```java
/**
 * COMP215-Programming Project 2: Multiple Sort Analysis.
 * MERGESORT is the Java implementation of the merge sort algorithm.
 * This implementation is based on psuedocode from "CLRS Algorithms."
 * This class is timed using the STOPWATCH class.
 * @author Andrew Parsons
 * @version 06 March 2017
 */

class MergeSort implements Sorter {

    /* --- INSTANCE VARIABLES --- */
    private Stopwatch stopwatch;
    private long elapsedTime;
    private static boolean debug = MainApp.debug;
    private Comparable[] mergeArray;

    /* --- METHODS --- */

    /**
     * (package-private): MERGESORT bifurcates the array, sorts the pieces, and merges them
     together
     * @param dataset Comparable[],
     * @param indexBegin int,
     * @param indexEnd int, the midpoint of a subarray
     * @return Comparable[], a sorted dataset
     */
    public Comparable[] sort(Comparable[] dataset, int indexBegin, int indexEnd) {

        stopwatch = new Stopwatch();
        mergeArray = new Comparable[dataset.length];
        // if the run size is less than two, then it is already sorted
        //if (indexEnd - indexBegin < 2)
            //return dataset;

        // divides the array, sorts both portions, and merges them. Recursive.
        if (indexBegin < indexEnd) {
            int indexMiddle = indexBegin + (indexEnd-indexBegin) / 2;
            sort(dataset, indexBegin, indexMiddle);
            sort(dataset,indexMiddle + 1, indexEnd);
            merge(dataset, indexBegin, indexMiddle, indexEnd);
            if (debug)  {
                assert assertionIsSorted(dataset);
            }
        }
        elapsedTime = stopwatch.elapsedTime();
        return dataset;
    }

    /**
     * (private): MERGE divides a dataset into smaller subarrays
     * @param dataset Comparable[], the dataset to sort
     * @param indexBegin int, the start of a subarray
     * @param indexMiddle int, the midpoint of a subarray
     * @param indexEnd int, the end of a subarray
     */
    private void merge(Comparable[] dataset, int indexBegin, int indexMiddle, int indexEnd) {

        for (int i = indexBegin; i <= indexEnd; i++) {
            mergeArray[i] = dataset[i];
        }

        if (debug) {
            assert assertionIsSorted(dataset, indexBegin, indexMiddle);
```

```
        assert assertionIsSorted(dataset, indexMiddle + 1, indexEnd);
    }

    int a = indexBegin;
    int b = indexMiddle + 1;
    int c = indexBegin;

    /* INVARIANT: At the start of each iteration of the for loop of lines twelve through
    seventeen,
    the subarray A[p…k-1] contains the k – p smallest elements of L[1…n+1] and R[1…m+1], in
    sorted order.
    Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been
    copied back into A. */

    /*INIT: c = indexBegin, so therefore dataset[indexBegin...c-1] is empty and therefore
    sorted. */

    while (a <= indexMiddle && b <= indexEnd) {
        if (mergeArray[a].compareTo(mergeArray[b]) < 1) {
            dataset[c] = mergeArray[a];
            a++;
            /*MAIN: dataset[indexBegin...c] contains c - indexBegin + 1 smallest elements */
        } else {
            dataset[c] = mergeArray[b];
            b++;
            /*MAIN: dataset[indexBegin...c] contains c - indexBegin + 1 smallest elements */
        }
        c++;
    }

    while (a <= indexMiddle) {
        dataset[c] = mergeArray[a];
        c++;
        a++;
    }
    /*TERM: dataset[indexBegin...indexEnd] contains smallest elements and is sorted. */

    /** Implementation from Cormen et al. did NOT work for some reason! */
    /*int length1 = indexMiddle - indexBegin + 1;
    int length2 = indexEnd - indexMiddle;
    Comparable[] leftDataset = new Comparable[length1+1];
    Comparable[] rightDataset = new Comparable[length2+1];

    if (debug) {
        assert assertionIsSorted(dataset, indexBegin, indexMiddle);
        assert assertionIsSorted(dataset, indexMiddle + 1, indexEnd);
    }

    for (int i = 0; i < length1; i++) {
        leftDataset[i] = dataset[indexBegin + i];
    }

    for (int j = 0; j < length2; j++) {
        rightDataset[j] = dataset[indexMiddle + j + 1];
    }

    leftDataset[length1] = Integer.MAX_VALUE;
    rightDataset[length2] = Integer.MAX_VALUE;
    int i = 0;
    int j = 0;

    *//*INIT: k = indexBegin, so therefore dataset[indexBegin...k-1] is empty and therefore
    sorted. *//*
```

```
        //while (i < length1 && j < length2) {
        for (int k = indexBegin; k <= indexEnd; k++) {
            if (leftDataset[i].compareTo(rightDataset[j]) < 1) {
                dataset[k] = leftDataset[i];
                i++;
                *//*MAIN: dataset[indexBegin...k] contains k - indexBegin + 1 smallest elements
                *//*
            } else {
                dataset[k] = rightDataset[j];
                *//*MAIN: dataset[indexBegin...k] contains k - indexBegin + 1 smallest elements
                *//*
                j++;
            }
            k++;
        } *//*TERM: dataset[indexBegin...indexEnd] contains smallest elements and is sorted. *//*
    */
    }

    /** (package-private): getter method for timing */
    public long getElapsedTime() {
        return elapsedTime;
    }

    /** check if subarray is sorted in ascending order (called by assert) */
    private static boolean assertionIsSorted(Comparable[] dataset, int low, int high) {
        for (int i = low + 1; i <= high; i++) {
            if (lessThan(dataset[i], dataset[i - 1])) return false;
        } return true;
    }

    /** check if subarray is sorted in ascending order (called by assert) */
    private static boolean assertionIsSorted(Comparable[] a) {
        return assertionIsSorted(a, 0, a.length-1);
    }

    /** check if an element is less than another one */
    private static boolean lessThan(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    @Override // DO NOT USE THIS METHOD
    public Comparable[] sort(Comparable[] dataset) {
        return new Comparable[0];
    }
}
```