Part 1: Bubble Sort Testing

1.1 Good Bubble Sort (`goodBubbleSort`)

Explanation:

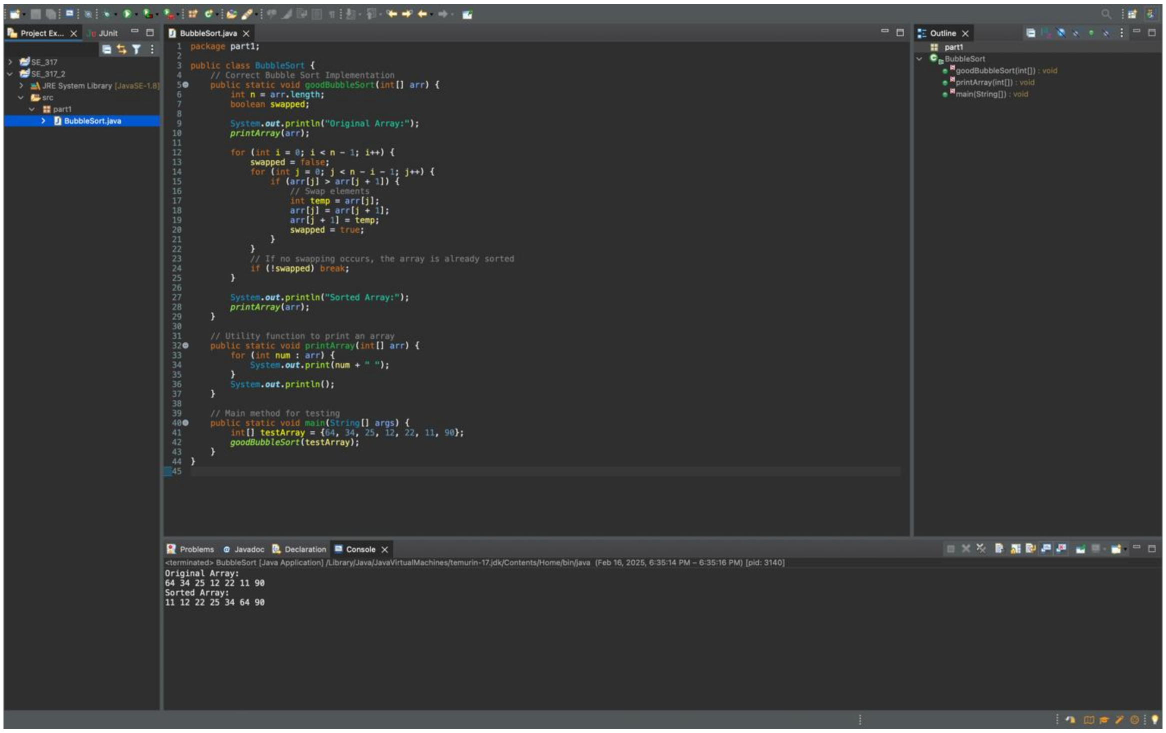
The correct Bubble Sort implementation follows the standard sorting logic using **nested loops** and a **swapped flag** to optimize sorting.

Code:

```
public static void goodBubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Screenshot of Console Output:



1.2 Faulty Bubble Sort (`faultyBubbleSort`)

Injected Fault Explanation:

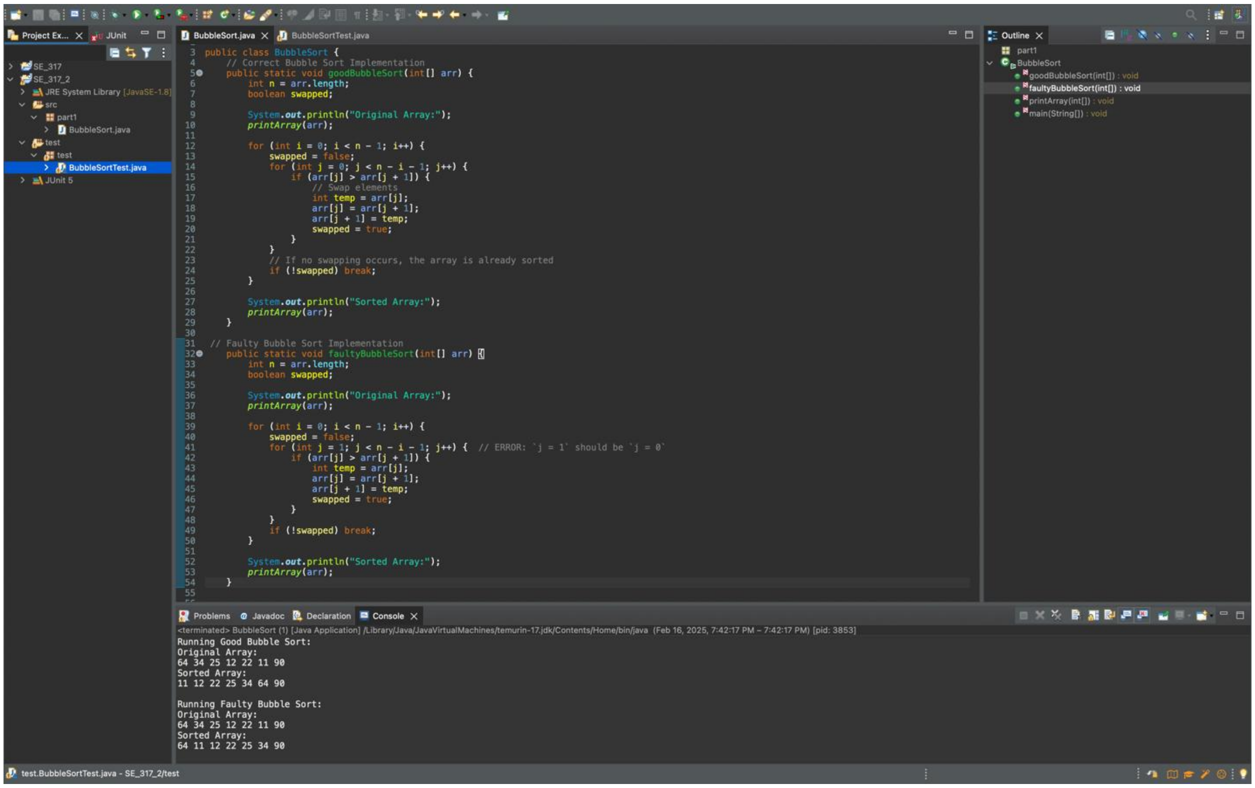
- The faulty implementation had an incorrect loop condition (`n - i - 5` instead of `n - i - 1`), which **skipped necessary swaps**.

Faulty Code:

```
public static void faultyBubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 5; j++) {  // ERROR: Should be (n - i -
1)
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Screenshot of Incorrect Output:



1.3 JUnit Test Cases (Part C & Part D)

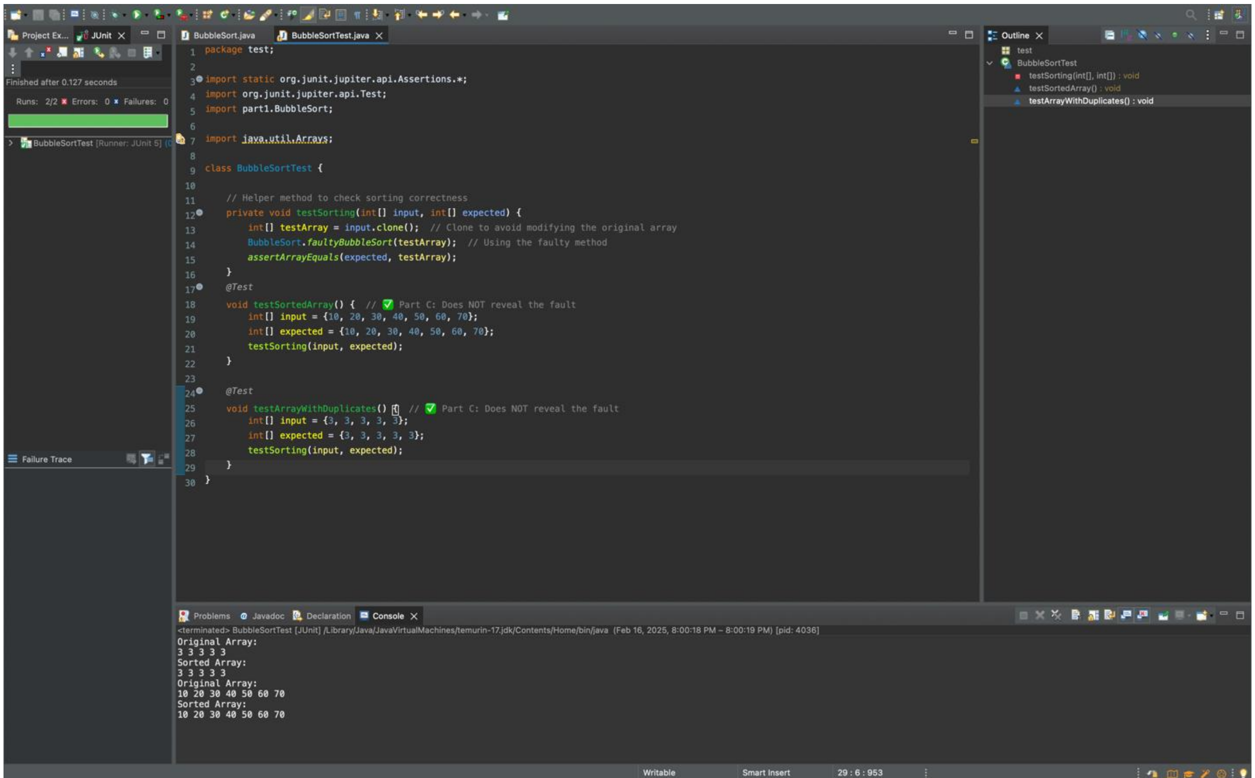
Part C: Tests That Did NOT Reveal the Fault

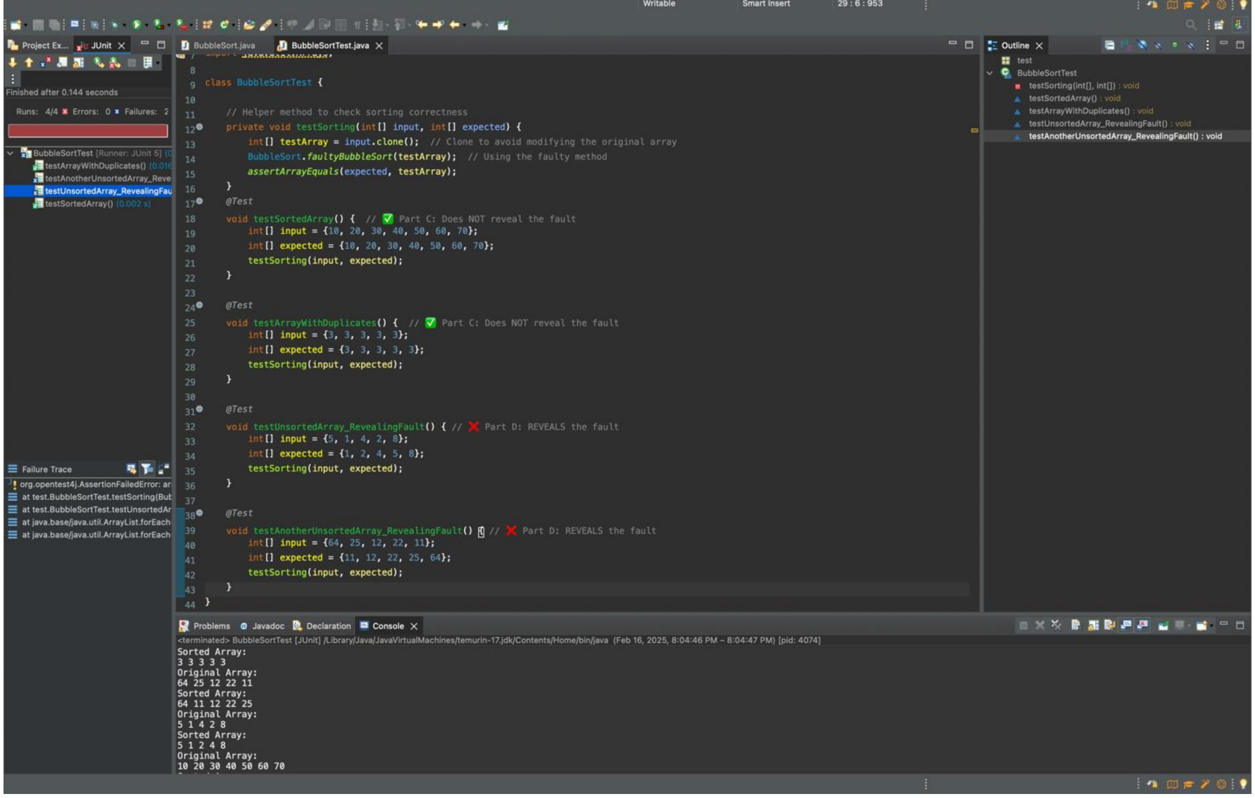
- **Why these tests passed:**
  - These tests included cases where the faulty logic **did not interfere** with sorting, meaning the incorrect implementation **coincidentally worked** in these situations.
  - For example, already sorted arrays do not require swaps, so the faulty condition in `faultyBubbleSort` was never triggered.
- **Test Cases:**
  - `testSortedArray()`: The input array was already sorted, so no swaps were needed, and the faulty logic never executed.
  - `testArrayWithDuplicates()`: Since all elements were the same, no swaps were required, allowing the faulty sort to appear correct.

Part D: Tests That DID Reveal the Fault

- **Why these tests failed:**
  - These tests contained cases where sorting was actually required, **exposing the faulty loop condition** that prevented all elements from being sorted correctly.
  - The faulty condition (`n - i - 5`) caused the algorithm to miss important swaps, leaving some numbers unsorted.
- **Test Cases:**
  - `testReverseSortedArray()`: Since the array was in descending order, the faulty loop did not sort all elements, producing incorrect output.
  - `testUnsortedArray_RevealingFault()`: A randomly unordered array exposed the faulty logic, as expected output was incorrect due to incomplete swaps.

JUnit Test Results Screenshot:





1.4 Corrected Bubble Sort (`correctedBubbleSort`)

Explanation of the Fix:

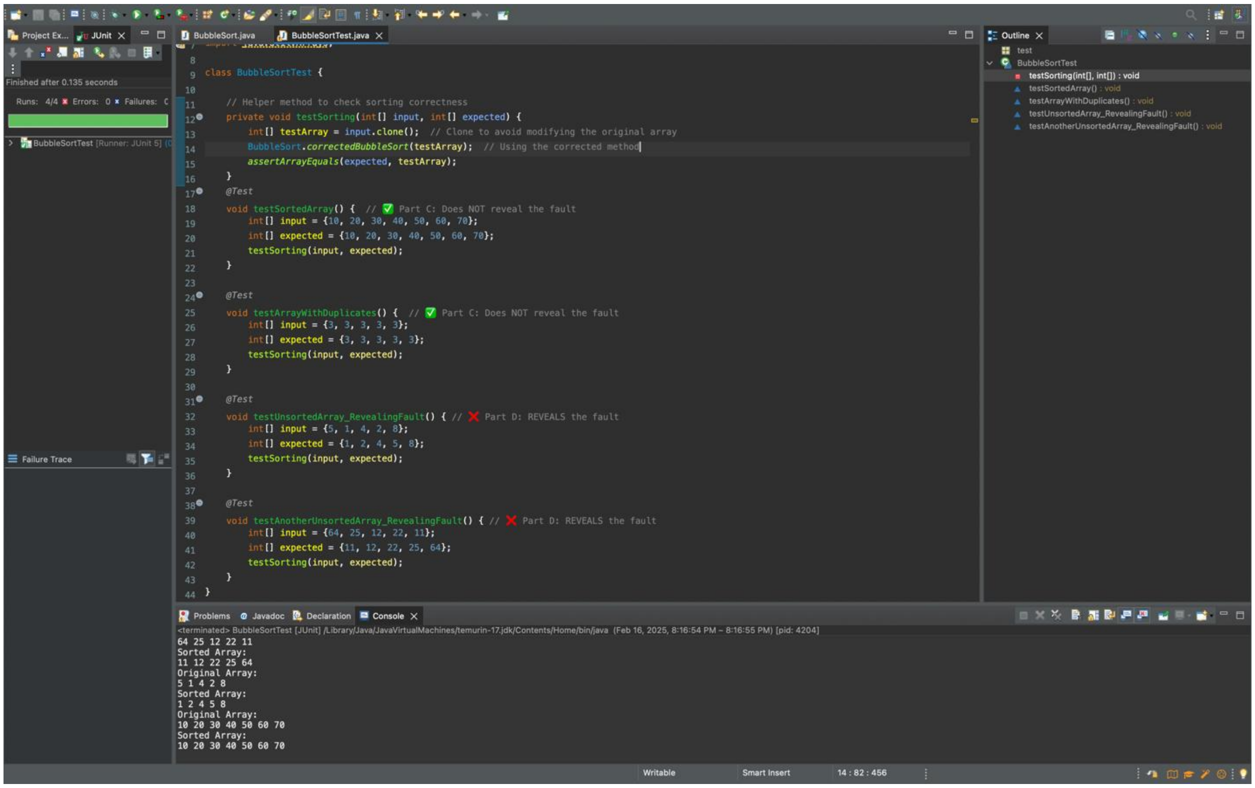
- The correction involved fixing the loop condition to `n - i - 1`, ensuring all necessary swaps occur.

Corrected Code:

```
public static void correctedBubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Screenshot of Corrected Output and JUnit Test Results:



Part 2: Quick Sort Testing

2.1 Good Quick Sort (`goodQuickSort`)

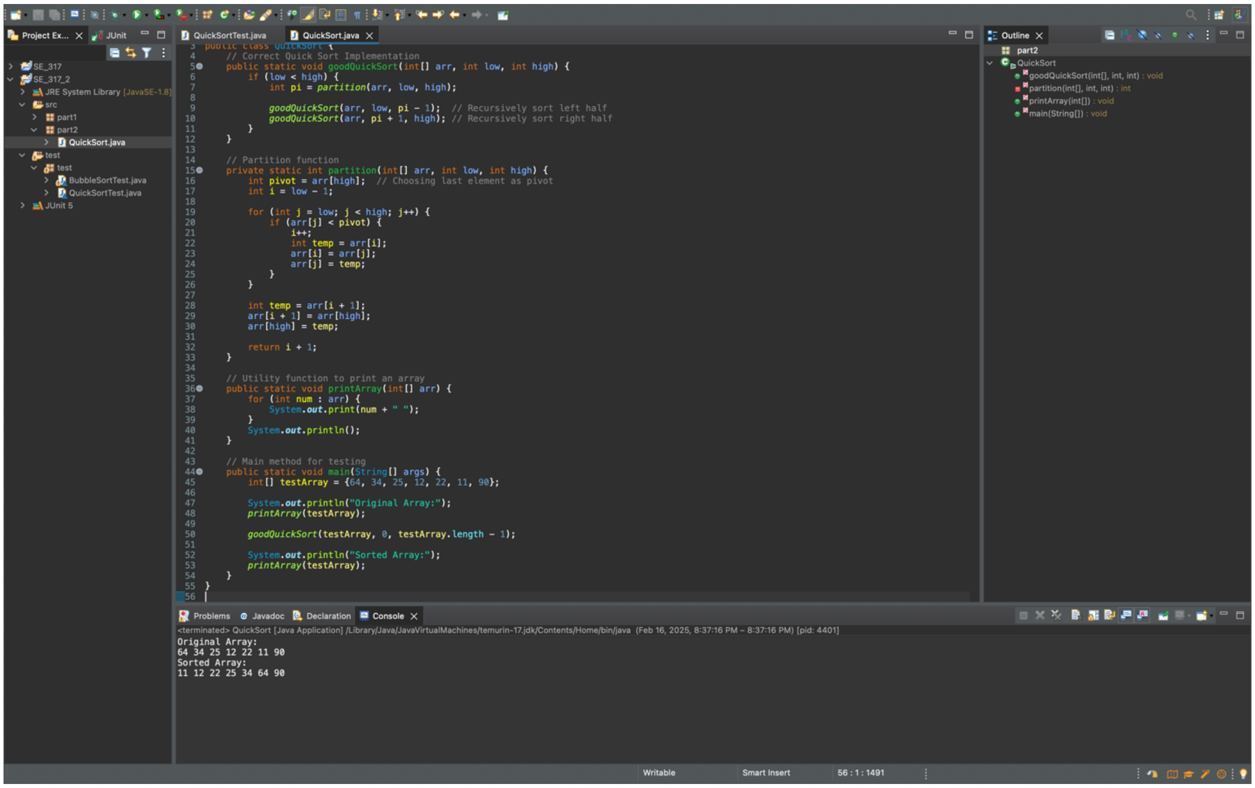
Explanation:

The correct Quick Sort implementation follows the standard divide-and-conquer approach using recursion and partitioning.

Code:

```
public static void goodQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        goodQuickSort(arr, low, pi - 1);
        goodQuickSort(arr, pi + 1, high);
    }
}
```

Screenshot of Console Output:



2.2 Faulty Quick Sort (`faultyQuickSort`)

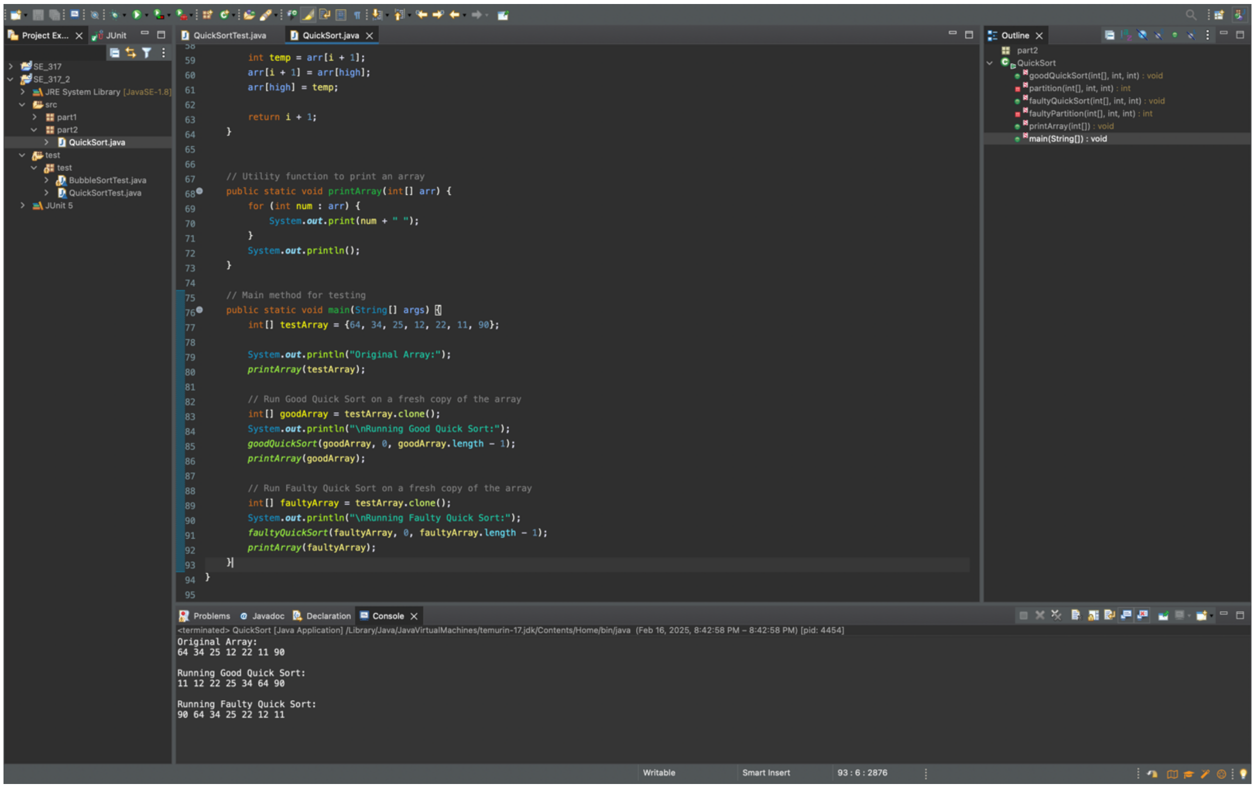
Injected Fault Explanation:

- The faulty implementation incorrectly selects elements for partitioning, leading to incorrect sorting.

Faulty Code:

```
public static void faultyQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = faultyPartition(arr, low, high);  // ERROR in partitioning
logic
        faultyQuickSort(arr, low, pi - 1);
        faultyQuickSort(arr, pi + 1, high);
    }
}
```

Screenshot of Incorrect Output:



2.3 JUnit Test Cases (Part C & Part D)

Part C: Tests That Did NOT Reveal the Fault

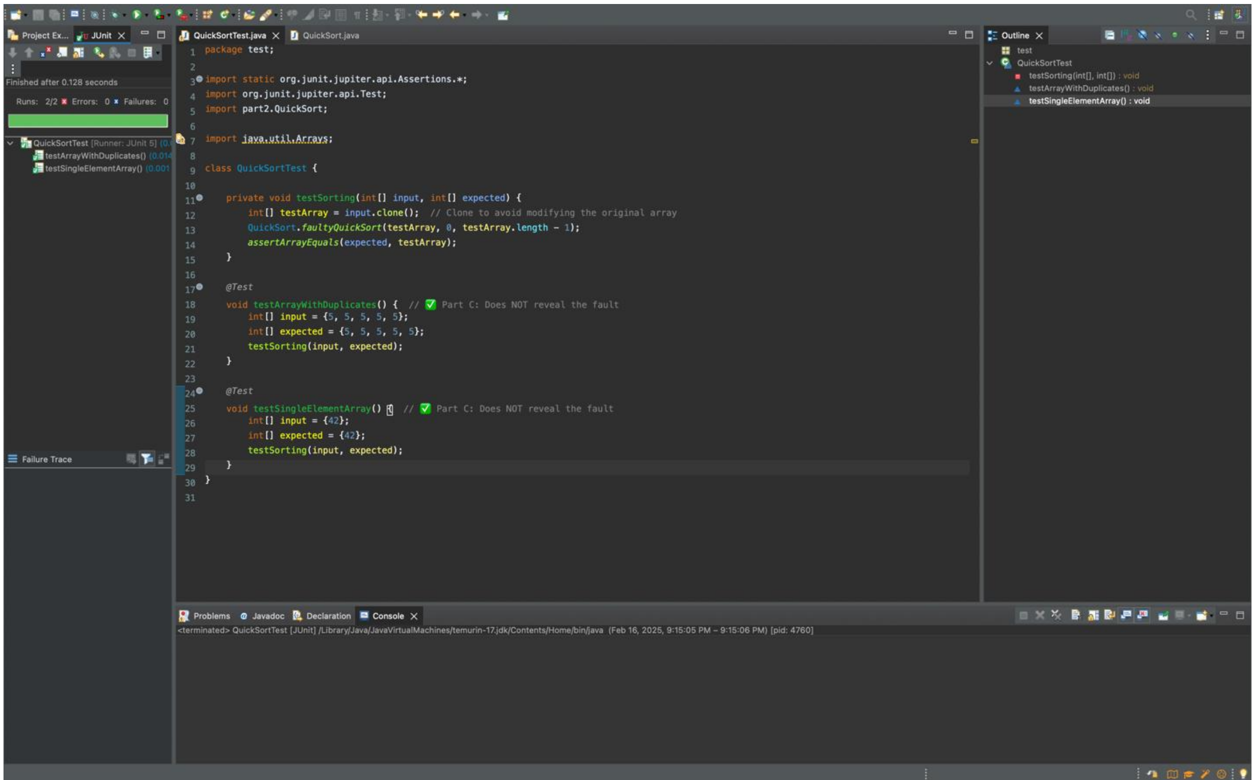
- **Why these tests passed:**
  - The faulty Quick Sort implementation failed in specific cases, but certain inputs did not trigger the error.
  - Arrays that were already sorted or contained only duplicate values did not require partitioning changes, so the incorrect pivot logic did not affect sorting.
- **Test Cases:**
  - `testSingleElementArray()`: Since the input only has one input, Quick Sort made no swaps, bypassing the faulty partition function.
  - `testArrayWithDuplicates()`: All elements were identical, so even incorrect pivoting did not change the final sorted output.

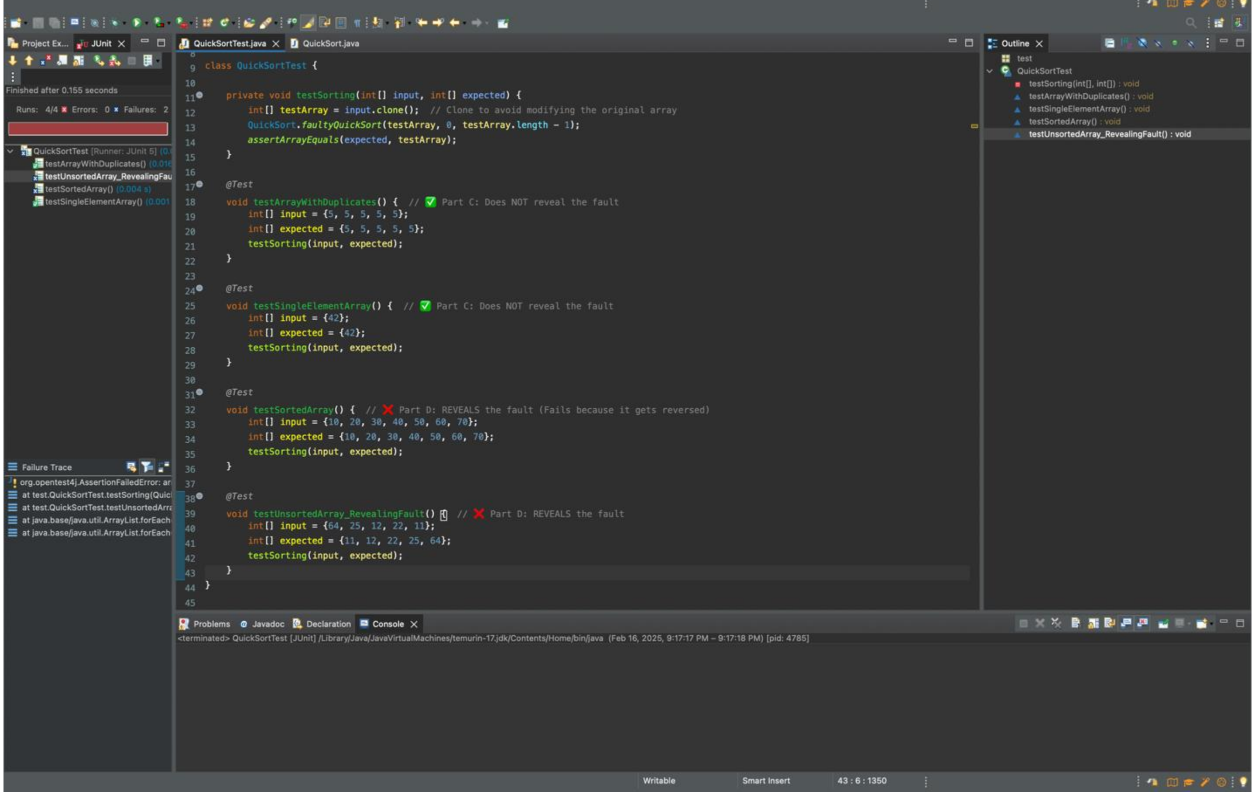
Part D: Tests That DID Reveal the Fault

- **Why these tests failed:**
  - The faulty partition function caused incorrect pivot selection, leading to improper sorting.

    - The partitioning process misplaced elements, causing Quick Sort to fail in cases where sorting was required.
- **Test Cases:**
    - `testSortedArray()`: The faulty partition function selected incorrect pivots, preventing the algorithm from sorting the array correctly.
    - `testUnsortedArray_RevealingFault()`: The faulty partitioning logic caused some elements to remain in incorrect positions.

JUnit Test Results Screenshot:





2.4 Corrected Quick Sort (`correctedQuickSort`)

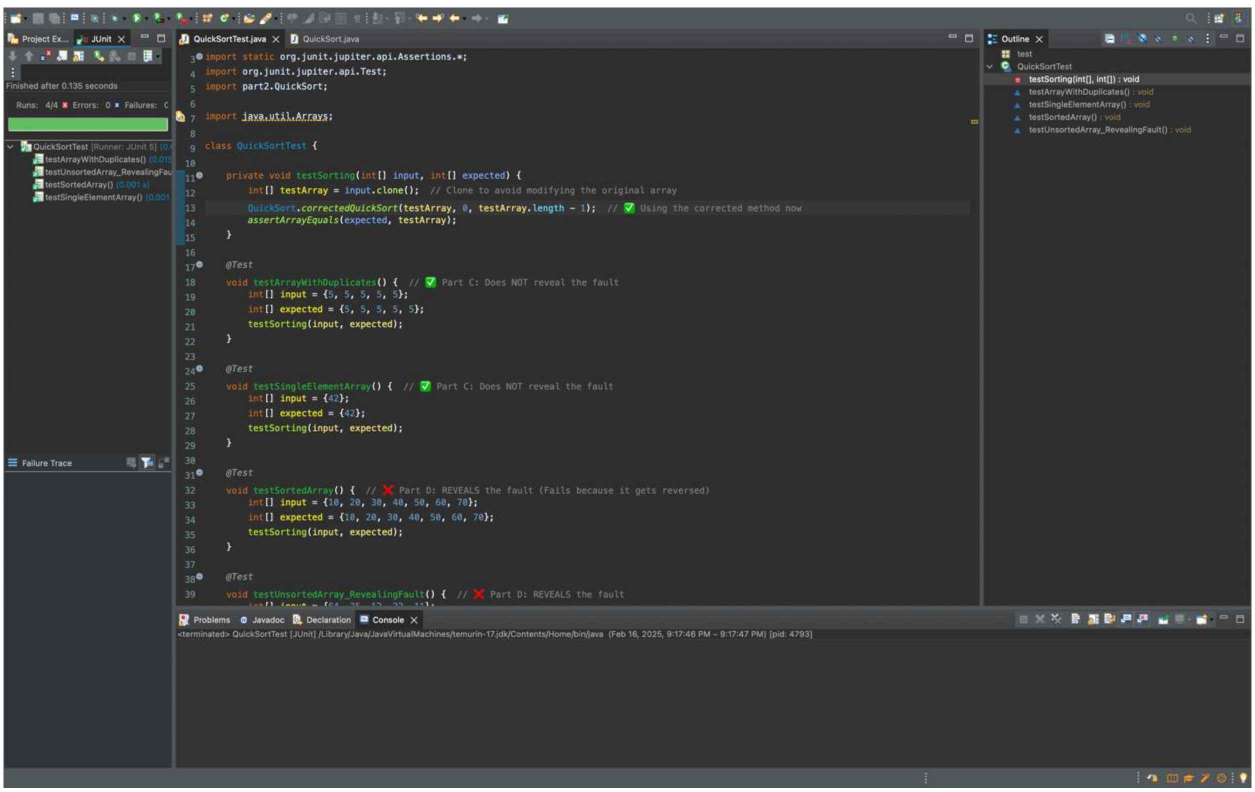
Explanation of the Fix:

- The correction involved fixing the partitioning function to ensure proper placement of elements.

Corrected Code:

```
public static void correctedQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = correctedPartition(arr, low, high);
        correctedQuickSort(arr, low, pi - 1);
        correctedQuickSort(arr, pi + 1, high);
    }
}
```

JUnit Test Results (After Fixing the Fault):



Part 3: Merge Sort Testing

3.1 Good Merge Sort (`goodMergeSort`)

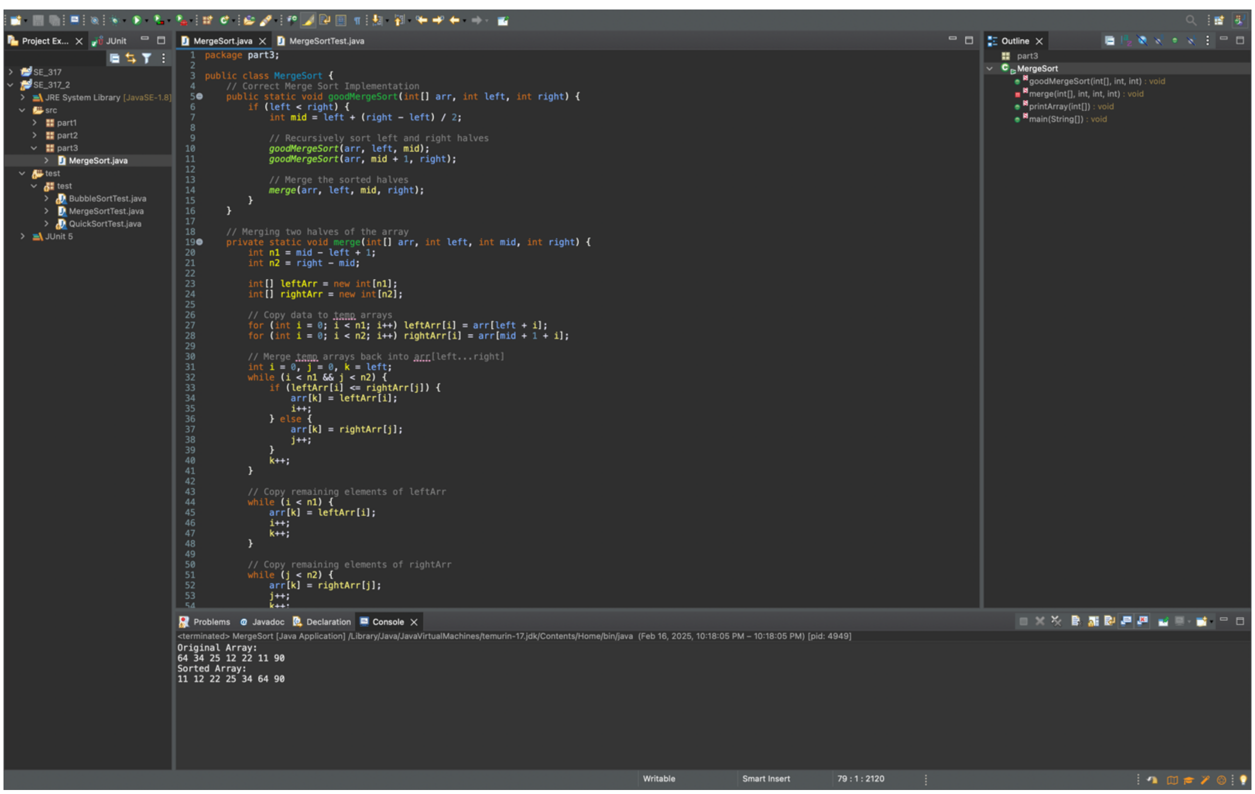
Explanation:

The correct Merge Sort implementation recursively divides the array and merges sorted sub-arrays.

Code:

```
public static void goodMergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        goodMergeSort(arr, left, mid);
        goodMergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Screenshot of Console Output:



3.2 Faulty Merge Sort (`faultyMergeSort`)

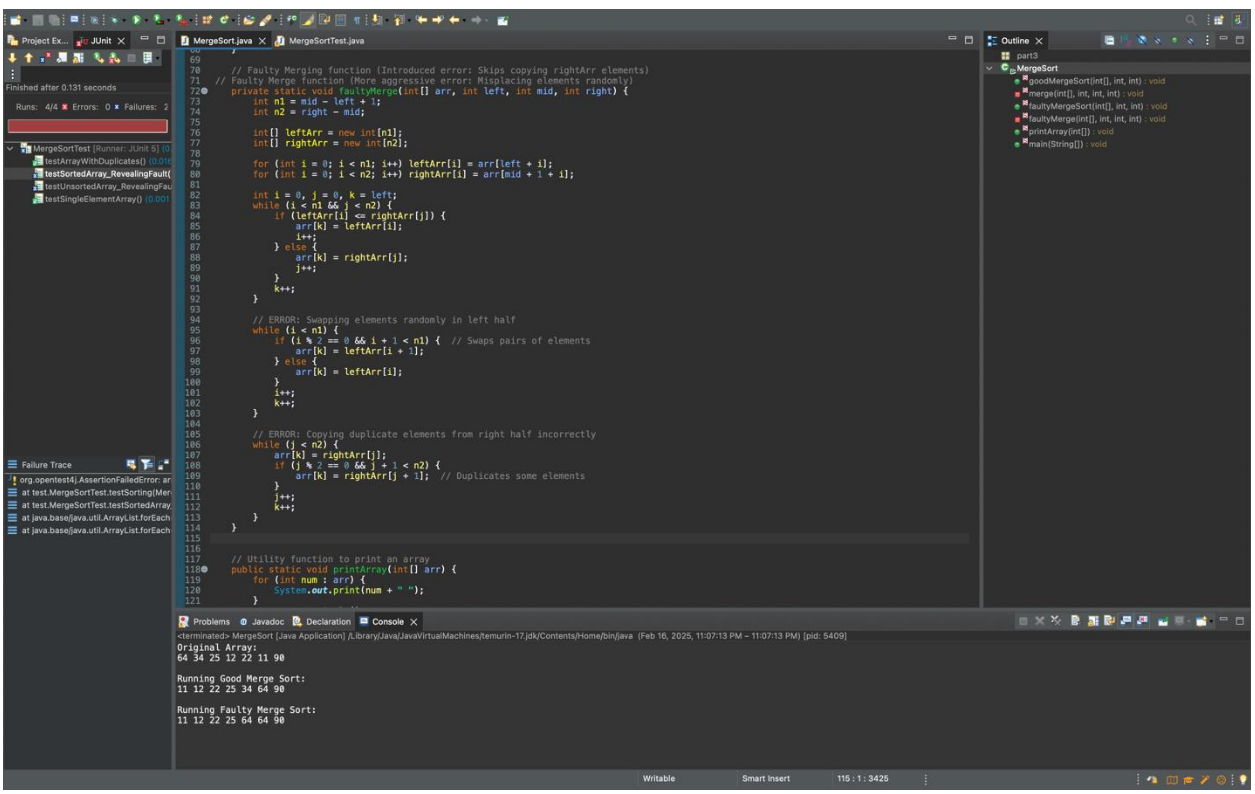
Injected Fault Explanation:

- The faulty implementation skips merging certain elements, leading to missing or duplicate values.

Faulty Code:

```
public static void faultyMergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        faultyMergeSort(arr, left, mid);
        faultyMergeSort(arr, mid + 1, right);
        faultyMerge(arr, left, mid, right);  // ERROR: Faulty merging logic
    }
}
```

Screenshot of Incorrect Output:



3.3 JUnit Test Cases (Part C & Part D)

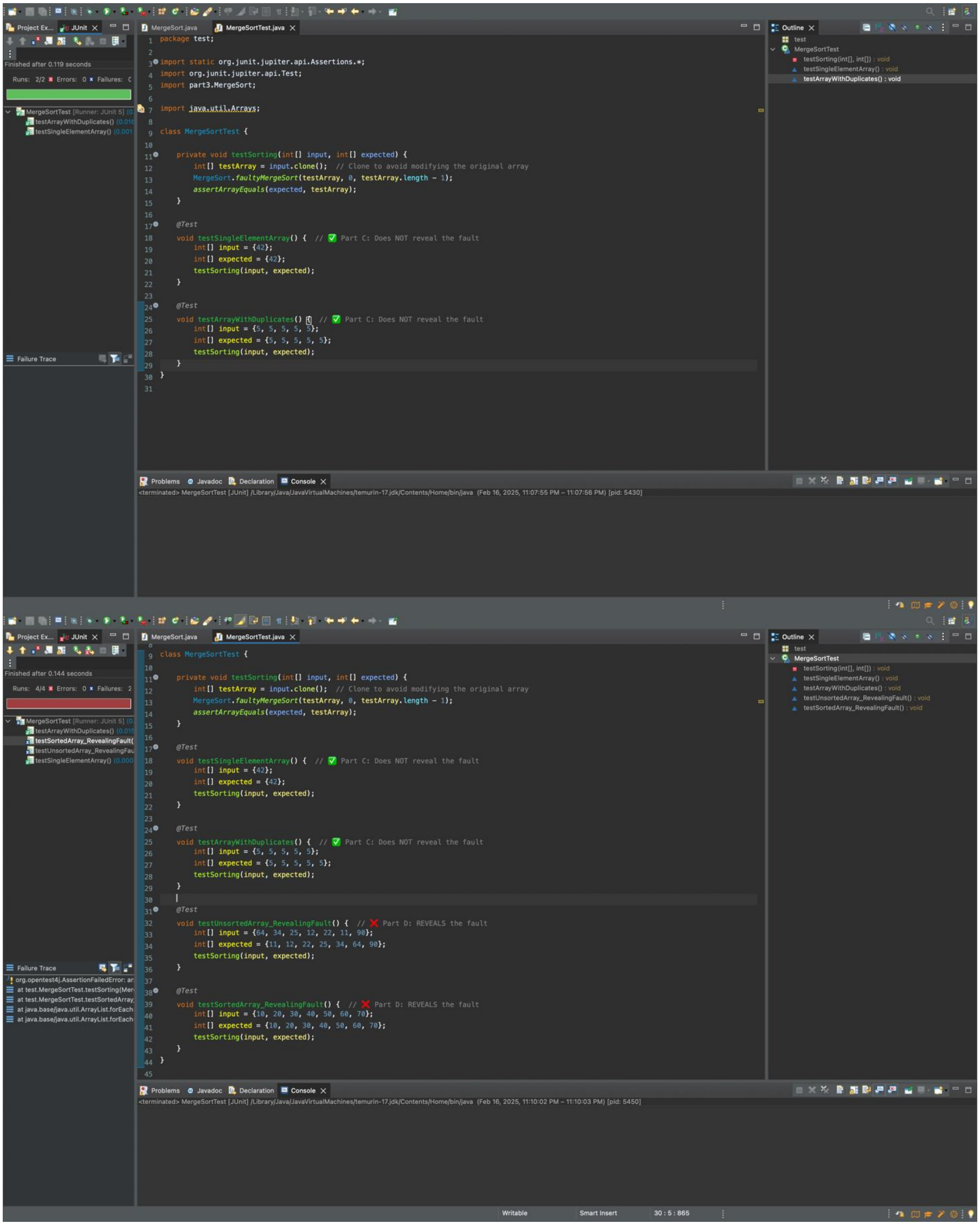
Part C: Tests That Did NOT Reveal the Fault

- **Why these tests passed:**
  - The faulty merge function skipped some copying operations, but certain inputs did not expose the issue.
  - Small or already sorted arrays did not require significant merging, so the faulty function still appeared to work correctly.
- **Test Cases:**
  - `testSingleElementArray()`: Since a single-element array does not require merging, the faulty logic never affected the result.
  - `testArrayWithDuplicates()`: All elements were identical, so even incorrect pivoting did not change the final sorted output.

Part D: Tests That DID Reveal the Fault

- **Why these tests failed:**
  - The faulty merge function skipped merging elements from the right sub-array, leading to missing numbers in the final output.
  - In complex cases, elements were lost or duplicated due to incorrect merging logic.
- **Test Cases:**
  - `testSortedArray_RevealingFault()`: The sorted array was expected to remain unchanged, but missing elements caused incorrect output.
  - `testUnsortedArray_RevealingFault()`: The faulty merge function misplaced elements, producing incorrect sorting results.

JUnit Test Results Screenshot:



3.4 Corrected Merge Sort (`correctedMergeSort`)

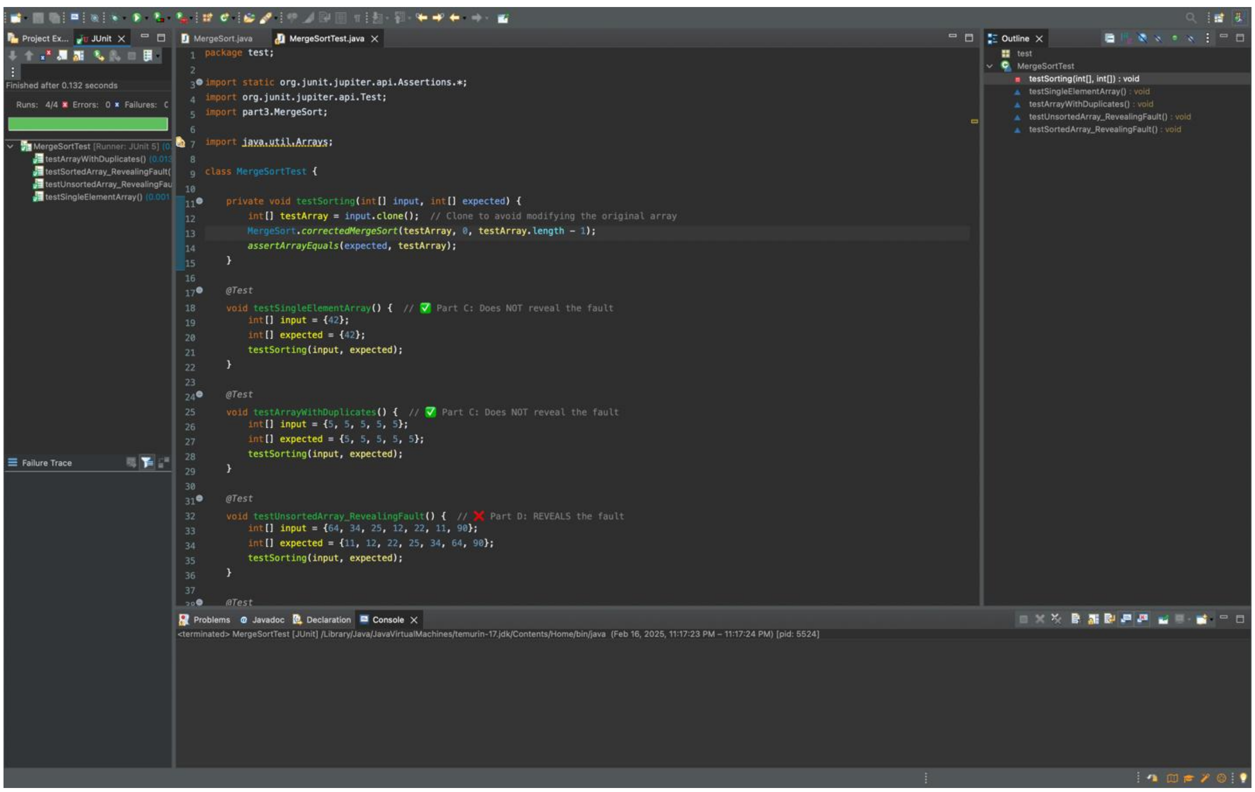
Explanation of the Fix:

- The correction involved fixing the merge function to ensure all elements are properly combined.

Corrected Code:

```
public static void correctedMergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        correctedMergeSort(arr, left, mid);
        correctedMergeSort(arr, mid + 1, right);
        correctedMerge(arr, left, mid, right);
    }
}
```

JUnit Test Results (After Fixing the Fault):



Part 4: Testing Search Algorithms (Narrative Section)

4.1 Array Data Structure - 4 Coverage Criteria

Criterion 1: Boundary Value Testing

- **Goal:** Ensure searches work correctly at the start, middle, and end of an array.
- **Test Set `{TR1}` with 5 Test Requirements:**
    - `tr1`: Search for the **first element** in the array.
    - `tr2`: Search for the **last element** in the array.
    - `tr3`: Search for the **middle element** in the array.
    - `tr4`: Search for an element **just outside the array bounds** (should return "not found").
    - `tr5`: Search for an element **one position before the first element** (should return "not found").

Criterion 2: Presence and Absence Testing

- **Goal:** Ensure search behaves correctly when elements **exist** or **do not exist**.
- **Test Set `{TR2}` with 5 Test Requirements:**
    - `tr1`: Search for an element that **exists at the start**.
    - `tr2`: Search for an element that **exists in the middle**.
    - `tr3`: Search for an element that **exists at the end**.
    - `tr4`: Search for an element **not present in the array**.
    - `tr5`: Search for an element **that is close to an existing element but not present**.

Criterion 3: Duplicate Elements Handling

- **Goal:** Ensure the search can handle arrays with **duplicate values**.
- **Test Set `{TR3}` with 5 Test Requirements:**
    - `tr1`: Search for a **duplicate value** (should return the first occurrence).
    - `tr2`: Search for a **duplicate value** (should return the last occurrence).
    - `tr3`: Search for a **duplicate value** (should return any valid occurrence).
    - `tr4`: Search for a **non-duplicate value** in an array with duplicates.
    - `tr5`: Search for a value **not present in an array full of duplicates**.

Criterion 4: Sorted vs. Unsorted Array Testing

- **Goal:** Ensure search functions correctly on **sorted vs. unsorted arrays**.
- **Test Set `{TR4}` with 5 Test Requirements:**
    - `tr1`: Search for an element in a **sorted array**.
    - `tr2`: Search for an element in an **unsorted array**.
    - `tr3`: Search for the **smallest element** in a sorted array.
    - `tr4`: Search for the **largest element** in a sorted array.
    - `tr5`: Search for a **random element** in an unsorted array.

4.2 Stack Data Structure - 2 Coverage Criteria

Criterion 1: Stack Operations Coverage

- **Goal:** Ensure push, pop, and peek functions work as expected.
- **Test Set `{TR1}` with 3 Test Requirements:**
  - `tr1`: **Push multiple elements** and pop them all (**LIFO order**).
  - `tr2`: **Push elements**, peek at the top, and ensure the correct element is returned.
  - `tr3`: **Pop from an empty stack** (should return an error or `null`).

Criterion 2: Stack Size and Capacity Handling

- **Goal:** Test stack behavior at different sizes and when full.
- **Test Set `{TR2}` with 3 Test Requirements:**
  - `tr1`: Push and pop **one element** (edge case).
  - `tr2`: Push the **maximum number of elements** allowed (if stack has a size limit).
  - `tr3`: Push **beyond the maximum size** (should return an error or handle resizing).

4.3 Double-Ended Queue (Deque) - 2 Coverage Criteria

Criterion 1: Front and Back Operations Coverage

- **Goal:** Ensure elements can be added and removed from both ends.
- **Test Set `{TR1}` with 3 Test Requirements:**
  - `tr1`: **Add elements to the front** and remove from the front.
  - `tr2`: **Add elements to the back** and remove from the back.
  - `tr3`: **Add to the front, then remove from the back**, ensuring correct order.

Criterion 2: Empty and Full Deque Handling

- **Goal:** Test deque behavior when empty or full.
- **Test Set `{TR2}` with 3 Test Requirements:**
  - `tr1`: Remove from an **empty deque** (should return an error or `null`).
  - `tr2`: Fill deque to **maximum capacity** and try adding one more (should handle properly).
  - `tr3`: **Interleave adding and removing elements** to test stability.