;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include necessary headers

class Solution {
public:
    int minIncrementForUnique(vector<int>& nums) {
        // Sort the input array to arrange numbers in non-decreasing order.
        sort(nums.begin(), nums.end());

        // Initialize the variable to store the minimum increments needed.
        int minIncrements = 0;

        // Iterate through the array starting from the second element.
        for (int i = 1; i < nums.size(); ++i) {
            // Check if the current element is less than or equal to the previous element.
            if (nums[i] <= nums[i - 1]) {
                // Calculate the difference needed to make the current number unique.
                int difference = nums[i - 1] - nums[i] + 1;

                // Increment the current number by the calculated difference.
                nums[i] += difference;

                // Add the difference to the total number of increments needed.
                minIncrements += difference;
            }
        }

        // Return the total minimum increments needed to make all the nums unique.
        return minIncrements;
    }
};

```

Typescript Solution

```typescript
// Import array and algorithm functionality
function sortArray(nums: number[]): number[] {
    return nums.sort((a, b) => a - b);
}

function minIncrementForUnique(nums: number[]): number {
    // Sort the input array to arrange numbers in non-decreasing order.
    nums = sortArray(nums);

    // Initialize the variable to store the minimum increments needed.
    let minIncrements: number = 0;

    // Iterate through the array starting from the second element.
    for (let i = 1; i < nums.length; i++) {
        // Check if the current element is less than or equal to the previous element.
        if (nums[i] <= nums[i - 1]) {
            // Calculate the difference needed to make the current number unique.
            const difference: number = nums[i - 1] - nums[i] + 1;

            // Increment the current number by the calculated difference.
            nums[i] += difference;

            // Add the difference to the total number of increments needed.
            minIncrements += difference;
        }
    }

    // Return the total minimum increments needed to make all the numbers unique.
    return minIncrements;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is determined by several factors:

1. Sorting the input list, which is `nums.sort()`. This operation is typically implemented using an algorithm like Timsort (in Python's sort function), which has a time complexity of `O(n log n)` where `n` is the number of elements in the list.
2. A single `for` loop that iterates over the sorted list `nums`, which adds a time complexity of `O(n)`.

Hence, the total time complexity is dominated by the sorting operation, which gives us:

```
Time Complexity: O(n log n)
```

Space Complexity

The space complexity of the provided code is :

1. No extra space is used apart from the initial input list and a variable `ans` that keeps track of the increments needed to make each element in the list unique. This results in a constant amount of additional space being used, i.e., `O(1)`.

```
Space Complexity: O(1)
```