Problem Statement:

- Given an integer array nums, rotate the array to the right by k steps, where k is a non-negative integer. The rotation should modify the array in-place and use O(1) extra space.

Example 1:

- **Input:** nums = [1, 2, 3, 4, 5, 6, 7], k = 3
- **Output:** [5, 6, 7, 1, 2, 3, 4]
- **Explanation:**
  - ***Rotate the array 1 step to the right:*** [7, 1, 2, 3, 4, 5, 6]
  - ***Rotate the array 2 steps to the right:*** [6, 7, 1, 2, 3, 4, 5]
  - ***Rotate the array 3 steps to the right:*** [5, 6, 7, 1, 2, 3, 4]

Example 2:

- **Input:** nums = [-1, -100, 3, 99], k = 2
- **Output:** [3, 99, -1, -100]
- **Explanation:**
  - ***Rotate the array 1 step to the right:*** [99, -1, -100, 3]
  - ***Rotate the array 2 steps to the right:*** [3, 99, -1, -100]

Constraints:

- 1 <= nums.length <= 10^5
- -2^{31} <= nums[i] <= 2^{31} - 1
- 0 <= k <= 10^5

Follow-up:

- The problem challenges you to come up with multiple solutions, particularly one that modifies the array in-place with O(1) extra space. There are at least three different methods to approach this problem.

Approach:

To rotate the array in-place with O(1) extra space, the most efficient method involves reversing parts of the array. *This approach can be broken down into three key steps:*

1. **Reverse the entire array:** This moves the elements to the rightmost position.
2. **Reverse the first k elements:** This brings the last k elements to the front.
3. **Reverse the remaining n k elements:** This reorders the remaining part of the array to its correct rotated position.

Detailed Steps:

1. Reverse the entire array:

- The first step is to reverse the entire array, which places the elements that need to be rotated to the front in their desired final positions, but in reverse order.
- ***Example:*** nums = [1, 2, 3, 4, 5, 6, 7] becomes [7, 6, 5, 4, 3, 2, 1].

2. Reverse the first k elements:

- Next, reverse the first k elements, which reorders the elements in their final positions.
- ***Example:*** For k = 3, reversing the first k elements of [7, 6, 5, 4, 3, 2, 1] results in [5, 6, 7, 4, 3, 2, 1].

3. **Reverse the remaining elements:**

- Finally, reverse the elements from index k to the end of the array. This restores the rest of the elements to their correct positions after the rotation.
- ***Example:*** Reversing the elements from index k = 3 in [5, 6, 7, 4, 3, 2, 1] results in [5, 6, 7, 1, 2, 3, 4].

Why This Works:

- Rotating an array can be thought of as shifting its elements in circular fashion. By reversing different segments of the array, you effectively rearrange the elements in the desired rotated order. The reverse operation, when applied correctly, allows us to rotate the array using in-place operations, without requiring any additional arrays or extra memory, thus achieving O(1) space complexity.

Time Complexity:

- Each reversal of a section of the array takes O(n) time, where n is the length of the array. Since we are performing three reversal operations (reverse the entire array, reverse the first k elements, and reverse the remaining n k elements), the overall time complexity remains O(n).

Space Complexity:

- The solution uses only a constant amount of extra space (for variables like start, end, etc.). Hence, the space complexity is O(1).

Edge Cases:

1. **When k = 0:** No rotation is needed, so the array remains unchanged.

2. **When k >= len(nums):** Rotating the array by k steps is equivalent to rotating it by k % len(nums) steps because after every len(nums) rotations, the array returns to its original configuration.

3. **When nums has only one element:** No matter the value of k, the array remains unchanged as there is nothing to rotate.

Multiple Solutions:

1. Brute Force Approach:

- Shift the elements of the array one by one for k steps. This approach has a time complexity of O(k * n) and space complexity of O(1) but is inefficient for large arrays.

2. Using Extra Array:

- Create a new array to store the rotated values. This has a time complexity of O(n) but requires O(n) additional space.

3. Cyclic Replacement:

- Move elements directly to their final position by determining where each element will end up after the rotation. This method can be tricky to implement but also achieves O(n) time complexity and O(1) space complexity.