1)

- Linear search is a simple searching algorithm that checks each element of a list sequentially until the desired element is found or the list ends.

Ex:

```
public class LinearSearch {
    public static int linearSearch(int[] array, int key) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == key) {
                return i;
       }
        }
        return -1;
   }

    public static void main(String[] args) {
        int[] array = {2, 4, 0, 1, 9};
        int key = 1;

        int result = linearSearch(array, key);

        if (result == -1) {
            System.out.println("Element not found in the array");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }}
```

2)

- Binary search is an efficient searching algorithm that works on sorted arrays. It repeatedly divides the search interval in half and compares the middle element of the interval with the target value. If the middle element is equal to the target, the search is complete. If the target is less than the middle element, the search continues on the left half, otherwise, it continues on the right half. This process continues until the target is found or the interval is empty.

EX:

```
public class BinarySearch {
    public static int binarySearch(int[] array, int key) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int middle = left + (right - left) / 2;


            if (array[middle] == key) {
                return middle;
            }


            if (array[middle] < key) {
                left = middle + 1;
            } else {

                right = middle - 1;
            }
        }


        return -1;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        int key = 4;

        int result = binarySearch(array, key);

        if (result == -1) {
            System.out.println("Element not found in the array");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

3)

- Bubble Sort is not suitable for large datasets due to its inefficiency compared to more advanced algorithms like Quick Sort, Merge Sort, or Heap Sort. However, it is simple to implement and can be useful for educational purposes or small datasets

Ex:

```
public class BubbleSort {

    public static void bubbleSort(int[] array) {
        int n = array.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - 1 - i; j++) {
                if (array[j] > array[j + 1]) {

                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }


    public static void main(String[] args) {
        int[] array = {64, 34, 25, 12, 22, 11, 90};

        System.out.println("Unsorted array:");
        for (int num : array) {
            System.out.print(num + " ");
        }
        System.out.println();

        bubbleSort(array);

        System.out.println("Sorted array:");
        for (int num : array) {
            System.out.print(num + " ");
        }
    }}
```

4)

- Selection Sort is a simple and efficient sorting algorithm. The main idea is to repeatedly find the minimum element (considering ascending order) from the unsorted part and put it at the beginning.

Ex:

```
public class SelectionSort {

    public static void selectionSort(int[] array) {
        int n = array.length;

            for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

           int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }


    public static void main(String[] args) {
        int[] array = {64, 25, 12, 22, 11};

        System.out.println("Unsorted array:");
        for (int num : array) {
            System.out.print(num + " ");
        }
        System.out.println();

        selectionSort(array);

        System.out.println("Sorted array:");
        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

5)

- Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array one item at a time. It is much like sorting playing cards in your hands. The algorithm divides the array into a sorted and an unsorted region, and it iterates through the unsorted region, picking each element and inserting it into its correct position in the sorted region

- Insertion Sort is efficient for small datasets and nearly sorted data. It is more efficient than Bubble Sort and Selection Sort in practice, despite having the same worst-case time complexity

Ex:

```
public class InsertionSort {

   public static void insertionSort(int[] array) {
      int n = array.length;


      for (int i = 1; i < n; ++i) {
         int key = array[i];
         int j = i - 1;


         while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j = j - 1;
         }
         array[j + 1] = key;
      }
   }

     public static void main(String[] args) {
      int[] array = {12, 11, 13, 5, 6};

      System.out.println("Unsorted array:");
      for (int num : array) {
         System.out.print(num + " ");
      }
      System.out.println();

      insertionSort(array);

      System.out.println("Sorted array:");
      for (int num : array) {
         System.out.print(num + " ");
      }
   }}
```

6)

- The 2-Sum algorithm can be efficiently implemented in Java using a HashMap to store the numbers and their indices as you iterate through the array

- This approach ensures that the algorithm runs efficiently even for large arrays, making it a suitable solution for the 2-Sum problem.

Ex:

```
import java.util.HashMap;
import java.util.Map;

public class TwoSum {

    public static int[] twoSum(int[] nums, int target) {

        Map<Integer, Integer> map = new HashMap<>();

            for (int i = 0; i < nums.length; i++) {
                    int complement = target - nums[i];

                   if (map.containsKey(complement)) {
                         return new int[]{map.get(complement), i};
            }

                  map.put(nums[i], i);
        }

         throw new IllegalArgumentException("No two sum solution");
    }

    public static void main(String[] args) {
        int[] nums = {2, 7, 11, 15};
        int target = 9;

        try {
            int[] result = twoSum(nums, target);
            System.out.println("Indices: " + result[0] + ", " + result[1]);
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }}
```

7)

- To reverse an array in Java without using a temporary array, you can use a simple algorithm that involves swapping elements from the start and end of the array, moving towards the Center.

Ex:

```
public class ReverseArray {
    public static void main(String[] args) {
            int[] array = {1, 2, 3, 4, 5};

            System.out.println("Original array:");
            printArray(array);

            reverseArray(array);

            System.out.println("Reversed array:");
             printArray(array);
    }

        public static void reverseArray(int[] array) {
        int start = 0;
        int end = array.length - 1;

        while (start < end) {
             int temp = array[start];
            array[start] = array[end];
            array[end] = temp;


            start++;
            end--;
        }
    }

    // Method to print the array
    public static void printArray(int[] array) {
        for (int element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

8)

- To find duplicate numbers in an array in Java, you can use various approaches. Here are two common methods: using a HashSet for an efficient solution and using nested loops for a simpler approach.

Ex:

```
import java.util.HashSet;

public class FindDuplicates {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 3, 2, 7, 8, 8};


        findDuplicates(array);
    }

    public static void findDuplicates(int[] array) {
        HashSet<Integer> seen = new HashSet<>();
        HashSet<Integer> duplicates = new HashSet<>();

        for (int num : array) {
            if (seen.contains(num)) {
                duplicates.add(num);
            } else {
                seen.add(num);
            }
        }

        System.out.println("Duplicate numbers: " + duplicates);}}
```

9)

- To find the index of a specific element in an array in Java, you can use a simple loop to iterate through the array and compare each element with the target value

- Using Loop

Ex:

```
public class FindIndex {
    public static void main(String[] args) {
        int[] array = {10, 20, 30, 40, 50};
        int target = 30;

        int index = findIndex(array, target);

        if (index != -1) {
            System.out.println("Element " + target + " found at index: " + index);
        } else {
            System.out.println("Element " + target + " not found in the array.");
        }
    }

    public static int findIndex(int[] array, int target) {
```

```
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target) {
                return i;           }
        }
        return -1;
    }
}
```

- Using Arrays.binarySearch (for Sorted Arrays)

Ex:

```
import java.util.Arrays;

public class FindIndex {
    public static void main(String[] args) {
        int[] array = {10, 20, 30, 40, 50};
        int target = 30;


        int index = Arrays.binarySearch(array, target);

        if (index >= 0) {
            System.out.println("Element " + target + " found at index: " + index);
        } else {
            System.out.println("Element " + target + " not found in the array.");
        }
    }}
```

10)

- Insertion sort is a simple and intuitive sorting algorithm. It works similarly to how you might sort playing cards in your hands: you take each card and insert it into its proper position among the cards that are already sorted

Ex:

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] array = {12, 11, 13, 5, 6};

        System.out.println("Original array:");
        printArray(array);


        insertionSort(array);
```

```
    System.out.println("Sorted array:");
    printArray(array);
}


public static void insertionSort(int[] array) {
    for (int i = 1; i < array.length; i++) {
        int key = array[i];
        int j = i - 1;


        while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = key;
    }
}


public static void printArray(int[] array) {
    for (int num : array) {
        System.out.print(num + " ");
    }
    System.out.println();
}}
```

-kaif Zaki-