Code 1 : Single Number

```
/**
 * Approach 1 :
 */
class Solution {

    public int singleNumber(int[] nums) {
        // Create a HashMap to store the count of each element in the array.
        HashMap<Integer, Integer> count = new HashMap<>();

        // Iterate through the array and update the count of each element in the HashMap.
        for (int i = 0; i < nums.length; i++) {
            count.put(nums[i], count.getOrDefault(nums[i], 0) + 1);
        }

        // Iterate through the HashMap entries to find the element with a count of 1,
        // which represents the single number in the array.
        for (Map.Entry<Integer, Integer> entry : count.entrySet()) {
            if (entry.getValue() == 1) {
                // Return the key (single number) found in the HashMap.
                return entry.getKey();
            }
        }

        // If no single number is found, return 0.
        return 0;
    }
}

/* *
 * Time Complexity : O(n)
 * Space Complexity : O(1)
 */



/**
 * Approach 2 :
 */
class Solution {
    public int singleNumber(int[] nums) {
        // Initialize the result variable to store the single number.
        int ans = 0;

        // Iterate through the array and perform bitwise XOR on each element.
        // XOR operation cancels out duplicate numbers, leaving only the single number.
        for (int i = 0; i < nums.length; i++) {
```

```
            ans ^= nums[i];
        }

        // Return the result, which is the single number in the array.
        return ans;
    }
}

/* *
 * Time Complexity : O(n)
 * Space Complexity : O(1)
 */
```

Code 2 : First Repeating Element

```
class Solution {

    public static int firstRepeated(int[] arr, int n) {
        // Create a HashMap to store the count of each element in the array.
        HashMap<Integer, Integer> map = new HashMap<>();

        // Iterate through the array and update the count of each element in the HashMap.
        for (int i = 0; i < n; i++) {
            map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);
        }

        // Iterate through the array to find the index of the first repeated element.
        for (int i = 0; i < n; i++) {
            if (map.get(arr[i]) > 1) {
                // Return the index (1-based) of the first repeated element.
                return i + 1;
            }
        }

        // If no repeated element is found, return -1.
        return -1;
    }
}

/* *
 * Time Complexity : O(n)
 * Space Complexity : O(n)
 */
```

Code 3 : Key Pair

```
class Solution {
```

```
    boolean hasArrayTwoCandidates(int arr[], int n, int x) {
        // Sort the input array in ascending order.
        Arrays.sort(arr);

        // Initialize pointers for the start and end of the array.
        int start = 0, end = n - 1;

        // Iterate through the array using two pointers.
        while (start < end) {
            // Calculate the sum of elements at the current positions.
            int sum = arr[start] + arr[end];

            // Check if the sum equals the target.
            if (sum == x) {
                // Return true if a pair is found with the target sum.
                return true;
            }

            // If the sum is greater than the target, move the end pointer to the left.
            if (sum > x) {
                end--;
            } else {
                // If the sum is less than the target, move the start pointer to the right.
                start++;
            }
        }

        // If no pair is found with the target sum, return false.
        return false;
    }
}

/* *
 * Time Complexity : O(n log n)
 * Space Complexity : O(n)
 * Reason behind time and space complexity use Arrays.sort() method
 */
```