Lab 3 Quiz
ENGG 4450
November 22,2024
Bhavjot Gill

Link to git:https://github.com/Gill003/java-algorithms-implementation-master-F230Bhavjot

BubbleSort Corrected Code:

```
package com.jwetherell.algorithms.sorts;

public class BubbleSort<T extends Comparable<T>> {

    private BubbleSort() { }

    public static <T extends Comparable<T>> T[] sort(T[] unsorted) {
        boolean swapped = true;
        int length = unsorted.length;
        while (swapped) {
            swapped = false;
            for (int i = 1; i < length; i++) {
                if (unsorted[i].compareTo(unsorted[i - 1]) < 0) { // Fixed condition
for ascending order
                    swap(i, i - 1, unsorted);
                    swapped = true;
                }
            }
            length--;
        }
        return unsorted;
    }

    private static <T extends Comparable<T>> void swap(int index1, int index2, T[]
unsorted) {
        T value = unsorted[index1];
        unsorted[index1] = unsorted[index2];
        unsorted[index2] = value;
    }
}
```

BubbleSort Test Code:

```
package com.jwetherell.algorithms.sorts.test;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import com.jwetherell.algorithms.sorts.BubbleSort;

public class BubbleSortTest {

    @Test
    public void testSortIntegers() {
        Integer[] unsorted = {7, 1, 9, 6, 3};
        Integer[] expected = {1, 3, 6, 7, 9};
        Integer[] result = BubbleSort.sort(unsorted);
        assertArrayEquals(expected, result);
```

```
}

@Test
public void testSortStrings() {
    String[] unsorted = {"grape", "orange", "kiwi", "pear"};
    String[] expected = {"grape", "kiwi", "orange", "pear"};
    String[] result = BubbleSort.sort(unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortEmptyArray() {
    Integer[] unsorted = {};
    Integer[] expected = {};
    Integer[] result = BubbleSort.sort(unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortSingleElement() {
    Integer[] unsorted = {17};
    Integer[] expected = {17};
    Integer[] result = BubbleSort.sort(unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortAlreadySorted() {
    Integer[] unsorted = {2, 4, 6, 8, 10};
    Integer[] expected = {2, 4, 6, 8, 10};
    Integer[] result = BubbleSort.sort(unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortDescending() {
    Integer[] unsorted = {10, 8, 6, 4, 2};
    Integer[] expected = {2, 4, 6, 8, 10};
    Integer[] result = BubbleSort.sort(unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortArrayWithDuplicates() {
    Integer[] unsorted = {9, 3, 7, 9, 2, 7, 1};
    Integer[] expected = {1, 2, 3, 7, 7, 9, 9};
    Integer[] result = BubbleSort.sort(unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortArrayWithNegativeValues() {
    Integer[] unsorted = {-7, 3, -4, 6, -2};
    Integer[] expected = {-7, -4, -2, 3, 6};
    Integer[] result = BubbleSort.sort(unsorted);
```

```
        assertArrayEquals(expected, result);
    }

    @Test
    public void testSortLargeArray() {
        Integer[] unsorted = new Integer[1000];
        for (int i = 0; i < 1000; i++) {
            unsorted[i] = (int) (Math.random() * 2000 - 1000); // Random values
between -1000 and 999
        }
        Integer[] result = BubbleSort.sort(unsorted);
        // Test if the result is sorted
        for (int i = 1; i < result.length; i++) {
            assertTrue(result[i - 1] <= result[i]);
        }
    }
}
```

QuickSort code:

```
package com.jwetherell.algorithms.sorts;

import java.util.Random;

public class QuickSort<T extends Comparable<T>> {

    private static final Random RANDOM = new Random();

    public enum PivotType {
        FIRST, MIDDLE, RANDOM
    }

    private static PivotType pivotType = PivotType.RANDOM;

    private QuickSort() { }

    public static <T extends Comparable<T>> T[] sort(PivotType type, T[] array) {
        pivotType = type; // Set the pivot selection strategy
        quickSort(array, 0, array.length - 1);
        return array;
    }

    private static <T extends Comparable<T>> void quickSort(T[] array, int low, int
high) {
        if (low < high) {
            int partitionIndex = partition(array, low, high);
            quickSort(array, low, partitionIndex - 1); // Recursively sort left
partition
            quickSort(array, partitionIndex, high); // Recursively sort right
partition
        }
    }

    private static <T extends Comparable<T>> int partition(T[] array, int low, int
high) {
```

```
        T pivot = selectPivot(array, low, high);
        int left = low;
        int right = high;

        while (left <= right) {
            while (array[left].compareTo(pivot) < 0) left++; // Find larger on left
            while (array[right].compareTo(pivot) > 0) right--; // Find smaller on
right
            if (left <= right) {
                swap(array, left, right); // Swap out-of-place elements
                left++;
                right--;
            }
        }
        return left; // Return next partition index
    }

    private static <T extends Comparable<T>> T selectPivot(T[] array, int low, int
high) {
        int pivotIndex;
        if (pivotType == PivotType.RANDOM) {
            pivotIndex = RANDOM.nextInt(high - low + 1) + low; // Random index
        } else if (pivotType == PivotType.MIDDLE) {
            pivotIndex = (low + high) / 2; // Middle index
        } else {
            pivotIndex = low; // First element
        }
        T pivot = array[pivotIndex];
        swap(array, pivotIndex, high); // Move pivot to end
        return pivot;
    }

    private static <T extends Comparable<T>> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

QuickSort Test Code:

```
package com.jwetherell.algorithms.sorts.test;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import com.jwetherell.algorithms.sorts.QuickSort;

public class QuickSortTest {

    @Test
    public void testSortIntegersRandomPivot() {
        Integer[] unsorted = {12, 7, 19, 4, 8};
        Integer[] expected = {4, 7, 8, 12, 19};
        Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.RANDOM, unsorted);
```

```
    assertArrayEquals(expected, result);
}

@Test
public void testSortStringsMiddlePivot() {
    String[] unsorted = {"kiwi", "mango", "grape", "peach"};
    String[] expected = {"grape", "kiwi", "mango", "peach"};
    String[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.MIDDLE, unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortEmptyArray() {
    Integer[] unsorted = {};
    Integer[] expected = {};
    Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.RANDOM, unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortSingleElement() {
    Integer[] unsorted = {78};
    Integer[] expected = {78};
    Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.FIRST, unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortAlreadySorted() {
    Integer[] unsorted = {3, 6, 9, 12, 15};
    Integer[] expected = {3, 6, 9, 12, 15};
    Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.MIDDLE, unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortDescending() {
    Integer[] unsorted = {20, 15, 10, 5, 0};
    Integer[] expected = {0, 5, 10, 15, 20};
    Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.RANDOM, unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortArrayWithDuplicates() {
    Integer[] unsorted = {4, 9, 2, 4, 7, 9, 6};
    Integer[] expected = {2, 4, 4, 6, 7, 9, 9};
    Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.RANDOM, unsorted);
    assertArrayEquals(expected, result);
}

@Test
public void testSortArrayWithNegativeValues() {
    Integer[] unsorted = {-12, 8, -5, 10, -2};
    Integer[] expected = {-12, -5, -2, 8, 10};
```

```
        Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.RANDOM, unsorted);
        assertArrayEquals(expected, result);
    }

    @Test
    public void testSortLargeArray() {
        Integer[] unsorted = new Integer[1000];
        for (int i = 0; i < 1000; i++) {
            unsorted[i] = (int) (Math.random() * 2000 - 1000); // Random values
between -1000 and 999
        }
        Integer[] result = QuickSort.sort(QuickSort.PIVOT_TYPE.RANDOM, unsorted);
        // Test if the result is sorted
        for (int i = 1; i < result.length; i++) {
            assertTrue(result[i - 1] <= result[i]);
        }
    }
}
```

How Were Errors Fixed?

The BubbleSort implementation error was fixed by correcting the comparison logic in the if statement within the for loop. Initially, the condition unsorted[i].compareTo(unsorted[i - 1]) > 0 was causing the array to sort in descending order. This was updated to unsorted[i].compareTo(unsorted[i - 1]) < 0, ensuring the array is sorted in ascending order. Additionally, tests were updated with new test cases, including arrays with negative numbers, duplicates, and large random arrays, to verify the fix and confirm the sorting behavior is consistent across edge cases.

The QuickSort implementation error was addressed by refining the pivot index calculation for the selected pivot type (first, middle, or random). For the random pivot, the pivot index is now selected using RAND.nextInt(finish - start + 1) + start. The partitioning logic was corrected to compare and swap elements properly around the pivot, and the recursion was adjusted to sort both left and right partitions after partitioning. New test cases, such as sorting strings and handling arrays with varying sizes, were added to ensure the algorithm works correctly with different pivot types and data inputs.

Test Case Pass Confirmations