DSA In-Place Removal

Give DSA notes for this problem: Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.

Consider the number of elements in nums which are not equal to val be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the elements which are not equal to val. The remaining elements of nums are not important as well as the size of nums.
Return k.

Here are concise **DSA notes** for solving the problem:

Problem Analysis

1. You are tasked with **removing occurrences of a specific value (`val`)** from an array `nums` **in-place** (without using extra space for another array).
2. The key is to rearrange the array such that the first `k` elements are the elements that are **not equal to** `val`.
3. The output is the value of `k`, i.e., the count of elements not equal to `val`.

Constraints

- Modify the array in-place.
- Order of elements does not matter after rearranging.
- Use **O(1) extra space**.

Approach

Two-Pointer Technique

1. **Pointers Involved:**
   - `i`: Tracks the current position to check in the array.
   - `k`: Tracks the position to place the next valid element (not equal to `val`).
2. **Key Idea:**
   - Traverse the array with `i`.
   - When you find an element not equal to `val`, move it to the `k`th position and increment `k`.
3. **Algorithm Steps:**
   - Initialize `k = 0` (start writing at index `0`).
   - Loop through the array using `i`:
     - If `nums[i]` is not equal to `val`, place it at `nums[k]` and increment `k`.
   - After the loop, the first `k` elements of `nums` will contain elements not equal to `val`.
   - Return `k`.

Pseudocode

```text
1. Initialize k = 0.
2. For i in range(0, len(nums)):
     - If nums[i] != val:
         - nums[k] = nums[i]
         - Increment k
3. Return k
```

Key Observations

- **Why in-place?**
  - Instead of shifting elements or creating a new array, we overwrite elements from the front (`k` pointer).
- **Unordered Remaining Elements:**
  - The remaining part of the array after index `k` can be ignored as it is not required.

Time and Space Complexity

- **Time Complexity:** $O(n)$, where n is the length of the array, because each element is processed once.
- **Space Complexity:** $O(1)$, as no additional space is used.

Example

Input:

```text
nums = [3, 2, 2, 3]
val = 3
```

Execution:

1. Start with `k = 0`.
2. Loop:
   - `i = 0`: `nums[0] == 3` → Skip.
   - `i = 1`: `nums[1] == 2` → `nums[0] = 2`, `k = 1`.
   - `i = 2`: `nums[2] == 2` → `nums[1] = 2`, `k = 2`.
   - `i = 3`: `nums[3] == 3` → Skip.
3. Output: `k = 2`, `nums = [2, 2, ...]`.

Output:

```text
k = 2
```

Key Edge Cases

1. **Empty Array (`nums = []`):**
   - Output: `k = 0`.
2. **All elements equal to `val`:**
   - Example: `nums = [3, 3, 3], val = 3`.
   - Output: `k = 0`, `nums` is unimportant.
3. **No elements equal to `val`:**
   - Example: `nums = [1, 2, 4], val = 3`.
   - Output: `k = len(nums)`, all elements remain.

Let me know if you'd like to explore or optimize further!

Is there any optimize approach than this ?

The **two-pointer technique** described above is already optimal for this problem in terms of **time** and **space complexity**. Here's why and whether optimization is possible:

Why It's Already Optimal

1. **Time Complexity:**

- The array is traversed exactly **once**, making it $O(n)$.
- This is the minimum time complexity achievable, as you must examine each element to determine if it equals `val`.

2. **Space Complexity:**
   - Uses **O(1)** extra space because it modifies the array in-place.
   - No auxiliary data structures are used, which is already space-optimal.

Can We Optimize Further?

There are no faster or more space-efficient approaches for this specific problem because:

- $O(n)$ time is required to process all n elements.
- The in-place modification constraint ensures space usage is already minimized to $O(1)$.

Alternative Two-Pointer Variation

While the standard two-pointer technique is efficient, here's a slight variation that can reduce unnecessary writes:

1. **Two Pointers from Opposite Ends:**
   - Use two pointers: `left` starting at the beginning and `right` starting at the end of the array.
   - The idea is to swap elements that are `val` with elements from the end of the array.
2. **Key Steps:**
   - If `nums[left] == val`, swap `nums[left]` with `nums[right]` and decrement `right`.
   - Otherwise, increment `left`.
   - Stop when `left > right`.
3. **Why Use This?**
   - Reduces the number of write operations in the array by swapping only when necessary.

- However, the overall time complexity is still $O(n)$, and it might not make a big difference in practice.

Pseudocode for Variation

```text
1. Initialize left = 0, right = len(nums) - 1.
2. While left <= right:
     - If nums[left] == val:
         - Swap nums[left] with nums[right].
         - Decrement right.
     - Else:
         - Increment left.
3. Return left as the count of valid elements.
```

Example Execution for Variation

Input:

```text
nums = [3, 2, 2, 3], val = 3
```

Execution:

1. Start with `left = 0`, `right = 3`.
2. Iteration:
   - `nums[0] == 3`: Swap `nums[0]` and `nums[3]`. Array becomes `[3, 2, 2, 3]`. Decrement `right = 2`.
   - `nums[0] == 3`: Swap `nums[0]` and `nums[2]`. Array becomes `[2, 2, 3, 3]`. Decrement `right = 1`.
   - `nums[0] != 3`: Increment `left = 1`.

- `nums[1] != 3`: Increment `left = 2`.

3. Stop when `left > right`.
4. `left = 2` is the count of valid elements.

Output

- `k = 2`, `nums = [2, 2, ..., ...]`.

Comparison

- Both approaches have **O(n)** time complexity and **O(1)** space complexity.
- The swapping variation may slightly reduce writes but does not improve asymptotic performance.

Conclusion

The **standard two-pointer approach** is already optimal in terms of both time and space. The swapping variation can be considered in cases where reducing the number of writes matters (e.g., memory writes in hardware-level optimizations).