Insertion Sort Analysis in Java

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. More specifically, insertion sort is analysed to determined what the run time might be for sorting large amounts of data on any modern computer.

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. How will this procedure perform when handling high volumes of data? How will it perform when executed on different machines? Because insertion sort is a more basic sorting algorithm, numerous papers and articles have been written answering these two questions.

III. Procedure

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

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:     $i = 2$
3:     **while** i upto $A.length$ **do**
4:         $key = A[i]$
5:         $j = i - 1$
6:         **while** j downto 1 and $key < A[j]$ **do**
7:             $A[j + 1] = A[j]$
8:             $j = j - 1$
9:         **end while**
10:         $A[j + 1] = key$
11:         $i = i + 1$
12:     **end while**
13: **end procedure**

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

Outer-Loop Initialization: The outer-loop invariant holds because A'[1 ... i - 1] and A[1 ... i - 1]

both contain the same one element.

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 maybe in different orders.

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].

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

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.

Inner-Loop Maintenance: At the beginning of each iteration of the loop the inner-loop invariants 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].

Inner-Loop Termination: The negation of the implies that j = A.length and A'[1 ... j] has been entirely traversed and sorted in strictly non-decreasing order which maintains the inner-loop invariant.

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.

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 [NEED CITATION]. It is important to note that the stopwatch class used takes the elapsed real-time between the start of the insertion sort algorithm and the end of the insertion 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
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

B. Problems Encountered

One major issue encountered during the development of this insertion sort was that after completing the sort, A' was completely 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
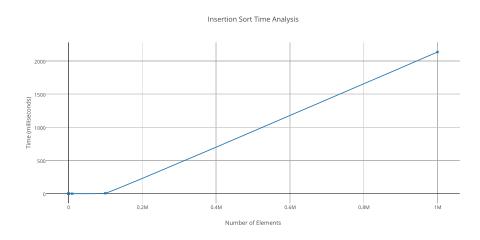
Insertion Sort Time Analysis

Figure 1. Insertion Sort Time Analysis

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.

V. Experimental Analysis

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

VI. Conclusions

References

APPENDIX

Listing 1
TESTDRIVER

```java
/*
*        @Author Preston Stosur−Bassett
*        @Date Jan 25, 2015
*        @Class TestDriver
*        @Description this class will create test data to run through and
    test the other classes in this directory.
*/

import java.util.ArrayList;

public class TestDriver {

        /*
        *        @Description This serves as the Driver function for this
           program, run this class to execute the program
        */
        public static void main(String args[]) {
                //Turn on debugger
                Debug debugger = new Debug();
                debugger.turnOn();

                //Test debug
                debugger.print("Debug is on.");

                //Create test data for sorts class
                //Note that this Array is for testing purposes only, the
                   Algorithm can handle all Comparable Generic Types
                ArrayList<Integer> testList = new ArrayList<Integer>();
                testList = DummyData.runArrayList(0, 10, 0, 10, testList)
                   ;
                //testList = DummyData.identicalElement(0, 1000, 10,
                   testList);
                //ArrayList<String> testList = new ArrayList<String>();
                //testList = DummyData.runArrayList(0, 1000, testList);

                System.out.println("Un−Sorted List");
                System.out.println(testList);

                //Test sort List
                Sort sorter = new Sort<Integer>();
                //Sort sorter = new Sort<String>();
                Stopwatch watchStopper = new Stopwatch();
                testList = sorter.insertion(testList);

                //Print out the results.
                System.out.println("Sorted List");
```

```
                System.out.println(testList);
                System.out.println("Time to complete: "+watchStopper.
                    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;

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
```

6

```
        */
        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();
                        // TODO: Write invariant for this loop
                        while(i > 1) {
                                assert actual.get(i - 1).compareTo(actual
                                    .get(i - 2)) >= 0;

                                i--;
                        }
```

```
            }
        }
}
```

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  start  the  starting  value  to  add  to  the  array
            list
         *      @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  start,  int  end,
            int  min,  int  max,  ArrayList<Integer>  list) {
                Random  random = new  Random();
                // INVARIANT: TODO: Come  up  with  invariant
                // INITIALIZATION: TODO: Assert  initialization
                while(start < end) {
                        // MAINTANANCE: TODO: Assert  Maintanance
                        Integer  intToAdd = new  Integer(random.nextInt((
                            max − min + 1) + min));
                        list.add(intToAdd);

                        //Count  up  on  the  iterator
                        start++;
                }
                //TERMINATION: TODO: Assert  Termination
```

```
                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 start the starting value to add to the array
          list
         *      @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 start, int end,
           ArrayList<String> list) {
                Random random = new Random();
                // TODO: Write loop invariant
                while(start < end) {
                        Integer intToString = new Integer(random.nextInt
                           ((1000000 - 1) + 1));
                        String intString = String.valueOf(intToString);
                        list.add(intString);

                        //Count up on the iterator
                        start++;
                }

                return list;
        }

        /*
         *      @Description: identicalElement will take an element and
          add it to the ArrayList<Integer> for a given amount of times
         *      @param int start the starting value to be used as an
          iterator for the loop
         *      @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 start, int
           end, int element, ArrayList<Integer> list) {
```

```
                        // TODO: Write loop invariant
                        //The element to add over and over again
                        Integer iden = new Integer(element);
                        while(start < end) {
                                list.add(iden);

                                //Count up on the iterator
                                start++;
                        }

                        return list;
                }
}
```

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() {
```

10

```
        long now = System.currentTimeMillis();
        return (now − start) / 1000.0;
    }


}
```

<div align="center">

Listing 5
SORT

</div>

```
/*
 *        @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> 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: Insertion 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> insertion(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] */
                    // TODO: ensurelist is in stricly non decreasing
                        order
                    Integer integerI = new Integer(i);
                    Integer sortedSizeO = new Integer(sorted.size());
                    debugger.assertEquals(sortedSizeO, integerI);
                    debugger.assertEquals(unsorted, sorted);
            }
            return sorted;
        }
}
```