

Java Sorting Algorithms Comparison

Preston Stosur-Bassett

Abstract

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. Heap 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.

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.

Algorithm 1 INSERTION-SORT(A)

```

1: procedure INSERTION-SORT(A)
2:   if  $A.length < 2$  then
3:     return A
4:   end if
5:    $i = 2$ 
6:   while  $i$  upto  $A.length$  do
7:      $key = A[i]$ 
8:      $j = i - 1$ 
9:     while  $j$  downto 1 and  $key < A[j]$  do
10:       $A[j + 1] = A[j]$ 
11:       $j = j - 1$ 
12:    end while
13:     $A[j + 1] = key$ 
14:     $i = i + 1$ 
15:  end while
16:  return A
17: end procedure

```

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

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

Insertion Sort Outer-Loop Maintenance: The outer-loop invariant holds because $A'[1 \dots i - 1]$ and $A[1 \dots i - 1]$ 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 $A'[1 \dots i - 1]$ contains all the elements in $A[1 \dots i - 1]$.

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

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

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 $A'[1 \dots j]$ 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 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 While Loop Invariant: Left contains i elements, all of which can be found in $A[0 \dots i \text{ upto } mid]$.

Merge Sort First While Loop Invariant Initialization: The invariant holds true vacuously because i is 0 and left contains no elements.

Merge Sort First While 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 While Loop Invariant Termination: The invariant holds true because i has been 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 While Loop Invariant: Right contains i elements, all of which can be found in $A[mid \dots A.length]$.

Merge Sort Second While Loop Invariant Initialization: The invariant holds true vacuously because i is 0 and right contains no elements.

Merge Sort Second While 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 .

Algorithm 2 MERGE-SORT(*A*)

```

1: procedure MERGE-SORT(A)
2:   if A.length < 2 then
3:     return A
4:   end if
5:   mid = A.length/2
6:   for i = 1 upto mid do
7:     left[left.length] = A[i]
8:   end for
9:   for i = mid upto A.length do
10:    right[right.length] = A[i]
11:  end for
12:  left = Merge-Sort(left)
13:  right = Merge-Sort(right)
14:  A = Merge(left, right)
15:  return A
16: end procedure

17: procedure MERGE(left, right)
18:   var i = 0
19:   var y = 0
20:   var x = 0
21:   while left.length != i and right.length != y do
22:     if left[i] < right[y] then
23:       combined[x] = left[i]
24:       i = i + 1
25:       x = x + 1
26:     end if
27:     if right[y] < left[i] then
28:       combined[x] = right[y]
29:       y = y + 1
30:       x = x + 1
31:     end if
32:   end while
33:   while left.length != i do
34:     combined[x] = left[i]
35:     i = i + 1
36:     x = x + 1
37:   end while
38:   while right.length != y do
39:     combined[x] = right[y]
40:     y = y + 1
41:     x = x + 1
42:   end while
43:   return combined
44: end procedure

```

Merge Sort Second While 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 $left[0 \dots i]$ or $right[0 \dots y]$ in strictly 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 $left[i] \geq right[y]$.

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 $left[i] \geq right[y]$.

Merge Together Second While Loop Invariant: Combined contains x number of elements where $x \geq i$ and those elements are contained in $left[0 \dots i]$ in strictly 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 $left[i] \geq right[y]$.

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 $right[0 \dots y]$ 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 $left[i] \geq right[y]$.

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.

Algorithm 3 HEAP-SORT(*A*)

```

1: procedure HEAPIFY(A, i, total)
2:   var left = i * 2
3:   var right = left + 1
4:   var iPrime = i
5:   if left <= total and A[left] > A[i] then
6:     i = left
7:   end if
8:   if right <= total and A[right] > A[i] then
9:     i = right
10:  end if
11:  if i! = iPrime then
12:    var temp = A[iPrime]
13:    A[iPrime] = A[i]
14:    A[i] = tmp
15:    A = heapify(A, i, total)
16:  end if
17:  return A
18: end procedure
19: procedure HEAP-SORT(A)
20:   var size = A.length
21:   for var i = size/2 downto 1 do
22:     A = heapify(A, i, size)
23:   end for
24:   for var i = size downto 1 do
25:     var tmp = A[1]
26:     A[0] = A[i]
27:     A[i] = tmp
28:     size = size - 1
29:     A = heapify(A, 1, size)
30:   end for
31:   return A
32: end procedure

```

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 Sedgewock and Kevin Wayne. 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.

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 extends Comparable < T >>* fixed this issue. [?]

V. EXPERIMENTAL ANALYSIS

VI. CONCLUSIONS

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: 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 elems 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 primitive value to assert is
     *               strictly greator than the second arguement
     * @param int small an integer primitive 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 primitive value to assert is
     *               stricly less than the second arguement
     * @param int large an integer primitive 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 primitive value to assert is
     *               greator than or equal to the second arguement
     * @param int small an integer primitive 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 argument is less
    *      than or equal to the second argument
    *      @param int small an integer primitive value to assert is
    *      less than or equal to the second argument
    *      @param int large an integer primitive value to assert the
    *      first argument 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 the loop
    , 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 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;
}
}

```

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:
     * @Description: heapify is a helper method for heapSort

```

```

        that keeps the heap in order so that the root node is the
        largest 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 strictly 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 For Loop: TODO: write this
//Invariant for Second For Loop: TODO: write this
public ArrayList<T> heapSort(ArrayList<T> unsorted) {

```

```

    int arrSize = unsorted.size() - 1;
    //Initialization: TODO: write this
    for(int i = arrSize / 2; i >= 0; i--) {
        //Maintanance: TODO: write this
        unsorted = heapify(unsorted, i, arrSize);
        //Termination: TODO: write this
    }
    for(int i = arrSize; i > 0; i--) { //Initialization: TODO
        : write this
        //Maintanance: TODO: write this
        T tmp = unsorted.get(0);
        unsorted.set(0, unsorted.get(i));
        unsorted.set(i, tmp);
        arrSize--;
        unsorted = heapify(unsorted, 0, arrSize);
        //Termination: TODO: write this
    }
    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) {
    //Maintenance: 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) {
    //Maintenance: Our invariant holds true at the
        beginning 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
    terminatio 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 because
    //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 strictly
// 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;
}
}
```