October 10

Problem 1: Remove Duplicates from Sorted Array

Problem Statement:

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

- Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
- Return k.

Link to problem:

https://leetcode.com/problems/remove-duplicates-from-sorted-array/description/

Example 1:

Input: nums = [1,1,2]

Output: 2, nums = [1,2,_]

Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,0,1,1,1,2,2,3,3,4]

Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]

Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

Solution:

```
class Solution {
  public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        // Initialize a pointer for the next unique element's position
        int i = 0;
        // Iterate through the array starting from the second element
        for (int j = 1; j < nums.length; j++) {
            // If a new unique element is found
            if (nums[j] != nums[i]) {
                i++;
                // Place the unique element in the next position
                nums[i] = nums[j];
            }
        }
        // The number of unique elements is i + 1
        return i + 1;
    }
}
```

Explanation:

- This problem uses the **Two Pointer** technique:
    - One pointer (i) keeps track of the position where the next unique element should be placed.
    - Another pointer (j) iterates through the array.
- If nums[j] is different from nums[i], it means we have found a new unique element. We increment i and assign nums[j] to nums[i].
- At the end of the iteration, i + 1 gives us the count of unique elements in the array.

Edge Cases:

- If the input array is empty, return 0 as there are no elements.

Time Complexity:

- O(n), where n is the length of the array. We traverse the array once with the two pointers.

Space Complexity:

- **O(1)**, since we modify the array in place and use no extra space except for the pointers.

Problem 2: Best Time to Buy and Sell Stock

Problem Statement:

You are given an array prices where prices[i] is the price of a given stock on the ith day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Link to problem:

https://leetcode.com/problems/best-time-to-buy-and-sell-stock/description/

Example 1:

- **Input:** `prices = [7,1,5,3,6,4]`
- **Output:** `5`
- **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6 - 1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

- **Input:** `prices = [7,6,4,3,1]`
- **Output:** `0`
- **Explanation:** In this case, no transactions are done, and the max profit = 0.

Solution:

```
class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length == 0) return 0;

        int minPrice = Integer.MAX_VALUE;  // Store the minimum price encountered so far
        int maxProfit = 0;  // Store the maximum profit calculated so far

        for (int i = 0; i < prices.length; i++) {
            // Update the minimum price if a lower price is found
            if (prices[i] < minPrice) {
                minPrice = prices[i];
            }
            // Calculate the profit if selling on day 'i', and update the maxProfit if it's higher
            else if (prices[i] - minPrice > maxProfit) {
                maxProfit = prices[i] - minPrice;
            }
        }
        return maxProfit;
    }
}
```

Explanation:

- The idea is to **buy at the lowest price** encountered so far and **sell at the highest price** that occurs after the buying day to maximize profit.

- We iterate through the array, keeping track of the **minimum price** we have seen. Then, at each step, we check the profit we would make if we sold on that day (prices[i] - minPrice), and we update the maximum profit accordingly.

Edge Cases:

- If the array is empty or contains only one price, no transactions can be made, so return `0`.

- If prices are continuously decreasing, the maximum profit is `0` because there is no day to sell the stock at a higher price than the buying price.

Time Complexity:

- O(n), where n is the number of days (or the length of the prices array). We only traverse the array once.

Space Complexity:

- O(n) ,since we are using only two variables (minPrice and maxProfit) to store intermediate results.