Removing Duplicates from Sorted Array

Problem Statement

The problem "Remove Duplicates from Sorted Array" is a common coding challenge found on platforms like LeetCode. In this problem, the task is to manipulate a sorted array in-place to remove any duplicate elements, ensuring that each element appears only once. The goal is to modify the original array to achieve this outcome and return the new length of the modified array. It is important to note that the relative order of elements in the array must be preserved during this process, and the solution should not utilize any additional space to store a separate array.

This problem challenges programmers to devise an efficient algorithm that can handle the removal of duplicates without using extra memory space. By focusing on modifying the existing array without creating a new one, the solution aims to optimize both time and space complexity. This problem is a classic example of array manipulation and requires careful consideration of array indices and element comparisons to achieve the desired outcome.

When tackling the "Remove Duplicates from Sorted Array" problem, it is crucial to understand the constraints imposed by the requirement to perform the operation in-place. This constraint adds an extra layer of complexity to the problem, emphasizing the need for an algorithm that can efficiently update the array elements while maintaining their relative order.

Examples

To further illustrate the problem of "Remove Duplicates from Sorted Array," let's consider the following example inputs and outputs:

1. **Example input with an already unique list:**
   - Input: [1, 2, 3, 4, 5]
   - Output: The modified array remains the same as the input since there are no duplicate elements to remove. The new length of the array would be 5.
2. **Example input with some duplicate values:**
   - Input: [1, 1, 2, 2, 3, 4, 4, 5]
   - Output: After removing duplicates in-place, the modified array would be [1, 2, 3, 4, 5]. The new length of the array would be 5.
3. **Edge case with an empty array:**
   - Input: []

- Output: With an empty array as input, the modified array remains empty as there are no elements to remove duplicates from. The new length of the array would be 0.

These examples showcase different scenarios that the algorithm for removing duplicates from a sorted array should be able to handle efficiently. By testing with unique lists, lists containing duplicates, and edge cases like an empty array, we can ensure that the solution is robust and effective in various situations.

Constraints

The constraints for the problem "Remove Duplicates from Sorted Array" play a crucial role in defining the boundaries within which the solution must operate. These constraints provide essential guidelines for managing the array manipulation process effectively while ensuring the desired outcome of a sorted, duplicate-free array. Here are the key constraints for this problem:

- **Array Length Restrictions:** The length of the input array is constrained to be between 0 and 3 times 10 to the power of 4 (0 <= array.length <= 3 * 10^4). This limitation on the array size influences the efficiency of the algorithm, prompting the need for a solution that can handle arrays of varying lengths within this specified range.
- **Value Range within the Array:** Each element within the input array falls within the range of -100 to 100 (-100 <= array[i] <= 100). Understanding this boundary is essential for correctly identifying and removing duplicate elements while preserving the non-decreasing order of the array. The limited range of values guides the algorithm's decision-making process during element comparisons.
- **Sorted Order Requirement:** The input array is sorted in non-decreasing order, emphasizing the importance of maintaining this order throughout the removal of duplicates. The constraint of a sorted array influences the approach taken to update array elements in-place, ensuring that the final array remains sorted after removing any duplicate occurrences.

By adhering to these constraints, programmers can develop a solution that efficiently handles the removal of duplicates from a sorted array while staying within the specified limitations. Understanding and incorporating these constraints into the algorithm design process is essential for creating a robust and effective solution that meets the requirements of the problem statement.

Explanation

To solve the problem of "Remove Duplicates from Sorted Array," a detailed approach is required to efficiently manipulate the given array in-place without utilizing additional memory space. The key steps involved in solving this problem include iterating through the array, identifying duplicate elements, and overwriting duplicates by shifting elements to the left.

Approach Overview:

1. **Iterating through the Array:**
   - Start by traversing the sorted array from the second element (index 1) onwards.
   - Compare each element with its previous element to identify duplicates.
2. **Identifying and Overwriting Duplicates:**
   - If a duplicate is found, shift all subsequent elements one position to the left to overwrite the duplicate.
   - Maintain a pointer to track the position where the next unique element should be placed.
3. **Maintaining O(1) Additional Memory Complexity:**
   - Avoid using any additional data structures like sets or arrays to store unique elements.
   - Instead, update the existing array in-place to achieve the desired outcome.

Detailed Steps:

1. Initialize a pointer variable to track the position of the next unique element (let's call it uniquePointer) and set it to 1 initially.
2. Iterate through the array starting from the second element (index 1).
3. Compare the current element with the previous element:
   - If they are the same, continue iterating.
   - If they are different, overwrite the element at the uniquePointer position with the current element and increment the uniquePointer.
4. Continue this process until the end of the array is reached.
5. After iterating through the entire array, the elements up to the uniquePointer index will be the modified array with duplicates removed.
6. The length of the modified array will be equal to the value of uniquePointer.

By following this approach and ensuring O(1) additional memory complexity, the problem of removing duplicates from a sorted array can be effectively solved while preserving the relative order of elements within the array.

Simplified Explanation

To solve the 'Remove Duplicates from Sorted Array' problem, follow these simple steps:

1. **Start from the second element:**

- Begin iterating through the sorted array starting from the second element.

2. **Check for duplicates:**
   - Compare each element with the previous one to identify duplicates.
3. **Move duplicates to the end:**
   - If a duplicate is found, shift all following elements to the left to overwrite the duplicate.
4. **Track unique elements:**
   - Maintain a counter to keep track of the position where the next unique element should be placed.
5. **Avoid extra space:**
   - Ensure that you update the array in-place without using additional memory.
6. **Update array elements:**
   - Overwrite duplicates with unique elements while maintaining the order of elements.
7. **Finalize the unique array:**
   - The array up to the unique counter will contain the modified array without duplicates.
8. **Count the new length:**
   - The new length of the modified array will be equal to the unique counter value.

By following these straightforward steps, you can efficiently remove duplicates from a sorted array while adhering to the constraints and requirements of the problem. This simplified approach emphasizes the importance of iterating through the array, identifying duplicates, and updating the array in-place to achieve a unique element list without utilizing additional memory space.

Python Solution for Removing Duplicates from Sorted Array

```python
def removeDuplicates(nums):
    # Initialize pointer to track unique elements
    uniquePointer = 1

    # Iterate from the second element
    for i in range(1, len(nums)):
        # Compare current element with previous element
        if nums[i] != nums[i - 1]:
            # Overwrite duplicate elements
            nums[uniquePointer] = nums[i]
            uniquePointer += 1
```

```
    return uniquePointer

# Example usage:
nums = [1, 1, 2, 2, 3, 4, 4, 5]
new_length = removeDuplicates(nums)

print("Modified Array:", nums[:new_length])
print("New Length:", new_length)
```

This Python code provides a solution to the "Remove Duplicates from Sorted Array" problem by efficiently removing duplicate elements from a sorted array in-place. The removeDuplicates function takes the input array nums and iterates through it, identifying duplicates and overwriting them to create a unique element list. The function returns the new length of the modified array, which represents the count of unique elements.

In the example usage provided, the input array nums contains duplicate values, and after applying the removeDuplicates function, the modified array without duplicates is displayed along with its new length. This implementation adheres to the constraints of the problem, ensuring efficient removal of duplicates while maintaining the sorted order of elements in the array.

Detailed Explanation of Each Line of Code

Let's break down the Python code provided for removing duplicates from a sorted array and understand the significance of each line:

1. def removeDuplicates(nums):
    - This line defines a function named removeDuplicates that takes in a single parameter nums, which is the input sorted array from which duplicates need to be removed.
2. uniquePointer = 1
    - Here, a variable uniquePointer is initialized to 1. This pointer tracks the position where the next unique element should be placed in the modified array.
3. for i in range(1, len(nums)):
    - This line initiates a loop that iterates through the input array nums starting from the second element (index 1) to compare each element with its previous element.
4. if nums[i] != nums[i - 1]:
    - Within the loop, this conditional statement checks if the current element nums[i] is not equal to the previous element nums[i - 1]. This condition identifies when a duplicate element is encountered.
5. nums[uniquePointer] = nums[i]

    - When a unique element is found (i.e., when the current element is different from the previous one), this line overwrites the element at the uniquePointer position with the current unique element nums[i].
6. uniquePointer += 1
    - After replacing duplicate elements with unique ones, the uniquePointer is incremented by 1 to move to the next position where the next unique element should be placed in the modified array.
7. return uniquePointer
    - Finally, the function returns the value of uniquePointer, which represents the new length of the modified array with duplicate elements removed. This length corresponds to the count of unique elements in the array.
8. nums = [1, 1, 2, 2, 3, 4, 4, 5]
    - This line initializes the input array nums with duplicate values for testing the removal of duplicates.
9. new_length = removeDuplicates(nums)
    - The function removeDuplicates is called with the input array nums, and the returned value (new length of the modified array) is stored in the variable new_length.
10. print("Modified Array:", nums[:new_length])
    - This line prints out the modified array containing unique elements by slicing the array up to the new_length index.
11. print("New Length:", new_length)
    - Finally, this line displays the new length of the modified array after removing duplicates, indicating the count of unique elements present in the array.

Understanding each line of this Python code snippet is crucial for grasping how duplicates are efficiently removed from a sorted array in-place while maintaining the relative order of elements.