October 3

Problem: Find the Duplicate Number

Problem Statement: Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive, there is only one repeated number in nums. Return this repeated number. You must solve the problem without modifying the array nums and using only constant extra space.

Link to problem:

https://leetcode.com/problems/find-the-duplicate-number/description/

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

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

Example 3:
Input: nums = [3,3,3,3,3]
Output: 3

Solution:

1. Set-based Approach (Commented Out):

You initially wrote a solution using a HashSet, which works by keeping track of numbers we've seen so far. Once we encounter a number we've already seen, that number is the duplicate.

Set-based approach:

- Time Complexity: **O(n)**, because we loop through all elements once.
- Space Complexity: **O(n)**, since we're using a HashSet to store the elements.

```
Set<Integer> set = new HashSet<>();

int ans = 0;

for (int i : nums) {

   if (set.contains(i)) {

      ans = i;
```

```
        break;

    }

    set.add(i);

}

return ans;
```

2. Two-Pointer (Floyd's Tortoise and Hare) Approach:

Next is two-pointer approach, which finds the duplicate by detecting a cycle in the array. Here's how it works:

- Two pointers (`slow` and `fast`) traverse the array at different speeds (slow by 1 step, fast by 2 steps).
- They meet at some point inside the cycle caused by the duplicate.
- After that, you reset one pointer and move both pointers one step at a time until they meet again at the entrance of the cycle — this is the duplicate number.

This approach works in **constant space** and doesn't modify the array.

Key Points:

- **Time Complexity**: **O(n)**, because we essentially pass through the array twice.
- **Space Complexity**: **O(1)**, as only two pointers are used (`slow` and `fast`).

```
class Solution {

    public int findDuplicate(int[] nums) {

        int slow = nums[0]; // Initialize slow pointer

        int fast = nums[0]; // Initialize fast pointer



        // Step 1: Find the intersection point in the cycle

        do {

            slow = nums[slow]; // Move slow pointer by 1 step

            fast = nums[nums[fast]]; // Move fast pointer by 2 steps

        } while (slow != fast); // Continue until they meet
```

```
        // Step 2: Find the entrance to the cycle
        slow = nums[0]; // Reset slow pointer to the start
        while (slow != fast) { // Move both pointers at the same speed
            slow = nums[slow]; // Move slow pointer by 1 step
            fast = nums[fast]; // Move fast pointer by 1 step
        }

        // Return the duplicate number
        return slow; // Both pointers meet at the duplicate number
    }
}
```

Explanation:

- We use Floyd's Tortoise and Hare (Cycle Detection) algorithm to find the duplicate number.
- First, we initialize two pointers: slow and fast. slow moves one step at a time, while fast moves two steps at a time.
- We loop until both pointers meet; this indicates that there is a cycle in the array caused by the duplicate number.
- Once they meet, we reset the slow pointer to the start of the array.
- We then move both pointers at the same speed (one step at a time) until they meet again. The meeting point will be the duplicate number.
- This approach guarantees we do not modify the original array and uses constant space

---

Step-by-Step Execution:

1. **Initialization:**
    - We initialize two pointers:
        - slow = nums[0] = 1
        - fast = nums[0] = 1
2. **Step 1: Find the intersection point in the cycle**

- Now we move the slow pointer by one step at a time and the fast pointer by two steps at a time:
  - First iteration:
    - slow = nums[slow] = nums[1] = 3
    - fast = nums[nums[fast]] = nums[nums[1]] = nums[3] = 2
  - Second iteration:
    - slow = nums[slow] = nums[3] = 2
    - fast = nums[nums[fast]] = nums[nums[2]] = nums[4] = 2
  - Both pointers meet at 2. This confirms there is a cycle, which is caused by the duplicate number.

3. **Step 2: Find the entrance to the cycle**
   - Now we reset the slow pointer to the beginning of the array (slow = nums[0] = 1) and move both pointers by one step at a time:
     - First iteration:
       - slow = nums[slow] = nums[1] = 3
       - fast = nums[fast] = nums[2] = 2
     - Second iteration:
       - slow = nums[slow] = nums[3] = 2
       - fast = nums[fast] = nums[2] = 2
     - Both pointers meet again at 2, which is the duplicate number.

Time Complexity:

- O(n), where n is the number of elements in the array. We traverse the array a limited number of times.

Space Complexity:

- O(1), as we are only using a few pointers for tracking the positions.
