

Insertion Sort Analysis

Melany Diaz

March 13, 2016

Abstract

Using a program made for this report, this report will be validating and analyzing the run-time complexity of Insertion Sort. By implementing the Insertion Sort algorithm in a class, we will analyze and compare the complexity behaviors of best case, worse case, and average case run times for multiple arrays of varying size n .

MOTIVATION AND BACKGROUND

By definition, an algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output. Some algorithms, such as sorting random objects, can seem very intuitive and straight-forward to a person. However, to a computer, sorting a collection of objects can be a complex and strenuous process.

One of the most important aspects of an algorithm is how fast it is, or its time complexity. Algorithm speed is essential for a timely and effective execution of a program. In some cases, the use of an appropriately implemented, well-written algorithm can make the slowest computer execute a program faster than the fastest computer using a poorly used and written algorithm. In a world where instant results are expected from a computer, effective algorithms can be more valuable than the most advanced hardware.

Since the exact speed of an algorithm depends on the details of its implementation, the run-time of an algorithm is typically discussed in terms of the size of its input. For example, if the algorithm must process N objects, it might have a run-time proportional to $O(N^2)$ (i.e. Insertion Sort) or $O(N \lg N)$ (i.e. Merge Sort). However, the execution time of many sorting algorithms can vary due to factors other than the size of the input. For example, if a sorting algorithm must sort a set of objects that are already in sorted order, it would take much less time to re-organize than a set of objects in random order. Consequently, when analyzing the time complexity of an algorithm, one must keep in mind that algorithm's best case, worst case, and average case run-times. Typically, the best case is when an algorithm is given a collection of objects to sort that is already in sorted order, it's worst case is when it is given a collection of objects sorted in the opposite order, and it's average case is when it is given a collection sorted in no particular order.

There exists many different kinds of algorithms used to sort objects such as Bubble Sort, Quick Sort, or Insertion Sort. Each of these have their own benefits and disadvantages, as well as particular moments when it would be more appropriate to use one sort style versus another. Because of this, it is important for a computer scientist to familiarize themselves with the different run-time behaviors and complexities of each. The purpose of this report is to study and analyze the complexity of InsertionSort. By creating a class that implements InsertionSort to sort different objects, we will be validating and comparing the run-time behaviors of the class with the asymptotic run-time complexity of Insertion Sort.

By executing this program and comparing the run-time behaviors of InsertSort's best, worst, and average cases, we hope to gain a better understanding of run-time complexities and asymptotic run-times. We also hope to address the following questions:

- What inputs are required to generate average complexity? How about best and worst cases?
- How do you create your test driver so that it exercises your programs?
- How do you create the sorting class so that it will be extensible and re-usable for future projects?
- What input size n_0 is required to begin to exhibit asymptotic complexity?
- How does measured run-time correspond to operation counts that we use in abstract complexity analysis?

PROCEDURE

There were two things that I had to consider when creating this program: Making a user-friendly program that could be easily used in future projects, and appropriately implementing and calling on an Insertion Sort method.

I knew that this program had to be extensible and re-usable for our future projects. Consequently, I built my program following an Object Oriented Design; any methods specific to a sorting style were created in a class outside of the main class. Thus, the main class is solely responsible of making objects and executing methods on them. I also wanted my program to be very user friendly and easy for the user to have some say and understanding of the results. As such, the main class is also responsible for getting user inputs from a scanner (such as size, or whether the user would like to see a printed version on the arrays) and printing the results (such as the timing of each method)

Here are two examples of user involvement with my program:

```
<terminated> SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 1, 2016, 12:15:23 AM)
Enter number of elements:
9
If the array has less than 20 elements, would you like to see it? Type Y for yes
Y
Average case: [3, 7, 4, 6, 5, 5, 0, 2, 0]
Best case: [0, 0, 2, 3, 4, 5, 5, 6, 7]
Worst case: [7, 6, 5, 5, 4, 3, 2, 0, 0]
-----
TIMES NEEDED TO SORT EACH

Average case: 10997 Nanoseconds

Best case: 3666 Nanoseconds

Worst case: 8798 Nanoseconds
```

A user specifying what size the arrays to be sorted should be and choosing to see them printed out

```
<terminated> SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 1, 2016, 12:16:19 AM)
Enter number of elements:
19
If the array has less than 20 elements, would you like to see it? Type Y for yes
No
-----
TIMES NEEDED TO SORT EACH

Average case: 20527 Nanoseconds

Best case: 6598 Nanoseconds

Worst case: 38122 Nanoseconds
```

A user specifying what size the arrays to be sorted should be and choosing not to see them printed out

Before considering how to implement the InsertSort method to my arrays, I had to find a way to make three arrays, one for the average, best, and worst cases. These are some questions I had to answer in order to correctly create these arrays:

- What inputs are required to generate average complexity?
- What about best-case complexity?

- What about worst-case complexity?

The answer to these three questions lied in how I generated my average complexity array. In order to make the average array used for this, I asked the user to specify how large the array should be, and then proceeded to make an array of that size with random elements (most often integers). After succeeding in making the *average[]* array (a better name would be random) I used it to make the arrays for best and worst case.

For insertion sort, the best case scenario would be an array whose elements were already sorted in increasing order. This called for making an array, called *increasing[]* whose elements were sorted already. To make such an array, I implemented the *InsertionSort* class (more details following) sorted *average[]* and saved it as *increasing[]*.

Similarly, the worst case scenario for insertion sort is an array whose elements are sorted in decreasing order. As above, I made the third array, called *decreasing[]* whose elements were sorted in decreasing order. To make this array, I implemented the *InsertionSortDecreasing* class, and used it to sort *average[]* and saved it under *decreasing[]*.

In order to make the *InsertionSort* and *InsertionSortDecreasing* classes, I closely followed the pseudocode for InsertSort, making sure to correctly implement the invariant, and pre and post conditions.

Pseudocode for Insertion Sort:

```

INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key

```

Loop Invariant for Insertion Sort: At the start of each iteration of the **for** loop of lines 1-8, the subarray *A'*[1..*j* - 1] consists of the elements originally in *A*[1..*j* - 1], but in sorted order.

Input and Output for Insertion Sort:

Input: A sequence of *n* numbers $\langle a'_1, a'_2 \leq \dots \leq a'_n \rangle$.

Output: a permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $\langle a'_1, a'_2 \leq \dots \leq a'_n \rangle$.

Insertion Sort pre-condition: An array, *A*, that must be sorted

Insertion Sort post-condition: An array, *A'*, such that *A'* is a permutation of *A* where $\forall m, n \in 1..n, m < n \implies A'[m] \leq A'[n]$ where *n* is the size of the array.

In order to illustrate the correct usage of program headers and the implementation of my pre/post conditions and invariant in my program, here is an image of my InsertSort method:

```

/**
 * This method uses insertion sort to sort an array in increasing order
 *
 * @param startArray, the array needed to be sorted
 * **written according to the pseudocode precondition**
 * @precondition An original array, A, to be sorted (and j=1)
 *
 * @return startArray, a permutation of the original
 * array but sorted in increasing order
 * **written according to the pseudocode postconditions**
 * @postconditions A permutation of the original array, A', such
 * that for all m,n in {1..n} m < n: A'[m] < A'[n]
 */
public Comparable[] SortInsertion(Comparable[] startArray) {
    for (int j = 1; j < startArray.length; j++) Follows Preconditions/input
    {
        Integer key = (Integer) startArray[j];

        //Insert StartArray[j] into the sorted sequence StartArray[0,...,j-1]
        int i = j-1;

        while (i >= 0 && startArray[i].compareTo(key) == 1) {
            startArray[i+1] = startArray[i];
            i = i-1; For loop follows pseudocode
        } //end while loop

        startArray[i+1] = key;
    } //end for loop
    return (startArray); Follows Postconditions/output
}

```

After completing these tasks, I implemented a method that could time how long it took to sort my three arrays using InsertSort.

TESTING

Program testing is the process used to help identify the correctness, completeness, and quality of a class. The process of testing involves executing a program with the intent of finding errors and bugs. One of our goals for this project was to find a way to create a test driver so that it exercised my programs. In order to do so and test my program, I tried to find any bugs that my classes might encounter when tackling edge cases or unexpected bounds. The following matrix shows the tests I put my program through, my expected result, the actual result, and how I fixed it.

Test	Expected Result	Actual Result	Remedy
Array of length zero*	A new User prompt asking for an input greater than 0	As expected	N/A
Array of negative length*	A new User prompt asking for a positive integer	As expected	N/A
Array with just one element	Since an array of size one is already sorted, I expect that the times for Best, Worst, Average will be the same	After multiple trials, the results were either the same time, or a time differing by (at most) 2000 nanoseconds	Since the times were never more than 2000 nanoseconds apart from each other, I think it is safe to assume that those seconds come from the <i>TimeToSort()</i> method, rather than the actual <i>InsertSort</i> method itself.
Array with null items	Since the user <i>has</i> to input an array size, I don't think a null array could be made. This was also confirmed during the assertion phase of the coding process	As Expected	N/A
Passing a letter or character as Array size	My program will probably throw an exception since the scanner was expecting an integer	As expected	Implemented a while loop to guarantee that the user inputs a valid integer as an array size (See Image)
Replying anything other than "Y" to second prompt	The program will assume that the user does not want to see the array printed out	As expected	N/A

*Further explained in following section.

The following demonstrates how the program would react if the user were to input an

array of negative size or size 0.

```

SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 31, 2016)
Enter number of elements:
0
The number of elements should be positive, try again:
-9
The number of elements should be positive, try again:

```

Receiving an inappropriate input from the user for the array size was not something that had occurred to me as I began coding. Thus, if the user ever inserted any response other than an integer, my program would throw an exception:

```

File Edit Source Refactor Navigate Search Project Run Window Help
SortingAnalysis.java InsertionSort.java InsertionSortDecreasing.java
SortingAnalysis src (default package) SortingAnalysis main(String[]) : void
/*
 * The following code will get an integer, n, as from the user as input
 * it will be used to determine the size of the array
 */
System.out.println("Enter number of elements: ");
Scanner s = new Scanner(System.in);
n = s.nextInt();
while (n <= 0)
{
    System.out.println("The number of elements should be positive, try again");
    s = new Scanner(System.in);
    n = s.nextInt();
}

<terminated> SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 31, 2016, 11:33:47 PM)
Enter number of elements:
a
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at SortingAnalysis.main(SortingAnalysis.java:86)

```

After realizing this, I implemented a solution that involved a guaranteed integer from the user for the array size:

```

Java - SortingAnalysis/src/SortingAnalysis.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

SortingAnalysis.java | InsertionSort.java | InsertionSortDecreasing.java
SortingAnalysis | src | (default package) | SortingAnalysis | main(String[]) : void

/*
 * The following code will get an integer, n, as from the user as input
 * it will be used to determine the size of the array
 */
System.out.println("Enter number of elements: ");
Scanner s = new Scanner(System.in);
while(!s.hasNextInt()) {
    s.next();
    System.out.println("The number of elements must be a postive integer, tr
} int n = s.nextInt();

while (n <= 0)
{
    System.out.println("The number of elements should be positive, try again
    s = new Scanner(System.in);
    while(!s.hasNextInt()) {
        s.next();
    }
    n = s.nextInt();
}

Console
SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 31, 2016, 11:32:31 PM)
Enter number of elements:
a
The number of elements must be a postive integer, try again:
$
The number of elements must be a postive integer, try again:
3
If the array has less than 20 elements, would you like to see it? Type Y for yes

```

PROBLEMS ENCOUNTERED

As with any programming project, a programmer must be able to keep track of any problems encountered and of the solutions found to encounter them. Following is a description of the problems I found while programming my classes and how I chose to tackle them.

Copying the Arrays In order to make an array for best, worst, and average cases I was making an array full of random numbers (and calling it my average array) and then using that array to make my increasing array (best case) and decreasing array (worst case)

Originally I was setting my increasing and decreasing arrays equal to my average array and working from there. It didn't take me long to realize that I was making my code work with the same array, called by three different names, rather than working with three different arrays, each with their own name.

After reading the Array Documentation, I learned that for what I was doing, I shouldn't be setting arrays equal to each other, but rather using the *copyOf()* method. Thus, I learned that I should replace a line such as

```
increasing = average;
```

with the more appropriate method

```
increasing = Arrays.copyOf(average, n);
```

Where *n* is the number of elements in the array.

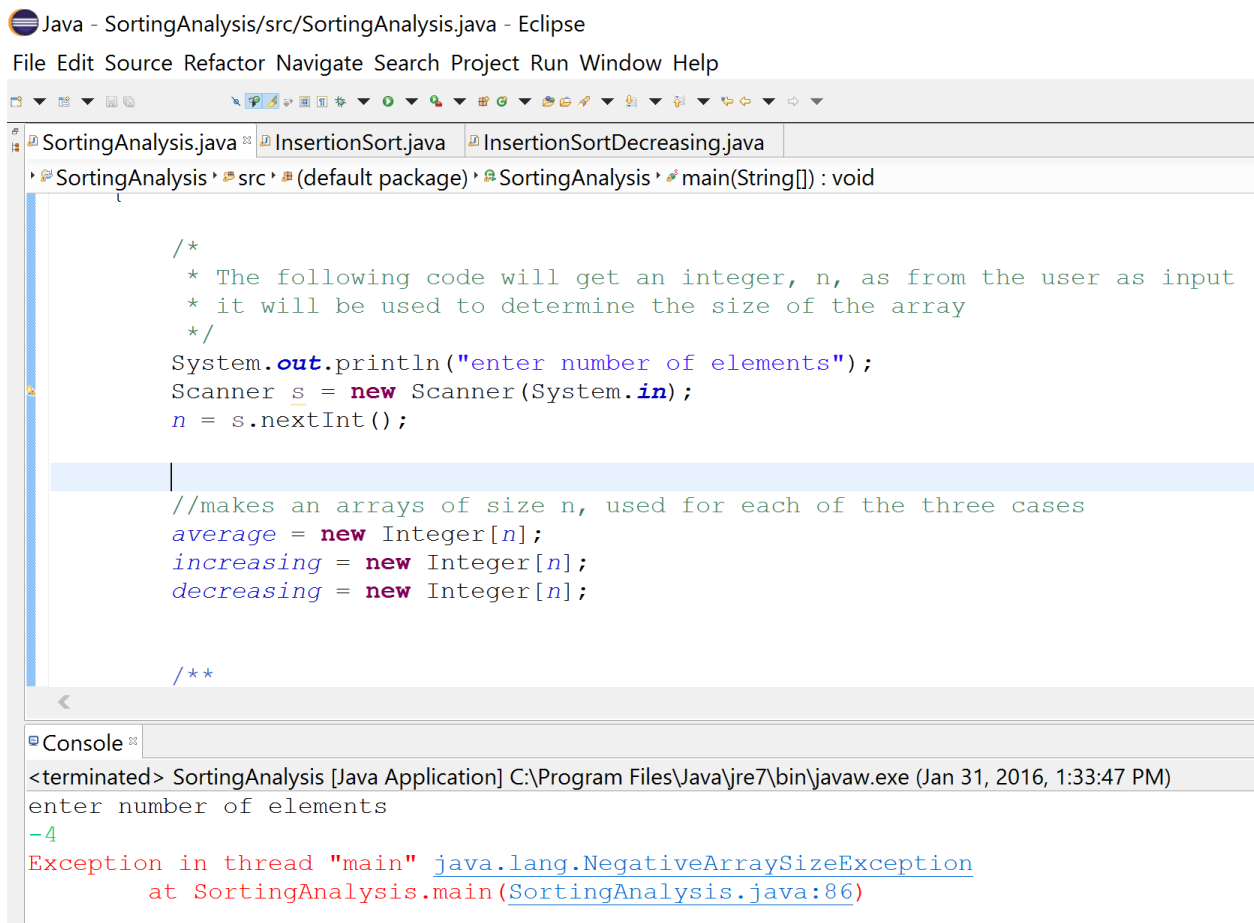
Printing out the Arrays for testing In order to see if my program was working, I thought it would be practical to print out each array to see how they had handled the sorting method. Originally, I tried to write my own print method using a for loop. After a while of realizing that was not a very good idea, I consulted the Array documentation and found that there already exists a *toString()* method for arrays. Naturally, I decided to use that.

After testing whether my code worked or not, it wasn't entirely necessary to print out the arrays. However, I thought seeing the arrays was very helpful, and decided to give the user the option to see them or not. Since the arrays can get very large, I wanted to limit the display option to arrays of size smaller than 20. Even though finding the *toString()* method was originally supposed to be part of my testing and debugging phase, I'm glad I discovered it, since I used it later on for this component of my program.

Dealing with Arrays of Negative or Empty Size The precondition to Insertion-Sort is that you must have an array that must be sorted. Since I implemented my code so that the user could decide how many elements the arrays should have, I risked the user ordering an array of negative size, or an empty array (an array of size 0).

Before I found a solution to this, I would run into null-pointer exceptions and array size

exceptions if the user were to make the array of these sizes.



The screenshot shows the Eclipse IDE interface. The title bar reads "Java - SortingAnalysis/src/SortingAnalysis.java - Eclipse". The menu bar includes "File", "Edit", "Source", "Refactor", "Navigate", "Search", "Project", "Run", "Window", and "Help". The toolbar contains various icons for file operations and development tools. The Package Explorer on the left shows the project structure: "SortingAnalysis" (src) containing "(default package)" with "SortingAnalysis" and "main(String[]): void". The Editor window displays the source code for "SortingAnalysis.java". The code includes a comment block explaining the input, followed by code to read an integer 'n' from the user and create three Integer arrays of size 'n'. The console at the bottom shows the execution output: "<terminated> SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 31, 2016, 1:33:47 PM)", the prompt "enter number of elements", the user input "-4", and a red error message: "Exception in thread \"main\" java.lang.NegativeArraySizeException at SortingAnalysis.main(SortingAnalysis.java:86)".

```
Java - SortingAnalysis/src/SortingAnalysis.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

SortingAnalysis.java | InsertionSort.java | InsertionSortDecreasing.java
SortingAnalysis | src | (default package) | SortingAnalysis | main(String[]): void

/*
 * The following code will get an integer, n, as from the user as input
 * it will be used to determine the size of the array
 */
System.out.println("enter number of elements");
Scanner s = new Scanner(System.in);
n = s.nextInt();

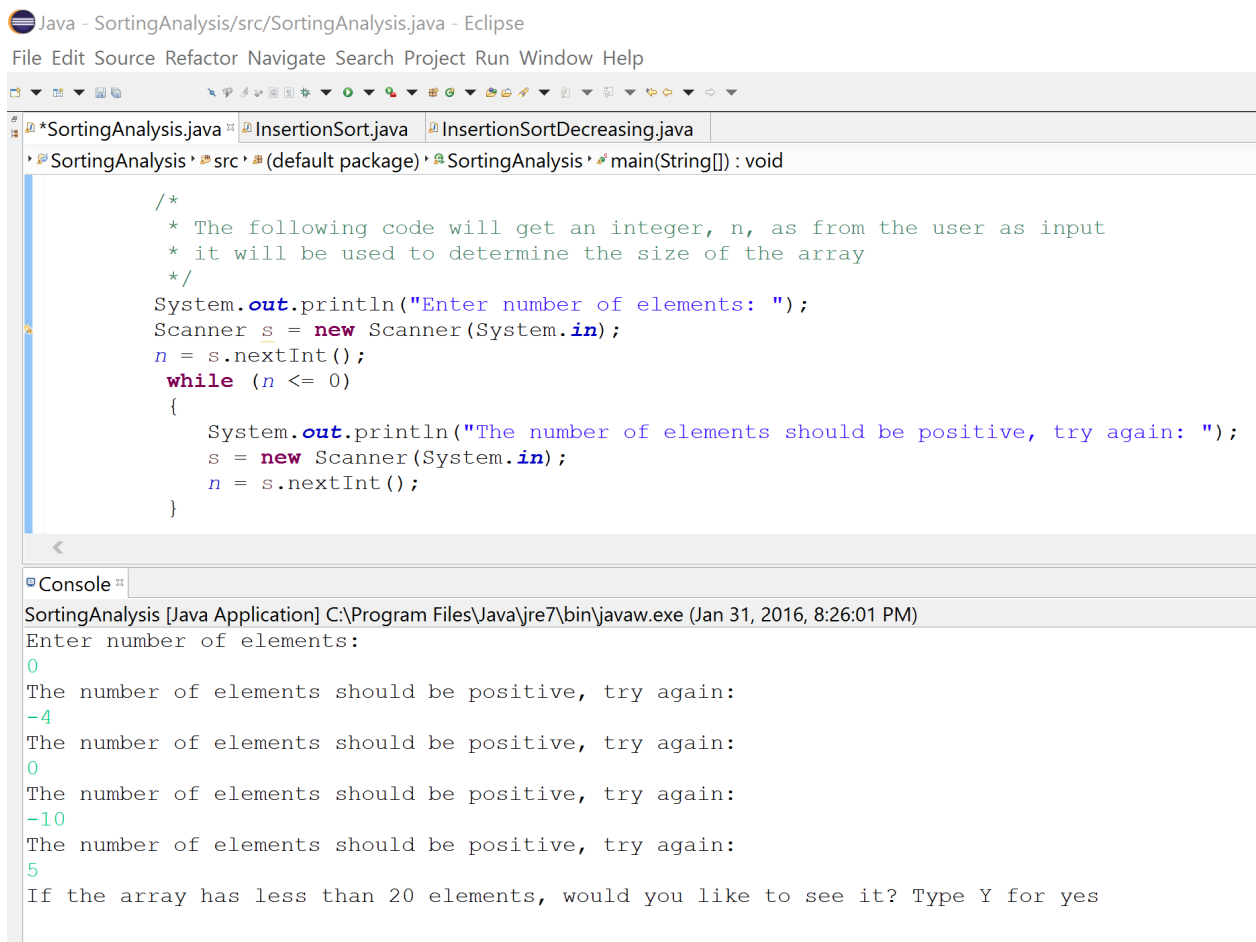
//makes an arrays of size n, used for each of the three cases
average = new Integer[n];
increasing = new Integer[n];
decreasing = new Integer[n];

/**

<terminated> SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 31, 2016, 1:33:47 PM)
enter number of elements
-4
Exception in thread "main" java.lang.NegativeArraySizeException
    at SortingAnalysis.main(SortingAnalysis.java:86)
```

In order to avoid this problem, I decided to implement a while loop that would guarantee

that my user inputed a number greater than zero. The results were as expected.



```
Java - SortingAnalysis/src/SortingAnalysis.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

/*
 * The following code will get an integer, n, as from the user as input
 * it will be used to determine the size of the array
 */
System.out.println("Enter number of elements: ");
Scanner s = new Scanner(System.in);
n = s.nextInt();
while (n <= 0)
{
    System.out.println("The number of elements should be positive, try again: ");
    s = new Scanner(System.in);
    n = s.nextInt();
}

SortingAnalysis [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 31, 2016, 8:26:01 PM)
Enter number of elements:
0
The number of elements should be positive, try again:
-4
The number of elements should be positive, try again:
0
The number of elements should be positive, try again:
-10
The number of elements should be positive, try again:
5
If the array has less than 20 elements, would you like to see it? Type Y for yes
```

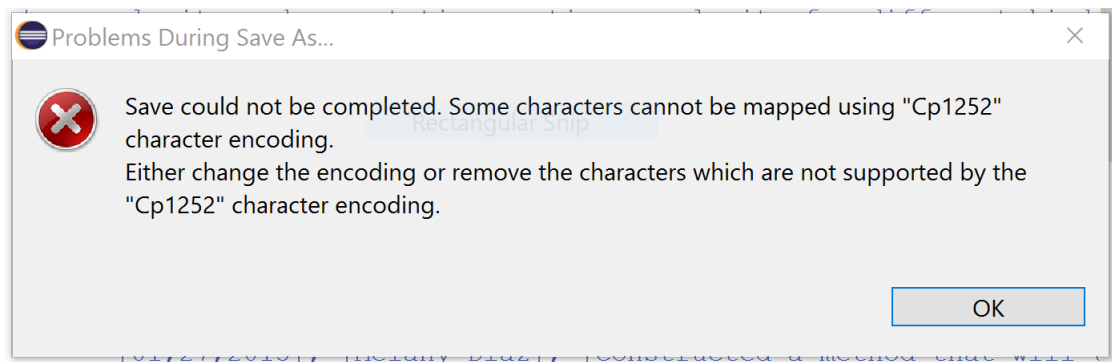
Changing from Integers to Comparables I originally coded my program so that it would use arrays full of integers ranging from 0-9. I did this strategically so that I could use inequalities in my comparisons and logically organize the arrays from the smallest integer to the largest, as well as easily fill my average array with random integers. The assignment specifications, however, required us to use Comparable Objects, and use the compareTo() method in our InsertionSort. After confirming my original code worked, I tackled the task of changing my arrays to be from int[] to Comparable[] and accordingly change all int variables to become Integer variables.

The biggest issue I encountered while doing this was not using compareTo() rather than inequalities (as I had previously expected) but it was to fill my arrays with random Comparable Objects. Originally I had used a random number generator to make my arrays with random integers (which I could then use as my average, best, and worst cases) However,

there does not exist a random comparable object generator, and finding a way to implement an array with random Comparable Objects was not as straightforward as I had hoped.

In the interest of time, I decided the most practical solution to this problem was to use the fact that Integer is a Comparable Object, and implemented my Comparable arrays to use Integers as their elements. Had I had more time or incentive to find an accurate way to fill an array with comparable objects, I'm sure that I could have. However, due to lack of time, I reasoned that using Integer values would suffice for the project.

Problem Saving Classes After implementing my code to demonstrate the use of pre and postconditions in my methods, I kept receiving an error that would prevent me from saving or running my class.



After a little troubleshooting, I learned that this error showed whenever there was a character in the code that Java couldn't read. After searching through my class, I found that in the comments, I was using \forall and \exists in my comments to talk about my conditions.

```
/**
 * This method uses insertion sort to sort an array in increasing order
 *
 * @param startArray, the array needed to be sorted
 * **written according to the pseudocode precondition**
 * @precondition An original array, A, to be sorted (and j=1)
 *
 * @return startArray, a permutation of the original
 * array but sorted in increasing order
 * **written according to the pseudocode postconditions**
 * @postconditions A permutation of the original array, A', such
 * that  $\forall m, n \in \{1..n\} \ m < n: A'[m] < A'[n]$ 
 */
```

After I replaced those two characters with "for all" and "there exists" I was able to save an run my classes without any problems.

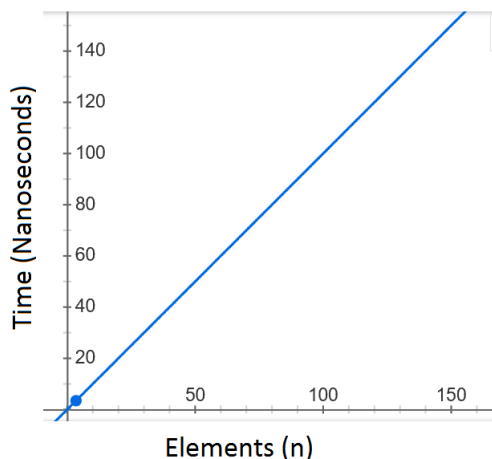
EXPERIMENTAL ANALYSIS AND ASYMPTOTIC RUN-TIME COMPARISON

After confirming that the program does meet every post condition, and that the program produces the required results, the last thing to do is compare the run-time complexity of the program with the asymptotic run-time complexity of InsertSort. To compare the two run-times, I ran my program for different input sizes, n . In order to gain an accurate estimate of the timings of each run, I gathered the average of three runs per each n (see Appendix D). Due to the scaling of the graph, and the n elements chosen to test Insertion Sort, my graphs do not appear as expected (A group of plotting point bellow n_0 that seem hap-hazardous and after n_0 a more persistent tie to the asymptotic line). Because of this outcome, I decided to use Microsoft Excel's *trend line* function, which takes a set a data points and estimates their corresponding function as accurately as possible. Using this, I found that even though my data points weren't visually as expected, mathematically they were. The following subsections will describe my results for the average, best, and worst cases of Insert Sort.

n	Average	Best	Worst
50	100682.3333	8064.667	88343
100	490214	19305	814497
500	3421477	77466	7233707
1000	7896448	34457	1032479
5000	19968754	54739	18868339
10000	77620512	103370	79537293
50000	2942304830	596263	3168755939
100000	17283158032	1658076	19209443187

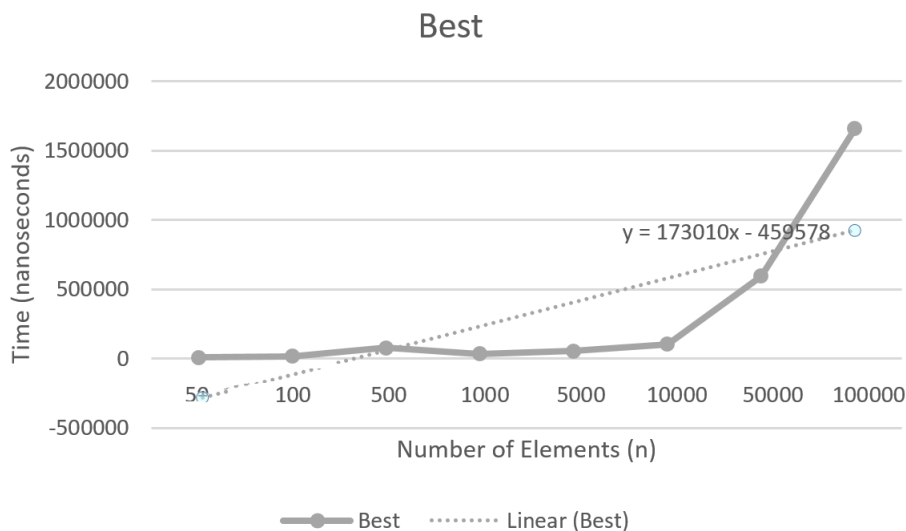
Best Case

Asymptotically, the best case input for insertion sort is an array that is already sorted in increasing order. In this case, Insert Sort typically has linear running time (i.e. $O(cn)$) Where c is a constant.



Visual representation of best case: $O(n)$

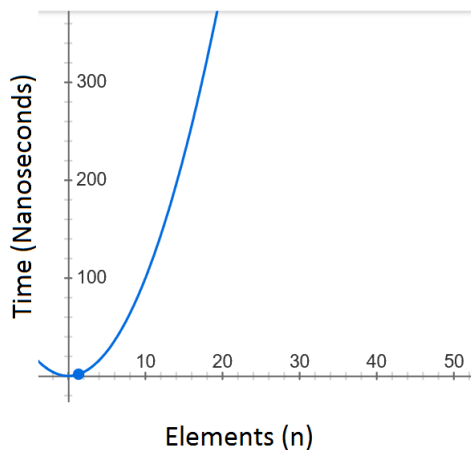
After plotting the best case data points found using the program, I found the trend line of the plotting to be $y = 173010n - 459578$ which agrees with the asymptotic run-time, implying that $c = 173010$.



Just by visually estimating from the graph and data table(s), I would confidently assume that n_0 happens around $n = 50$.

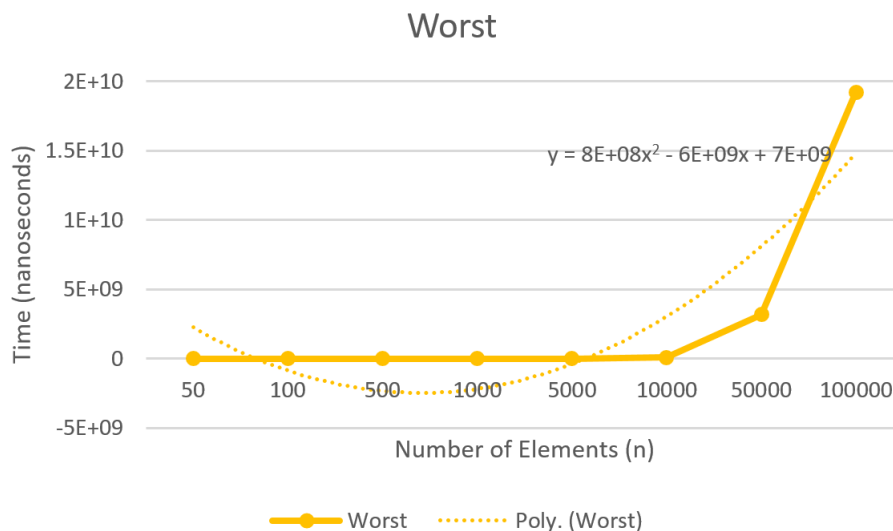
Worst Case

Asymptotically, The worst case input for insertion sort is an array that is sorted in the opposite order, in this case decreasing order. In this case, every iteration of the method will scan and shift the entire sorted subarray before continuing, thus producing a quadratic running time (i.e. $O(n^2)$) Where c is a constant.



Visual representation of worst case: $O(n^2)$

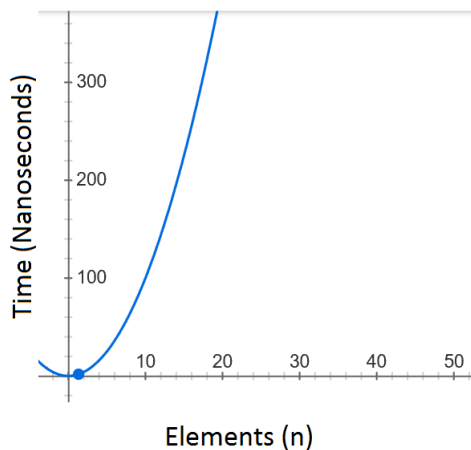
After plotting the worst case data points found using the program, I found the trend line of the plotting to be $y = (8E + 08)n^2 - (6E + 09)n + 7E + 09$ which agrees with the asymptotic run-time, implying that $c = 8E + 08$.



Just by visually estimating from the graph and data table(s), I would confidently assume that n_0 happens around $n = 100$.

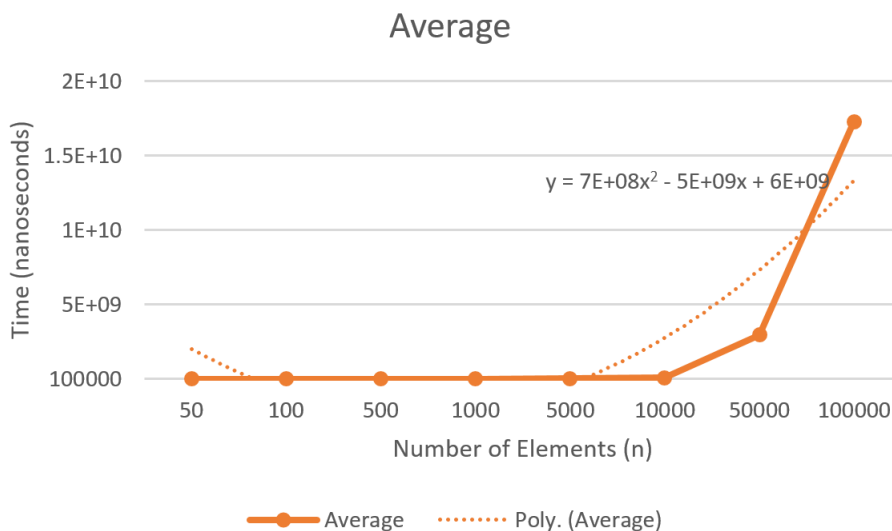
Average Case

The asymptotic average case for insertion sort is typically also quadratic, due to the implementation of the method. Thus, like with the worst-case behavior, the asymptotic running time of the average case is $O(cn^2)$, Where c is a constant.



Visual representation of worst case: $O(n^2)$

After plotting the average case data points found using the program, I found the trend line of the plotting to be $y = 7E + 09N^2 - 5E + 09N + 6E + 09$ which agrees with the asymptotic run-time, implying that $c = 7E + 0$.



As with Worst-Case, just by visually estimating from the graph and data table(s), I would confidently assume that n_0 happens around $n = 100$.

CONCLUSIONS

After constructing an InsertSort program and using it to analyze its run-time behaviors, we were able to compare its results to the asymptotic run-time complexities for best, worst, and average cases. With a better understanding of run-time complexities and asymptotic run-times, we have learned how to generate different complexities, implement and scrutinize a class so that it surpasses an intensive testing phase, and have extensively understood the Insertion Sort algorithm and its corresponding run-time complexities.

APPENDIX A: Main Class

The following is the Java class written for the main class. This uses the InsertSort and InsertSortDecreasing classes provided for in the following appendices. This class is responsible for making the array that will be sorting them, and printing the times it took to sort them using the other classes.

```

import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;

// import java.util.ArrayList;

/**
 * COMP 215: Design and Analysis of Algorithms:
 * Programming Assignment 1<br>
 *
 * The <code>SortingAnalysis</code> class provides a
 * main method for a program that validates the run-time
 * complexity and asymptotic run-time complexity for
 * different kinds of sort methods.
 *
 * This code will be used to compare the worst case,
 * best case, and average case run times for Insertion Sort
 *
 * <br> <br>
 * Created: <br>
 * [01/24/2016], [Melany Diaz]<br>
 * With assistance from: Dr. Gerry Howser<br>
 *
 * Modifications: <br>
 * [01/24/2016], [Melany Diaz],
 * [implemented code to sort in increasing and
 * decreasing orders, made an array of user
 * inputted size of random integers]<br>
 *
 * [01/27/2016], [Melany Diaz],
 * [discovered arrays have a toString
 * method, so changed how to print the arrays,
 * refactored main class and update documentary,
 * added timing mechanisms]<br>
 *

```

```

*          [01,31,2016], [Melany Diaz],
*          [perfected comments and headers(specifying
*          pre and post conditions), added assert
*          statements and changed the program so that
*          it wouldn't just compare integers, but any
*          comparable object, had the user choose if
*          the array should be printed, confirmed use of
*          preconditions, postconditions and invariants]
*
*  @author [Melany Diaz]    [with assistance from Dr. Gerry Howser
*  ]
*/

```

```

public class SortingAnalysis
{
    /*
     * Instance Variables
     */

    //used to make an array of random elements
    static Random generator = new Random();
    int randomInt;

    //User-input for how many elements the array should have
    static int n;

    //User-input for if the array should be displayed or not
    static String answer;
    static Boolean show = false;

    //the average, best, and worst case arrays
    static Comparable[] average;
    static Comparable increasing[];
    static Comparable decreasing[];

    //Objects of each sorting class
    static InsertionSort insertionSort;
    static InsertionSortDecreasing insertionSortDecreasing;

    /**
     * The main function initiates execution of this program.
     * Makes the worst, best, and average arrays and then times
     how

```

```

* long it takes to sort them in ascending order.
* Will print out each array if they are smaller than 20
  elements
* and will print the amount of time it took to sort them.
* The user will specify how big each array will be.
*
* @param String[] args not used in this program
**/
public static void main(String[] args)
{

    /*
     * The following code will get an integer, n, as from the
       user as input
     * it will be used to determine the size of the array
     */
    System.out.println("Enter number of elements: ");
    Scanner s = new Scanner(System.in);
    while(!s.hasNextInt()) {
        s.next();
        System.out.println("The number of elements
            must be a positive integer, try again: ");
    } int n = s.nextInt();

    while (n <= 0)
    {
        System.out.println("The number of elements should
            be positive, try again: ");
        s = new Scanner(System.in);
        while(!s.hasNextInt()) {
            s.next();
        }
        n = s.nextInt();
    }

    /*
     * Used for testing, the user can decide if the array
       will be printed or not
     */
    System.out.println("If the array has less than 20
        elements, would you like to see it? Type Y for yes");
    Scanner display = new Scanner(System.in);
    answer = display.nextLine();
    if (answer == "Y")

```

```

        show = true;

//makes an arrays of size n, used for each of the three
cases
average = new Comparable[n];
increasing = new Comparable[n];
decreasing = new Comparable[n];

/**
 * The following three blocks will make the best, worst,
 * and average cases
 */

/*
 * Make the array used for average case and fill the
 * average array
 * with n random objects.
 */
for(int i = 0; i < n; i++)
    average[i] = generator.nextInt(n);

//if the array is small enough for the user to benefit in
seeing it,
//then we will display it. If it is too big, then it is
not worth seeing it
if (show){
    if(n <20)
        System.out.println("Average_case:_ " + Arrays.
            toString(average));
    else
        System.out.println("Array_is_too_big_to_display");
}

/*
 * The following code uses InsertionSort and its methods
 * to find the best case scenario. InsertionSort will
 * sort the array items in an increasing order.
 */
insertionSort = new InsertionSort();

//make a new array with the same elements but in

```

```

        increasing order
increasing = Arrays.copyOf(average,n);
increasing = insertionSort.SortInsertion(increasing);
if(show && n < 20)
    System.out.println("Best_case:_"+ Arrays.toString
        (increasing));

/*
 * The following code uses InsertionSortDecreasing and its
 * methods
 * to find the best case scenario. InsertionSortDecreasing
 * will
 * sort the array items in decreasing order.
 */
insertionSortDecreasing = new InsertionSortDecreasing();

//make a new array with the same elements but in
    decreasing order
decreasing = Arrays.copyOf(average,n);
decreasing = insertionSortDecreasing.SortDecreasing(
    decreasing);
if(show && n < 20)
    System.out.println("Worst_case:_"+ Arrays.
        toString(decreasing));

/*
 * The following will print the times required to sort
 * three arrays with the same
 * elements of size n, one in already increasing order(
 * best case)
 * one in already decreasing order (worst case)
 * and one with the elements sorted randomly (average case
 * )
 */
System.out.println("
    _____");
System.out.println("TIMES_NEEDED_TO_SORT_EACH");

for (int i = 0; i < 3; i++){
    Comparable[] toTime;
    System.out.println("");
    if (i == 0){
        toTime = Arrays.copyOf(average,n);
    }
}

```

```
        //System.out.println(Arrays.toString(toTime));
        System.out.println("Average_case:_" +
            insertionSort.TimeToSort(toTime) + "_" +
            Nanoseconds");}
    else if (i == 1){
        toTime = Arrays.copyOf(increasing,n);
        //System.out.println(Arrays.toString(toTime));
        System.out.println("Best_case:_" +
            insertionSort.TimeToSort(toTime)+ "_" +
            Nanoseconds");}
    }
    else {
        toTime = Arrays.copyOf(decreasing,n);
        //System.out.println(Arrays.toString(toTime));
        System.out.println("Worst_case:_" +
            insertionSort.TimeToSort(toTime) + "_" +
            Nanoseconds");}
    }

}

} //end main

} //end class
```

Appendix B: InsertionSort Class

The following is the Java class written for Insertion Sort. This class provides a method that uses the Insertion Sort algorithm. The method produces a permutation of the array to be sorted with all of the elements sorted in increasing order.

Written in accord with the corresponding pseudocode, this class requires an array that must be sorted and will produce another array, based off of the original and with the same elements, but in sorted order: going from lowest to highest.

Since the assignment requires a timing mechanism to be able to time how long it takes to execute an InsertSort method, this class provides a second method, which times the execution process.

```
import java.util.ArrayList;
import java.util.Arrays;

/**
 * COMP 215: Design and Analysis of Algorithms:
 *      Programming Assignment 1<br>
 *
 *
 * The <code>InsertionSort</code> class provides an algorithm
 * that will sort a given
 * array into increasing order using the InsertionSort algorithm.
 * This is made for a program that validates the run-time
 * complexity and asymptotic run-time complexity for different
 * kinds of sort methods
 * of Insertion Sort
 *
 * This code will be used to compare the worst case, best case,
 * and average case run times
 * for Insertion Sort
 *
 * <br> <br>
 * Created: <br>
 *      [01/24/2016], [Melany Diaz]<br>
 *
 *      With assistance from: Dr. Gerry Howser<br>
 *
 * Modifications: <br>
 *      [01/24/2016], [Melany Diaz],
 *      [constructed the class and the first methods]<br>
```

```

*
*      [01,27,2015], [Melany Diaz],
*      [constructed a method that will time how long it takes
*      to run insertionSort]<br>
*
*      [01,31,2016], [Melany Diaz],
*      [perfected comments and headers(specifying pre and post
*      conditions and invariants), added assert statements and
*      changed the program so that it wouldn't just compare
*      integers, but any comparable object]
*
*      @author [Melany Diaz] [with assistance from Dr. Gerry Howser]
*/

public class InsertionSort
{
    // instance variables
    Comparable[] startArray;

    //Variables used to determine how long each method takes
    //to execute
    private long startTime;
    private long endTime;
    private long duration;

    /**
     * Constructs a new object of this class.
     */
    public InsertionSort()
    {
        // initializing instance variables
        this.startArray = startArray;
    }

    // Methods

    /**
     * This method uses insertion sort to sort an array in
     * increasing order
     *
     * @param startArray, the array needed to be sorted
     * **written according to the pseudocode precondition**
     * @precondition An original array, A, to be sorted (and j=1)
     */

```



```

* @return startArray, a permutation of the original
* array but sorted in increasing order
* **written according to the pseudocode postconditions**
* @postcondtions A permutation of the original array, A',
* such
* that for all m,n in {1..n} m < n: A'[m] < A'[n]
*
*/
public Comparable[] SortInsertion(Comparable[] startArray) {

    for (int j = 1; j < startArray.length; j++)
    {
        Integer key = (Integer) startArray[j];

        //Insert StartArray[j] into the sorted sequence
        //StartArray[0,...,j-1]
        int i = j-1;

        while(i >= 0 && startArray[i].compareTo(key) == 1)
        {
            startArray[i+1] = startArray[i];
            i = i-1;
        } //end while loop

        startArray[i+1] = key;
    } //end for loop
    return (startArray);
}

/**
* This method times how long it takes to run insertion sort
* for an array passed as a parameter
*
* @param Array Array, the array who is to be sorted
* @return long time, the amount it takes to sort that array
*/

public long TimeToSort(Comparable[] Array){
    long start= System.nanoTime();
    this.SortInsertion(Array);
    long end = System.nanoTime();
    long duration = end - start;
    return duration;
}
}

```

Appendix C: InsertionSortDecreasing Class

The following is the Java class written for Insertion Sort in Decreasing order. This class provides a method that uses the InsertionSortDescending algorithm. The method produces a permutation of the array to be sorted with all of the elements sorted in decreasing order.

Written in accord with the corresponding pseudocode, this class requires an array that must be sorted and will produce another array, based off of the original and with the same elements, but in sorted order: going from highest to lowest.

```
import java.util.ArrayList;

/**
 * COMP 215: Design and Analysis of Algorithms: Programming
 * Assignment 1<br>
 *
 * The <code>InsertionSortDecreasing</code> class provides an
 * algorithm that will sort a given
 * array into decreasing order using the Insertion Sort algorithm
 * . This is made for a program
 * that validates the run-time complexity and asymptotic run-time
 * complexity for different
 * kinds of sort methods. This will sort in descending order
 *
 * This code will be used to compare the worst case, best case,
 * and average case run times
 * for Insertion Sort
 *
 * <br> <br>
 * Created: <br>
 * [01/24/2016], [Melany Diaz]<br>
 * With assistance from: Dr. Gerry Howser<br>
 *
 * Modifications: <br>
 * [01/24/2016], [Melany Diaz],
 * [constructed the class and the first methods]<br>
 *
 * [01,27,2015], [Melany Diaz],
 * [constructed a method that will time how long it takes
 * to run insertionSort]<br>
 *
 * [01,31,2016], [Melany Diaz],
 * [perfected comments and headers (specifying pre and post
```

```

*          conditions), added assert statements and changed
*          the program so that it wouldn't
*          just compare integers, but any comparable object]
*          and invariants
*
*  @author [Melany Diaz][with assistance from Dr. Gerry Howser]
*/

public class InsertionSortDecreasing
{
    // instance variables
    Comparable[] startArray;

    /**
     * Constructs a new object of this class.
     */
    public InsertionSortDecreasing()
    {
        // initializing instance variables
        this.startArray = startArray;
    }

    // Methods

    /**
     * This method uses insertion sort to sort an array in
     * decreasing order
     *
     * @param startArray, the array needed to be sorted
     * **written according to the pseudocode precondition**
     * @precondition An original array, A, to be sorted (and j=1)
     *
     * @return startArray, a permutation of the original
     * array but sorted in increasing order
     * **written according to the pseudocode postconditions**
     * @postconditions A permutation of the original array, A',
     * such
     * that for all m,n in {1..n} m > n: A'[m] > A'[n]
     *
     */
    public Comparable[] SortDecreasing(Comparable[] startArray) {
        for (int j = 1; j < startArray.length; j++)
        {

```

```
Integer key = (Integer) startArray[j];

//Insert StartArray[j] into the sorted sequence
//StartArray[0,...,j-1]
int i = j;

while(i>0 && startArray[i-1].compareTo(key)==-1){
    startArray[i] = startArray[i-1];
    i = i-1;
} //end while loop

startArray[i] = key;
} //end for loop
return (startArray);
}
```

Appendix D: Data Analysis for Comparisons

In order to gain an accurate estimate of the timings of each run, I gathered the average of three runs per each n . The following shows the data received for each run and how the average was estimated.

	n	first run	second run	third run	average of runs
Average Case	50	152490	69647	79910	100682
Best Case	50	9530	7332	7332	8065
Worst Case	50	126830	124639	13560	88343
Average Case	100	441339	554973	474329	490214
Best Case	100	15395	27859	14662	19305
Worst Case	100	797636	844555	801301	814497
Average Case	500	2818851	4208116	3237463	3421477
Best Case	500	61582	63781	107036	77466
Worst Case	500	9020323	5535798	7144999	7233707
Average Case	1000	855552	9061378	13772414	7896448
Best Case	1000	77711	13196	12463	34457
Worst Case	1000	1623130	764645	709661	1032479
Average Case	5000	19641782	19240765	21023715	19968754
Best Case	5000	46919	66714	50585	54739
Worst Case	5000	18514975	19061883	19028160	18868339
Average Case	10000	77607268	77627268	77627000	77620512
Best Case	10000	103370	103370	103370	103370
Worst Case	10000	79557576	82157956	76896347	79537293
Average Case	50000	2957817674	2901758831	2967337986	2942304830
Best Case	50000	600400	600426	587963	596263
Worst Case	50000	3197818817	3276286975	3032162025	3168755939
Average Case	100000	16503799400	16902719220	18442955475	17283158032
Best Case	100000	1475040	1208184	2291003	1658076
Worst Case	100000	18229827622	19535154260	19863347680	19209443187

REFERENCES

Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

Lbackstrom. "The Importance of Algorithms Topcoder." The Importance of Algorithms Topcoder. Accessed January 31, 2016. <https://www.topcoder.com/community/data-science/data-science-tutorials/the-importance-of-algorithms/>.

"Nairaland Forum." The Importance Of Software Testing And Not Just Software Programming. April 01, 2008. Accessed January 31, 2016. <http://www.nairaland.com/124053/importance-software-testing-not-just>.