[Click on the logo to find **Problem Statement**]

Intuition

The given code aims to remove duplicates from a sorted array in-place using a two-pointer approach. The two pointers, named `slow` and `fast`, traverse the array simultaneously. The `slow` pointer is responsible for keeping track of the unique elements in the array, while the `fast` pointer iterates through the array to identify duplicates.

The algorithm works by comparing the elements at the `slow` and `fast` positions. If they are not equal, it indicates the discovery of a new unique element. In such cases, the `slow` pointer is incremented, and the value at the `fast` position is assigned to the `slow` position, effectively updating the unique element at the `slow` pointer.

The process continues until the `fast` pointer reaches the end of the array. At this point, the `slow` pointer points to the last unique element in the modified array. The length of the modified array, represented by `slow + 1`, is then returned.

This approach allows the algorithm to modify the input array in-place without using additional data structures, resulting in an efficient solution with a time complexity of O(n) and a space complexity of O(1), where n is the length of the input array.

Approach

1. **Initialization**: Set two pointers, `slow` and `fast`, initially pointing to the first and second elements of the array, respectively.

2. **Iterative Comparison**: Enter a while loop that continues until the `fast` pointer reaches the end of the array. Within the loop:

   a. Compare the elements at positions `nums[slow]` and `nums[fast]`.

   b. If the elements are different, it indicates the discovery of a new unique element.

3. **Updating Unique Elements**: Increment the `slow` pointer to move to the next position and update the value at that position with the element at `nums[fast]`. This step effectively stores the unique element at the `slow` pointer.

4. **Continue Traversal**: Increment the `fast` pointer to continue traversing the array.

5. **Return Length**: Once the traversal is complete, return `slow + 1`, which represents the length of the modified array containing unique elements.

Complexity

- Time complexity: O(n)
- Space complexity: O(1)

Code

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        slow, fast = 0, 1

        while fast < len(nums):
            if nums[slow] != nums[fast]:
                slow += 1
                nums[slow] = nums[fast]
            fast += 1

        return slow + 1
```

If you want to see more solutions to coding problems, you can visit:

GitHub