## Problem 1

Given an integer list nums, write a function that solves the following problem; return true if any value appears at least twice in the list, and return false if every element is distinct.

Example 1:

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

Output: true

Example 2:

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

Output: false

Example 3:

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

Output: true

## Solution

```
public boolean code(int[] nums){
    Set<Integer> set = new HashSet<Integer>();
    for(int n: nums){
        if(set.contains(n)) return true;
        set.add(n);
    }
    return false;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

Utilizing “Set” data structure we can tackle this problem. A Set is a collection of unique elements. A HashSet is a specific implementation of a Set in which the operations, such as addition of a new element, lookup of an element, among others, can be done in constant time (O(1)), thanks to hashing. By traversing the integer list and asking if the element is already in the set (before adding it) we can either return true or continue by adding said element and moving on to the next one. The logic being that, if at any point in the traversal we find out that we had that number already registered it would mean that there are duplicates in the list; but if we reach the end of the traversal and that doesn’t happen that means that all the elements are unique, so we return false.

## Problem 2

Given a sorted list of distinct integers and a target value, write a function that solves the following problem; return the index if the target is found. If not, return the index where it would be if it were inserted in order.

Example 1:

Input: nums = [1,3,5,6], target = 5

Output: 2

Example 2:

Input: nums = [1,3,5,6], target = 2

Output: 1

Example 3:

Input: nums = [1,3,5,6], target = 7

Output: 4

## Solution

```
public int code(int[] nums){
    int left = 0, right = nums.length - 1;
    while(left <= right){
        int mid = left + (right - left)/2;
        if(target == nums[mid]) return mid;
```

```
            else if(target > nums[mid]) left = mid + 1;
            else right = mid - 1;
        }
        return left;
}
```

Time Complexity: O(log n)

Space Complexity: O(1)

To solve this problem, we use binary search to look for the index. The way binary search works is by jumping to the middle point of our array and checking if we are at our target, if not we check whether we have gone past the target or not far enough to reach it, depending on our current situation we change our variables to change our starting and ending pointers and keep checking on the correct portion (the portion where the element must be) of the list and we keep doing this until we find the target on the list or when our pointers are touching and we haven't found said element to then return the index where we should insert the element.