26. Remove Duplicates from Sorted Array

Easy Array Two Pointers

Problem Description

The problem presents an integer array called `nums` that is sorted in non-decreasing order. The goal is to remove duplicates from `nums` in a way that each unique element appears only once while maintaining the relative order of elements. This task must be done in-place, which means that we should not allocate extra space for another array, and we have to modify the original array instead.

Once duplicates are removed, we are required to return the new length of the array, which represents the number of unique elements. The first part of the array up to this new length should contain all unique elements in their original order, and whatever follows after doesn't matter.

To adhere to the problem's constraints, the final arrangement of the array and the value of `k` are checked against the expected result using the provided custom judge implementation. In simple terms, after calling the solution function, the array `nums` should have its first `k` elements without duplicates and `k` should equal the expected number of unique elements.

Intuition

To arrive at the solution, we realize that since the array is already sorted, duplicates will be adjacent to each other. Hence, we can simply iterate through the array and compare each element with the previous one (except for the first element) to identify duplicates.

Our intuition should tell us that we need two pointers here: one for iterating over the array (`x` in this case) and another to keep track of the location in `nums` where the next unique element should be placed (`k`).

Starting with `k` at 0, we move through the array with `x` and when `x` is not equal to `nums[k - 1]` (which would mean `x` is not a duplicate), we place `x` at `nums[k]` and increment `k`. This works because when `x` is a duplicate of `nums[k - 1]`, we simply skip it by not incrementing `k`, hence overwriting the duplicates in subsequent steps.

The process continues to the end of the array, ensuring that all unique elements are moved to the front of the array, and `k` represents the number of unique elements. By the end of the iteration, we return `k` as the result.

Learn more about Two Pointers patterns.

Solution Approach

The solution to this problem uses a technique commonly known as the two-pointer approach. This approach is often applied in array manipulation problems, especially when we need to modify the array in-place to satisfy certain constraints.

Here's how the two-pointer technique is applied step by step:

Step 1: Initialize Pointer `k`

A pointer `k` is initialized to 0. This pointer is used to track the position where the next unique element should be placed in the array.

Step 2: Iterate Through the Array

We iterate over each element `x` in the array `nums`. During iteration, `k` is used as a slow-runner pointer, while the loop index (implicitly represented by the iteration over `x`) acts as a fast-runner pointer.

Step 3: Identify Unique Elements

We only want to act when we come across a unique element. A unique element in this sorted array is identified by checking if it is different than the element at the position just before the current `k`. This is represented by the condition `x != nums[k - 1]`.

Step 4: Place Unique Elements and Increment Pointer `k`

When a unique element is found, we copy it to `nums[k]` and then increment `k`. This effectively shifts the unique elements to the front of the array and steps over any duplicates.

The first element is a special case because there is no `nums[k - 1]` when `k` is 0, so it's always considered unique, and we start `k` at 0 by default.

Step 5: Handle Duplicates

When a duplicate element (which is not unique) is encountered, the body of the `if` condition does not execute, which means that `k` is not incremented. This implies that the duplicate value will be overwritten in the next iteration when a new unique element is encountered.

Step 6: Return the Count of Unique Elements

After the iteration is completed, `k` holds the total count of unique elements, as it was incremented only when encountering a unique element. We return `k` at the end.

Here is how the implementation looks in Python:

```python
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        k = 0
        for x in nums:
            if k == 0 or x != nums[k - 1]:
                nums[k] = x
                k += 1
        return k
```

In this code, the `if k == 0` check allows us to handle the first element correctly, and the subsequent `x != nums[k - 1]` checks help in identifying if the current element is different from the element that was last identified as unique.

Example Walkthrough

Let's assume we have the following sorted integer array `nums`:

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

Our goal is to remove duplicates using the algorithm described.

Initial Setup

We initialize `k` to `0`. At this point, the array is unchanged.

Iteration 1

Element at index `0` is `1`. Since `k` is `0`, the first element is considered unique by default. We assign `nums[k]` to `1` and increment `k` to `1`.

Updated array state:

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

Iteration 2

Element at index `1` is also `1`. This is a duplicate of `nums[k - 1]`. We do not increment `k`.

Updated array state (no changes, duplicate ignored):

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

Iteration 3

Element at index `2` is `2`. Now, `nums[k - 1]` is `1`, so `2` is unique. We assign `nums[k]` to `2` and increment `k` to `2`.

Updated array state:

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

Iteration 4

Element at index `3` is `2`. This is a duplicate of `nums[k - 1]`. Again, `k` is not incremented.

Updated array state (no changes, duplicate ignored):

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

Iteration 5

Element at index `4` is `3`. Now, `nums[k - 1]` is `2`, so `3` is unique. We assign `nums[k]` to `3` and increment `k` to `3`.

Updated array state:

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

Conclusion

We've finished iterating over `nums`. The first `k` (3) elements [1, 2, 3] are the unique numbers from the original array. The rest of the array doesn't matter for this problem.

We return `k`, which is `3`, indicating the number of unique elements in the modified array.

The final state of `nums` is:

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

And the returned value is `3`.

Python Solution

```python
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        # Initialize the array position marker k to 0.
        # This will keep track of the position in the array
        # where the next unique element should be placed.
        k = 0

        # Loop through each number in the nums array.
        for x in nums:
            # Check if k is 0, which means we're at the start of the array
            # or if the current number is not equal to the last unique number we've seen.
            # The condition k == 0 is true for the first element,
            # so we store it as the first unique element.
            if k == 0 or x != nums[k - 1]:
                # Place the current unique element at the k-th position in the array.
                nums[k] = x
                # Increment k to indicate that the next unique element should be placed in the next position.
                k += 1

        # Return the length of the array that contains all unique elements,
        # which is also the new length of the array without duplicates.
        return k

```

Java Solution

```java
class Solution {
    // Method to remove duplicates from sorted array
    // and return the length of the array after duplicates have been removed.
    public int removeDuplicates(int[] nums) {
        // Initialize the count for unique elements
        int uniqueCount = 0;

        // Iterate over each element in the array
        for (int currentNum : nums) {
            // If it's the first element or is not equal to the previous element
            // (which means it's not a duplicate)
            if (uniqueCount == 0 || currentNum != nums[uniqueCount - 1]) {
                // Assign the current number to the next unique position in the array
                nums[uniqueCount++] = currentNum;
            }
        }

        // Return the count of unique elements, which is also the new length of the array
        return uniqueCount;
    }
}

```

C++ Solution

```cpp
#include <vector> // Required to use the std::vector container

// Solution class encapsulates the method to remove duplicates from an array.
class Solution {
public:
    // Method to remove duplicates from a sorted vector in-place.
    // @param nums is a reference to a vector of integers.
    // @return The new length of the vector after duplicates have been removed.
    int removeDuplicates(std::vector<int>& nums) {
        // The std::unique function returns an iterator to the element that follows the last element
        // not removed. The range between this iterator and the end of the vector contains all the
        // elements that need to be removed.
        auto newEnd = std::unique(nums.begin(), nums.end());

        // Erase the non-unique elements from the vector using the erase function.
        // This mutates the vector and adjusts its size accordingly.
        nums.erase(newEnd, nums.end());

        // Return the new size of the vector after removing duplicates.
        return nums.size();
    }
};

```

Typescript Solution

```typescript
/**
 * Removes all duplicates from a sorted array of numbers.
 * Duplicates are removed in-place and the function returns
 * the new length of the array after duplicates have been removed.
 *
 * @param {number[]} nums - A sorted array of numbers from which to remove duplicates.
 * @return {number} The new length of the array after duplicate removal.
 */
function removeDuplicates(nums: number[]): number {
    // Initialize the index for the next unique element.
    let nextUniqueIndex: number = 0;

    // Iterate through the array of numbers.
    for (const currentElement of nums) {
        // If we're at the start of the array or the current element is not
        // the same as the last unique element, we treat it as unique.
        if (nextUniqueIndex === 0 || currentElement !== nums[nextUniqueIndex - 1]) {
            // Store the current element at the next unique position.
            nums[nextUniqueIndex] = currentElement;
            // Increment the index for the next unique element.
            nextUniqueIndex++;
        }
        // If the current element is the same as the last unique element,
        // the loop continues without doing anything, effectively skipping duplicates.
    }

    // Return the new length of the array after all duplicates have been removed.
    // This length is equivalent to the next unique element's index.
    return nextUniqueIndex;
}

```

Time and Space Complexity

Time Complexity:

The given algorithm iterates through each element of the list exactly once. During each iteration, the algorithm performs a constant number of operations: a comparison, an assignment (when necessary), and an increment of the `k` counter. The time complexity is therefore linear relative to the length of the input list `nums`. If `n` represents the number of elements in `nums`, the time complexity can be expressed as `O(n)`.

Space Complexity:

The algorithm modifies the list `nums` in place and does not require any additional space that grows with the size of the input, except for the counter variable `k`. Therefore, the auxiliary space requirement is constant, and the space complexity is `O(1)`.