October 13

Problem 1: Find First and Last Position of Element in Sorted Array

Problem Statement:

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

Link to problem:

https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/description/

Example 1:

- Input: nums = [5,7,7,8,8,10], target = 8
- Output: [3,4]

Example 2:

- Input: nums = [5,7,7,8,8,10], target = 6
- Output: [-1,-1]

Solution:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] result = {-1, -1};

        result[0] = findFirst(nums, target);
        result[1] = findLast(nums, target);

        return result;
    }

    private int findFirst(int[] nums, int target) {
        int index = -1;
        int start = 0, end = nums.length - 1;
        while (start <= end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] >= target) {
                end = mid - 1;
```

```
            } else {
                start = mid + 1;
            }
            if (nums[mid] == target) index = mid;
        }
        return index;
    }

    private int findLast(int[] nums, int target) {
        int index = -1;
        int start = 0, end = nums.length - 1;
        while (start <= end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] <= target) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
            if (nums[mid] == target) index = mid;
        }
        return index;
    }
}
```

Explanation:

- We use binary search twice—once to find the first occurrence and once to find the last occurrence of the target.
- This approach ensures a time complexity of O(log n), which is efficient for large arrays.

Time Complexity: O(log n) (Binary Search)

Space Complexity: O(1)

Problem 2: Search in Rotated Sorted Array

Problem Statement:

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of* target *if it is in* nums*, or* -1 *if it is not in* nums.

You must write an algorithm with O(log n) runtime complexity.

Link to problem:

https://leetcode.com/problems/permutations/description/

Example 1:

- **Input:** `nums = [4,5,6,7,0,1,2], target = 0`
- **Output:** `4`

Example 2:

- **Input:** `nums = [4,5,6,7,0,1,2], target = 3`
- **Output:** `-1`

Solution:

```
class Solution {
    public int search(int[] nums, int target) {
        int start = 0, end = nums.length - 1;

        while (start <= end) {
            int mid = start + (end - start) / 2;

            if (nums[mid] == target) return mid;

            // Check if the left half is sorted
            if (nums[start] <= nums[mid]) {
                if (target >= nums[start] && target < nums[mid]) {
                    end = mid - 1;
                } else {
                    start = mid + 1;
                }
            }
            // Otherwise, the right half is sorted
            else {
                if (target > nums[mid] && target <= nums[end]) {
                    start = mid + 1;
                } else {
                    end = mid - 1;
                }
            }
        }
```

```
        return -1;
    }
}
```

Explanation:

- **Divide the Array into Halves**: The idea is to divide the array into two halves at each step using the middle element (mid). Since the array is rotated, one half of the array will still be sorted.
- **Check Which Half is Sorted**: When we calculate mid, one of two things will be true:
  - Either the **left half** (from start to mid) is sorted, or
  - The **right half** (from mid to end) is sorted.
- **Decide Where to Search**: Based on the sorted half, you can decide whether the target is likely in that half or the unsorted half:
  - If the **left half is sorted**:
    - Check if the target is within the range of the sorted half (nums[start] <= target < nums[mid]). If so, search in the left half.
    - Otherwise, search in the right half.
  - If the **right half is sorted**:
    - Check if the target is within the range of the sorted half (nums[mid] < target <= nums[end]). If so, search in the right half.
    - Otherwise, search in the left half.
- **Repeat Until Found or Not Found**: Keep adjusting the search range (start and end indices) until the target is found, or the search range becomes invalid (start > end), meaning the target is not in the array.

Time Complexity:

O(log n): Since we are halving the search space at each step, the time complexity is logarithmic, typical of binary search.

Space Complexity:

O(1): We are only using a few extra variables (start, end, mid), so the space complexity is constant.
