October 6

Problem 1: Array Rotation by K Steps

Problem Statement:

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Link to problem:

https://leetcode.com/problems/rotate-array/description/

Example 1:

Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 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 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]

Solution:

```
class Solution {
    public void rotate(int[] nums, int k) {
        int n = nums.length;  // Get the length of the array
        k = ((k % n) + n) % n; // Normalize k for both positive and negative values

        // Step 1: Reverse the entire array
        swap(nums, 0, n - 1);

        // Step 2: Reverse the first part (0 to k-1)
        swap(nums, 0, k - 1);

        // Step 3: Reverse the second part (k to n-1)
        swap(nums, k, n - 1);
    }

    static void swap(int[] arr, int i, int j) {
        while (i <= j) {
            int temp = arr[i];  // Store the element temporarily
            arr[i] = arr[j];    // Swap the elements
            arr[j] = temp;
            i++;
            j--;
        }
    }
}
```

Explanation:

- **Normalization of k**:When k is negative, rotating to the left can be interpreted as rotating to the right by n - |k| steps. The formula k = ((k % n) + n) % n ensures that k always falls within the valid range [0, n-1], making the solution work for both positive and negative k.

- **Reverse Technique**:
    - The array is reversed in three main steps:
        - First, reverse the entire array.
        - Then, reverse the first k elements.
        - Finally, reverse the remaining part of the array from k to the end.

- **Reversing Elements**: The reverse function swaps the elements between the indices i and j, effectively reversing the array.

Time Complexity:

- **O(n)**, where `n` is the length of the array. We perform three linear passes (one to reverse the entire array and two more to reverse parts of it), making the time complexity linear.

Space Complexity:

- **O(1)**, as the rotation is done in-place without any additional data structures or space, except for a few extra variables.

Problem 2: Intersection of Two Arrays

Problem Statement:

Given two integer arrays nums1 and nums2, return an array of their **intersection**. Each element in the result must be **unique**, and you may return the result in any order.

Link to problem:

https://leetcode.com/problems/intersection-of-two-arrays

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2]

Example 2:

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [9,4]

Explanation: [4,9] is also accepted.

Solution:

```
class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
        // Create two sets to store unique elements
        Set<Integer> set1 = new HashSet<>();
        Set<Integer> resultSet = new HashSet<>();

        // Add all elements of nums1 to set1
        for (int num : nums1) {
            set1.add(num);
        }

        // Check each element of nums2, if it's in set1, add to resultSet
        for (int num : nums2) {
            if (set1.contains(num)) {
                resultSet.add(num);
            }
        }

        // Convert resultSet to array and return
        int[] result = new int[resultSet.size()];
        int index = 0;
        for (int num : resultSet) {
            result[index++] = num;
        }
        return result;
    }
}
```

Explanation:

- **Sorting**: We sort both arrays nums1 and nums2 to allow a two-pointer comparison.

- **Two Pointers**: We use two pointers i for nums1 and j for nums2. We compare elements:
  - If nums1[i] == nums2[j], we add the element to the result list if it's not already there (to ensure uniqueness).
  - If nums1[i] < nums2[j], we increment i to move forward in nums1.
  - If nums1[i] > nums2[j], we increment j to move forward in nums2.
- Finally, we convert the result list into an array and return it.

Time Complexity:

- O(n log n + m log m): Sorting both arrays takes this time.
- O(n + m): After sorting, we do a linear comparison using two pointers.

Space Complexity:

- O(1) extra space for the algorithm (ignoring the output array). The result list requires O(min(n, m)) space.
