October 4

Problem: Two Sum

Problem Statement:

Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Link to problem:

https://leetcode.com/problems/two-sum/

Example 1:
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:
Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:
Input: nums = [3,3], target = 6
Output: [0,1]

Solution:

```
class Solution {
  public int[] twoSum(int[] nums, int target) {
    int[] res = new int[2];  // Array to store the result
    Map<Integer, Integer> map = new HashMap<>();  // HashMap to store the number and its index

    for (int i = 0; i < nums.length; i++) {
        // If the complement exists in the map, return the result
        if (map.containsKey(target - nums[i])) {
            res[0] = map.get(target - nums[i]);  // Index of the complement
            res[1] = i;  // Current index
            return res;  // Return immediately after finding the pair
        }
        // If complement not found, store the current number and index in the map
        map.put(nums[i], i);
```

```
    }

    return res;  // Return result (this line should never be reached due to the problem's guarantee)
}
}
```

Explanation:

- **We use a HashMap** to store the numbers and their indices as we loop through the array.
- For **each element**, we check if the **complement** (i.e., target - current number) is present in the map.
- If found, it means **we've found a pair** whose sum equals the target, and we return their indices.
- If not found, we add the current number and its index to the map for future checks.

Time Complexity:

- **O(n)**, where n is the number of elements in the array. We traverse the array once, and lookups in the HashMap are constant time on average.

Space Complexity:

- **O(n)**, since we use a HashMap to store the elements and their indices.
