SE 317, Lab 3

Name: Aina Qistina Binti Azman

Net ID: 457 464 051

Part 1: Bubble Sort Testing

a) Write a faulty program (include any fault of your choice in your code). Make sure your code still compiles successfully.

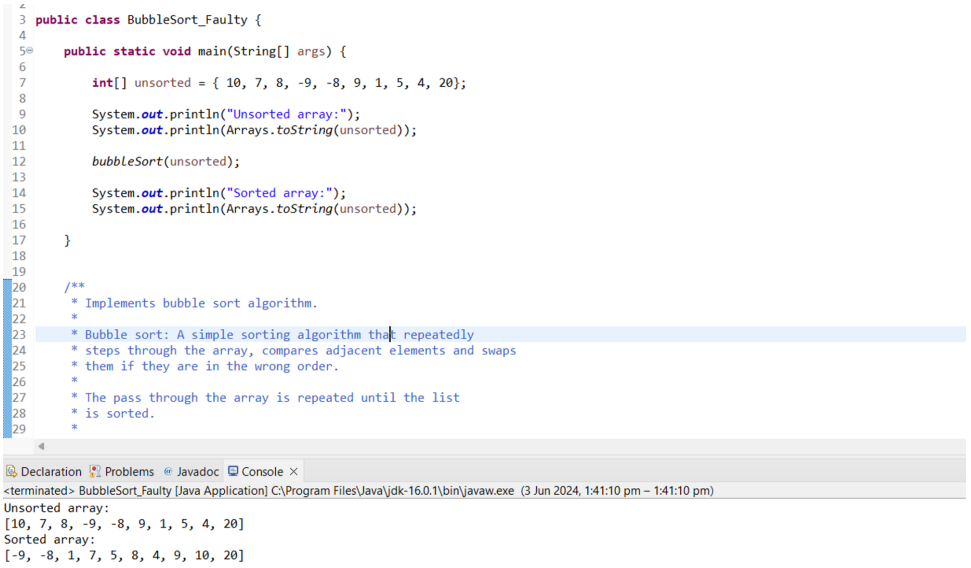
i.
Write down your source code in Java. The code output will display the original (input) array and the new array on different lines. Make sure all your code is clearly commented.

```
import java.util.*;

public class BubbleSort_Faulty {

    public static void main(String[] args) {

        int[] unsorted = { 10, 7, 8, -9, -8, 9, 1, 5, 4, 2
0};

        System.out.println("Unsorted array:");
        System.out.println(Arrays.toString(unsorted));

        bubbleSort(unsorted);

        System.out.println("Sorted array:");
        System.out.println(Arrays.toString(unsorted));

    }
```

```
    /**
     * Implements bubble sort algorithm.
     *
     * Bubble sort: A simple sorting algorithm that repeate
dly
     * steps through the array, compares adjacent elements
and swaps
     * them if they are in the wrong order.
     *
     * The pass through the array is repeated until the lis
t
     * is sorted.
     *
     * @param x the array to be sorted
     */
     public static void bubbleSort(int[] x) {

        //Initialize the number of rounds of sorting
        int round = 1;

        //Loop as many times as the length of the array
        while(round < x.length) {

            //Iterate over the array up to the length minus
the number of completed rounds
            for(int i = 0; i < x.length - round; i++) {

                //Swap if the current element is greater th
an next element
                if(x[i] > x[i+1]) {

                    int temp;
                    temp = x[i];

                    x[i] = x[i+1];
                    x[i+1] = temp;
                }
```

```
            }

            //Increment the round counter
            round += 2; //Faulty
        }
    }

}
```

ii.
Compile your code and take a screenshot.



b) Write two tests that do NOT reveal the fault.

i.
Each test case will include an input array of your choice and the expected output array.

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.*;

public class BubbleSort_FaultyTest_NotReveal {
```

```
    @Test
    public void test1() {

        int[] x = {1, 2, 3, 4, 5, 6, 7, 8};
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));

        BubbleSort_Faulty.bubbleSort(x);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }

    @Test
    public void test2() {

        int[] x = {2, 1, 4, 3, 5, 6, 7, 8};
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        BubbleSort_Faulty.bubbleSort(x);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }

}
```

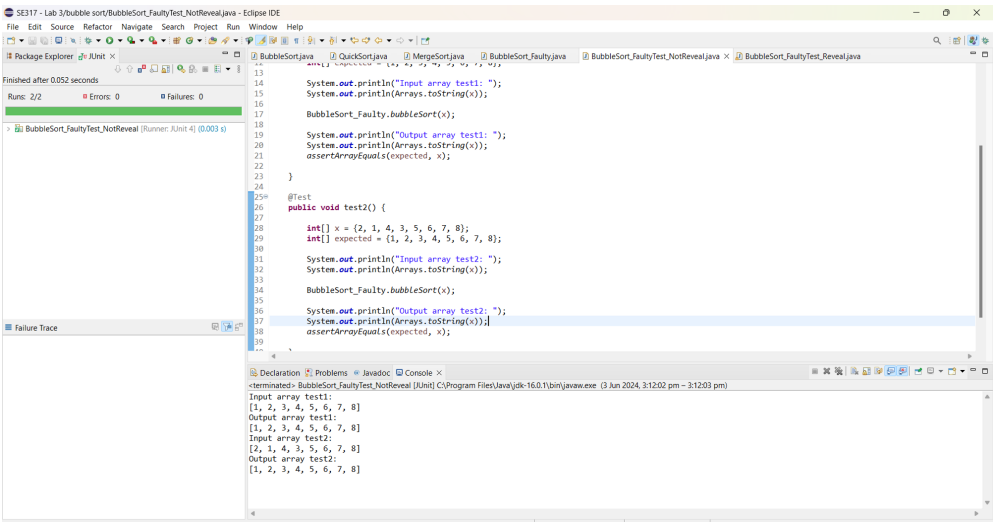
Explanation

These test cases do not reveal the fault because:-

- test1: The input array is already sorted. The algorithm did make comparisons but the elements of the array does not need any swaps. Even with the faulty increment on the round counter, the array remains sorted.
- test2: The input array only requires one swap in the first round to sort it. Subsequent rounds, affected by the faulty incrementation of the round counter, do not impact the already sorted portion of the array.

ii.

Run your code using the test array as input and take a screenshot of the input and actual output of each test case.



c) Write two tests that do reveal the fault.

i.

Each test case will include an input array of your choice and the expected output array

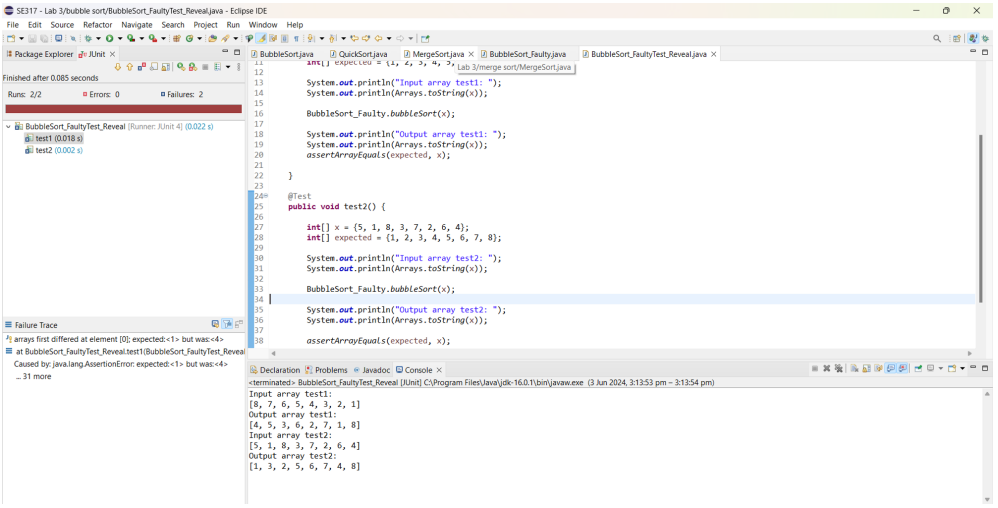
```
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.*;
```

```
public class BubbleSort_FaultyTest_Reveal {

    @Test
    public void test1() {

        int[] x = {8, 7, 6, 5, 4, 3, 2, 1};
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));

        BubbleSort_Faulty.bubbleSort(x);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }

    @Test
    public void test2() {

        int[] x = {5, 1, 8, 3, 7, 2, 6, 4};
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        BubbleSort_Faulty.bubbleSort(x);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);

    }
}
```

Explanation

- test1: The input array is in descending order. The faulty implementation , with its incorrect round counter incrementation, will fail to complete the required number of rounds, leaving the array partially sorted or completely unsorted.
- test2: The input array is randomized. The incorrect round counter incrementation will likely prevent the algorithm from completing the necessary rounds to fully sort the array, leading to an incorrect output.

ii.
Run your code using the test array as input and take a screenshot of the input and actual output of each case.



d) Identify and remove the fault.

i.
Explain your correction in the comments of your source code.

```
import java.util.Arrays;

public class BubbleSort {

    public static void main(String[] args) {
```

```
        int[] unsorted = { 10, 7, 8, 9, 1, 5};

        System.out.println("Unsorted array:");
        System.out.println(Arrays.toString(unsorted));

        bubbleSort(unsorted);

        System.out.println("Sorted array:");
        System.out.println(Arrays.toString(unsorted));

    }


    /**
     * Implements bubble sort algorithm.
     *
     * Bubble sort: A simple sorting algorithm that repeate
dly
     * steps through the array, compares adjacent elements
and swaps
     * them if they are in the wrong order.
     *
     * The pass through the array is repeated until the lis
t
     * is sorted.
     *
     * @param x the array to be sorted
     */
     public static void bubbleSort(int[] x) {

        //Initialize the number of rounds of sorting
        int round = 1;

        //Loop as many times as the length of the array
        while(round < x.length) {

            //Iterate over the array up to the length minus
```

```
the number of completed rounds
            for(int i = 0; i < x.length - round; i++) {

                //Swap if the current element is greater th
an next element
                if(x[i] > x[i+1]) {

                    int temp;
                    temp = x[i];

                    x[i] = x[i+1];
                    x[i+1] = temp;
                }

            }

            //Increment the round counter
            round++; //Increment correctly by 1
        }
    }

}
```

Explanation of the Fault & Fix:

The fault in the faulty bubble sort algorithm was the incorrect incrementation of the round counter. In the faulty implementation, the round counter was incremented by 2 in each iteration of the outer loop, which caused the algorithm to skip every other round of comparisons and swaps. As a result, the sorting process was incomplete, leading to arrays that were only partially sorted or completely unsorted.

To fix this fault, I modified the algorithm to correctly increment the round counter by 1 in each iteration of the outer loop. This change ensures that the algorithm performs the necessary number of rounds of comparisons and swaps to fully sort the array. With the corrected round counter incrementation, the bubble sort algorithm functions as intended, accurately sorting arrays in ascending order.

ii.

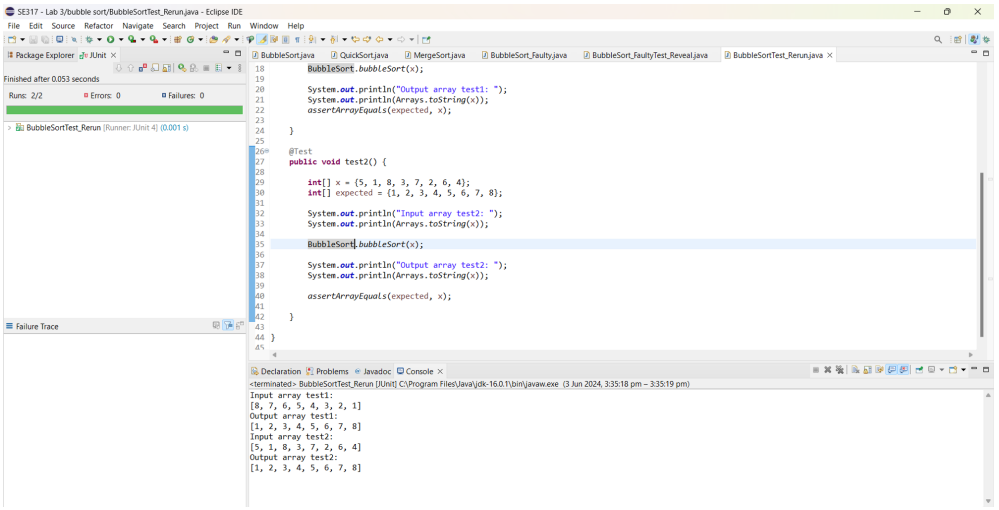
Run the test cases of part c again after fixing the fault.

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;

public class BubbleSortTest_Rerun {

    @Test
    public void test1() {

        int[] x = {8, 7, 6, 5, 4, 3, 2, 1};
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));

        BubbleSort.bubbleSort(x);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }

    @Test
    public void test2() {

        int[] x = {5, 1, 8, 3, 7, 2, 6, 4};
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));
```

```
        BubbleSort.bubbleSort(x);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);

    }

}
```

iii.

Take a screenshot of the input and actual output of each case.



Part 2: Quick Sort Testing

a) Write a faulty program (include any fault of your choice in your code). Make sure your code still compiles successfully.

i.

Write down your source code in Java. The code output will display the original

(input) array and the new array on different lines. Make sure all your code is clearly commented.

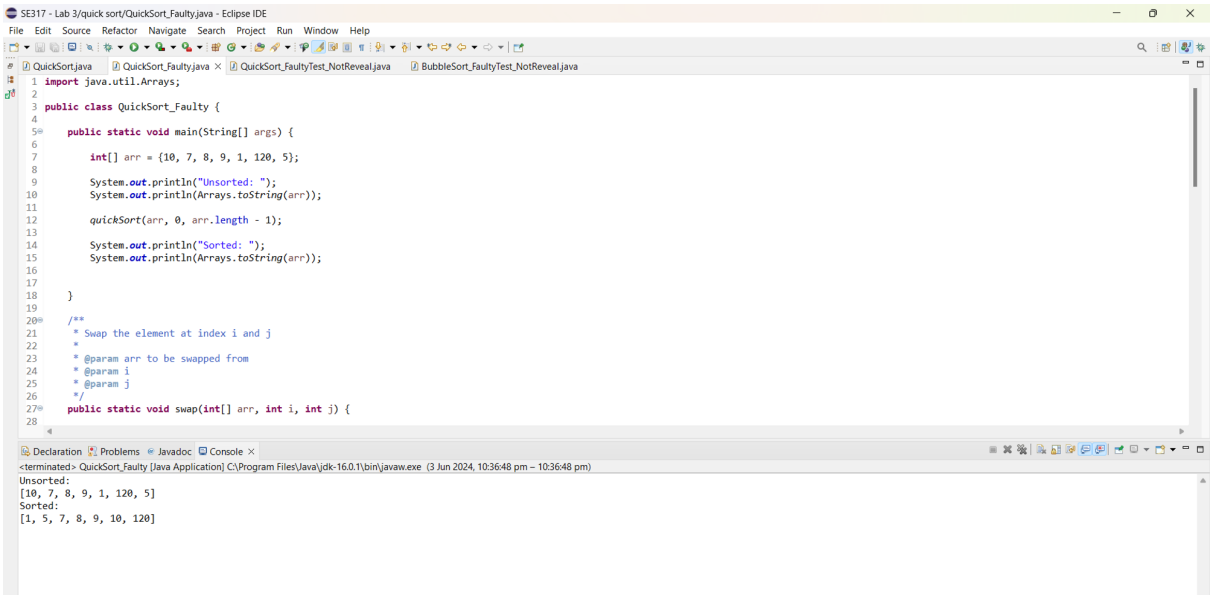
```
import java.util.Arrays;

public class QuickSort_Faulty {

    public static void main(String[] args) {

        int[] arr = {10, 7, 8, 9, 1, 120, 5};

        System.out.println("Unsorted: ");
        System.out.println(Arrays.toString(arr));

        quickSort(arr, 0, arr.length - 1);

        System.out.println("Sorted: ");
        System.out.println(Arrays.toString(arr));


    }

    /**
     * Swap the element at index i and j
     *
     * @param arr to be swapped from
     * @param i
     * @param j
     */
    public static void swap(int[] arr, int i, int j) {

        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;


    }
```

```
    /**
     * Last element of an array is chosen to be a pivot.
     * This function place the pivot element at its correct
position.
     *
     * @param arr to be partitioned
     * @param low the lowest index being compared
     * @param high the highest index being compared
     * @return
     */
    public static int partition(int[] arr, int low, int hig
h) {

        int pivot = arr[high]; //Last element of the array
as pivot

        int i = low - 1;

        for(int j = low; j <= high; j++) {

            //If current element is less than pivot
            if(arr[j] < pivot) {

                //Increment index of smaller element and sw
ap
                i++;
                swap(arr, i, j);
            }
        }

        //Place the pivot at the correct position
        swap(arr, i+1, high);

        //Return the pivot current index
        return i; //Faulty
```

```
    }

    /**
     * Implements quick sort
     *
     * @param arr array to be sorted
     * @param low starting index
     * @param high ending index
     */
    public static void quickSort(int[] arr, int low, int hi
gh) {

        if(low < high) {

            int p_index = partition(arr, low, high); // Fin
d the index of the current pivot

            quickSort(arr, low, p_index - 1); //Sort elemen
ts before the pivot
            quickSort(arr, p_index + 1, high); //Sort eleme
nts after the pivot
        }



    }

}
```

ii.

Compile your code and take a screenshot.



b) Write two tests that do NOT reveal the fault.

i.

Each test case will include an input array of your choice and the expected output array.

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;

public class QuickSort_FaultyTest_NotReveal {

    @Test
    public void test1() {

        int[] x = {10, 7, 8, 9, 1, 120, 5};
        int[] expected = {1, 5, 7, 8, 9, 10, 120};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));
```

```
        QuickSort_Faulty.quickSort(x, 0, x.length - 1);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);
    }

    @Test
    public void test2() {

        int[] x = {1, 3, 5, 7, 9, 11};
        int[] expected = {1, 3, 5, 7, 9, 11};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        QuickSort_Faulty.quickSort(x, 0, x.length - 1);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }

}
```

Explanation

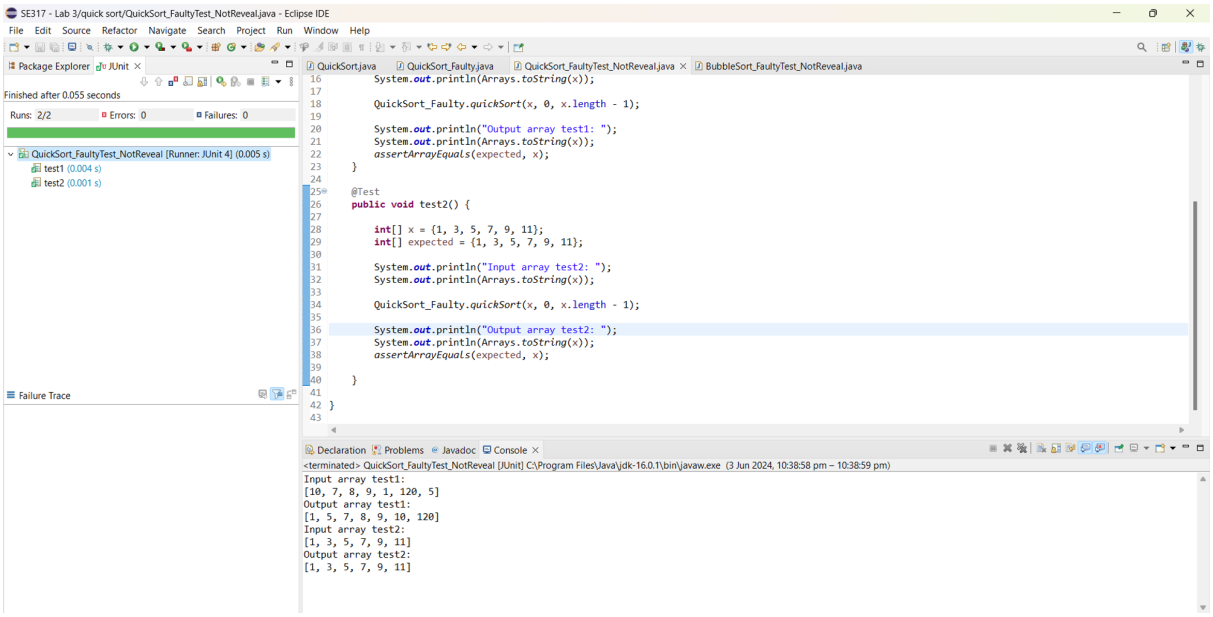
These test cases do not reveal the fault because:-

- test1: The array is sorted correctly despite returning ‘i’ instead of ‘i+1’ in the *partition* method is because of the specific arrangement of elements and pivot choices. In this case, the return of ‘i’ by the partition is 0. Hence, when recursive calling for *quickSort* method happens to sort the first part of the subarray, *quickSort*(arr, 0, -1), the low value is indeed greater than the high value, which leads to coincidentally correct repositioning.

- test2: The input array is already sorted. Each partition will choose the last element as the pivot, and since the array is already sorted, no swaps happened. Returning ‘i’ instead of ‘i+1’ in the *partition* method will still result in correctly sorted subarrays.

ii.

Run your code using the test array as input and take a screenshot of the input and actual output of each test case.



c) Write two tests that do reveal the fault.

i.

Each test case will include an input array of your choice and the expected output array

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;

public class QuickSort_FaultyTest_Reveal {
```

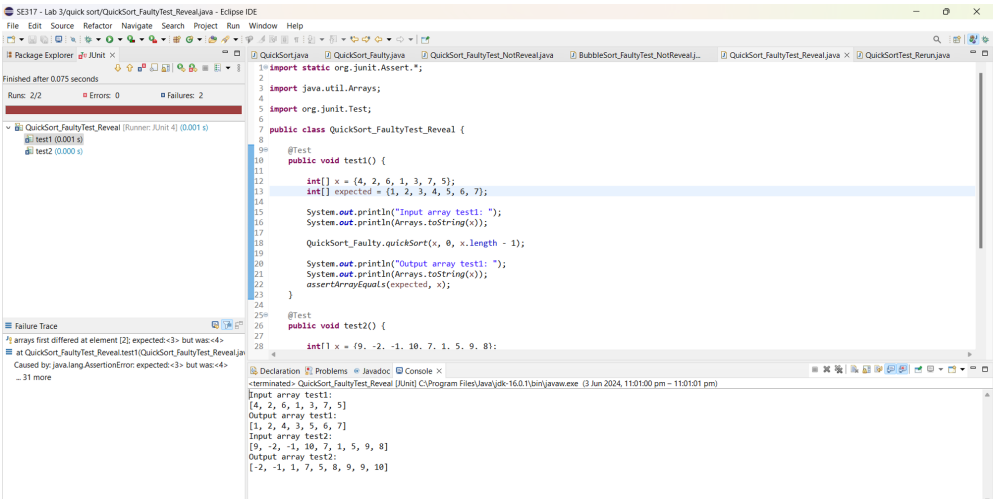
```
    @Test
    public void test1() {

        int[] x = {4, 2, 6, 1, 3, 7, 5};
        int[] expected = {1, 2, 3, 4, 5, 6, 7};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));

        QuickSort_Faulty.quickSort(x, 0, x.length - 1);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);
    }

    @Test
    public void test2() {

        int[] x = {9, -2, -1, 10, 7, 1, 5, 9, 8};
        int[] expected = {-2, -1, 1, 5, 7, 8, 9, 9, 10};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        QuickSort_Faulty.quickSort(x, 0, x.length - 1);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }
}
```

Explanation

- test1: The incorrect partition return value causes the pivot, 5, not to be correctly placed, leading to incorrect subarray partitioning.

- test2: The incorrect partition return value causes the pivot, 8, not to be correctly placed, leading to incorrect subarray partitioning.

ii.
Run your code using the test array as input and take a screenshot of the input and actual output of each case.



d) Identify and remove the fault.

i.
Explain your correction in the comments of your source code.

```
import java.util.*;

public class QuickSort {

    public static void main(String[] args) {

        int[] arr = {10, 7, 8, 9, 1, 5};

        System.out.println("Unsorted: ");
        System.out.println(Arrays.toString(arr));

        quickSort(arr, 0, arr.length - 1);
```

```
        System.out.println("Sorted: ");
        System.out.println(Arrays.toString(arr));


    }

    /**
     * Swap the element at index i and j
     *
     * @param arr to be swapped from
     * @param i
     * @param j
     */
    public static void swap(int[] arr, int i, int j) {

        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;


    }

    /**
     * Last element of an array is chosen to be a pivot.
     * This function place the pivot element at its correct
position.
     *
     * @param arr to be partitioned
     * @param low the lowest index being compared
     * @param high the highest index being compared
     * @return
     */
    public static int partition(int[] arr, int low, int hig
h) {

        int pivot = arr[high]; //Last element of the array
as pivot
```

```
        int i = low - 1;

        for(int j = low; j <= high; j++) {

            //If current element is less than pivot
            if(arr[j] < pivot) {

                //Increment index of smaller element and sw
ap
                i++;
                swap(arr, i, j);
            }
        }

        //Place the pivot at the correct position
        swap(arr, i+1, high);

        //Return the pivot current index
        return i+1; //Return the correct index of pivot
    }

    /**
     * Implements quick sort
     *
     * @param arr array to be sorted
     * @param low starting index
     * @param high ending index
     */
    public static void quickSort(int[] arr, int low, int hi
gh) {

        if(low < high) {

            int p_index = partition(arr, low, high); // Fin
d the index of the current pivot

            quickSort(arr, low, p_index - 1); //Sort elemen
ts before the pivot
```

```
            quickSort(arr, p_index + 1, high); //Sort eleme
nts after the pivot
        }

    }

}
```

Explanation of the Fault & Fix:

The fault in the quick sort algorithm was in the *partition* method. Specifically, the issue was with the return value of the pivot index. The method incorrectly returned 'i' instead of 'i+1'. This caused the recursive calls of *quickSort* method to receive incorrect sub-array boundaries, leading to improper sorting.

To fix the issue, I corrected the return value of the *partition* method to 'i+1'. This ensures that the pivot element is placed in its correct sorted position, and the subsequent recursive calls to *quickSort* method work with the correct sub-array boundaries.

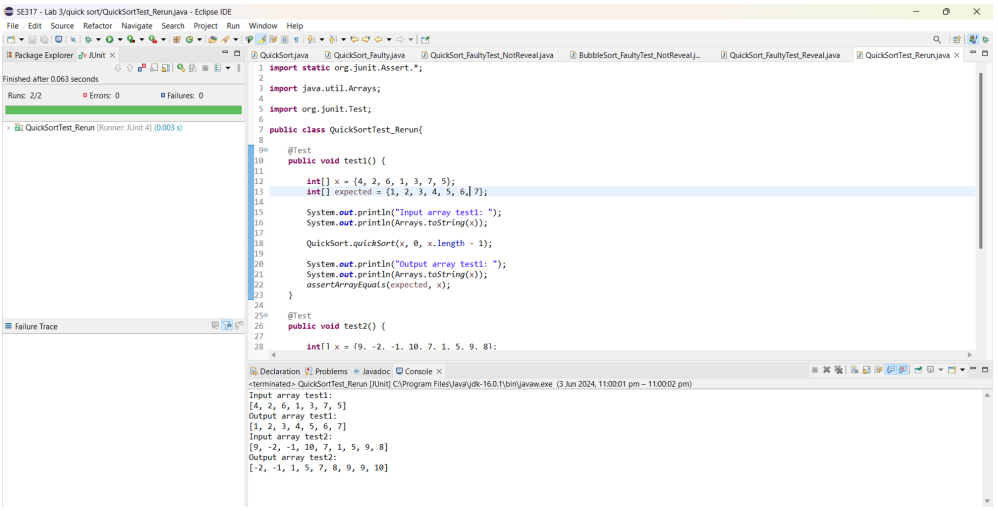
ii.
Run the test cases of part c again after fixing the fault.

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;

public class QuickSort_FaultyTest_Reveal {

    @Test
    public void test1() {

        int[] x = {4, 2, 6, 1, 3, 7, 5};
        int[] expected = {1, 2, 3, 4, 5, 6, 7};

        System.out.println("Input array test1: ");
```

```
        System.out.println(Arrays.toString(x));

        QuickSort_Faulty.quickSort(x, 0, x.length - 1);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);
    }

    @Test
    public void test2() {

        int[] x = {9, -2, -1, 10, 7, 1, 5, 9, 8};
        int[] expected = {-2, -1, 1, 5, 7, 8, 9, 9, 10};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        QuickSort_Faulty.quickSort(x, 0, x.length - 1);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));
        assertArrayEquals(expected, x);

    }
}
```

iii.

Take a screenshot of the input and actual output of each case.



Part 3: Merge Sort Testing

a) Write a faulty program (include any fault of your choice in your code). Make sure your code still compiles successfully.

i.

Write down your source code in Java. The code output will display the original (input) array and the new array on different lines. Make sure all your code is clearly commented.

```
import java.util.Arrays;

public class MergeSort_Faulty{

    public static void main(String args[]) {

     int[] arr = {10, 7, 8, 9, 1, 5};

        System.out.println("Unsorted: ");
        System.out.println(Arrays.toString(arr));

        mergeSort(arr, 0, arr.length - 1);
```

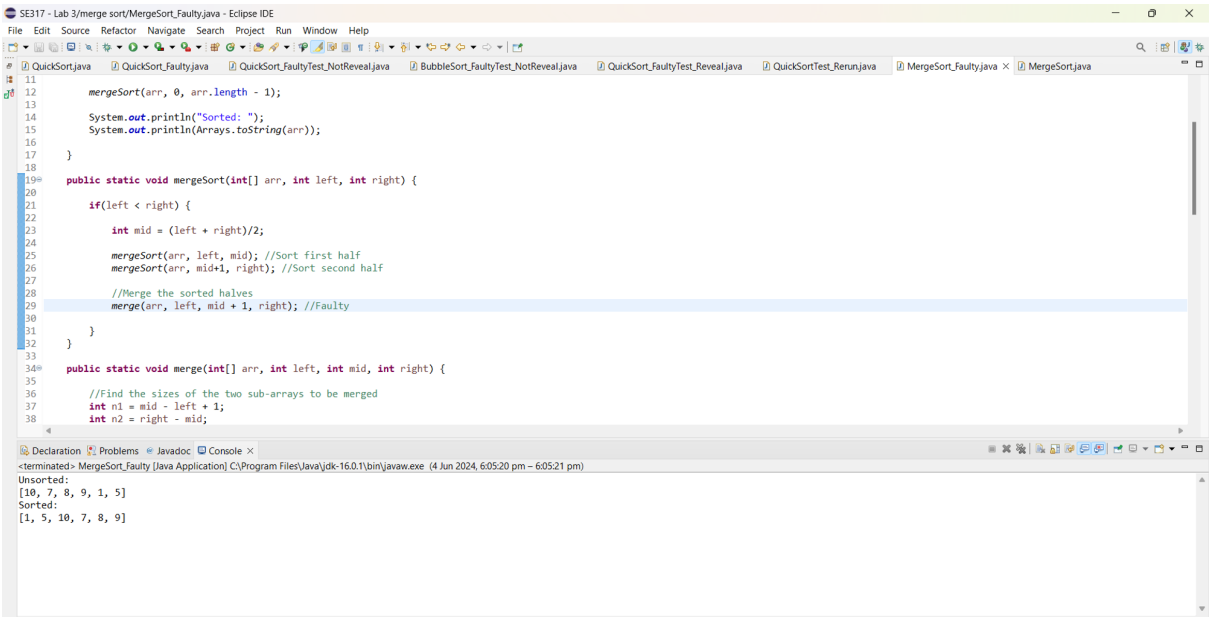
```
        System.out.println("Sorted: ");
        System.out.println(Arrays.toString(arr));

    }

    /**
     * Sorts an array using the merge sort algorithm
     *
     * @param arr the array to be sorted
     * @param left the starting index of the sub-array to b
e sorted
     * @param right the ending index of the sub-array to be
sorted
     */
    public static void mergeSort(int[] arr, int left, int r
ight) {

        if(left < right) {

            int mid = (left + right)/2;

            mergeSort(arr, left, mid); //Sort first half
            mergeSort(arr, mid+1, right); //Sort second hal
f

            //Merge the sorted halves
            merge(arr, left, mid + 1, right); //Faulty

        }
    }

    /**
     * Merges two sub-arrays of arr[].
     *
     * @param arr the array to be merged
     * @param left the starting index of the first sub-arra
y
     * @param mid the ending of index of the first sub-arra
```

```
y
     * @param right the ending index of the second sub-arra
y
     */
    public static void merge(int[] arr, int left, int mid,
int right) {

        //Find the sizes of the two sub-arrays to be merged
        int n1 = mid - left + 1;
        int n2 = right - mid;

        //Create temp arrays
        int[] left_arr = new int[n1];
        int[] right_arr = new int[n2];

        //Copy data to temp arrays
        for(int i = 0; i < n1; i++) {

            left_arr[i] = arr[left + i];
        }

        for(int j = 0; j < n2; j++) {

            right_arr[j] = arr[mid + j + 1];

        }

        //Merge the temp arrays
        int i = 0, j = 0; //Initial indexes of first and se
cond sub-array

        int k = left;

        while(i < n1 && j < n2) {

            if(left_arr[i] <= right_arr[j]) {

                arr[k] = left_arr[i];
```

```
                i++;
            }

            else {

                arr[k] = right_arr[j];
                j++;
            }

            k++;
        }

        //Copy remaining of left_arr[] if any
        while(i < n1) {

            arr[k] = left_arr[i];
            i++;
            k++;
        }

        //Copy remaining of right_arr[] if any
        while(j < n2) {

            arr[k] = right_arr[j];
            j++;
            k++;
        }
    }


}
```

ii.

Compile your code and take a screenshot.



b) Write two tests that do NOT reveal the fault.

i.

Each test case will include an input array of your choice and the expected output array.

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;

public class MergeSort_FaultyTest_NotReveal {

    @Test
    public void test1() {

        int[] x = {2, 4, 6, 8, 10, 12, 14, 16, 90};

        int[] expected = {2, 4, 6, 8, 10, 12, 14, 16, 90};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));
```

```
        MergeSort_Faulty.mergeSort(x, 0, x.length - 1);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);
    }


    @Test
    public void test2() {

        int[] x = {3, 4, 4, 1, 5, 9, 10};

        int[] expected = {1, 3, 4, 4, 5, 9, 10};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        MergeSort_Faulty.mergeSort(x, 0, x.length - 1);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);
    }


}
```

Explanation

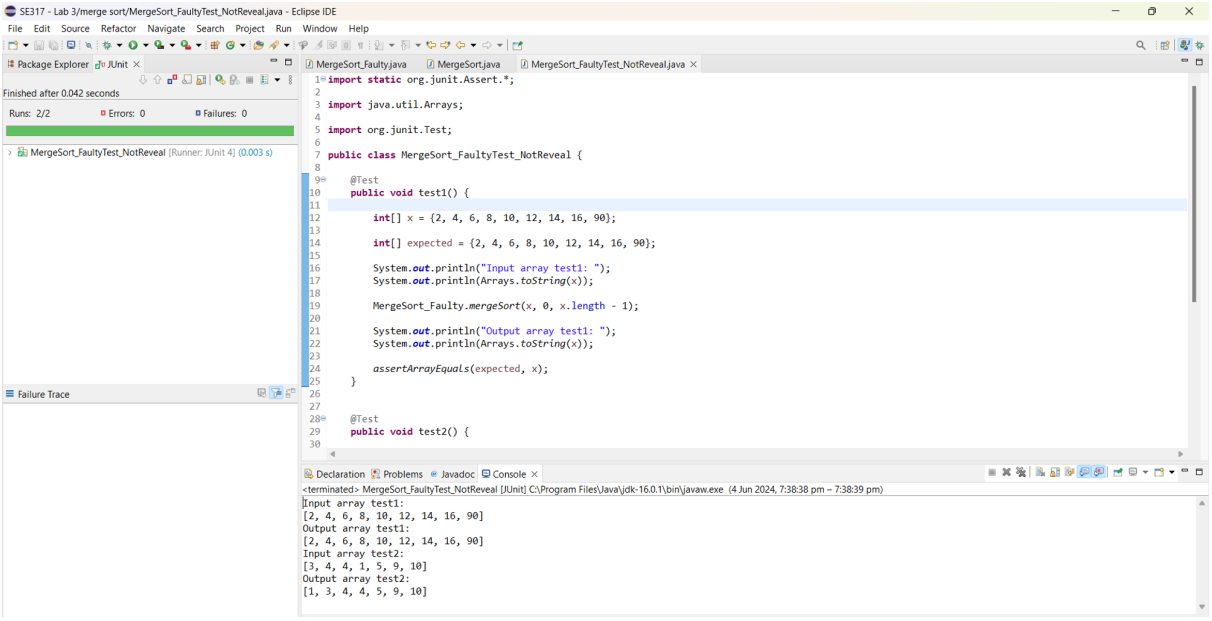
These test cases do not reveal the fault because:-

- test1: The input array is fully sorted. Even though the *merge* function is called with an incorrect parameter, ‘mid + 1’, it does not affect the already sorted array since there is no need for any element reordering.

- test2: The structure of the input array ensures that despite the incorrect merge call, the merging process between the sub-arrays does not disturb the overall order. The merging at these steps coincidentally happen to align correctly due to how the elements are positioned.

ii.

Run your code using the test array as input and take a screenshot of the input and actual output of each test case.



c) Write two tests that do reveal the fault.

i.

Each test case will include an input array of your choice and the expected output array

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;
```

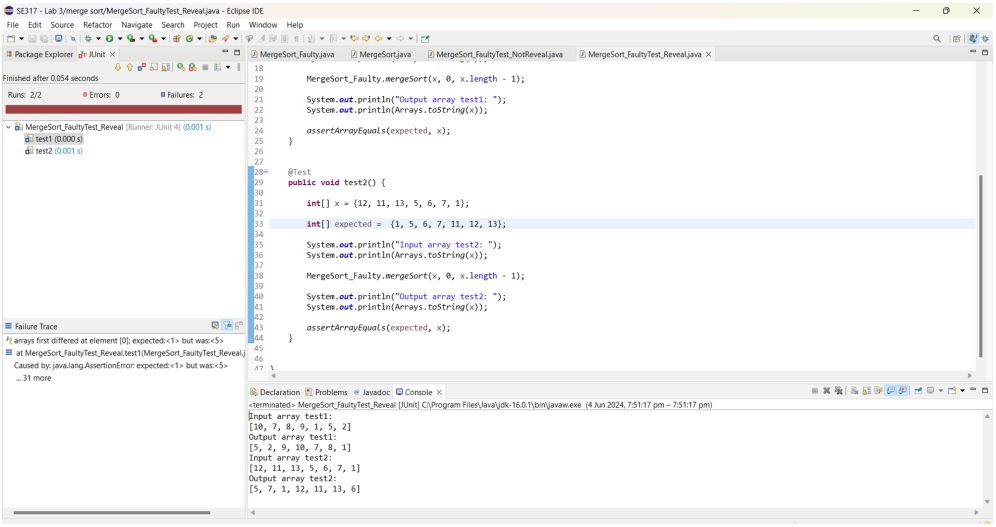
```
public class MergeSort_FaultyTest_Reveal {

    @Test
    public void test1() {

        int[] x = {10, 7, 8, 9, 1, 5, 2};

        int[] expected = {1, 2, 5, 7, 8, 9, 10};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));

        MergeSort_Faulty.mergeSort(x, 0, x.length - 1);

        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);
    }


    @Test
    public void test2() {

        int[] x = {12, 11, 13, 5, 6, 7, 1};

        int[] expected =  {1, 5, 6, 7, 11, 12, 13};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        MergeSort_Faulty.mergeSort(x, 0, x.length - 1);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);
    }
```

```
}
```

Explanation

- Both test1 and test2 have elements that is structured in a way that it would be affected by the incorrect parameter passed to the *merge* method.

ii.
Run your code using the test array as input and take a screenshot of the input and actual output of each case.



d) Identify and remove the fault.

i.
Explain your correction in the comments of your source code.

```
import java.util.Arrays;

public class MergeSort {

    public static void main(String args[]) {
```

```
     int[] arr = {10, 7, 8, 9, 1, 5};

        System.out.println("Unsorted: ");
        System.out.println(Arrays.toString(arr));

        mergeSort(arr, 0, arr.length - 1);

        System.out.println("Sorted: ");
        System.out.println(Arrays.toString(arr));

    }

    /**
     * Sorts an array using the merge sort algorithm
     *
     * @param arr the array to be sorted
     * @param left the starting index of the sub-array to b
e sorted
     * @param right the ending index of the sub-array to be
sorted
     */
    public static void mergeSort(int[] arr, int left, int r
ight) {

        if(left < right) {

            int mid = (left + right)/2;

            mergeSort(arr, left, mid); //Sort first half
            mergeSort(arr, mid+1, right); //Sort second hal
f

            //Merge the sorted halves
            merge(arr, left, mid, right); //Corrected to mi
d

        }
    }
```

```
    /**
     * Merges two sub-arrays of arr[].
     *
     * @param arr the array to be merged
     * @param left the starting index of the first sub-arra
y
     * @param mid the ending of index of the first sub-arra
y
     * @param right the ending index of the second sub-arra
y
     */
    public static void merge(int[] arr, int left, int mid,
int right) {

        //Find the sizes of the two sub-arrays to be merged
        int n1 = mid - left + 1;
        int n2 = right - mid;

        //Create temp arrays
        int[] left_arr = new int[n1];
        int[] right_arr = new int[n2];

        //Copy data to temp arrays
        for(int i = 0; i < n1; i++) {

            left_arr[i] = arr[left + i];
        }

        for(int j = 0; j < n2; j++) {

            right_arr[j] = arr[mid + j + 1];

        }

        //Merge the temp arrays
        int i = 0, j = 0; //Initial indexes of first and se
cond sub-array
```

```
        int k = left;

        while(i < n1 && j < n2) {

            if(left_arr[i] <= right_arr[j]) {

                arr[k] = left_arr[i];
                i++;
            }

            else {

                arr[k] = right_arr[j];
                j++;
            }

            k++;
        }

        //Copy remaining of left_arr[] if any
        while(i < n1) {

            arr[k] = left_arr[i];
            i++;
            k++;
        }

        //Copy remaining of right_arr[] if any
        while(j < n2) {

            arr[k] = right_arr[j];
            j++;
            k++;
        }
    }
```

```
}
```

Explanation of the Fault & Fix:

The fault in the merge sort algorithm was due to an incorrect parameter passed to the *merge* method within the *mergeSort* function. Specifically, the parameter ‘mid + 1’ was used instead of ‘mid’, causing the merging process to combine non-adjacent elements and disrupt the sorting order.

To address the fault, I corrected the parameter passed to the *merge* method, ensuring that it correctly defines the sub-arrays to be merged. By using ‘mid’ instead of ‘mid + 1’, the algorithm now properly merges adjacent elements within the sub-arrays.

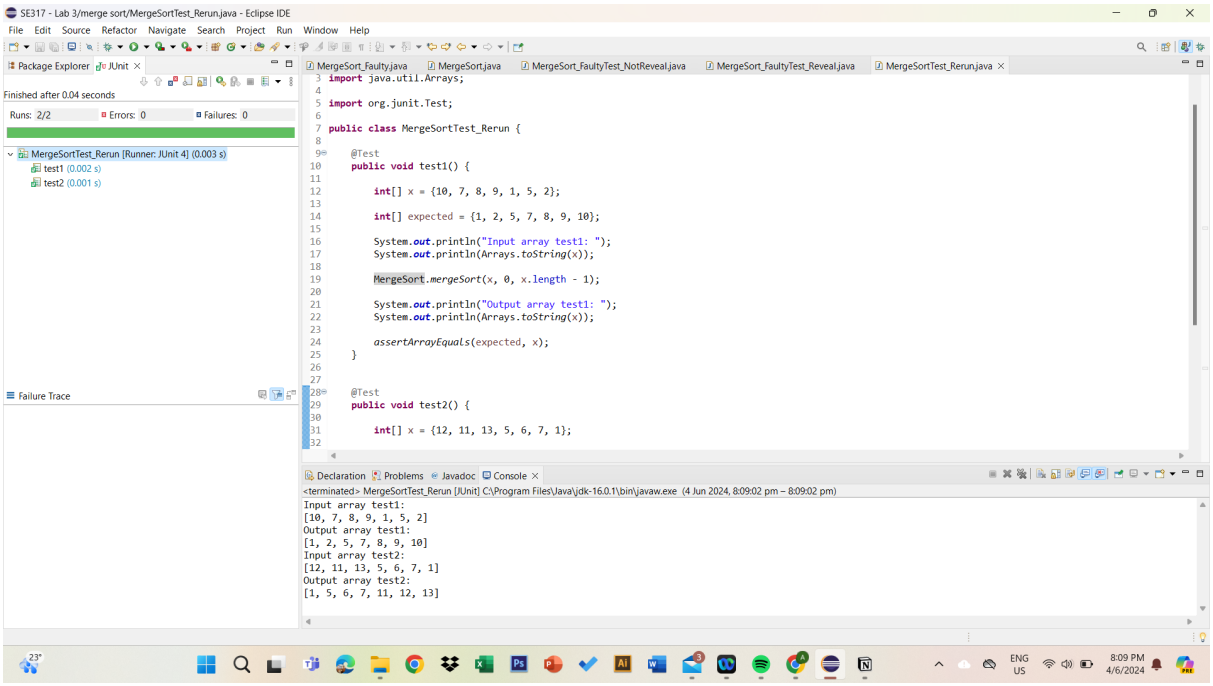
ii.
Run the test cases of part c again after fixing the fault.

```
import static org.junit.Assert.*;

import java.util.Arrays;

import org.junit.Test;

public class MergeSortTest_Rerun {

    @Test
    public void test1() {

        int[] x = {10, 7, 8, 9, 1, 5, 2};

        int[] expected = {1, 2, 5, 7, 8, 9, 10};

        System.out.println("Input array test1: ");
        System.out.println(Arrays.toString(x));

        MergeSort.mergeSort(x, 0, x.length - 1);
```

```
        System.out.println("Output array test1: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);
    }


    @Test
    public void test2() {

        int[] x = {12, 11, 13, 5, 6, 7, 1};

        int[] expected =  {1, 5, 6, 7, 11, 12, 13};

        System.out.println("Input array test2: ");
        System.out.println(Arrays.toString(x));

        MergeSort.mergeSort(x, 0, x.length - 1);

        System.out.println("Output array test2: ");
        System.out.println(Arrays.toString(x));

        assertArrayEquals(expected, x);
    }


}
```

iii.

Take a screenshot of the input and actual output of each case.


SE317 - Lab 3/merge sort/MergeSortTest_Rerun.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer JUnit
Finished after 0.04 seconds
Runs: 2/2 Errors: 0 Failures: 0
MergeSortTest_Rerun [Runner: JUnit 4] (0.003 s)
test1 (0.002 s)
test2 (0.001 s)
Failure Trace
MergeSort_Faulty.java
MergeSort.java
MergeSort_FaultyTest_NotReveal.java
MergeSort_FaultyTest_Reveal.java
MergeSortTest_Rerun.java
import java.util.Arrays;
import org.junit.Test;
public class MergeSortTest_Rerun {
@Test
public void test1() {
int[] x = {10, 7, 8, 9, 1, 5, 2};
int[] expected = {1, 2, 5, 7, 8, 9, 10};
System.out.println("Input array test1: ");
System.out.println(Arrays.toString(x));
MergeSort.mergeSort(x, 0, x.length - 1);
System.out.println("Output array test1: ");
System.out.println(Arrays.toString(x));
assertArrayEquals(expected, x);
}
@Test
public void test2() {
int[] x = {12, 11, 13, 5, 6, 7, 1};
Declaration Problems Javadoc Console
<terminated> MergeSortTest_Rerun [JUnit] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (4 Jun 2024, 8:09:02 pm – 8:09:02 pm)
Input array test1:
[10, 7, 8, 9, 1, 5, 2]
Output array test1:
[1, 2, 5, 7, 8, 9, 10]
Input array test2:
[12, 11, 13, 5, 6, 7, 1]
Output array test2:
[1, 5, 6, 7, 11, 12, 13]
ENG US
8:09 PM 4/6/2024