80. Remove Duplicates from Sorted Array II

Medium | Array | Two Pointers | Leetcode Link

Problem Description

Imagine you have a list of numbers that are sorted in ascending order, but some numbers appear more than once. Your task is to modify this list so that each unique number appears no more than twice. However, the challenge is to do this without using any additional space and to modify the original list directly—that means you can't create a new list to hold the result. You need to ensure the final list still remains sorted.

For example, if your list is `[1,1,1,2,2,3,3,3,3]`, your goal is to change it to something like `[1,1,2,2,3,3,__,__,__]` (with the underscores representing spaces you don't care about).

Ultimately, you'll return the length of the modified list (in the example above, it would be 6 since there are six numbers in the list after duplicates beyond the second instance are removed).

Intuition

The solution uses a two-pointer approach that exploits the fact that the input array is already sorted. The essence of the approach is to iterate over the array and make sure that we're copying over each unique element at most twice to the 'front' part of the array.

We use a variable `k` as a pointer to keep track of the '有效段' (or the valid segment) of the list – the portion that contains no more than two instances of any number. When we find a number that should be part of the valid segment, we copy it to the position indicated by `k` and increment `k`.

As we go through the list, for each new element we check:

- If `k` is less than 2, which means we're still filling up the first two slots, we can safely add the number without any checks.
- If the current number is not the same as the element two places before it in our '有效段' (valid segment of the list), we know that we haven't yet seen it twice, so we copy it to the current `k` position.

This way, once we've gone through the entire list, `k` points just past the last element of the desired valid segment. We don't care what's beyond it, and we return `k` as the new length of the non-duplicated (up to twice) list.

Solution Approach

The solution provided employs a simple algorithm with no additional data structures, adhering to an in-place modification constraint which is necessary for this problem. We use the two-pointer technique, but with just one variable `k` as the slow-runner pointer, while the `for x in nums` loop acts as the fast-runner pointer.

Here's a step-by-step explanation of the implementation:

1. We initialize the pointer `k` to zero. This will keep track of the position in the array where we will place the next unique element that we want to keep, which should appear at most twice.

2. We iterate over each number `x` in `nums` using a `for` loop.

3. For each number, we have two conditions to check:

   - If `k < 2`: This means we are at the beginning of `nums`, and since we can have at least two of the same element, we don't need to check for duplicates yet.
   - If `x != nums[k - 2]`: This is checked when `k` is greater than or equal to 2. Since the array is sorted, if the current number `x` is different from the two places before the current `k` index, it means `x` is different from at least the last two numbers in our "valid segment", so we can safely include `x` in our result.
4. If either condition is true, we assign the current number `x` to the `k`th position in the array, thereby ensuring it is part of the final array, and increment `k`.

5. After the loop finishes, `k` is now the length of the array with no duplicates (allowing up to two instances of the same number). We return `k` as the result.

The key to this algorithm is understanding that since the array is sorted, duplicates are always adjacent. By checking two steps back, we ensure that we only keep at most two instances of any element. Furthermore, using only the variable `k` to manage the valid part of the array ensures that we comply with the O(1) extra space constraint of the problem, as we're just rearranging the elements and not using any extra space.

This is the heart of the solution—the algorithm relies solely on the sorted nature of the array and the clever use of a single index to keep track of our "valid segment".

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose our input array is:

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

According to the problem description, we want to modify the array so that no number appears more than twice and we want to do this in place. Here's how we apply the two-pointer technique with the variable `k`:

1. We initialize `k` to zero.

2. Start iterating over each number in `nums`.

   a. `x = 1`, `k = 0` (k < 2 is True). Place `1` at `nums[0]`, increment `k` to 1.

   b. `x = 1`, `k = 1` (k < 2 is True). Place `1` at `nums[1]`, increment `k` to 2.

   c. `x = 1`, `k = 2` (k < 2 is False, but `x != nums[k - 2]` is False since `nums[0]` is 1). We skip this step.

   d. `x = 2`, `k = 2` (Since `x` is different from `nums[k - 2]` ⇒ 2 != `nums[0]`). Place `2` at `nums[2]`, increment `k` to 3.

   e. `x = 3`, `k = 3` (Since `x` is different from `nums[k - 2]` ⇒ 3 != `nums[1]`). Place `3` at `nums[3]`, increment `k` to 4.

   f. `x = 3`, `k = 4` (Since `x` is different from `nums[k - 2]` ⇒ 3 != `nums[2]`). Place `3` at `nums[4]`, increment `k` to 5.

   g. `x = 4`, `k = 5` (Since `x` is different from `nums[k - 2]` ⇒ 4 != `nums[3]`). Place `4` at `nums[5]`, increment `k` to 6.

3. By the end of the iteration, our array looks like this:

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

Here, __ represents the space we don't care about. The array `nums` now contains each unique number no more than twice, in sorted order, and `k` (which is 6) indicates the length of the modified array that we are concerned with. Therefore, we return `k` as the answer which is `6` in this case.

Python Solution

```python
from typing import List

class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        # Initialize the count of unique elements
        unique_count = 0

        # Iterate over each number in the input list
        for num in nums:
            # Check if the current number is different from the number
            # at position unique_count - 2.
            # This is to allow a maximum of two duplicates.
            if unique_count < 2 or num != nums[unique_count - 2]:
                # If condition met, copy the current number to the next position in the array.
                nums[unique_count] = num
                # Increment the count of unique elements.
                unique_count += 1

        # Return the length of the array containing no more than two duplicates of each element.
        return unique_count
```

Java Solution

```java
class Solution {
    public int removeDuplicates(int[] nums) {
        // 'k' is the index for placing the next unique element
        // or the second occurrence of an existing element
        int index = 0;

        // Iterate over each element in the array
        for (int num : nums) {
            // If the current position is less than 2 (i.e., we are at the start of the array)
            // or if the current element is different than the element two positions behind
            // then consider it for inclusion in the array
            if (index < 2 || num != nums[index - 2]) {
                // Place the current element at the 'index' position and increment 'index'
                nums[index] = num;
                index++;
            }
        }

        // The 'index' represents the length of the array without duplicates
        // allowing up to two occurrences
        return index;
    }
}
```

C++ Solution

```cpp
#include <vector>  // Include vector header for using the vector container

// Solution class containing the method to remove duplicates
class Solution {
public:
    // Method to remove duplicates from sorted array allowing at most two occurrences of each element
    int removeDuplicates(vector<int>& nums) {
        // Initialize the counter for the new length of the array
        int newLength = 0;

        // Iterate through each number in the input vector
        for (int num : nums) {
            // Check if we have seen less than 2 occurrences or if the current number
            // is not a duplicate of the number at newLength - 2 position
            if (newLength < 2 || num != nums[newLength - 2]) {
                // If the condition is true, copy the current number to the new position
                // and increase the length counter
                nums[newLength++] = num;
            }
        }

        // Return the new length of the array after duplicates are removed
        return newLength;
    }
};
```

Typescript Solution

```typescript
function removeDuplicates(nums: number[]): number {
    // Initialize the count, k, to be the index at which we insert the next unique element.
    let count = 0;

    // Iterate through each number in the given array.
    for (const current of nums) {
        // If the count is less than 2 or the current number is not equal to
        // the number two places before in the array, it is not a duplicate (or it's
        // the second occurrence of a number that is allowed twice), so we add it to the array.
        if (count < 2 || current !== nums[count - 2]) {
            nums[count] = current;
            count++; // Increment the count since we've added a unique number.
        }
    }

    // Return the new length of the array after duplicates have been removed.
    // Elements after the returned length are considered irrelevant.
    return count;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the input list `nums`. This is because the code consists of a single loop that goes through all elements of the list exactly once.

Space Complexity

The space complexity of the code is `O(1)`. No additional space is required that is dependent on the input size. The variable `k` is used to keep track of the position in the array while overwriting duplicates, but this does not scale with the size of the input.