ChatGPT

Give DSA notes based on this code: class Solution {
    public int removeElement(int[] nums, int val) {
        int i = 0; // Pointer for the next position to place non-val elements
        for (int j = 0; j < nums.length; j++) { // Traverse the array
            if (nums[j] != val) { // If the current element is not equal to val
                nums[i] = nums[j]; // Place it at index i
                i++; // Move the pointer for valid elements
            }
        }
        return i; // Return the new length of the array
    }
}

DSA Notes: Removing an Element from an Array

The given code addresses the problem of removing all occurrences of a specified value (`val`) from an array `nums` **in-place**. Below are the detailed notes on the approach:

Problem Analysis

- **Input**: An array `nums` of integers and an integer `val`.
- **Output**: Modified array with all occurrences of `val` removed, and the length of the new array returned.
- **Constraints**:
    - The operation must be performed **in-place** (i.e., no extra space for another array is allowed).
    - The relative order of the remaining elements must be preserved.

Approach: Two-Pointer Technique

1. **Purpose**: Use two pointers to filter out `val` from the array without using extra space.
2. **Pointers**:
   - `i`: Tracks the position where the next non-`val` element should be placed.
   - `j`: Iterates through the array.
3. **Logic**:
   - Iterate over the array with `j`.
   - If `nums[j] != val`, copy `nums[j]` to `nums[i]` and increment `i`.
   - At the end of the loop, all elements from index `0` to `i-1` are valid, and the rest of the array is irrelevant.

Code Walkthrough

```java
class Solution {
    public int removeElement(int[] nums, int val) {
        int i = 0; // Pointer for the next position to place non-val elements
        for (int j = 0; j < nums.length; j++) { // Traverse the array
            if (nums[j] != val) { // If the current element is not equal to val
                nums[i] = nums[j]; // Place it at index i
                i++; // Move the pointer for valid elements
            }
        }
        return i; // Return the new length of the array
    }
}
```

Example

Input: `nums = [3, 2, 2, 3]`, `val = 3`
Output: New length = `2`, Modified array = `[2, 2, _, _]`
Explanation:

- Initial array: `[3, 2, 2, 3]`
- `j = 0`: `nums[j] == 3`, skip it.
- `j = 1`: `nums[j] == 2`, copy `nums[1]` to `nums[0]`.
- `j = 2`: `nums[j] == 2`, copy `nums[2]` to `nums[1]`.
- `j = 3`: `nums[j] == 3`, skip it.
- Final array: `[2, 2, _, _]` (valid elements: first two).

Time and Space Complexity

- **Time Complexity**:
  - The algorithm runs in $O(n)$, where n is the length of the array. Each element is processed once.
- **Space Complexity**:
  - The algorithm uses $O(1)$ extra space since it operates directly on the input array.

Key Takeaways

1. **Two-Pointer Technique**:
   - Effective for in-place array modifications.
   - Maintains the relative order of elements in most implementations.
2. **In-Place Modification**:
   - Modify the array without creating a new one, saving space.
3. **Edge Cases**:
   - Empty array: Return `0`.
   - All elements are `val`: Return `0` after skipping all elements.
   - No elements are `val`: Return the original array length.

This method is commonly used in scenarios like:

- Filtering arrays.
- Removing duplicates or unwanted elements.
- Rearranging arrays based on conditions.