🎯 Question (LeetCode 27: Remove Element)

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` **in-place**. The relative order of the elements may be changed. Return the number of elements in `nums` which are not equal to `val`.

Example:

- Input: nums = [3,2,2,3], val = 3
- Output: 2, nums = [2,2]
- Input: nums = [0,1,2,2,3,0,4,2], val = 2
- Output: 5, nums = [0,1,3,0,4]

🏷️ Remove Element

📁 1. Definition and Purpose

- Remove all instances of a specific value from an array in-place.
- Useful in filtering datasets or modifying arrays without extra space.

📁 2. Syntax and Structure (Python)

```
# nums: list of integers
# val: integer value to remove
```

📁 3. Two Approaches

🧾 Approach 1: Brute Force

- Create a new array excluding `val` and copy back.

```
def remove_element_bruteforce(nums, val):
    temp = [x for x in nums if x != val]  # Step 1: Filter out val
    nums[:len(temp)] = temp                # Step 2: Copy back
    return len(temp)
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(n)

🧾 Approach 2: Optimized (Two Pointers In-place)

- Use two pointers to overwrite elements equal to `val`.
- Achieves O(1) extra space.

📁 4. Optimized Pseudocode

```
i = 0  # slow pointer for position to overwrite
for j in range(len(nums)):  # fast pointer
    if nums[j] != val:
        nums[i] = nums[j]
        i += 1
return i  # new length
```

📁 5. Python Implementation with Detailed Comments

```python
def remove_element(nums: list[int], val: int) -> int:
    """
    Remove all occurrences of val in-place and return new length.
    """
    i = 0  # slow pointer
    for j in range(len(nums)):  # fast pointer
        if nums[j] != val:  # Only keep elements not equal to val
            nums[i] = nums[j]  # Overwrite element at slow pointer
            i += 1  # Move slow pointer
    return i

# Example Usage
nums = [0,1,2,2,3,0,4,2]
val = 2
new_length = remove_element(nums, val)
print(new_length)  # Output: 5
print(nums[:new_length])  # Output: [0,1,3,0,4]
```

📁 6. Internal Working

- Fast pointer traverses the array.
- Slow pointer keeps track of position for valid elements.
- Overwrites unwanted elements in-place.

📁 7. Best Practices

- Use in-place method for memory efficiency.
- Ensure the returned length is used when slicing the array.
- Avoid creating new arrays for large datasets.

📁 8. Related Concepts

- Two-pointer technique
- In-place array manipulation
- Filtering elements in arrays

📂 9. Complexity Analysis

- **Optimized Approach:**
  - Time: O(n)
  - Space: O(1)
- **Brute Force Approach:**
  - Time: O(n)
  - Space: O(n)

📂 10. Practice and Application

- LeetCode: 27 Remove Element, 80 Remove Duplicates from Sorted Array
- Useful in real-time data filtering and preprocessing tasks.