Java Sorting Algorithms Comparison

Preston Stosur-Bassett

Abstract

The present study investigates the insertion sort, merge sort, and heap sort sorting algorithms and how they perform on data sets of different sizes and types when implemented in Java. Insertion sort is a sorting algorithm that runs in $\theta(n^2)$. For sorting large sets of data, insertion sort does not perform exemplary, however, for simple sorting of data sets under 10,000, insertion sort performs reasonably quickly on modern day computers. Merge sort and heap sort both perform at $\theta(n * \lg(n))$. Both merge and heap sort perform exceptionally well when sorting both small and large sets of data, even upto 10,000,000 elements.

I. Motivation

In order to show how an algorithm might run on a given set of hardware, and how the algorithm will perform when given large amounts of data, algorithms are analysed. Sorting algorithms sort data into a natural order. By analysing sorting algorithms, the fastest algorithm for a given problem can be determined.

II. Background

A sorting algorithm is used to sort data with a natural order. One such sorting algorithm is insertion sort, which sorts by iterating through a list of data, taking the current position, and repositioning it into a more appropriate place in the list. Merge sort is another sorting algorithm that greatly differs than insertion sort in that it uses a divide, conquer, and combine method; meaning that it breaks the set it is sorting into subsets until the subsets can no longer be broken up and then merge sort combines the subsets together rendering the correct answer. Heap sort creates a heap data structure and then swaps the first index with the last index value and re-heaps the data structure until the entire set is sorted.

III. Procedure

An insertion sort can be implemented in a multitude of languages using the pseudocode provided in Algorithm 1.

Insertion Sort Pre-Condition: A is a non-empty array of data with a natural order.

Insertion Sort Post-Condition: A' is a permutation of A (containing all the same elements) in strictly non-decreasing order.

Insertion Sort Outer-Loop Invariant: The subarray A'[1 ... i - 1] contains all the same elements as the subarray A[1 ... i - 1].

Insertion Sort Outer-Loop Initialization: The outer-loop invariant holds because A'[1 ... i - 1] and A[1 ... i - 1] both contain the same one element.

Insertion Sort Outer-Loop Maintenance: The outer-loop invariant holds because A'[1 ... i - 1] and A[1 ... i - 1] both contain the same elements, although they may be in different orders.

Insertion Sort Outer-Loop Termination: When the outer-loop terminates, i = A.length, which implies that the entire array has been traversed and the guard has been negated. The negation of the guard implies that A'[1 ... i - 1] contains all the elements in A[1 ... i - 1].

Insertion Sort Inner-Loop Invariant: A'[1 ... j] is sorted in strictly non-decreasing order.

Insertion Sort Inner-Loop Initialization: Before the first iteration of the loop, j = 1, meaning the subarray A'[1 ... j] contains exactly one element, which is already sorted.

Algorithm 1 INSERTION-SORT(A)

```
procedure INSERTION-SORT(A)
    if A.length < 2 then
        return A
    end if
    i = 2
    while i upto A.length do
        key = A[i]
        j = i − 1
        while j downto 1 and key < A[j] do
            A[j + 1] = A[j]
            j = j − 1
        end while
        A[j + 1] = key
        i = i + 1
    end while
    return A
end procedure
```

Insertion Sort Inner-Loop Maintenance: At the beginning of each iteration of the loop the inner-loop invariant holds because j counts down from i, and A'[j+1] is swapped with A'[j] only if A'[j+1] is less than A[j].
Insertion Sort Inner-Loop Termination: The negation of the guard implies that j = A.length and that A'[1 ... j] has been entirely traversed and sorted in strictly non-decreasing order, which maintains the inner-loop invariant.

Insertion Sort Conclusion: The termination of both the inner and outer loops implies that the entire array has been traversed, A' is a permutation of A containing all the same elements in strictly non-decreasing order. This satisfies the post condition.

A merge sort can be implemented in a variety of languages using the pseudocode provided in Algorithm 2.
Merge Sort Pre-Condition: A is a non-empty array of a comparable data type with a natural order.
Merge Sort Post-Condition: A' is a permutation of A (containing all the same elements) in strictly non-decreasing order.
Merge Pre-Condition: Left and right are both non-empty arrays of a comparable data type with a natural order in strictly non-decreasing order.
Merge Post-Condition: Combined has all the elements of both left and right in strictly non-decreasing order.
Merge Sort First For Loop Invariant: Left contains i elements, all of which can be found in A[1 ... i upto mid].
Merge Sort First For Loop Invariant Initialization: The invariant holds true vacuously because i is 0 and left contains no elements.
Merge Sort First For Loop Invariant Maintenance: The invariant holds true because i is incremented at the same rate that elements are added to left from the same i index value in A.
Merge Sort First For Loop Invariant Termination: The invariant holds true because i has been

Algorithm 2 MERGE-SORT(A)

```
procedure MERGE-SORT(A)
    if A.length < 2 then
        return A
    end if
    mid = A.length/2
    for i = 1 upto mid do
        left[left.length] = A[i]
    end for
    for i = mid upto A.length do
        right[right.length] = A[i]
    end for
    left = Merge-Sort(left)
    right = Merge-Sort(right)
    A = Merge(left, right)
    return A
end procedure

procedure MERGE(left, right)
    var i = 0
    var y = 0
    var x = 0
    while left.length ! = i and right.length ! = y do
        if left[i] < right[y] then
            combined[x] = left[i]
            i = i + 1
            x = x + 1
        end if
        if right[y] < left[i] then
            combned[x] = right[y]
            y = y + 1
            x = x + 1
        end if
    end while
    while left.length ! = i do
        combined[x] = left[i]
        i = i + 1
        x = x + 1
    end while
    while right.length ! = y do
        combined[x] = right[y]
        y = y + 1
        x = x + 1
    end while
    return combined
end procedure
```

incremented at the same rate that elements have been added to left from the same i index value in A upto mid.

Merge Sort Second For Loop Invariant: Right contains i elements, all of which can be found in A[mid ... A.length].
Merge Sort Second For Loop Invariant Initialization: The invariant holds true vacuously because i is 0 and right contains no elements.
Merge Sort Second For Loop Invariant Maintenance: The invariant holds true because i is incremented at the same rate that elements are added to right from the same i index value in A.
Merge Sort Second For Loop Invariant Termination: The invariant holds true because i has been incremented at the same rate that elements have been added to right from the same i index value in A upto to A.length.

Merge Together First While Loop Invariant: Combined contains x number of elements where x is the sum of i and y and those elements are contained in left[1 ... i] or right[1 ... y] in stricly non-decreasing order.
Merge Together First While Loop Invariant Initialization: Our invariant holds true vacuously before the first iteration of the loop because $i = 0$, $y = 0$, $i + y = 0$, $x = 0$ therefore $x = i + y$ and combined contains no elements and is vacuously in order.
Merge Together First While Loop Invariant Maintenance: Our invariant holds true at the beginning of each iteration of the loop because x is incremented whenever i and y are incremented, and elements are added to combined from left only when $left[i] < right[y]$ and added from right only when $left[i] >= right[y]$.
Merge Together First While Loop Invariant Termination: Our invariant holds true when the loop terminates because x has been incremented whenever i and y are incremented, and elements have only been added to combined from left when $left[i] < right[y]$ and from right only when $left[i] >= right[y]$.

Merge Together Second While Loop Invariant: Combined contains x number of elements where $x >= i$ and those elements are contained in left[1 ... i] in stricly non-decreasing order.
Merge Together Second While Loop Invariant Initialization: The invariant holds true before the first iteration of the loop because x has been incremented when i has been incremented and elements have been added to combined from left if and only if $left[i] < right[y]$ and from right if and only if $left[i] >= right[y]$.
Merge Together Second While Loop Invariant Maintenance: The invariant holds true at the beginning of each iteration of the loop because x has been incremented when i has been incremented and elements have been added to combined from left in order.
Merge Together Second While Loop Invariant Termination: The invariant holds true at the termination of the loop because x has been incremented when i has been incremented and elements have been added to combined from left in order.

Merge Together Third While Loop Invariant: Combined contains x number of elements where x is greater than or equal to y and those elements are contained in right[1 ... y] in strictly non-decreasing order.
Merge Together Third While Loop Invariant Initialization: The invariant holds true before the first initialization of the loop because x has been incremented when y has been incremented and elements have been added to combined from left if and only if $left[i] < right[y]$ and from right if and only if $left[i] >= right[y]$.
Merge Together Third While Loop Invariant Maintenance: The invariant holds true at the beginning of each iteration of the loop because x has been incremented when y has been incremented

and elements have been added to combined from right in order.
Merge Together Third While Loop Invariant Termination: The invariant holds true at the termination of the loop because x has been incremented when y has been incremented and elements have been added to combined from right in order.

A heap sort can be implemented in a variety of languages using the pseudocode below in Algorithm 3.
Heap Sort Pre-Condition: A is a non-empty array with a natural order.
Heap Sort Post-Condition: A' is a permutation of A in strictly non-decreasing order.
Heapify Pre-Condition: A is a non-empty array with a comparable data type in a natural order.
Heapify Post-Condition: Each parent node is more extreme than its child node.

Algorithm 3 HEAP-SORT(A)

```
procedure HEAPIFY(A, i, total)
    var left = i * 2
    var right = left + 1
    var iPrime = i
    if left <= total and A[left] > A[i] then
        i = left
    end if
    if right <= total and A[right] > A[i] then
        i = right
    end if
    if i != iPrime then
        var temp = A[iPrime]
        A[iPrime] = A[i]
        A[i] = tmp
        A = heapify(A, i, total)
    end if
    return A
end procedure
procedure HEAP-SORT(A)
    var size = A.length
    for var i = size/2 downto 1 do
        A = heapify(A, i, size
    end for
    for var i = size downto 1 do
        var tmp = A[1]
        A[0] = A[i]
        A[i] = tmp
        size = size - 1
        A = heapify(A, 1, size)
    end for
    return A
end procedure
```

Heap Sort First Loop Invariant: A[i] is the parent element in a heap.
Heap Sort First For Loop Invariant Initialization: The invariant holds true before the first

iteration of the loop because i is less than the size of the array, and therefore has a child node.
Heap Sort First For Loop Invariant Maintenance: The invariant holds true at the beginning of each iteration of the loop because A[i] will always be smaller than the size of the array, and therefore have children nodes.
Heap Sort First For Loop Invariant Termination: The invariant holds true at the termination of the loop because i will be the smallest index value of A and therefore will have children nodes.

Heap Sort Second For Loop Invariant: All elements in A at the index value greater than i are in stricly non-decreasing order.
Heap Sort Second For Loop Invariant Initialization: The invariant holds vacuously true before the first iteration of the loop because there are no elements in A that are at an index value greater than i.
Heap Sort Second For Loop Invariant Maintenance: The invariant holds true at the beginning of each iteration of the loop because each most extreme element is moved to the very end of the array at an index value greater than i.
Heap Sort Second For Loop Invariant Termination: The invariant holds true at the termination of the loop because i decreases as each largest element is moved to the end of A until the entire array has been traversed, so that all elements greater than i are in stirctly non-decreasing order.

IV. Testing

A. Testing Plan and Results

All arrays used in testing are Java ArrayList<Integer> unless otherwise specified. All times are recorded in milliseconds using a stopwatch class borrowed from Robert Sedgwick and Kevin Wayne [1] It is important to note that the stopwatch class used takes the elapsed real-time between the start of the sort algorithm and the end of the sort algorithm as opposed to taking the elapsed processor-time because these tests were run on a multi-core computer. In the table below, A denotes Array. Times in the table below are given as averages out of 10 trials in milliseconds.

Table I
Insertion Sort Test Results

Tested Input	Expected Results	Actual Results	Time
Empty A	Empty A	Empty A	0.0003
A of 1000 Strings	Sorted A 1000 Strings	Sorted A 1000 Strings	0.021
A 1 Element	Original A	Original A	0.0003
A 10 Elements	Sorted A 10 Elements	Sorted A 10 Elements	0.0005
A 100 Elements	Sorted A 100 Elements	Sorted A 100 Elements	0.0021
A 1000 Elements	Sorted A 1000 Elements	Sorted A 1000 Elements	0.019
A 10000 Elements	Sorted A 10000 Elements	Sorted A 10000 Elements	0.129
A 100000 Elements	Sorted A 100000 Elements	Sorted A 100000 Elements	6.4923
A 1000000 Elements	Sorted A 1000000 Elements	Sorted A 1000000 Elements	2135.5007
A 10000000 Elements	Sorted A 10000000 Elements	OS Crash	N/A
A 1000 Identical Elements	Original Array	Original Array	0.0052

Table II
Merge Sort Test Results

Tested Input	Expected Results	Actual Results	Time
Empty A	Empty A	Empty A	0.0002
A of 1000 Strings	Sorted A 1000 Strings	Sorted A 1000 Strings	0.0297
A 1 Element	Original A	Original A	0.0001
A 10 Elements	Sorted A 10 Elements	Sorted A 10 Elements	0.0004
A 100 Elements	Sorted A 100 Elements	Sorted A 100 Elements	0.0028
A 1000 Elements	Sorted A 1000 Elements	Sorted A 1000 Elements	0.0294
A 10000 Elements	Sorted A 10000 Elements	Sorted A 10000 Elements	0.2098
A 100000 Elements	Sorted A 100000 Elements	Sorted A 100000 Elements	2.1904
A 1000000 Elements	Sorted A 1000000 Elements	Sorted A 1000000 Elements	22.9314
A 10000000 Elements	Sorted A 10000000 Elements	Sorted A 10000000 Elements	241.0322
A 1000 Identical Elements	Original A	Original A	0.0298

Table III
Heap Sort Test Results

Tested Input	Expected Results	Actual Results	Time
Empty A	Empty A	Empty A	0.0002
A of 1000 Strings	Sorted A 1000 Strings	Sorted A 1000 Strings	.0188
A 1 Element	Original A	Original A	0.0003
A 10 Elements	Sorted A 10 Elements	Sorted A 10 Elements	0.0006
A 100 Elements	Sorted A 100 Elements	Sorted A 100 Elements	0.002
A 1000 Elements	Sorted A 1000 Elements	Sorted A 1000 Elements	0.0162
A 10000 Elements	Sorted A 10000 Elements	Sorted A 10000 Elements	0.0468
A 100000 Elements	Sorted A 100000 Elements	Sorted A 100000 Elements	0.889
A 1000000 Elements	Sorted A 1000000 Elements	Sorted A 1000000 Elements	23.0635
A 10000000 Elements	Sorted A 10000000 Elements	Sorted A 10000000 Elements	244.1952
A 1000 Identical Elements	Original A	Original A	0.0156

B. Problems Encountered

One major issue encountered during the development of this insertion sort was that after completing the sort, A' was sorted properly except for the first element in the array. No matter what value the first element of A had, it did not change position in A'. For example, if A[5, 6, 3, 4, 7] was passed to the insertion sort algorithm, the returned array would look like A'[5, 3, 4, 6, 7]. Changing the guard for the inner for loop (see Algorithm 1 line 6) from *key < A[j] and j downto 1* to *j downto 1 and key < A[j]* corrected this issue.

Another issue encountered during the development of this program was executing Java *assert* statements. When assertions were wrapped in parenthesis, they evaluated to a boolean, and the assert would throw an *AssertionError* with the explanation 'cannot compare to boolean'. Removing the parenthesis wrapping the assertion fixed this issue.

The last issue encountered during the development process of this program was generic programming. The program would not compile because not all Objects have a compareTo method. Adding the statement $< T\ \mathit{extends}\ \mathit{Comparable} < T >>$ fixed this issue. [2]

V. Experimental Analysis

The insertion sort demonstrated in Algorithm 1, 2, and 3 was implemented in Java and executed on an HP SpectreXT TouchSmart with a 4 core Intel i7 processor clocked at 1.9GHz running Ubuntu Gnome 14.10 64-bit.

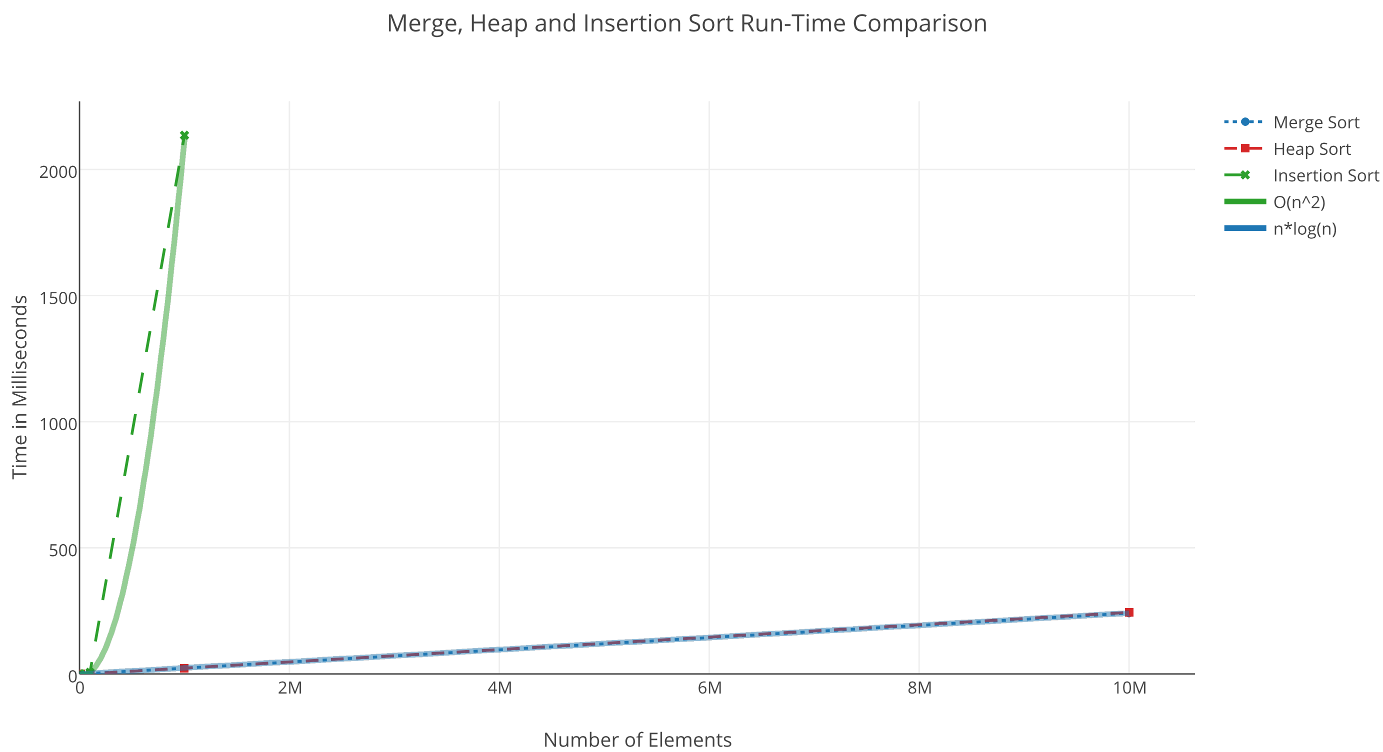

Figure 1. Merge, Heap and Insertion Sort Run-Time Comparison

The expected growth of insertion sort as the number of elements (n) grows large can be represented as $\theta(n^2)$ where $\theta()$ represents the asymptotically tightly bound running time. [3] At n_0 the algorithm took 0.0003 milliseconds to complete. At n_1 the algorithm also took 0.0003 milliseconds to complete. For both these values of n, the algorithm runs at $\theta(1)$, which is a constant value because the algorithm completes before a loop is run. As n grows larger the time to complete the experimental data correlates quite accurately with the expected growth. For n_{1*10^6} the data matches up perfectly. This is to be expected, because as n grows larger the constants and lower orders of the actual running time of the insertion sort start to affect the running time less and less as the highest order of n^2 is so large. It is expected for lower values of n to not correlate well with $\theta(n^2)$ because the system running the algorithm may be executing superfolous commands such as checking for system. For the data graphed above (see Fig. 1) this was in fact the case. To aid this issue, averages of 10 trails were taken and used in Figure 1.

The expected growth of merge sort as the number of elements (n) grows large can be represented as $\theta(n * \lg(n))$. At n_0 the sort took 0.0002 milliseconds to complete, however, at n_1 the algorithm took 0.0001 milliseconds to complete. The discrepency can be attributed to superfolous commands being executed by the system. As n grows larger the time to complete correlates with the expected running time. It is expected for lower values of n to not correlate as nicely with the expected running time because the system constant will have a larger impact on the smaller values of n. It is important to note that merge sort ran much faster than insertion sort at large values of n such as n_{1*10^6} and was able to process n_{1*10^7} where insertion sort crashed the system. The best case running time for this implementation of merge sort is $\theta(1)$ and is elicited only when $n <= 1$ because if $n <= 1$ the array is vacuously sorted and can be returned immediately without executing any loop, and run at a constant time.

Heap sort is very similar to merge sort in that its expected running time is $\theta(n * \lg(n))$. At lower values of n heap sort ran marginally quicker than merge sort, however, as n grew larger merge sort began to run marginally faster than heap sort. This difference is neglagable because of superfolous commands being executed by the system and other running processes. For all intensive purposes merge sort and heap sort have the same run time and the major difference between the two is the system constant. For n_0 heap sort completed in 0.0002 milliseconds and for n_1 heap sort completed in 0.0003 milliseconds. In both merge sort and insertion sort the sort completes before any loop is executed if $n <= 1$ however in this implementation of heap sort the algorithm runs through the entire sort. We can tell from this that the constant time is 0.0002 milliseconds, which correlates nicely with both merge and insertion sort. $\theta(n * \lg(n))$ is the both the best and worst case run-time for heap sort.

VI. Conclusions

With a running time of $\theta(n * \lg(n))$ both heap sort and merge sort perform much faster than insertion sort at a running time of $\theta(n^2)$, especially when n grows large. While faster comptuers will be able to execute insertion sort faster, as n grows large, even on a slower machine, both merge sort and heap sort will perform faster. Because it is easier to setup and maintain, insertion sort is a good choice when dealing with small sets of data that don't go beyond n_{1*10^5}, however, if data is expected to grow larger than that, either heap or merge sort is the best choice.

References

[1] R. Sedgewick and K. Wayne, "Stopwatch," Java Class.
[2] E. Hartig, "Generic programming," Conversation, January 2015.
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, Massachusetts: The MIT Press, 2009.

Appendix

Listing 1
Driver

```
/*
* @Author: Preston Stosur-Bassett
* @Date: Feb, 23, 2015
* @Class: Driver
* @Description: This class will test the functionality of the overall
   program by serving as a driver that runs through and calls all other
   required classes
*/

import java.util.ArrayList;

public class Driver{
  public static void main(String args[]) {
    DummyData testData = new DummyData();

    ArrayList<Integer> testList = new ArrayList<Integer>();
    testList = testData.runArrayList(10, 0, 100, testList);
    //testList = testData.runArrayList(1000, testList);

    System.out.println("Unsorted list: ");
    System.out.println(testList);

    Sort sorter = new Sort();

    Stopwatch watchman = new Stopwatch();
    //testList = sorter.mergeSort(testList);
    testList = sorter.heapSort(testList);

    System.out.println("Sorted List: ");
    System.out.println(testList);
    System.out.println("Time To Complete: "+watchman.elapsedTime());
  }
}
```

Listing 2
Debug

```
/*
*        @Author Preston Stosur-Bassett
*        @Date Jan 21, 2015
*        @Class Debug
*        @Description This class will help debugging by being able to turn
    on and turn off debug messages easily
*/

import java.util.List;
import java.util.ArrayList;
```

```
public class Debug<T> {
        boolean debugOn; //Variable to keep track of whether or not debug
            is on

        /*
        *       @Description constructor method that sets the default
           value of debugOn to false so that debug statements will not
           automatically print
        */
        public void Debug() {
                debugOn = false;
        }

        /*
        *       @Description turn on debugging print statements
        */
        public void turnOn() {
                debugOn = true;
        }

        /*
        *       @Description turn off debugging print statements
        */
        public void turnOff() {
                debugOn = false;
        }

        /*
        *       @Description will print messages only when debugOn
           boolean is set to true
        *       @param String message the string to print when debugging
           is turned on
        */
        public void print(T message) {
                if(debugOn == true) {
                        System.out.println(message);
                }
        }

        /*
        *       @Pre-Condition <code>T expected</code> and <code>T actual
           </code> are both of the same type T
        *       @Post-Condition If <code>T expected</code> and <code>T
           actual</code> are found to be equal, the program moves on,
           otherwise the program halts with <code>AssertionError</code>
           is thrown
        *       @Description runs an assert statement against an expected
            value and the actual value that are passed as parameters only
            when <code>debugOn == true</code>
```

```
*       @param T expected the expected value to assert against
   the actual value
*       @param T actualt he actual value to assert against the
   expected value
*/
public void assertEquals(T expected, T actual) {
        if(debugOn == true) {
                assert actual.equals(expected);
        }
}

/*
* @Pre-Condition: <code>List<Integer> actual</code> is a iterable
    list of Integer objects
*       @Post-Conditions: If the List of Integer objects is in
   stricly non-decreasing order, the program moves on normally,
   if not, the program halts with an <code>AssertionError</code>
* @Description: runs an assertion statement against a list of
   Integer objects to ensure that for <code>k = actual.size(); A[
   k - 2] <= A[k - 1];</code>
*       @param List<Integer> actual the list to assert is in
   stricly non-decreasing order
*/
public void assertOrder(List<Integer> actual) {
        if(debugOn == true) {
                int i = actual.size();
                while(i > 1) {
                        assert actual.get(i - 1).compareTo(actual
                           .get(i - 2)) >= 0;

                        i--;
                }
        }
}

/*
*       @Pre-Condition: <code>ArrayList<Integer> actual</code> is
    an ArrayList of Integer Objects
*       @Post-Condition: If the ArrayList of Integer Objects is
   in stricly non-decreasing order, the program moves on normally
   , if not, the pgoram halts with an <code>AssertionError</code>
*       @Description: runs an assertion statement against an
   ArrayList of Integer Objects to ensure that for <code>k =
   actual.size(); A[k-2] <= A[k-1];</code>
*       @param ArrayList<Integer> actual the ArrayList to assert
   is in stricly non-decreasing order
*/
public void assertOrder(ArrayList<Integer> actual) {
        if(debugOn == true) {
                int i = actual.size();
```

```
                while(i > 1) {
                        assert actual.get(i - 1).compareTo(actual
                           .get(i - 2)) >= 0;

                        i--;
                }
        }
}

/*
*       @Pre-Condition: ArrayList is an ArrayList of Integers and
    i is less than or equal to half of the size of actual
*       @Post-Condition: If elements exist past i the assertion
   holds
*       @Description: runs an assertion statement against an
   ArrayList of Integer Objects to ensure that there are children
    nodes of actual[i].
*       @param ArrayList<Integer> actual the array to test
   against
*       @param int i the index value to check has children nodes.
*/
public void assertChildren(ArrayList<Integer> actual, int i) {
        if(debugOn == true) {
                assert actual.size() > i;
        }
}

/*
*       @Pre-Condition: actual and expected both contain Integer
   Objects
*       @Post-Condition: If all the elements inside of the actual
    arraylist are also contained in the expected arraylist, then
   the assertion holds true
*       @Description: Tests to ensure a given ArrayList of
   Integer Objects contains all the elements of another given
   ArrayList of Integer Objects
*       @param ArrayList<Integer> expected the list to check
   contains against
*       @param ArrayList<Integer> actual the list to check to
   make sure all its elements are contained in the other
   arraylist
*/
public void assertContains(ArrayList<Integer> expected, ArrayList
   <Integer> actual) {
        if(debugOn == true) {
                for(int i = 0; i < actual.size(); i++) {
                        assert expected.contains(actual.get(i));
                }
        }
}
```

```
/*
*       @Pre-Condition: expectedOne, expectedTwo, and actual all
   contain Integer Objects
*       @Post-Condition: If all the elemts inside of the actual
   ArrayList are also contined in either the expectedOne
   ArrayList or the expectedTwo ArrayList, then the assertion
   holds true
*       @Description: Tests to ensure a given ArrayList of
   Integer Objects contains all the elements of another given
   ArrayList of Integer Objects
*       @param: ArrayList<Integer> expectedOne one of the lists
   to check to see if the given ArrayList actual's elements are
   contained in
*       @param: ArrayList<Integer> expectedTwo one of the lists
   to check to see if the given ArrayList actual's elements are
   contained in
* @param: ArrayList<Integer> actual the list to check ot make
   sure all its elements are contained in either expectedOne or
   expectedTwo
*/
public void assertContains(ArrayList<Integer> expectedOne,
   ArrayList<Integer> expectedTwo, ArrayList<Integer> actual) {
        if(debugOn == true) {
                for(int i = 0; i < actual.size(); i++) {
                        assert expectedOne.contains(actual.get(i)
                           ) || expectedTwo.contains(actual.get(i
                           ));
                }
        }
}

/*
* @Description: asserts that the first arguement is stricly
   greator than the second arguement
*       @param int large an integer primative value to assert is
   strictly greator than the second arguement
*       @param int small an integer primative value to assert the
    first arguement is strictly greator than.
*/
public void assertStrictGreat(int large, int small) {
        if(debugOn == true) {
                assert large > small;
        }
}

/*
*       @Description: asserts that the first arguement is
   strictly less than the second arguement
```

```
	*	@param int small an integer primative value to assert is stricly less than the second arguement
	*	@param int large an integer primative value to assert the first arguement is strictly less than.
	*/
	public void assertStrictLess(int small, int large) {
		if(debugOn == true) {
			assert small < large;
		}
	}

	/*
	*	@Description: asserts that the first arguement is greator than or equal to the second arguement
	*	@param int large an integer primative value to assert is greator than or equal to the second arguement
	*	@param int small an integer primative value to assert the first arguement is greator than or equal to.
	*/
	public void assertGreatEquals(int large, int small) {
		if(debugOn == true) {
			assert large >= small;
		}
	}

	/*
	*	@Description: asserts that the first arguement is less than or equal to the second arguement
	*	@param int small an integer primative value to assert is less than or equal to the second arguement
	*	@param int large an integer primative value to assert the first arguement is less than or equal to.
	*/
	public void assertLessEquals(int small, int large) {
		if(debugOn == true) {
			assert small <= large;
		}
	}
}
```

Listing 3
DummyData

```
/*
*	@Author Preston Stosur-Bassett
*	@Date Jan 25, 2015
*	@Class DummyData
*	@Description This class contains methods to generate dummy data given a set of parameters.
*/
```

```
import java.util.ArrayList;
import java.util.Random;

public class DummyData {

        /*
        *       @Description runArrayList<Integer> will take an ArrayList
             of Integer Objects and add a given amount of values to it
        *       @param int end the ending value to denote when to stop
           adding to the array list
        *       @param int min the minimum value of the randomly
           generated data.
        *       @param int max the maximum value of the randomly
           generated data.
        *       @param ArrayList<Integer> list the list to add value to
           and return
        *       @return ArrayList<Integer> the list after it has been
           updated with the randomly generated data
        */
        public static ArrayList<Integer> runArrayList(int end, int min,
           int max, ArrayList<Integer> list) {
                Random random = new Random();
                Debug debugger = new Debug();
                int start = 0;
                // INVARIANT: A.length >= start
                // INITIALIZATION: start = 0, A.length can be longer than
                    0 when initially passed, but not smaller, so our
                   invariant holds
                debugger.assertGreatEquals(list.size(), start);
                while(start < end) {
                        // MAINTANANCE: At the beginning of each
                           iteration, one element was added to A and
                           start was increased by one, therefore, our
                           invariant holds true.
                        debugger.assertGreatEquals(list.size(), start);
                        Integer intToAdd = new Integer(random.nextInt((
                           max - min + 1) + min));
                        list.add(intToAdd);

                        //Count up on the iterator
                        start++;
                }
                /*TERMINATION: The negation of the guard implies that (
                   end - start) number of elements have been added to A,
                   since start is initialized as 0 at the beginning of
                   the method and is
                        incremented by 1 each iteration of the loop,
                           which means that start amount of elements have
                            been added to A, and so our invariant holds
                           true.   */
```

```
        debugger.assertGreatEquals(list.size(), start);

        return list;
    }

    /*
    *       @Description: runArrayList<String> will take an ArrayList
        of String Objects and add a given amount of String numerical
       values to it
    *       @param int end the ending value to denote when to stop
       adding to the array list
    *       @param ArrayList<String> list the list to add String
       values to and return
    *       @return ArrayList<String> the list after it has been
       updated with the randomly generated numerical String values
    */
    public static ArrayList<String> runArrayList(int end, ArrayList<
       String> list) {
        Random random = new Random();
        Debug debugger = new Debug();
        int start = 0;
        // INVARIANT: A.length >= start
        // INITIALIZATION: Before the first iteration of the loop
           , start = 0 and A.length cannot be less than 0, so our
             invariant holds true
        debugger.assertGreatEquals(list.size(), start);
        while(start < end) {
            // MAINTENANCE: At the beginning of each
               iteration of the loop our invariant holds
               because for each iteration of the loop one
               element is added to A and start is incremented
                by 1
            debugger.assertGreatEquals(list.size(), start);
            Integer intToString = new Integer(random.nextInt
               ((1000000 - 1) + 1));
            String intString = String.valueOf(intToString);
            list.add(intString);

            //Count up on the iterator
            start++;
        }
        /* TERMINATION: The negation of the guard implies that (
           end - start) number of elements have been added to A,
           since start is initialized as 0 at the beginning of
           the method and is
                    incremented by 1 each iteration of the
                       loop, which means that start amount of
                        elements have been added to A, and so
                        our invariant holds true. */
        debugger.assertGreatEquals(list.size(), start);
```

```
        return list;
}

/*
*       @Description: identicalElement will take an element and
   add it to the ArrayList<Integer> for a given amount of times
*       @param int end the ending value to denote when to stop
   adding elements to the array
*       @param int element the element to add over and over again
    to the array
*       @param ArrayList<Integer> list the list to add elements
   to
*       @return ArrayList<Integer> the list after it has been
   updated with the given data
*/
public static ArrayList<Integer> identicalElement(int end, int
   element, ArrayList<Integer> list) {
        // INVARIANT: A.length >= start
        int start = 0;
        Debug debugger = new Debug();
        //The element to add over and over again
        Integer iden = new Integer(element);
        // INITIALIZATION: Before the first iteration of th eloop
           , start = 0 and A.length cannot equal anything less
           than 0, so our invariant holds true
        debugger.assertGreatEquals(list.size(), start);
        while(start < end) {
                // MAINTENANCE: At the beginning of each
                   iteration of the loop our invariant holds
                   because for each iteration of the loop one
                   element is added to A and start is incremented
                    by 1
                debugger.assertGreatEquals(list.size(), start);
                list.add(iden);

                //Count up on the iterator
                start++;
        }
        /* TERMINATION: The negation of the gaurd implies that (
           end - start) number of elements hav ebeen added to A,
           since start is initialied as 0 at the beginning of the
            method and is
                        incremented by 1 each iteration of the
                           loop, which means that start amount of
                            elements have been added to A, and so
                            our invariant holds true    */
        debugger.assertGreatEquals(list.size(), start);

        return list;
```

```
        }
}
```

The class Stopwatch has not been altered from its original form.

Listing 4
STOPWATCH

```
/*************************************************************************
 *  Compilation:  javac Stopwatch.java
 *
 *
 *************************************************************************/


/**
 *  The <tt>Stopwatch</tt> data type is for measuring
 *  the time that elapses between the start and end of a
 *  programming task (wall-clock time).
 *
 *  See {@link StopwatchCPU} for a version that measures CPU time.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */


public class Stopwatch {

    private final long start;

    /**
     * Initialize a stopwatch object.
     */
    public Stopwatch() {
        start = System.currentTimeMillis();
    }


    /**
     * Returns the elapsed time (in seconds) since this object was
        created.
     */
    public double elapsedTime() {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }

}
```

Listing 5
Sort

```
/*
*       @Author: Preston Stosur-Bassett
*       @Date: Jan 24, 2015
*       @Class: Sort
*       @Description: This class will contain many methods that will sort
    generic data types using common sorting algorithms.
*/

import java.util.ArrayList;
import java.util.List;

public class Sort<T extends Comparable<T>> {
        /*
        *       @Pre-Condition: ArrayList<T> is a non-empty set of data
           where T is a comparable data type with a natural order
        *       @Post-Condition: Each parent node is more extreme than
           its child node.
        *       @Description: heapify is a helper method for heapSort
           that keeps the heap in order so that the root node is the most
            extreme element in the heap.
        *       @param ArrayList<T> unsorted is a non-empty set of data
           where T is a comparable data type with a natural order
        *       @param int i
        *       @param int total
        *       @return ArrayList<T> unsorted
        */
        private ArrayList<T> heapify(ArrayList<T> unsorted, int i, int
           total) {
                int left = i * 2;
                int right = left + 1;
                int originalI = i;

                if(left <= total && unsorted.get(left).compareTo(unsorted
                   .get(i)) > 0) {
                        i = left;
                }
                if(right <= total && unsorted.get(right).compareTo(
                   unsorted.get(i)) > 0) {
                        i = right;
                }
                if(i != originalI) {
                        T tmp = unsorted.get(originalI);
                        unsorted.set(originalI, unsorted.get(i));
                        unsorted.set(i, tmp);
                        unsorted = heapify(unsorted, i, total);
                }
```

```
        return unsorted;
    }

    /*
    *       @Pre-Condition: ArrayList<T> unsorted is a non-empty
       ArrayList<T> where T is a comparable data type with a natural
       order.
    *       @Post-Condition: ArrayList<T> sorted is a permutation of
       unsorted (it contains all the same elements) in stricly non-
       decreasing order
    *       @Description: heapSort will sort a given set of data in
       an ArrayList<T> in strictly non-decreasing order using the
       heap sort method.
    *       @param ArrayList<T> unsorted is a non-empty ArrayList<T>
       where T is a comparable data type with a natural order
    *       @return ArrayList<T> sorted is a permutation of unsorted
       in strictly non-decreasing order
    */
    //Invariant for First While Loop: unsorted[i] is the parent
       element in a heap
    //Invariant for Second While Loop: All elements in unsorted
       greater than the index value of y are in stricly non-
       decreasing order
    public ArrayList<T> heapSort(ArrayList<T> unsorted) {
        //Debug
        Debug debugger = new Debug();

        int arrSize = unsorted.size() - 1;
        int i = arrSize / 2;
        //Initialization: Our invariant holds true before the
           first iteration of the loop because unsorted[i] must
           have child elements
        debugger.assertChildren(unsorted, i);
        while(i >= 0) {
            //Maintanance: Our invariant holds true at the
               beginning of each iteration of the loop
               because unsorted[i] must have children
               elements
            debugger.assertChildren(unsorted, i);

            unsorted = heapify(unsorted, i, arrSize);

            i--;
        }
        //Termination: Our invariant holds true at the
           termination of the loop because i will be the smallest
            index value of the loop and must have children
           elements
        debugger.assertChildren(unsorted, i);
```

```
        int y = arrSize;
        //Initialization: Our invariant holds vacuously true
           before the first iteration of the loop because there
           are no elements in unsorted that are at an index value
            greater than y.
        debugger.assertStrictLess(arrSize, y+1);

        while(y > 0) {
                //Maintanance: Our invariant holds true at the
                   beginning of each iteration of the loop
                   because all elements greater than y are in
                   strictly non-decreasing order

                T tmp = unsorted.get(0);
                unsorted.set(0, unsorted.get(y));
                unsorted.set(y, tmp);
                arrSize--;
                unsorted = heapify(unsorted, 0, arrSize);

                y--;
        }
        //Termination: Our invariant holds true at the
           termination of the loop because y decreases as each
           largest element is moved to the end of the list until
           the entire array has been traversed, so that all
           elements greater than y are in stricly non-decreasing
           order
        debugger.assertOrder(unsorted);

        ArrayList<T> sorted = unsorted;
        return sorted;
}

/*
*       @Pre-Condition: ArrayList<T> left is a non-empty sorted
   array in stricly non-decreasing order where T is a comparable
   data type with a natural order
*       @Post-Condition: ArrayList<T> right is a non-empty sorted
    array in strictly non-decreasing order where T is a
   comparable data type with a natural order
*       @Description: mergeTogether is used by the mergeSort
   method to recombine the left and right sections of the
   ArrayList<T> that is being sorted by merge sort. Note that
   this is a helper method for the mergeSort method, and should
   not be called externally of this class.
*       @param ArrayList<T> left a non-empty ArrayList<T> where T
    is a comparable data type with a natural order.
*       @param ArrayList<T> right a non-empty ArrayList<T> where
   T is a comparable data type with a natural order.
```

```
*       @return ArrayList<T> combined should contain all the
   elements of left and right in stricly non-decreasing order
*/
//Invariant for First While Loop: combined contains x number of
   elements where x is the sum of i and y and those elements are
   contained in left[0 ... i] or right[0 ... y] in stricly non-
   decreasing order
//Invaraint for Second While Loop: combined contains x number of
   elements where x is greater than or equal to i and those
   elements are contined in left[0 ... i] in stricly non-
   decreasing order
//Invariant for Third While Loop: combined contains x number of
   elements where x is greater than or equal to y and those
   elements are contained in right[0 ... y] in stricly non-
   decreasing order
private ArrayList<T> mergeTogether(ArrayList<T> left, ArrayList<T
   > right) {
        ArrayList<T> combined = new ArrayList<T>();
        int i = 0;
        int y = 0;
        int x = 0;

        //Debug
        Debug debugger = new Debug();

        //Initialization: Our invariant holds true vacuously
           before the first execution of the loop because x, i,
           and y are all equal to zero, combined is empty and
           therefore in order
        debugger.assertEquals(0, i);
        debugger.assertEquals(0, y);
        debugger.assertEquals(0, x);
        debugger.assertEquals(i, combined.size());

        while(left.size() != i && right.size() != y) {
                //Maintanance: Our invariant holds true at the
                   beginning of each iteration of the loop
                   because x is incremented whenever i or y is
                   incremented and elements are added to combined
                    from left and right in order
                debugger.assertEquals(i+y, x);
                debugger.assertEquals(x, combined.size());
                debugger.assertOrder(combined);
                debugger.assertContains(right, left, combined);

                if(left.get(i).compareTo(right.get(y)) < 0) {
                        combined.add(x, left.get(i));
                        i++;
                        x++;
                }
```

```
        else {
                combined.add(x, right.get(y));
                y++;
                x++;
        }
}
//Termination: Our invariant holds tur at the termination
    of the loop because x has been incremented whenever i
    or y has been incremented, and elements are added to
   combined from left and right in order
debugger.assertEquals(i+y, x);
debugger.assertEquals(x, combined.size());
debugger.assertOrder(combined);
debugger.assertContains(right, left, combined);

//Initialization: Our invariant holds true before the
   first execution of the loop because x has been
   incremented whenever i has been incremented and
   elements have been added to combined from left in
   order
debugger.assertGreatEquals(x, i);
debugger.assertEquals(x, combined.size());
debugger.assertOrder(combined);
debugger.assertContains(left, combined);

while(left.size() != i) {
        //Maintanance: Our invariant holds true at the
           beginning of each iteration of the loop
           because x has been incremented whenever i is
           incremented and elements have been added to
           combined from left in order
        debugger.assertGreatEquals(x, i);
        debugger.assertEquals(x, combined.size());
        debugger.assertOrder(combined);
        debugger.assertContains(left, combined);

        combined.add(x, left.get(i));
        i++;
        x++;
}
//Termination: Our invariant holds true at the
   termination of the loop because x has been incremented
    whenever i was incremented and elements have been
   added to combined from left in order
debugger.assertGreatEquals(x, i);
debugger.assertEquals(x, combined.size());
debugger.assertOrder(combined);
debugger.assertContains(left, combined);
```

```
        //Initialization: Our invariant holds true before the
           first execution of the loop because x has been
           incremented whenever y has been incremented and
           elements have been added to combined from right in
           order
        debugger.assertGreatEquals(x, y);
        debugger.assertEquals(x, combined.size());
        debugger.assertOrder(combined);
        debugger.assertContains(right, combined);

        while(right.size() != y) {
                //Maintanance: Our invariant holds tur at the
                   beinning of each iteration of the loop because
                    x has been incremented whenever y is
                   incremented and elements have been added to
                   combined from right in order
                debugger.assertGreatEquals(x, y);
                debugger.assertEquals(x, combined.size());
                debugger.assertOrder(combined);
                debugger.assertContains(right, combined);

                combined.add(x, right.get(y));
                y++;
                x++;
        }
        //Termination: Our invariant holds true at the
           terminatino of the loop because x has been incremented
            whenever y was incremented and elements have been
           added to combined from right in order.
        debugger.assertGreatEquals(x, y);
        debugger.assertEquals(x, combined.size());
        debugger.assertOrder(combined);
        debugger.assertContains(right, combined);

        return combined;
}

/*
*       @Pre-Condition: ArrayList<T> unsorted is a set of data
   type T, where T is a Comparable data type with a natural order
   .
*       @Post-Condition: ArrayList<T> returnValue is a
   permutation of unsorted in strictly non-decreasing order.
*       @Description: mergeSort will sort a given set of data in
   ArrayList<T> using the merge sort method
*       @param a non-empty ArrayList<T> unsorted where T is a
   Comparable data type with a natural order
*       @return ArrayList<T> returnValue which is a permutation
   of unsorted, in strictly non-decreasing order,
*/
```

```
//Invariant for First While Loop: left contains i elements, all
   of which can be found in sorted
//Invariant for Second While Loop: right contains y elements, all
    of which can be found in sorted
public ArrayList<T> mergeSort(ArrayList<T> unsorted) {
        ArrayList<T> sorted = unsorted;
        ArrayList<T> left = new ArrayList<T>();
        ArrayList<T> right = new ArrayList<T>();
        ArrayList<T> returnValue;

        //Debug
        Debug debugger = new Debug();
        debugger.turnOn();

        if(sorted.size() <= 1) {
                returnValue = sorted;
        }
        else {
                int mid = (sorted.size() / 2);
                int i = 0;
                //Initialization: Our invariant holds true bcause
                    i is zero and left contains 0 elements before
                    the first iteration of the loop.
                debugger.assertEquals(i, left.size());

                while(i < mid) {
                        //Maintanance: Our invariant holds true
                           because i is increased at the same
                           rate elements are added to left from
                           the same i index in sorted
                        debugger.assertEquals(i, left.size());
                        debugger.assertContains(sorted, left);

                        T temp = sorted.get(i);
                        left.add(temp);

                        i++;
                }
                //Termination: Our invariant holds true because i
                    has been incremented at the same rate
                   elements are added to left from the same index
                    i in sorted
                debugger.assertEquals(i, left.size());
                debugger.assertContains(sorted, left);

                int y = mid;
                //Initialization: Our invariant holds true
                   because i is zero and right contains 0
                   elements before the first iteration of the
                   loop.
```

```
                debugger.assertEquals(y, right.size());

                while(y < sorted.size()) {
                        //Maintanance: Our invariant holds true
                           because y is increased at the same
                           rate elements are added to right from
                           the same y index in sorted.
                        debugger.assertEquals(y, right.size());
                        debugger.assertContains(sorted, right);

                        T temp = sorted.get(y);
                        right.add(temp);

                        y++;
                }
                //Termination: Our invariant holds true because y
                    has been incremented at the same rate
                   elements are added to left from the same index
                    y in sorted
                debugger.assertEquals(y, right.size());
                debugger.assertContains(sorted, right);

                left = mergeSort(left);
                right = mergeSort(right);
                returnValue = mergeTogether(left, right);
        }
        return returnValue;
}

/*
*       @Pre-Condition: ArrayList<T> unsorted is an unsorted
   ArrayList of a comparable data type that is non-empty
*       @Post-Condition: ArrayList<T> will return a permutation
   of <code>unsorted</code> that will be in increasing order
*       @Description: insertionSort will sort an ArrayList of
   generic type T in increasing order using an insertion sort
*       @param ArrayList<T> unsorted is a non-empty unsorted
   array list of T, where T is a comparable type
*       @return sorted is a permutation of <code>unsorted</code>
   where all the elements are sorted in increasing order
*/
// INVARIANT (Outer-Loop): The pre condition implies that sorted
   [0 ... i - 1] will contain all the same data as unsorted[0 ...
    i - 1].
// INVARIANT (Inner-Loop): sorted[0 ... j] is sorted in stricly
   non-decreasing order.
public ArrayList<T> insertionSort(ArrayList<T> unsorted) {
        Debug debugger = new Debug<List<T>>();
        debugger.turnOn();
        ArrayList<T> sorted = unsorted;
```

```
if(sorted.size() > 1) {
        int i = 1;
        /* INITIALIZATION (Outer-Loop): The invariant
           holds because i = 1, and there is one element
           in the subarray of sorted[0 ... i - 1] and
           unsorted[0 ... i - 1], */
        List<T> subSortedOI = sorted.subList(0, i - 1);
        List<T> subUnsortedOI = unsorted.subList(0, i -
           1);
        debugger.assertEquals(subUnsortedOI, subSortedOI)
           ;

        while(i < sorted.size()) {
                /* MAINTENANCE (Outer-Loop): At the
                   beginning of each iteration of the
                   loop, the loop invariant is maintained
                    because the subarray of sorted[0 ...
                   i - 1] contains all the same elements
                   as
                        unsorted[0 ... i - 1] */
                List<T> subSortedOM = sorted.subList(0, i
                    - 1);
                List<T> subUnsortedOM = unsorted.subList
                   (0, i - 1);
                debugger.assertEquals(subUnsortedOM,
                   subSortedOM);

                T value = sorted.get(i);
                int j = i - 1;
                // INITIALIZATION (Inner-Loop): Before
                   the first iteration of the loop, j =
                   0, the subarray of sorted[0 ... 0]
                   contains one elements and therefore
                   the invariants holds vacuously.
                List subSortedII = sorted.subList(0, j);
                debugger.assertOrder(subSortedII);

                while(j >= 0 && (value.compareTo(sorted.
                   get(j)) < 0)) {
                        // MAINTENANCE: (Inner-Loop): At
                           the beginning of each
                           iteration sorted[0 ... j] is
                           sorted in stricly non-
                           decreasing order
                        List subSortedIM = sorted.subList
                           (0, j);
                        debugger.assertOrder(subSortedIM)
                           ;

                        sorted.set(j+1, sorted.get(j));
```

```
                                j--;
                        }
                        sorted.set(j+1, value);
                        // TERMINATION (Inner-Loop): The negation
                            of the guard implies that the sorted
                           [0 ... j] has been traversed and is
                           stricly non-decreasing order.
                        List subSortedIT = sorted.subList(0, j+1)
                           ;
                        debugger.assertOrder(subSortedIT);

                        //Count up on the iterator
                        i++;
                }
                /* TERMINATION (Outer-Loop): When the loop
                   terminates, i is equal to sorted.size()
                   meaning the entire array has been traversed
                   and that the guard has been negated.
                        The negation of the guard implies that
                           sorted[0 ... i - 1] contains all the
                           elements of unsorted[0 ... i - 1] */
                List subSortedOT = sorted.subList(0, i - 1);
                debugger.assertOrder(subSortedOT);
                Integer integerI = new Integer(i);
                Integer sortedSizeO = new Integer(sorted.size());
                debugger.assertEquals(sortedSizeO, integerI);
                debugger.assertEquals(unsorted, sorted);
        }
        return sorted;
    }
}
```