# 27. Remove Element

Easy Array Two Pointers

Leetcode Link

## Problem Description

Given an array of integers `nums` and an integer `val`, the task is to remove all occurrences of `val` from `nums`, while performing the operations in-place. The elements can be rearranged, which means the order after removal does not need to be the same as the original order. The goal is to return the total count of elements that are not equal to `val` after this removal.

To solve this problem, you don't need to actually delete elements from the array; you only need to move elements that are not equal to `val` to the beginning of the array. Once you've done this, you can return the count of these non-`val` elements. The remaining elements in the array after this count are irrelevant for the problem and can be ignored.

## Intuition

When approaching this problem, the key is to realize that since the order of the elements doesn't matter after removal, you can simply overwrite elements that equal `val` with elements that do not. By maintaining a variable `k` that tracks the number of valid elements (not equal to `val`), you can iterate through the array and every time you come across a valid element, you move it to the position indicated by `k`.

As you traverse the array with a loop, each time you find an element that isn't `val`, you place it at the `k`th index of the array and increment `k`. This simultaneously builds the subarray of valid elements at the front of the array while keeping track of its size. After the loop ends, every element before `k` is not `val`, and `k` itself represents the number of these valid elements, which is the final answer.

By applying this in-place method, you avoid extra space usage and complete the operation with a single pass through the array, resulting in an efficient solution.

## Solution Approach

The implementation of the solution employs a straightforward yet efficient algorithm. It uses no additional data structures, relying only on the in-place modification of the input array `nums`.

The algorithm can be summarized in the following steps:

1. Initialize a counter `k` to `0`. This will keep track of the number of elements that are not equal to `val`, as well as the index where the next non-`val` element should be placed.
2. Iterate through each element `x` in the array `nums` using a loop.
3. In each iteration, check if `x` is not equal to `val`.
4. If `x` is not equal to `val`, perform the following:
   - Assign the value of `x` to `nums[k]`, effectively moving it to the index `k` of the array.
   - Increment the counter `k` by `1`. This prepares `k` for the next valid element and also increments the count of non-`val` elements found so far.
5. Once the loop is finished, all elements not equal to `val` are placed at the start of the array in the first `k` positions.
6. Return the counter `k`, which represents the number of elements in `nums` that are not equal to `val`.

The simplicity of this algorithm lies in its single-pass traversal of the input array `nums` and the constant space complexity due to the in-place approach. Since we only use a simple counter and overwrite elements within the original array, we avoid the overhead of additional data structures.

The use of the counter as both a measure of valid elements and an index for placement makes this algorithm a fine example of efficient problem-solving with minimal operation.

Here is the Python implementation of the above approach, as presented earlier:

```python
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        k = 0
        for x in nums:
            if x != val:
                nums[k] = x
                k += 1
        return k
```

By carefully setting up the `k` index and moving valid elements in place, the algorithm avoids unnecessary operations and concludes with the desired result, which is the size `k` of the new array that contains no occurrences of `val`.

### Example Walkthrough

Let's go through a small example to illustrate how the solution approach works. We are given the following array of integers, `nums`, and a value, `val`, that needs to be removed from `nums`:

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

We want to remove all occurrences of `3`. According to the solution approach, we use an index `k` to store the position where the next non-`val` element should be placed. Here's how the process unfolds:

1. Initialize `k` to `0`. The array `nums` looks like this:

   ```
   nums = [3, 2, 2, 3]
   k = 0
   ```

2. Begin iterating over `nums`. The first element `nums[0]` is `3`, which is equal to `val`, so we do nothing with it and move on.

3. The second element `nums[1]` is `2`, which is not equal to `val`. We place `2` at index `k` (which is `0`), and then increment `k`:

   ```
   nums = [2, 2, 2, 3]
   k = 1
   ```

4. The third element `nums[2]` is also `2`. We again put `2` at index `k` (which is now `1`), and then increment `k`:

   ```
   nums = [2, 2, 2, 3]
   k = 2
   ```

5. The fourth and final element `nums[3]` is `3`, which is equal to `val`. We do nothing with it.

6. We have finished the loop, and now all elements not equal to `val` are placed at the start of `nums`. The array looks like this:

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

   Note that the last element (`3`) is irrelevant, as we only care about the first `k` elements.

7. We return `k`, which is `2`. This represents the number of elements in `nums` that are not `3` (our `val`). The first `k` elements of `nums` are the ones that count.

In this manner, the solution algorithm effectively removes all occurrences of `val` from `nums` and returns the count of non-`val` elements, all in a single pass through the array, without using extra space.

## Python Solution

```python
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        # Initialize a new index for the updated list without the value 'val'
        new_length = 0

        # Iterate over each number in the input list
        for number in nums:
            # If the current number is not the value to remove, update the list
            if number != val:
                # Assign the number to the new index location
                nums[new_length] = number
                # Increment the new length to move to the next index
                new_length += 1

        # Return the new length of the list after all removals are completed
        return new_length

```

## Java Solution

```java
class Solution {
    // Method to remove all instances of a given value in-place and return the new length.
    public int removeElement(int[] nums, int val) {
        int newLength = 0; // Initialize a counter for the new length of the array
        // Iterate over each element in the array
        for (int num : nums) {
            // If the current element is not the value to be removed, add it to the array's new position
            if (num != val) {
                nums[newLength++] = num;
            }
        }
        return newLength; // The new length of the array after removing the value
    }
}

```

## C++ Solution

```cpp
#include <vector> // Include vector header for using the vector container

class Solution {
public:
    // Function to remove all occurrences of a value from an array and return the new length
    int removeElement(std::vector<int>& nums, int val) {
        int newLength = 0; // Initialize the length of the new array after removal

        // Iterate through all the elements of the array
        for (int element : nums) {
            // Check if the current element is not the one to remove
            if (element != val) {
                // If it isn't, add it to the new array and increment the new array's length
                nums[newLength++] = element;
            }
        }

        // Return the length of the array after removal
        return newLength;
    }
};

```

## Typescript Solution

```typescript
/**
 * Removes all instances of a specified value from the array in-place and returns the new length of the array after removal.
 * The order of elements can be changed. It doesn't matter what you leave beyond the new length.
 *
 * @param {number[]} numbers - The array of numbers from which we want to remove the value.
 * @param {number} valueToRemove - The value to be removed from the array.
 * @returns {number} The new length of the array after removing the specified value.
 */
function removeElement(numbers: number[], valueToRemove: number): number {
    // Initialize the index for the placement of elements that are not equal to valueToRemove
    let newLength: number = 0;

    // Iterate over each element in the array
    for (const number of numbers) {
        // If the current number is not equal to the value we want to remove, we place it at the newLength index
        if (number !== valueToRemove) {
            numbers[newLength] = number;
            newLength++; // Increment the index for the next placement
        }
    }

    // Return the new length of the array after all removals
    return newLength;
}

```

## Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the array `nums`. This is because the code iterates over all elements in the array exactly once.

The space complexity of the code is `O(1)`, indicating that the amount of extra space used does not depend on the size of the input array. The solution only uses a constant amount of extra space for the variable `k`.