Remove Element

The LeetCode problem "Remove Element" requires you to manipulate an array in-place and return the new length of the array after removing all instances of a specified value. The problem is asking for an efficient way to modify the array while maintaining the relative order of the remaining elements.

Function Signature:

```
def removeElement(nums: List[int], val: int) -> int:
```

Expected Input:

- nums: A list of integers representing the initial array.
- val: An integer representing the value that needs to be removed from the array.

Expected Output:

- An integer representing the new length of the array after removing all instances of the specified value val.

The approach to solving this problem typically involves using two pointers – one for iterating through the array and another for keeping track of the current valid position to place elements without the specified value. This algorithm optimizes the space complexity by modifying the array in-place without using extra space.

By understanding the problem requirements and the function signature, you can implement an efficient solution to the "Remove Element" problem on LeetCode.

Examples

Here are several example test cases for the "Remove Element" problem along with their expected outputs:

Example 1:

Input:

- nums = [3, 2, 2, 3]
- val = 3

Output:

- After removing all instances of val = 3, the modified array should be [2, 2].
- The new length of the array should be 2.

Example 2:

Input:

- nums = [0, 1, 2, 2, 3, 0, 4, 2]
- val = 2

Output:

- After removing all instances of val = 2, the modified array should be [0, 1, 3, 0, 4].
- The new length of the array should be 5.

Example 3:

Input:

- nums = [1, 2, 3, 4, 5]
- val = 6

Output:

- Since there are no instances of val = 6 in the array, the array remains unchanged.
- The new length of the array should be the same as the original length, which is 5.

These examples cover various scenarios such as removing the specified value from different positions in the array, handling cases where the specified value is not present, and ensuring the array length is updated correctly after removal operations.

Constraints

In the "Remove Element" problem on LeetCode, several constraints need to be considered when devising a solution. The problem statement typically outlines the following limitations to define the boundaries within which the solution must operate:

Array Size Limitation: The problem may specify constraints on the size of the input array nums. This constraint defines the maximum number of elements that the array can contain and influences the algorithm's scalability and efficiency.

Element Value Constraints: There may be restrictions on the values that the elements in the array nums can take. Understanding these constraints is crucial for implementing the removal logic accurately and efficiently.

Complexity Constraints: The problem statement often indicates the desired time and space complexity of the solution. This constraint guides the selection of algorithms and data structures to ensure that the solution meets the performance requirements.

By acknowledging and adhering to these constraints, developers can craft solutions that not only solve the "Remove Element" problem effectively but also meet the specified limitations to produce efficient and scalable code.

Explanation

In the "Remove Element" problem on LeetCode, the core challenge revolves around efficiently modifying an array in-place to eliminate all occurrences of a specified value while preserving the order of the remaining elements. To tackle this problem effectively, a strategy employing two pointers is commonly utilized.

The primary approach involves iterating through the array with one pointer while using another pointer to keep track of the current valid position where elements without the specified value should be placed. By leveraging this algorithm, the space complexity is optimized as the array is altered without the need for additional space allocation.

When implementing the solution, it is crucial to consider various factors to ensure the correctness and efficiency of the algorithm. One key aspect is understanding the problem requirements and the function signature, which lays the foundation for devising a successful solution. By grasping the essence of the problem and the designated function parameters, developers can tailor their implementation to meet the specified objectives.

Moreover, it is essential to handle edge cases diligently, such as scenarios where the specified value is absent in the array. In such instances, the array should remain unchanged, with the final length reflecting the original length of the array. This attention to detail ensures the solution's accuracy across diverse input scenarios.

By following a structured approach that accounts for the problem's nuances, developers can navigate the "Remove Element" problem effectively on LeetCode. The strategic use of pointers, thoughtful consideration of edge cases, and adherence to problem constraints collectively contribute to the successful resolution of this array manipulation challenge.

Simplified Explanation

The "Remove Element" problem on LeetCode tasks you with efficiently modifying an array to eliminate all instances of a specified value while maintaining the order of the remaining elements. To achieve this, a common strategy involves using two pointers – one to iterate through the array and another to track the valid position for elements without the specified value.

By following this approach, you can update the array in-place without requiring extra space, optimizing the solution's space complexity. Understanding the problem's requirements and the function signature is crucial for implementing an effective solution.

When solving this problem, consider scenarios where the specified value is either present or absent in the array. If the value is not found, the array remains unchanged, and the final length matches the original array's length.

Adhering to constraints like array size limitations, element value restrictions, and complexity requirements is essential for crafting efficient and scalable solutions. By

addressing these constraints and implementing the two-pointer technique thoughtfully, you can successfully navigate the "Remove Element" problem on LeetCode.

Code

```
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        write_index = 0  # Initialize the write index to 0

        for i in range(len(nums)):  # Iterate through each element in nums
            if nums[i] != val:  # If the current element is not equal to val
                nums[write_index] = nums[i]  # Write the current element to the write index
                write_index += 1  # Move the write index to the next position

        return write_index  # Return the number of elements not equal to val
```

This code snippet provides a clean and structured solution to the 'Remove Element' problem on LeetCode. The function removeElement takes a list of integers nums and an integer val to remove from the array. By utilizing two pointers, the code efficiently updates the array in-place, removing all instances of the specified value while preserving the order of the remaining elements. The algorithm optimizes space complexity by modifying the array without additional space allocation.

The code first initializes two pointers i and j to iterate through the array and track the valid position for elements without the specified value, respectively. It then iterates through the array using pointer i, checks if the current element is not equal to val, and places valid elements at the position indicated by pointer j. Finally, the function returns the new length j, representing the number of valid elements after removal.

By following this well-structured code snippet and understanding the comments provided for each part of the code, developers can implement an efficient solution to the 'Remove Element' problem that aligns with best practices and effectively addresses the problem requirements.

Explanation of Each Line of Code

Let's break down the code snippet provided for the "Remove Element" problem on LeetCode and explain the purpose and functionality of each line:

def removeElement(nums: List[int], val: int) -> int:

- This line defines the function removeElement with two parameters: nums, a list of integers, and val, an integer to be removed from the array. The -> int indicates that the function returns an integer representing the new length of the array after removal.

i = 0

- Here, the variable i is initialized to 0, serving as a pointer to iterate through the array.

j = 0

- The variable j is also initialized to 0, acting as a pointer to keep track of the valid position to place elements without the specified value.

while i < len(nums):

- This while loop iterates through the array as long as the pointer i is within the bounds of the array length.

if nums[i] != val:

- This if statement checks if the current element in the array at index i is not equal to the specified value val.

nums[j] = nums[i]

- If the condition in the previous line is met, this line assigns the current valid element (not equal to val) to the position indicated by pointer j.

j += 1

- After placing a valid element at position j, this line increments the pointer j to prepare for the next valid element.

i += 1

- Regardless of the condition in the if statement, the pointer i is incremented to move to the next element in the array.

return j

- Finally, the function returns the value of j, which represents the new length of the array after removing all instances of the specified value val.

This code snippet effectively implements the logic required to solve the "Remove Element" problem by efficiently updating the array in-place and maintaining the order of the remaining elements. The strategic use of two pointers simplifies the removal process and optimizes the space complexity of the algorithm.