189. Rotate Array

Medium Array Math Two Pointers

Problem Description

In this problem, we are given an array of integers named `nums`. The goal is to rotate the elements of the array to the right by `k` steps. When we rotate the array, every element moves `k` places to the right, and the elements at the end of the array wrap around to the beginning. For example, if the array is `[1, 2, 3, 4, 5]` and `k` is 2, the rotated array will be `[4, 5, 1, 2, 3]`. It is important to note that `k` is non-negative, which means rotating by `k` steps always moves elements to the right, and may be larger than the length of the array.

Intuition

The key to approaching this problem is to understand that rotating an array by `k` steps to the right is equivalent to taking the last `k` elements and moving them to the front, while shifting the remaining elements to make space. However, if `k` is larger than the length of the array, rotating the array `k` times would effectively be the same as rotating it `k % len(nums)` times since every `len(nums)` rotations, the array returns to its original configuration.

To optimize the array rotation, we can follow a three-step reversal strategy which essentially achieves the rotation without additional storage for the displaced elements:

1. Reverse the entire array. This step puts the last `k` elements at the front, but in reversed order.
2. Reverse the first `k` elements. This step puts these elements into the correct order at the start of the array.
3. Reverse the last `n - k` elements, where `n` is the length of the array. This will fix the order of the rest of the array.

This method capitalizes on the idea that reversing specific parts of the array can be used to manipulate the positions of the elements with respect to each other, achieving the same result required by the rotation.

Solution Approach

The given Python solution implements the concept of array rotation by leveraging array slicing and concatenation. This method can be thought of as a one-step approach to the three-step reversal strategy outlined in the Reference Solution Approach. Let's break down the solution approach step by step:

1. To handle cases where `k` is greater than the length of the array `n`, we use `k %= len(nums)`. This calculates `k` modulo `n`, which effectively reduces `k` to the minimum number of steps needed to achieve the same resulting rotation.
2. The operation `nums[-k:]` obtains a slice of the last `k` elements of the array. These are the elements that need to be moved to the front of the array.
3. The operation `nums[:-k]` gets the slice of the array without the last `k` elements. This is the part of the array that will be shifted rightward by `k` positions.
4. By concatenating `nums[-k:] + nums[:-k]`, we are effectively placing the last `k` elements in front of the rest of the array, thus completing the rotation.
5. Finally, the slice assignment `nums[:]` updates the entire original array in place with the rotated version. This allows us to modify the array without returning anything, as the function signature requires the transformation to be applied in place.

It's important to note that no additional data structures are used in this approach. The slicing and concatenation operations take advantage of the Python language's capabilities to handle list manipulation efficiently.

This solution highlights a pattern often used in array manipulation problems — the clever use of slicing to avoid a more complex and potentially less efficient series of array operations.

Example Walkthrough

Let's walk through an example to illustrate the solution approach given in the content above. Suppose we have an array `nums` with the elements `[6, 7, 8, 1, 2, 3]` and `k` equals `2`.

We would like to rotate this array `k` steps to the right. Here's how we do it step by step:

1. First, we need to take care of cases where `k` may be greater than the length of `nums`. In our case, `k` is less than the length of `nums`, so `k %= len(nums)` does not change the value of `k`. If the `nums` array was shorter or `k` larger, this step would be critical to prevent unnecessary rotations.
2. Next, we take the last `k` elements of the array with the slice `nums[-k:]`. Since `k` is 2, this gives us `[2, 3]`.
3. Now, we take the rest of the array without the last `k` elements using `nums[:-k]`. This gives us `[6, 7, 8, 1]`.
4. We then concatenate the slices obtained in steps 2 and 3. So, `nums[-k:] + nums[:-k]` equals `[2, 3] + [6, 7, 8, 1]`, which results in the array `[2, 3, 6, 7, 8, 1]`.
5. The final step is to update the original array `nums` with the rotated version by using the slice assignment `nums[:]`. So, `nums[:]` becomes `[2, 3, 6, 7, 8, 1]`.

The original array `nums` is now rotated `k` steps to the right, and our final result is `[2, 3, 6, 7, 8, 1]`. No extra space was used; we've simply manipulated `nums` to its rotated form in place by using array slicing and concatenation.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        """Rotates the elements of the array to the right by k steps."""
        # The effective rotation needed when k is larger than the array's length
        k %= len(nums)

        # Perform rotation
        # The last k elements are moved to the front and the remainder are appended
        nums[:] = nums[-k:] + nums[:-k]
```

Java

```java
class Solution {
    // Class-level variable to store the input array
    private int[] nums;

    /**
     * Rotates the given array to the right by k steps.
     * @param nums Array to be rotated.
     * @param k Number of steps to rotate the array to the right.
     */
    public void rotate(int[] nums, int k) {
        // Assign the input array to the class-level variable
        this.nums = nums;

        // Number of elements in the array
        int n = nums.length;

        // Normalize the number of steps k to avoid extra rotations
        k %= n;

        // Reverse the entire array
        reverse(0, n - 1);

        // Reverse the first part (up to k elements)
        reverse(0, k - 1);

        // Reverse the second part (from k to the end of the array)
        reverse(k, n - 1);
    }

    /**
     * Reverses elements in the array between indices i and j.
     * @param i Starting index for reversal.
     * @param j Ending index for reversal.
     */
    private void reverse(int i, int j) {
        // Using two pointers approach, swap elements until pointers meet or cross
        while (i < j) {
            // Temporary variable to hold a value during the swap
            int temp = nums[i];

            // Perform swap
            nums[i] = nums[j];
            nums[j] = temp;

            // Move pointers towards each other
            ++i;
            --j;
        }
    }
}
```

C++

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

class Solution {
public:
    // This function rotates the elements of 'nums' to the right by 'k' steps
    void rotate(vector<int>& nums, int k) {
        int num_elements = nums.size();  // Get the number of elements in the vector
        k %= num_elements;               // Ensure k is within the bounds of the vector's size

        // Reverse the entire vector
        reverse(nums.begin(), nums.end());  // This puts the last k elements in front

        // Reverse the first k elements to restore their original order
        reverse(nums.begin(), nums.begin() + k);

        // Reverse the remaining elements to restore their original order
        reverse(nums.begin() + k, nums.end());
    }
};
```

TypeScript

```typescript
/**
 * Rotates the array `nums` to the right by `k` steps.
 * This version of the function is written with clearer naming, added comments, and maintains
 * the same in-place strategy for modifying the input array.
 * @param nums The input array of numbers to be rotated.
 * @param k The number of steps to rotate the array.
 */
function rotate(nums: number[], k: number): void {
    const arrayLength: number = nums.length;
    // Ensure the rotation steps are within the array bounds.
    k %= arrayLength;

    /**
     * Reverses the elements within the portion of the array from index `start` to `end`.
     * @param start The starting index of the portion to reverse.
     * @param end The ending index of the portion to reverse.
     */
    const reverse = (start: number, end: number): void => {
        while (start < end) {
            // Swap the elements at the `start` index and the `end` index.
            const temp: number = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;

            // Move towards the middle of the array from both ends.
            start++;
            end--;
        }
    };

    // Reverse the whole array.
    reverse(0, arrayLength - 1);
    // Reverse the first part (0 to k-1).
    reverse(0, k - 1);
    // Reverse the second part (k to n-1).
    reverse(k, arrayLength - 1);
}
```

```python
from typing import List

class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        """Rotates the elements of the array to the right by k steps."""
        # The effective rotation needed when k is larger than the array's length
        k %= len(nums)

        # Perform rotation
        # The last k elements are moved to the front and the remainder are appended
        nums[:] = nums[-k:] + nums[:-k]
```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of the array, because it involves slicing the array into two parts and then concatenating them, both of which take `O(n)` operations.

The space complexity is `O(1)` because the rotation operation modifies the array in-place. Although slicing the array appears to create new arrays, this is handled under the hood by Python and does not require additional space proportional to the size of the input array.