Tahir Emre Semiz - 20200808058

isMajority Function:

```
    public static boolean isMajority(int[] A, int x) {
        int n = A.length;
        int count = 0;

        // Count how many times x appears in the array
        for (int i = 0; i < n; i++) {
            if (A[i] == x) {
                count++;
            }
        }

        // If x appears more than half the array size, it is the majority
        return count > n / 2;
    }
```

Definition:

- This function checks whether a number in the array is majority or not.
- The amount of x in the array is stored in a variable called count and if this number is more than half of the array, x is the majority and returns true. if not, it returns false.

Time Complexity:

- **Best case** is that x does not appear in the array at all. In this case the array will be checked with a loop from the beginning to the end and each time the if block will return false. In this case the time complexity will be equal to **O(n)**.
- **Average case** is that x appears in the array in a random amount. However, in order to check this, all elements in the array must be checked. So again the time complexity will be equal to **O(n)**.
- **Worst case** is when x is the majority in the array, for this case the array must be checked from beginning to end. Again in this case the time complexity will be equal to **O(n)**.

Space Complexity:

- Only two fixed size variables **n** and **count** are used. Therefore the space complexity of the function does not change as the array size increases and is set to **O(1).**

Steps:

- The function takes an integer array (A) and an integer (x).
- Variables are defined. n represents the length of the array. Count counts the number of times the specified number is seen in the array.
- A for loop is used to check all elements of the array.
- With the if block inside the for loop, each element in the array is compared with the x value. If the element is equal to x, 1 is added to the count value.
- If count is more than half of the array, the function returns true otherwise it returns false.

findMajority Function:

```
public static int findMajority(int[] A) {
        int n = A.length;

        // If there is only one number in the array, that is the majority
        if (n == 1) {
            return A[0];
        }

        // Split the array into two halves
        int[] leftHalf = Arrays.copyOfRange(A, 0, n / 2);
        int[] rightHalf = Arrays.copyOfRange(A, n / 2, n);

        // Recursively find the majority in each half
        int a = findMajority(leftHalf);
        int b = findMajority(rightHalf);

        // Check if either of them is the majority in the original array
        if (isMajority(A, a)) {
            return a;
```

```
        } else if (isMajority(A, b)) {
            return b;
        }

        // If no majority is found in either half, return -1
        return -1;
    }
```

Definition:

- This function finds the majority element in the array.
- As a base case, if there is only one element in the array, that element is the majority and in this case it returns that element directly.
- With the divide and conquer method, the array is divided into two parts and the majority element is searched recursively in both halves.
- If one of the two halves satisfies the majority condition, it returns that element. If no condition is satisfied, -1 is returned.

Time Complexity:

- In the **best case**, the majority element is found immediately in each recursive call and only part of it is checked at each level. The best-case time complexity can be **O(logn)** due to **O(1)** operations at each level. However, even in this case **O(n)** operations are performed at each level. Thus the time complexity for the best case is **O(nlogn)**.
- In the **average case**, there are **O(n)** operations at each level because the whole sequence is checked from start to finish to find the majority element at each level. So the average case time complexity is **O(nlogn)**.
- The **worst case** scenario is when no majority can be found. In this case, the running time of the isMajority function at each level would be **O(n)** and the complexity of the findMajority function would be **O(nlogn)**.

Space Complexity:

- At each level, two subarrays are created, leftHalf and rightHalf. These arrays contain half as many elements as the original array. Since these subarrays are created at each level, **O(n)** memory is used at each level. The space complexity will be **O(n)** as the height of the recursion tree **(logn)** levels. Therefore the total space comlexity will be equal to **O(nlogn)**.

Steps:

- The length of the given array is assigned to n.
- As a base case, if there is only one element in the array, it returns that element directly because that element makes up the majority of the array.
- If there is more than one element in the array, divide and conquer comes into play here. the array is divided into two equal parts. The left half (leftHalf) and the right half (rightHalf).
- Recursively divide the array as far as it can be divided and find the majorities of the left and right halves. The majority of the left half is assigned to variable a and the majority of the right half is assigned to variable b.
- The majority elements in both halves are checked for majority in the original array using the isMajority function respectively. a or b is returned whichever is the majority.
- If no majority elements are found in both halves, it returns -1.

Test:

```
public static void main(String[] args) {
        int[] A = {3, 3, 6, 2, 3, 3, 7, 3, 3, 4, 1};
        int result = findMajority(A);
        System.out.println("Majority Element: " + result);
}
```

Result:

```
Majority Element: 3

Process finished with exit code 0
```