Programming Assignment 2: Insertion Sort Analysis

Melany Diaz

March 12, 2016

Abstract

This report will be validating and analyzing the run-time complexity of Insertion Sort, Merge Sort, Heap Sort, and Quick Sort. We hope to discover the conditions, if any exist, where it would be most efficient to use one sort versus another by analyzing their best, worst, and random case run-times for multiple input sizes.

MOTIVATION AND BACKGROUND

Sorting data is a problem that often occurs in computer applications, like when sorting through a file system or easing the process of searching for a value. Different factors can affect the time required to sort, thus many algorithms exist to solve the task of organizing comparable values. Because the exact speed of an algorithm depends on the details of the data that must be sorted, the run-time of algorithms 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 linearly proportional to n, which would look like $O(n)$. Some other run-times proportional to n are exponential, polynomial or logarithmic. Yet the run-time of an algorithm isn't solely dependent on the size of the input; the execution time of many sorting algorithms can vary due to pre-existing order of the elements that must be sorted. For example, if a sorting algorithm must sort a set of objects that are already in sorted order, it could 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 random 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 random case is when it is given a collection sorted in no particular order.

Some sorting algorithms include Insertion Sort, Merge Sort, Heap Sort, and Quick 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. For example, Insertion Sort works fastest when there isn't much data, yet for larger amounts of input, the other sorting algorithms surpass the speed of Insertion Sort. Because of this, it is important for computer scientists to familiarize themselves with the different run-time behaviors and complexities of these sorting algorithms.

The purpose of this report is to study and analyze the complexity of Insertion Sort, Merge Sort, Heap Sort, and Quick Sort. By validating and comparing the run-time complexities of the best, worst, and random cases of each, we hope to gain a better understanding of these algorithm's run-times. Through analyzing each case, we will discover the conditions, if any exist, that would make one sort algorithm more efficient than the others. We also hope to discover at what input size n_0 will the run-time behaviors of our implementations begin to assimilate to that algorithm's asymptotic run-time.

As a final goal, we gain to discover which algorithm has the biggest "leading constant." Due to its reputation for being the surpassed in speed by other algorithms when n gets large, we predict, *a priori*, that Insertion Sort will have the biggest leading constant.

PROCEDURE

In order to implement Insert Sort, Merge Sort, Heap Sort and Quick Sort, it is important to under stand some details about each. Some of those details include their pre and post conditions, their invariants, and those invariant's properties.

Insert Sort

Our first algorithm, Insert Sort, is best known for being efficient with small input sizes. Insertion Sort moves from the beginning of the list to the end of it exactly once. During the process of sorting, if a value is out of place, it is moved back to where it should. Note its following conditions.

Input: A sequence of n elements $< a_1, a_2, ..., a_n >$.

Output: A permutation $< a'_1, a'_2, ..., a'_n >$ of the input sequence such that $a_1 \leq a_2 \leq ... \leq a_n$.

The pseudo code for Insert Sort is as follows.

INSERTION–SORT(A)

```
for j = 2 to A.length
        key = A[j]
        //insert A[j] into the sorted sequence A[1..j−1]
        i = j−1
        while i > 0 and A[i] > key
                A[i+1] = A[i]
                i = i − 1
        A[i+1] = key
```

It is important to note the loop invariant for insertion sort and it's correctness.

Loop invariant: 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.

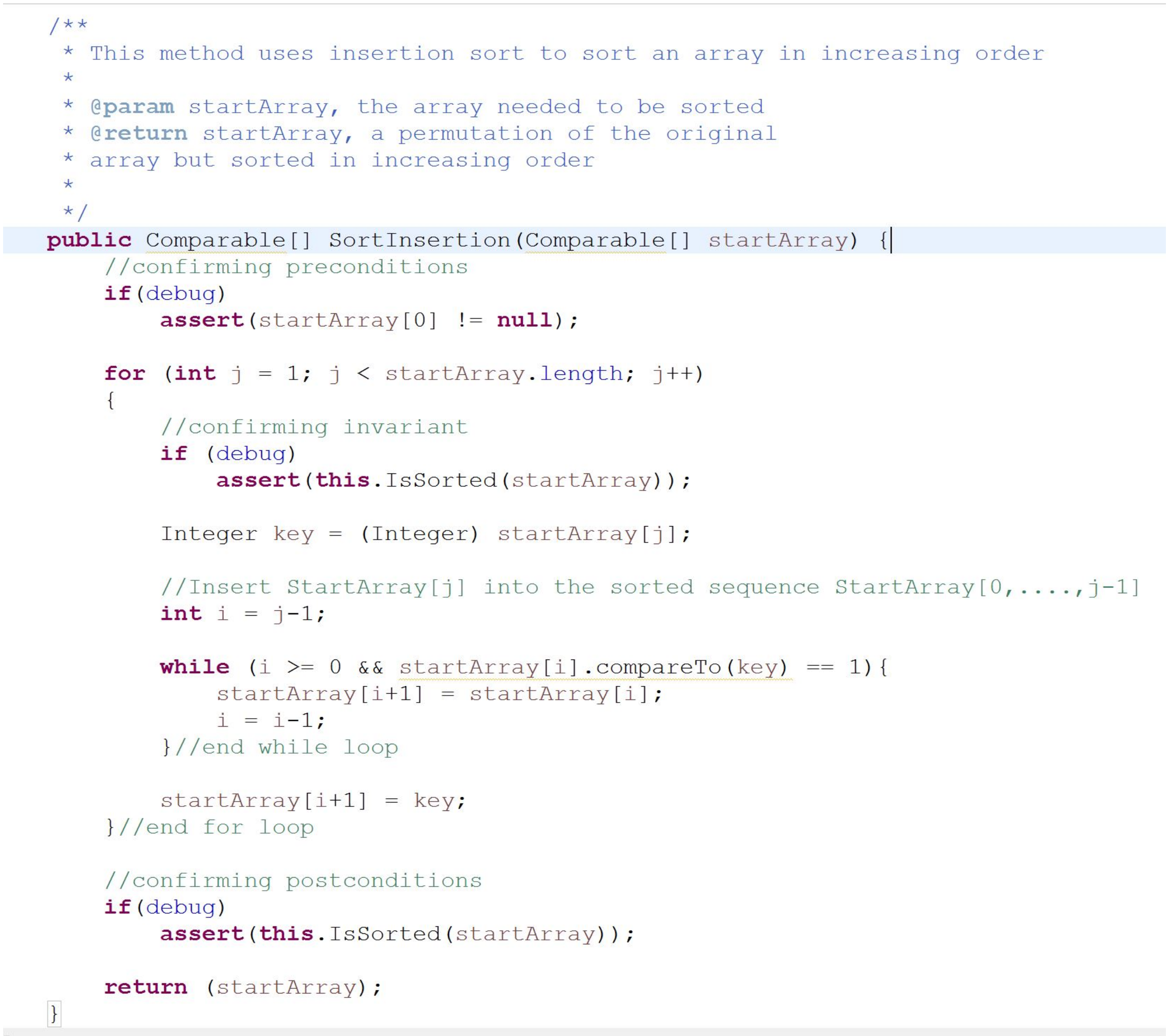
To show that the loop invariant implies the algorithm is correct, we must show it's initialization, maintenance and termination.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1..j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the for loop works by moving $A[j-1], A[j-2], A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 47), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Now that we know the algorithm's implementation details, we conclude by writing an insertion sort class that we may use to find the run-times for Insert Sort. An example of what a finalized insertion sort would look like may be as shown bellow.



```
/**
 * This method uses insertion sort to sort an array in increasing order
 *
 * @param startArray, the array needed to be sorted
 * @return startArray, a permutation of the original
 * array but sorted in increasing order
 *
 */
public Comparable[] SortInsertion(Comparable[] startArray) {
    //confirming preconditions
    if(debug)
        assert(startArray[0] != null);

    for (int j = 1; j < startArray.length; j++)
    {
        //confirming invariant
        if (debug)
            assert(this.IsSorted(startArray));

        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

    //confirming postconditions
    if(debug)
        assert(this.IsSorted(startArray));

    return (startArray);
}
```

Merge Sort

Our next algorithm, Merge Sort, is best known for comparing during a merging step. A merging step is when two lists are combined to output a single sorted list. During this, the first available element of each list is compared and the lower value is appended to the output list. When either list runs out of values, the remaining elements of the opposing list are appended to the output case.

The key operation of merge sort is that merging operation of the two sorted sequences. This is done by calling an auxiliary procedure called Merge. Another notable component of merge is the use of a sentinel, or a special value used to simplify the code. Note the following conditions necessary to implement Merge Sort.

Input: A sequence of n elements $< a_1, a_2, ..., a_n >$.

Output: A permutation $< a'_1, a'_2, ..., a'_n >$ of the input sequence such that $a_1 \leq a_2 \leq ... \leq a_n$.

The pseudo code for Merge Sort is as follows.

```
MERGE (A, p, q, r)

1         n1 = q − p + 1
2         n2 = r − q
3         let L[1.. n1 + 1] and R[1.. n2 + 1] be new arrays
4         for i = 1 to n1
5                   L[i] = A[p + i − 1]
6         for j = 1 to n2
7                   R[j] = A[q + j]
8         L[n1 + 1] = infinity
9         R[n2 + 1] = infinity
10        i = 1
11        j = 1
12        for k = p to r
13                  if L[i] <= R[j]
14                            A[k = L[i]
15                            i = i + 1
16                  else A[k] = R[j]
17                            j = j + 1
```

```
MERGE-SORT(A, p, r)

1 if p < r
2         q = floor.((p+r))2)
3         MERGE-SORT (A, p, q)
4         MERGE-SORT (A, q + 1, r)
5         MERGE (A, p, q, r)
```

It is important to note the loop invariant for insertion sort and it's correctness.

Loop invariant: At the start of each iteration of the **for** loop of lines 1217, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n]$ and $R[1..m]$, in sorted order. Moreover, $L[i]$ and $R[i]$ are the smallest elements of their arrays that have not been copied back into A.

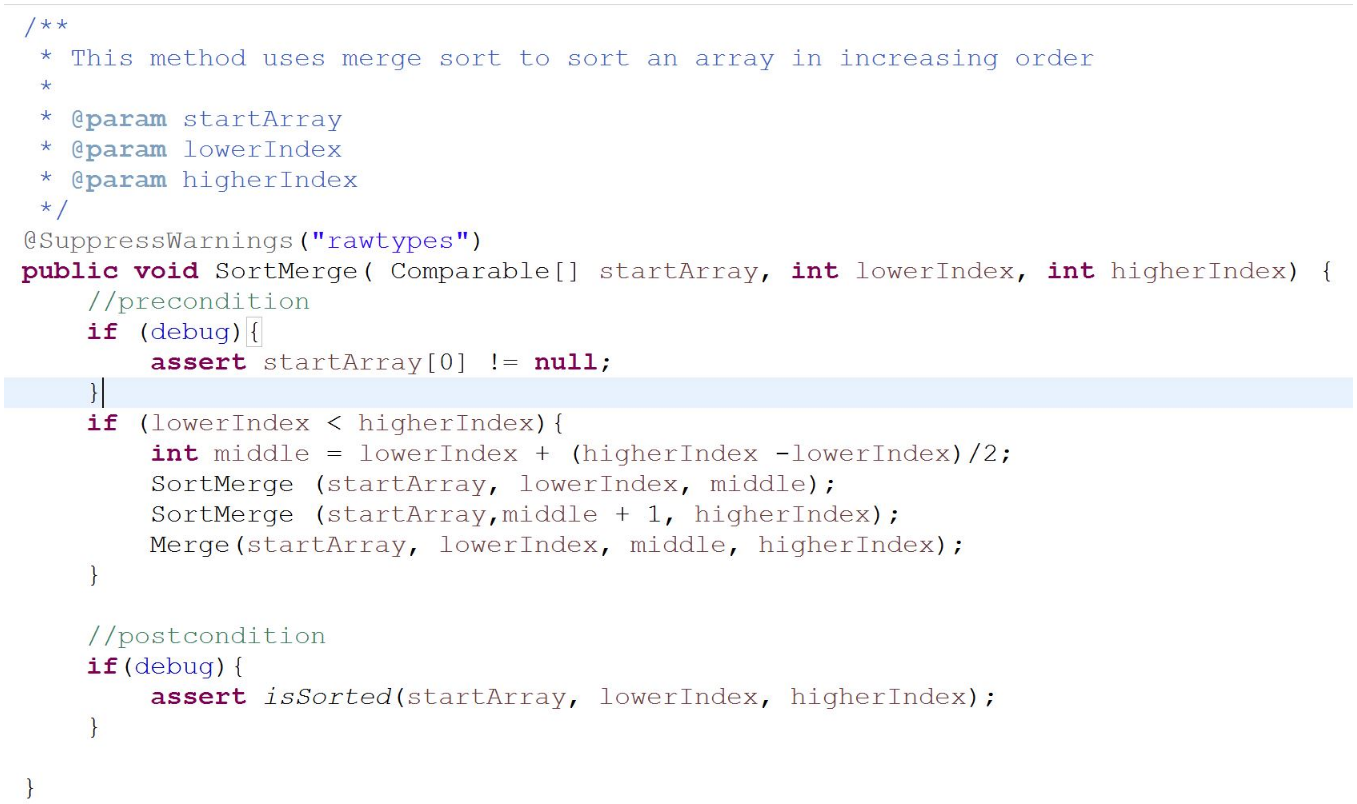
To show that the loop invariant implies the algorithm is correct, we must show it's initialization, maintenance and termination.

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k-p=0$ smallest elements of L and R, and since $i=j=1$, both $L[i]$ and $R[i]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k-p+1$ smallest elements, after line 14 copies $L[i]$ into the $A[i]$, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the for loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 1617 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k=r+1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k-p=r-p+1$ smallest elements of $L[1..n1+1]$ and $R[1..n2+1]$, in sorted order. The arrays L and R together contain $n1+n2+2=r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

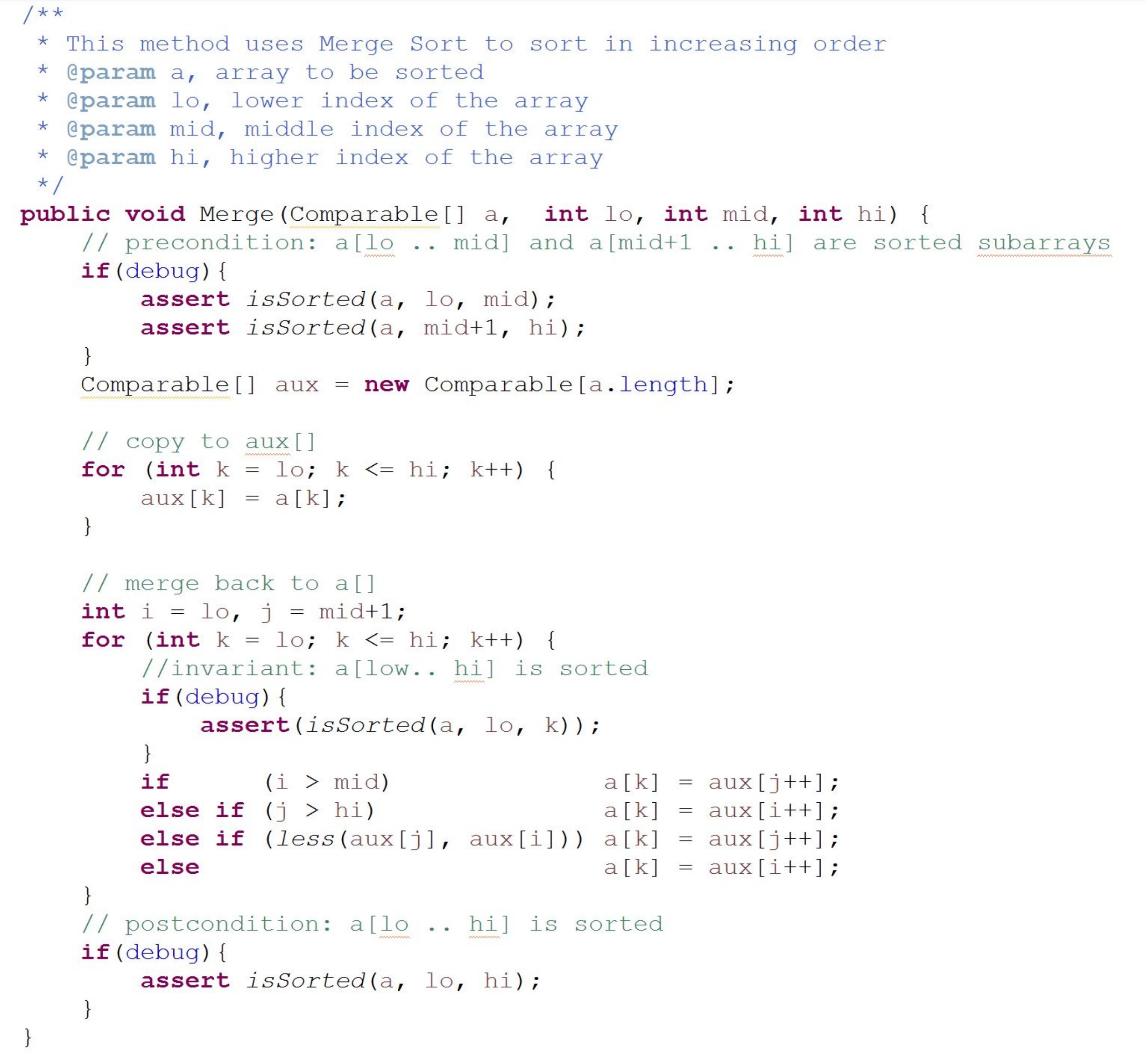
Now that we know the algorithm's implementation details, we conclude by writing an merge sort class that we may use to find the run-times for merge Sort. An example of what a finalized merge sort would look like may be as shown bellow.



```
/**
 * This method uses merge sort to sort an array in increasing order
 *
 * @param startArray
 * @param lowerIndex
 * @param higherIndex
 */
@SuppressWarnings("rawtypes")
public void SortMerge( Comparable[] startArray, int lowerIndex, int higherIndex) {
    //precondition
    if (debug){
        assert startArray[0] != null;
    }
    if (lowerIndex < higherIndex){
        int middle = lowerIndex + (higherIndex -lowerIndex)/2;
        SortMerge (startArray, lowerIndex, middle);
        SortMerge (startArray,middle + 1, higherIndex);
        Merge(startArray, lowerIndex, middle, higherIndex);
    }

    //postcondition
    if(debug){
        assert isSorted(startArray, lowerIndex, higherIndex);
    }

}
```



```
/**
 * This method uses Merge Sort to sort in increasing order
 * @param a, array to be sorted
 * @param lo, lower index of the array
 * @param mid, middle index of the array
 * @param hi, higher index of the array
 */
public void Merge(Comparable[] a,  int lo, int mid, int hi) {
    // precondition: a[lo .. mid] and a[mid+1 .. hi] are sorted subarrays
    if(debug){
        assert isSorted(a, lo, mid);
        assert isSorted(a, mid+1, hi);
    }
    Comparable[] aux = new Comparable[a.length];

    // copy to aux[]
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // merge back to a[]
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        //invariant: a[low.. hi] is sorted
        if(debug){
            assert(isSorted(a, lo, k));
        }
        if      (i > mid)              a[k] = aux[j++];
        else if (j > hi)               a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                           a[k] = aux[i++];
    }
    // postcondition: a[lo .. hi] is sorted
    if(debug){
        assert isSorted(a, lo, hi);
    }
}
```

Heap Sort

Our next algorithm, Heap Sort, is best known for using the data structure called a "heap." This indicates that the largest value is in the root, but also that any subtree of a heap is itself a heap. Like Insert Sort, but unlike Merge Sort, Heap Sort sorts in place. However, it is much faster than Insert Sort when working with large amounts of data. Thus, Heap Sort combines the better attributes of the two sorting algorithms we have already discussed [1]. Note its following conditions.

Input: A sequence of n elements $< a_1, a_2, ..., a_n >$.

Output: A permutation $< a'_1, a'_2, ..., a'_n >$ of the input sequence such that
$a_1 \leq a_2 \leq ... \leq a_n$.

The pseudo code for Heap Sort is as follows.

```
PARENT(i)
1         return floor(i/2)

LEFT(i)
1         return 2i

RIGHT(i)
1         return 2i + 1


MAX-HEAPIFY(A,i)

1         l = LEFT(i)
2         r = RIGHT(i)
3         if l <= A.heap-size and A[l] > A[i]
4                 largest = l
5         else largest = i
6         if r <= A.heap-size and A[r] > A[largest]
7                 largest = r
8         if largest != i
9                 exchange A[i] with A[largest]
10                MAX-HEAPIFY(A, largest)


BUILD-MAX-HEAP(A)

1         A.heap-size = A.length
2         for i = floor(A.largest/2) down to 1
3                 MAX-HEAPIFY(A,i)


HEAPSORT(A)

1         BUILD-MAX-HEAP(A)
2         for i = A.length down to 2
3                 exchange A[1] with A[i]
4                 A.heap-size = A.heap-size - 1
5                 MAX-HEAPIFY(A, 1)
```

It is important to note the loop invariant for insertion sort and it's correctness. In this case, we will be studying the loop invariant for BUILD-MAX-HEAP

Loop invariant: At the start of each iteration of the **for** loop of lines 2-3, each node

$i+1, i+2, ..., n$ is the root of a max-heap.

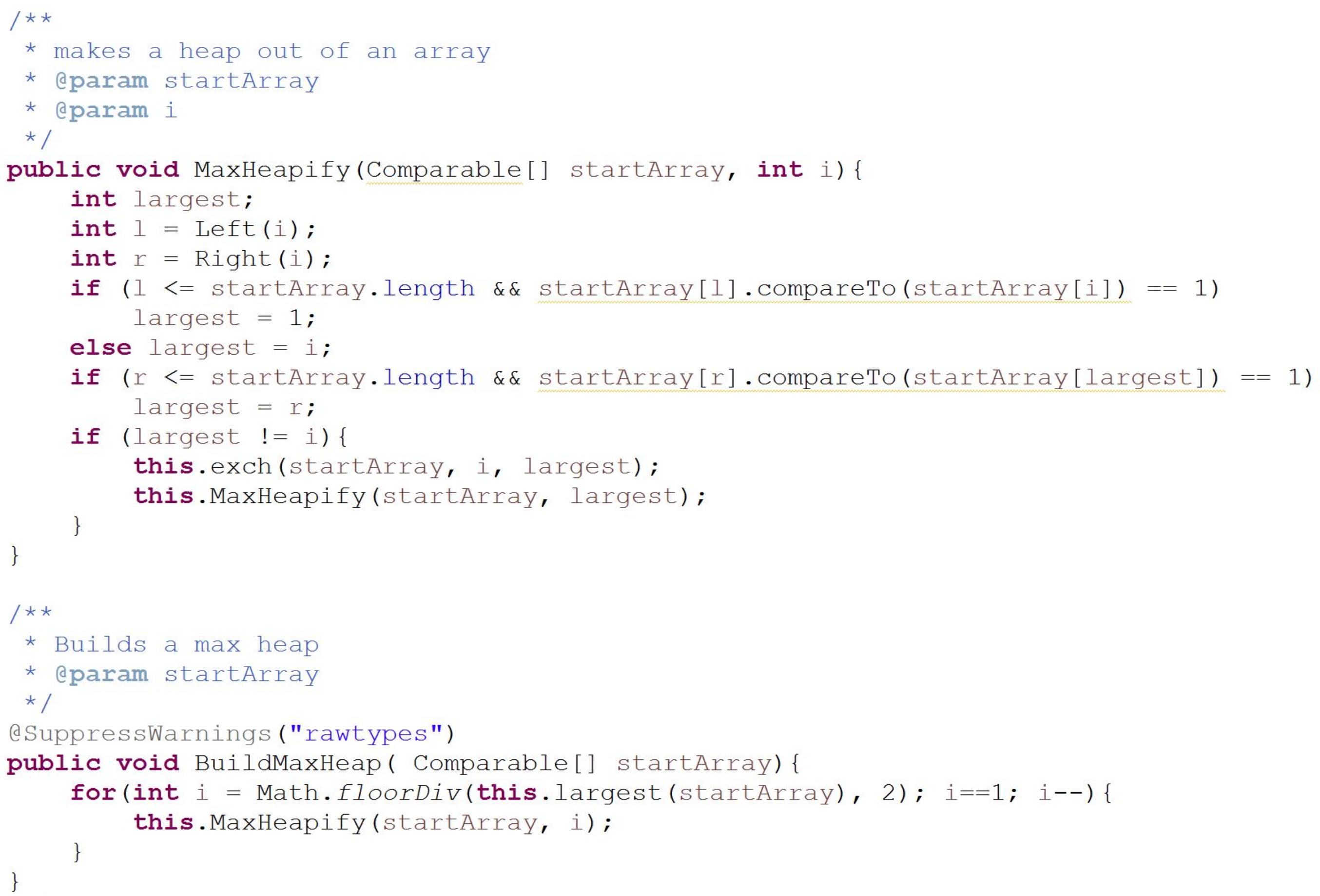
To show that the loop invariant implies the algorithm is correct, we must show it's initialization, maintenance and termination.

Initialization: Prior to the first iteration of the loop, $i = floor(n/2)$. Each node $floor(n/2)+1, floor(n/2)+2, ..., n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY$(A, i)/$ to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i+1, i+2, ..., n$ are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.

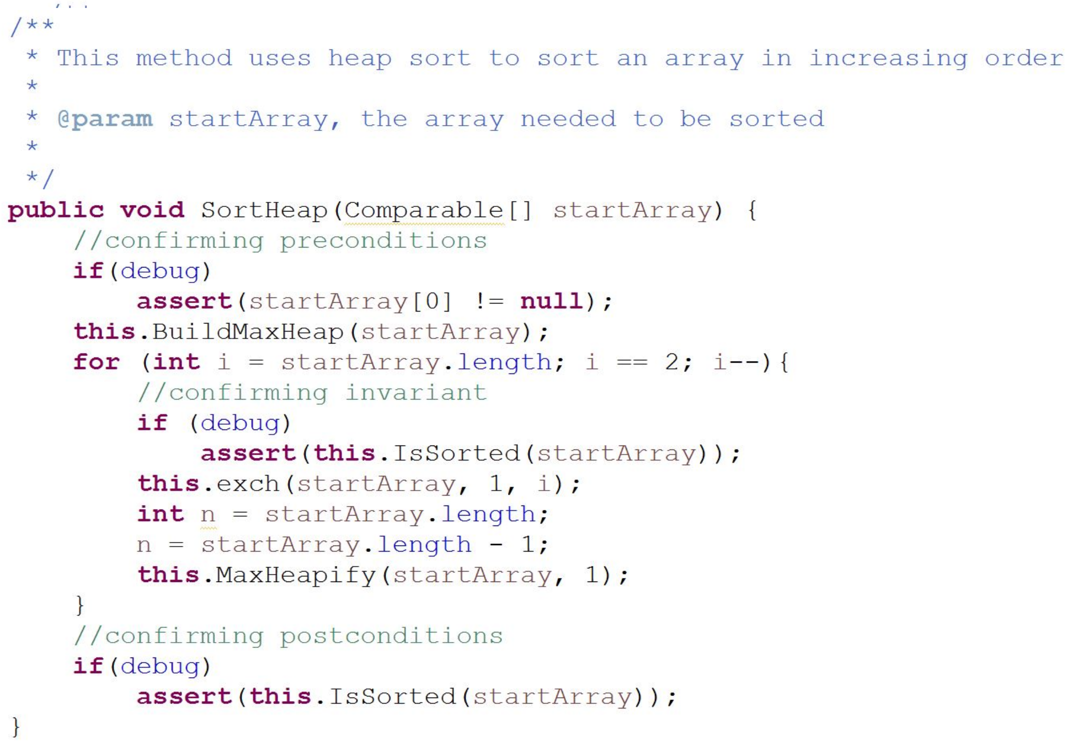
Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, ..., n$ is the root of a max-heap. In particular, node 1 is.

Now that we know the algorithm's implementation details, we conclude by writing an Heap sort class that we may use to find the run-times for Heap Sort. An example of what a finalized Heap sort would look like may be as shown bellow.



```java
/**
 * makes a heap out of an array
 * @param startArray
 * @param i
 */
public void MaxHeapify(Comparable[] startArray, int i){
    int largest;
    int l = Left(i);
    int r = Right(i);
    if (l <= startArray.length && startArray[l].compareTo(startArray[i]) == 1)
        largest = 1;
    else largest = i;
    if (r <= startArray.length && startArray[r].compareTo(startArray[largest]) == 1)
        largest = r;
    if (largest != i){
        this.exch(startArray, i, largest);
        this.MaxHeapify(startArray, largest);
    }
}

/**
 * Builds a max heap
 * @param startArray
 */
@SuppressWarnings("rawtypes")
public void BuildMaxHeap( Comparable[] startArray){
    for(int i = Math.floorDiv(this.largest(startArray), 2); i==1; i--){
        this.MaxHeapify(startArray, i);
    }
}
```



```java
/**
 * This method uses heap sort to sort an array in increasing order
 *
 * @param startArray, the array needed to be sorted
 *
 */
public void SortHeap(Comparable[] startArray) {
    //confirming preconditions
    if(debug)
        assert(startArray[0] != null);
    this.BuildMaxHeap(startArray);
    for (int i = startArray.length; i == 2; i--){
        //confirming invariant
        if (debug)
            assert(this.IsSorted(startArray));
        this.exch(startArray, 1, i);
        int n = startArray.length;
        n = startArray.length - 1;
        this.MaxHeapify(startArray, 1);
    }
    //confirming postconditions
    if(debug)
        assert(this.IsSorted(startArray));
}
```

Quick Sort

Our next algorithm, Quick Sort, is best known for its reputation of being the best practical choice for sorting. Quick Sort has a remarkably efficient random case. It also has the advantage, like Insert Sort and Heap Sort of working in place. Note its following conditions.

Input: A sequence of n elements $< a_1, a_2, ..., a_n >$.

Output: A permutation $< a'_1, a'_2, ..., a'_n >$ of the input sequence such that $a_1 \leq a_2 \leq ... \leq a_n$.

The pseudo code for Quick Sort is as follows.

```
QUICKSORT(A, p, r)

1       if p < r
2               q = PARTITION (A, p, r)
3               QUICKSORT(A, p, q-1)
4               QUICKSORT(A, q+1, r)


PARTITION (A, p, r)

1       x = A[r]
2       i = p - 1
3       for j = p to r-1
4               if A[j] <= x
5                       i = i +1
6                       exchange A[i] with A[j]
7       exchange A[i+1] with A[r]
8       return i+1
```

It is important to note the loop invariant for insertion sort and it's correctness. In this case, we will be studying the loop invariant of the PARTITION method.

Loop invariant: At the beginning of each iteration of the loop of lines 3-6, for any array index k,

1. if $p \leq k \leq i$, then $A[k] \leq x$.

2. if $i + 1 \leq k \leq j - 1$, then $A[k] > x$.

3. if $k = r$, then $A[k] = x$.

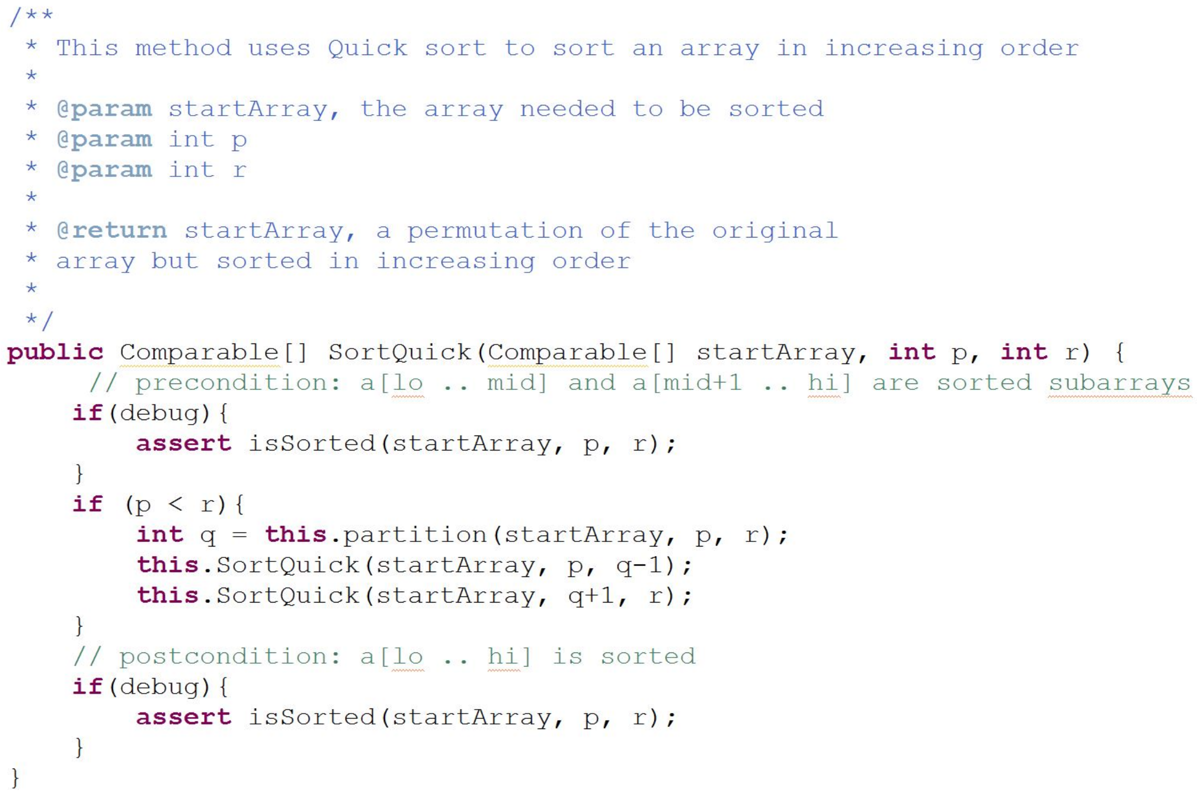
To show that the loop invariant implies the algorithm is correct, we must show it's initialization, maintenance and termination.

Initialization: Prior to the first iteration of the loop, $i = p - 1$ and $j = p$ Because no values lie between p and i and no values lie between $i+1$ and $j-1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

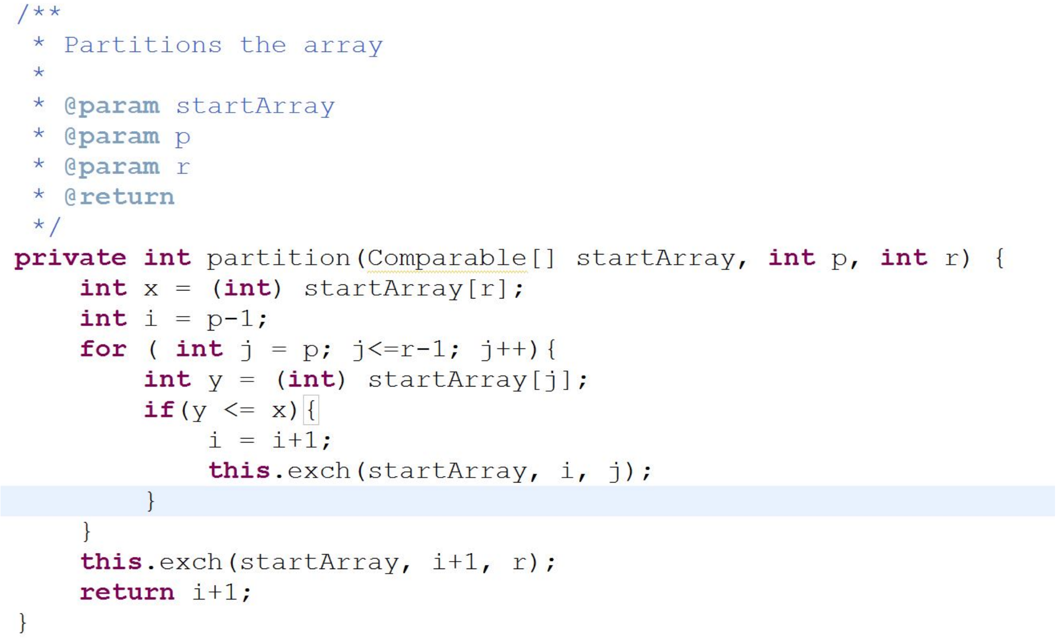
Maintenance: We consider two cases, depending on the outcome of the test in line 4. When $A[j] > x$ the only action in the loop is to increment j . After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. When $A[j] \leq x$ the loop increments i, swaps $A[i]$and $A[j]$ and then increments j . Because of the swap, we now have that $A[i] \leq x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

Now that we know the algorithm's implementation details, we conclude by writing an Quick sort class that we may use to find the run-times for Quick Sort. An example of what a finalized Quick sort would look like may be as shown bellow.



```
/**
 * This method uses Quick sort to sort an array in increasing order
 *
 * @param startArray, the array needed to be sorted
 * @param int p
 * @param int r
 *
 * @return startArray, a permutation of the original
 * array but sorted in increasing order
 *
 */
public Comparable[] SortQuick(Comparable[] startArray, int p, int r) {
      // precondition: a[lo .. mid] and a[mid+1 .. hi] are sorted subarrays
    if(debug){
        assert isSorted(startArray, p, r);
    }
    if (p < r){
        int q = this.partition(startArray, p, r);
        this.SortQuick(startArray, p, q-1);
        this.SortQuick(startArray, q+1, r);
    }
    // postcondition: a[lo .. hi] is sorted
    if(debug){
        assert isSorted(startArray, p, r);
    }
}
```



```
/**
 * Partitions the array
 *
 * @param startArray
 * @param p
 * @param r
 * @return
 */
private int partition(Comparable[] startArray, int p, int r) {
    int x = (int) startArray[r];
    int i = p-1;
    for ( int j = p; j<=r-1; j++){
        int y = (int) startArray[j];
        if(y <= x){
            i = i+1;
            this.exch(startArray, i, j);
        }
    }
    this.exch(startArray, i+1, r);
    return i+1;
}
```

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 the program. The following matrix shows the tests the program went through, the expected results, the actual results, and the solution.

Test	Expected Result	Actual Result	Remedy
Array of length zero	program continues with n = 0	For some sorts (like Insert Sort) This didn't cause any problem. Other sorts, like Merge Sort, gave an out of bounds exception	The exception came because these sorts require the array length to be divided in half, and 0/2 is out of bounds. The solution to this was found by inserting an assert statement that would guarantee the arrays would never analyze an array of length 0.
Array of negative length	an exception	As expected	Implemented the program so that an array of negative length would never be made
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 *TimeToSort()* method, rather than the actual *InsertSort* method itself.
Array with null items	an exception is thrown due to asserts	As Expected	N/A

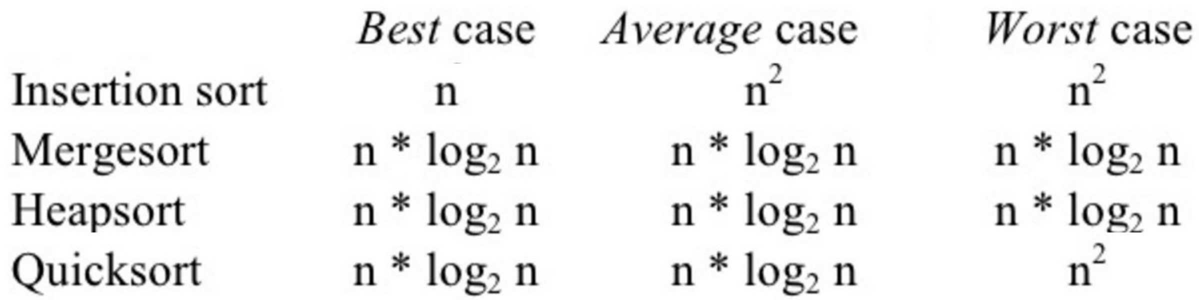
PROBLEMS ENCOUNTERED

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

An issue that consistently came up while implementing my program was making sure that the methods sorted any comparable element, and not just integers. The methods were originally implemented to analyze integers, however, when changing them to analyze other elements, a lot of errors on overriding, raw types, and casting kept showing up. After a long time of debugging, these issues were resolved.

Since each sorting algorithm has different conditions for their best, worst, and random cases, it was difficult at first to modify the original code (used for Programming Assignment 1: Insertion Sort) to work well with the new requirements. After some trial and error, the best solution was found to be refactoring most of the code in the main class to each of the sorting classes. That way, each class would determine its own best, worst, and average cases for each algorithm.

On the same note, finding how best, worst, and average arrays for input proved to be quite a challenge. Deducting how each array would need to be preordered for each case was as difficult as finding a way to implement that in the code.



	Best case	*Average* case	*Worst* case
Insertion sort	n	n^2	n^2
Mergesort	n * $\log_2$ n	n * $\log_2$ n	n * $\log_2$ n
Heapsort	n * $\log_2$ n	n * $\log_2$ n	n * $\log_2$ n
Quicksort	n * $\log_2$ n	n * $\log_2$ n	n^2

Figure 1: Growth Rates for Selected Sorting Algorithms

EXPERIMENTAL ANALYSIS AND ASYMPTOTIC RUN-TIME COMPARISON

After confirming that the program meets the conditions of each sort, and that it produces the required results, we now must compare the run-time complexities of the four sorting algorithms with their asymptotic run-times.

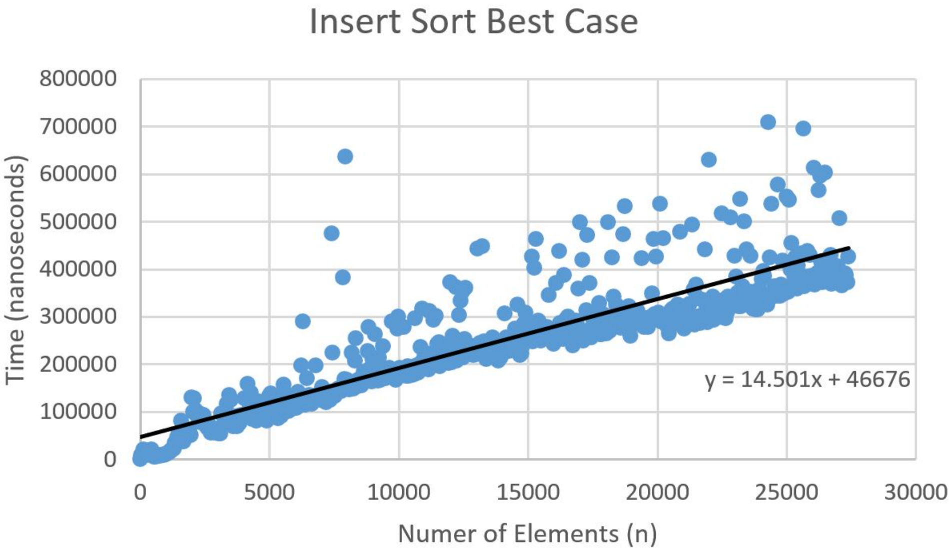
By running the program for different input sizes n, pre-ordered accordingly to find the best, worst, and random cases, we found the different run-times for each sorting algorithm. We predict that our results will act according to each algorithm's asymptotic growth-rates, as described by the table in figure 1. To best compare if our results align with this prediction, we plotted the time behavior for each case as a function of n. Our results are described in the following sections.

Insert Sort

Best Case

The best case input for insertion sort is an array whose elements are already sorted in increasing order. In this case, Insert Sort typically has linear running time (i.e. O(n)).

After plotting the best case data points found using the program, we found the results shown in the following chart.

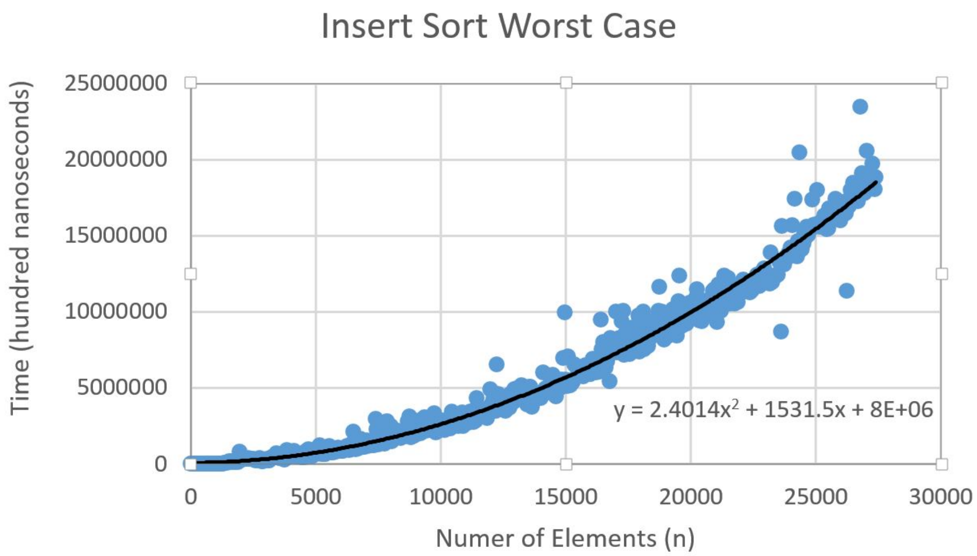


Note that even with the outliers, the graph still follows linear behavior, defined by the trendline $y = 14.501x + 46676$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 1500$.

Worst Case

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

After plotting the worst case data points found using the program, we found the results shown in the following chart.

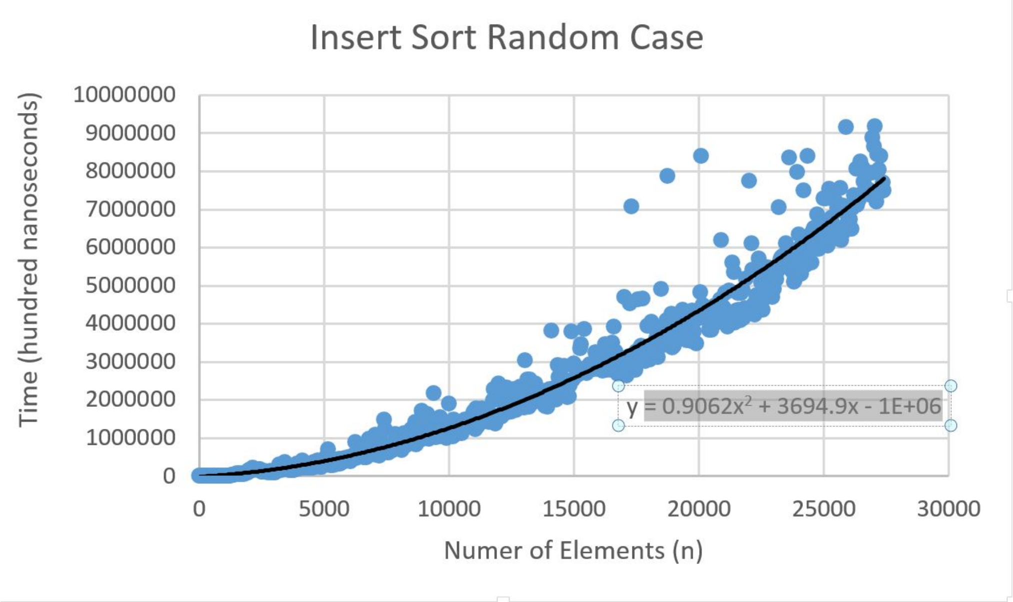


Note that this time we have fewer outliers than with the best case. Even so, the graph follows the predicted behavior of O(n^2) defined by the trendline $y = 2.4014x^2 + 1531.5x + 8E + 06$. As you can see from the graph, n_0 is very near the beginning, we could estimate that it falls around $n_0 = 50$.

Random Case

The asymptotic random case for insertion sort is typically also quadraic, due to the implementation of the method. Thus, like with the worst-case behavior, the asymptotic running time of the random case is O(n^2).

After plotting the random case data points found using the program, we found the results shown in the following chart.



It is interesting to see how similar the random case is to the the worst case with Insertion sort. Again, our data shows O(n^2) behavior with a trendline around $y = 0.9062x^2 + 3694.9x - 1E + 06$. Again, n_0 is very near the beginning, so we similarly estimate that it falls around $n_0 = 50$.

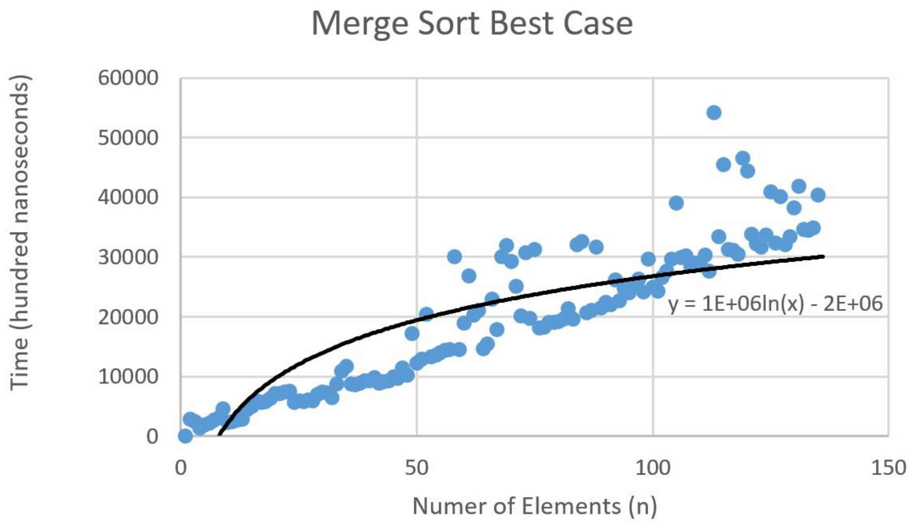
After analyzing our findings, we can conclude that the best conditions to use insertion sort is when there are not a lot of elements, or when they are already in sorted order, since the run-time is linearly proportional to the amount of elements to be sorted.

Merge Sort

Best Case

Merge sort's best case is when the largest element of one sorted sublist is smaller than the first element of its opposing sub-list, for every merge stem that occurs. Only one element from the opposing list is compared, which reduces the number of comparisons in each merge step to $n/2$. The best case of Merge sort typically has logarithmic running time (i.e. O($nlgn$)).

After plotting the best case data points found using the program, we found the results shown in the following chart.

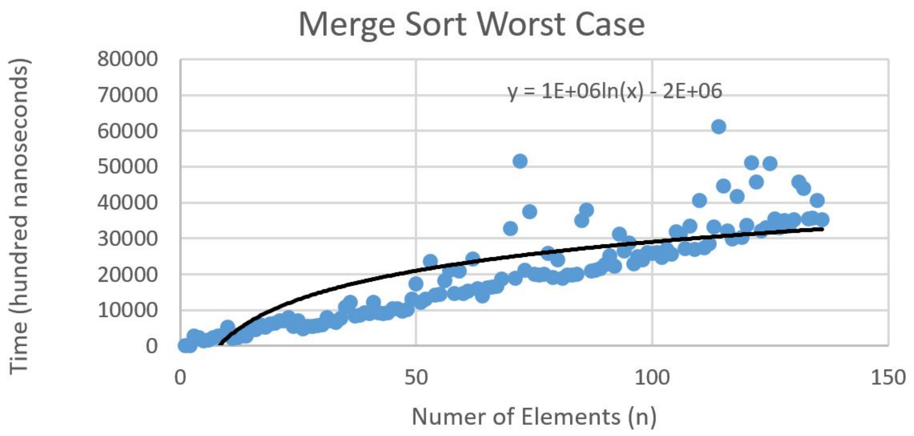


Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = 1E + 06ln(x) - 2E + 06$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 1000$.

Worst Case

The worst case scenario for Merge Sort is when, during ever merge step, exactly one value remains in the opposing list; in other words, no comparisons were skipped. This situation occurs when the two largest values in a merge stem are contained in opposing lists. When this situation occurs, Merge Sort must continue comparing list elements in each of the opposing lists until the two largest values are compared[6]. In this case, Merge Sort typically has logarithmic running time (i.e. O($nlgn$)).

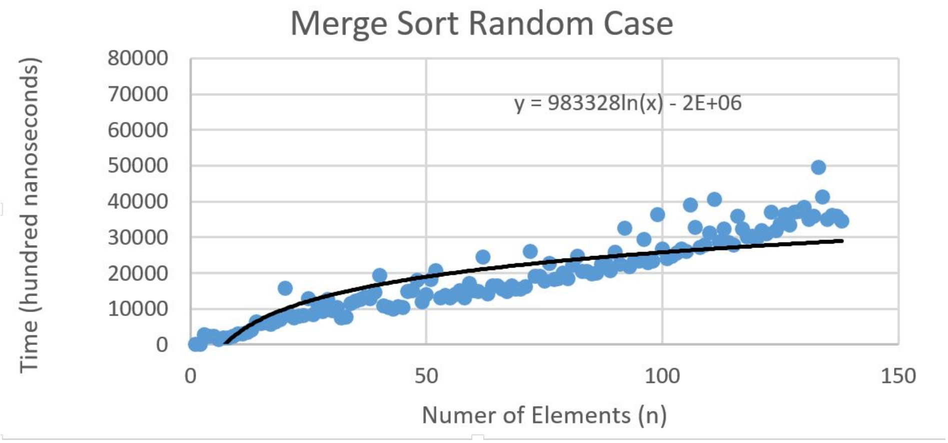
After plotting the worst case data points found using the program, we found the results shown in the following chart.



Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = 1E + 06ln(x) - 2E + 06$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 1000$.

Random Case

A quality of Merge Sort, is that its best, worst and random cases all have the same asymptotic running times, O($nlgn$). We can see this quality after plotting the random case data points found using the program.



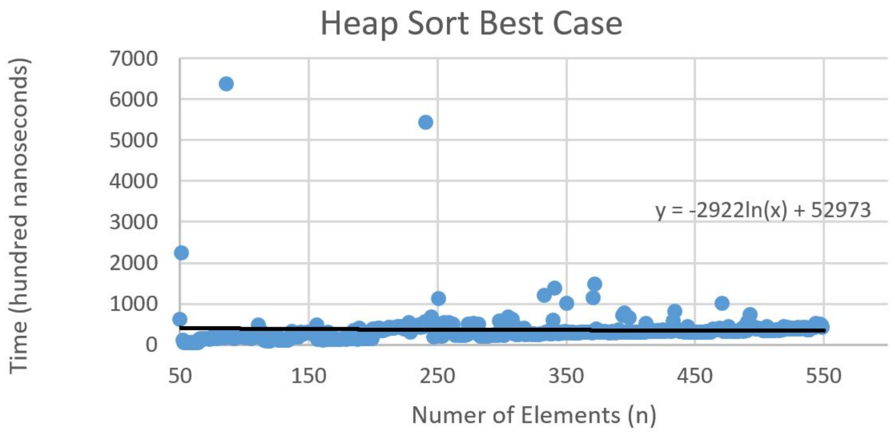
Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = 983328ln(x) - 2E + 06$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 1000$.

Heap Sort

When it comes to finding the best possible inputs for achieving the best, worst, and random cases, Heap Sort is an interesting algorithm to study. Since the first step of a heap sort is to build a heap with the initial elements, then the input is already in a specified order. This indicates that the time required to build that heap happens outside the Heap Sort algorithm (typically in max-heap). This is partly responsible for the fact that Heap Sort has the same running time for its best, worst and random case: O($nlgn$).

Best Case

As mentioned before, the asymptotic best case for Heap Sort is O($nlgn$) After plotting the best case data points found using the program, we found the results shown in the following chart.

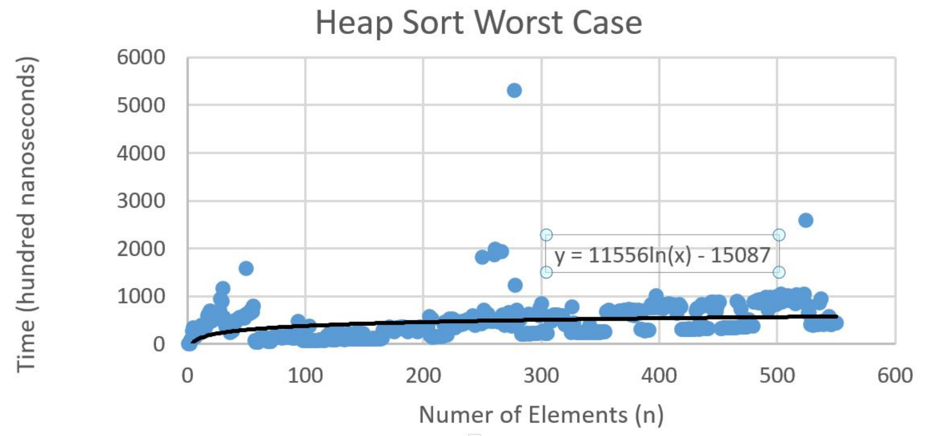


Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = -2922ln(n) + 52973$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 50$ Since right from the beginning our data aligned with the asymptotic trendline.

Worst Case

The asymptotic worst case for Heap Sort is O($nlgn$).

After plotting the best case data points found using the program, we found the results shown in the following chart.

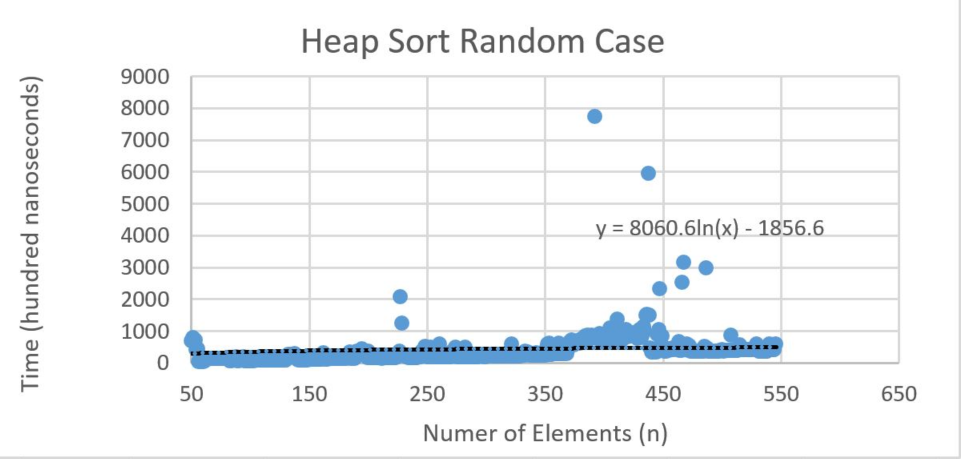


Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = 11556ln(n) - 15087$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 50$ Since right from the beginning our data aligned with the asymptotic trendline.

Random Case

The asymptotic random case for Heap Sort is also O($nlgn$).

After plotting the best case data points found using the program, we found the results shown in the following chart.



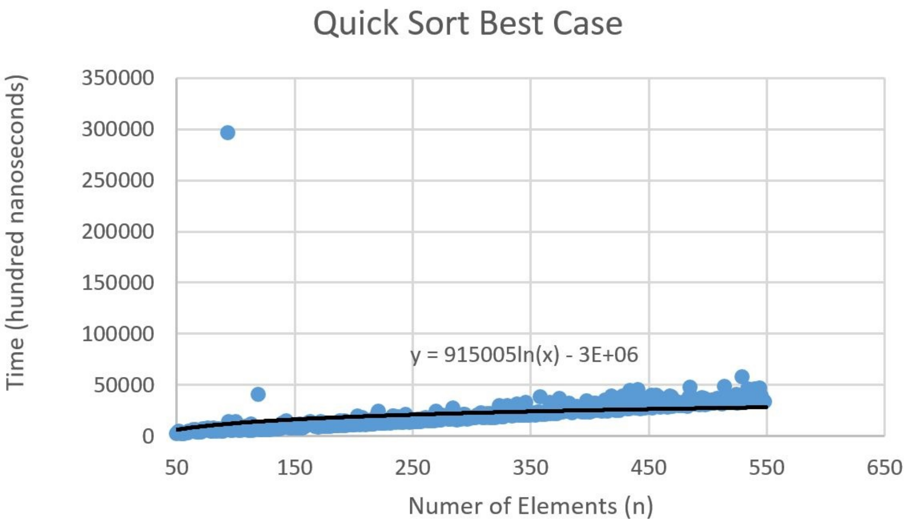
Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = 8060ln(n) - 1856.6$. A good estimation from the graph would indicate that n_0 would A good estimation from the graph would indicate that n_0 would lie around $n_0 = 50$ Since right from the beginning our data aligned with the asymptotic trendline.

Quick Sort

Best Case

The asymptotic best case for Quick Sort is O($nlgn$). This happens when the most even possible split, PARTITION produces two subproblems. Each subproblem would be of size no more than $n/2$, since one is of size $floor(n/2)$ and one of size $(n/2) - 1$[1].

After plotting the best case data points found using the program, we found the results shown in the following chart.

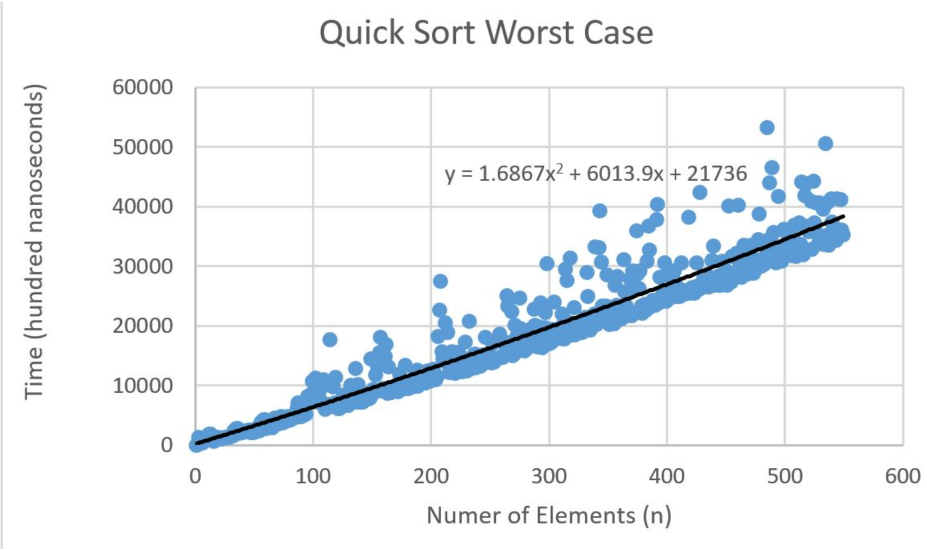


Note that even with the two outliers in the beginning, the graph still follows logarithmic behavior, defined by the trendline $y = 915005ln(n) - 3E + 06$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 50$ Since right from the beginning our data aligned with the asymptotic trendline.

Worst Case

The worst case behavior for Quick Sort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements.[6] The asymptotic run-time for Quick Sort's worst case is O(n^2), the same as Insert Sort. Moreover, this occurs when the input array is already completely sorted- a situation in which Insert Sort runs in O(n) time.

After plotting the worst case data points found using the program, we found the results shown in the following chart.

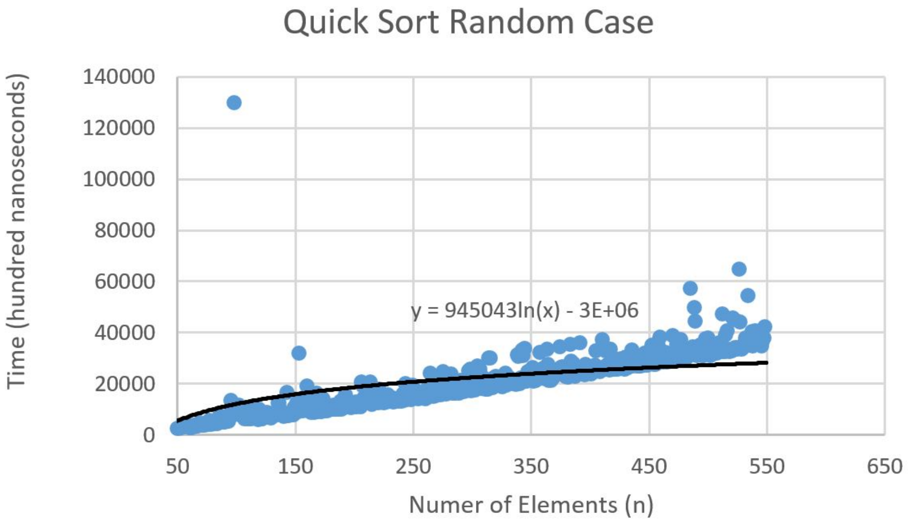


Note that even with the outliers, the graph follows the predicted behavior of O(n^2), defined by the trendline $y = 1.6867n^2 + 6013.9n + 21736$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 50$ Since right from the beginning our data aligned with the asymptotic trendline.

Random Case

The random case running time for Quick Sort is much closer to its best case than to its worst case[1]. It's running time is therefore O($nlgn$).

After plotting the random case data points found using the program, we found the results shown in the following chart.



Note that even with the outliers, the graph still follows logarithmic behavior, defined by the trendline $y = 945043ln(n) - 3E + 06$. A good estimation from the graph would indicate that n_0 would lie around $n_0 = 50$ Since right from the beginning our data aligned with the asymptotic trendline.

CONCLUSIONS

After constructing an sorting program that implemented the Insert Sort, Merge Sort, Heap Sort and Quick Sort algorithms, we were able to analyze and compare their run-time behaviors. We compared the time complexities we found to the asymptotic run-time complexities for best, worst, and average cases of each algorithm style. We have also found the conditions that make one algorithm better than the others and have discovered that the algorithm with the largest leading constant is InsertSort. With a better understanding of each sorting algorithm, 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, Merge Sort, Heap Sort, and Quick Sort algorithms and their corresponding run-time complexities.

We predicted, *a priori*, that Insert Sort would have the ”largest constant”. After finding the functions for each algorithms best, worst, and random case, we found that the example with the largest leading constant was Insert Sort’s best case with $y = 14.501x + 46676$.

APPENDIX A: Main Class

The following is the Java class written for the main class.

```
import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;
/**
 *  COMP 215: Design and Analysis of Algorithms:
 *                Programming Assignment 1
 *
 *  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,
 *  Merge Sort, Heap Sort, and Quick Sort
 *
 *
 *  Created:
 *     [01/24/2016], [Melany Diaz]
 *     With assistance from:  Dr. Gerry Howser
 *
 *  Modifications: <
 *                [01/24/2016], [Melany Diaz],
 *                [implemented code to sort in increasing and
 *                 decreasing orders, made an array of user
 *                 inputed size of random integers]
 *
 *                [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]
 *
 *                [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]
 *
 *                 [02,07,2015], [Melany Diaz],
 *                 [Changed the class to work with the second project
    ]
 *
 *   @author [Melany Diaz]
 *
 */

public class SortingAnalysis
{
          /*
           * Instance Variables
           */

          //Objects of each sorting class
          static InsertionSort insertionSort = new InsertionSort();
          static MergeSort mergeSort = new MergeSort();
          static HeapSort heapSort = new HeapSort();
          static QuickSort QuickSort = new QuickSort();



     /**
      *  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)
     {
          /**
           * find the time complexities of each sorting style
           */

          System.out.println("InsertSort" + "\n" + "n;␣Random;␣Best;
             ␣Worst");
          insertionSort.run();
```

```
        System.out.println("MergeSort" + "\n" + "n;␣Random;␣Best;␣
            Worst");
        mergeSort.run();

        System.out.println("HeapSort" + "\n" + "n;␣Random;␣Best;␣
            Worst");
        heapSort.run();

        System.out.println("QuickSort" + "\n" + "n;␣Random;␣Best;␣
            Worst");
        QuickSort.run();

    }//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;
import java.util.Random;


/**
 * COMP 215: Design and Analysis of Algorithms: Programming
   Assignment<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
 *
 * 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 and Dr. Alyce Brady
  <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 objct]
 *      and invariants
 *      [02,11,2016], [Melany Diaz], [added assertions and a
    confirmation method]
 *
 * @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;

    boolean debug = true;

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


    static InsertionSortDecreasing insertionSortDecreasing = new
       InsertionSortDecreasing();
    /**
     * 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
 * @return startArray, a permutation of the original
 * array but sorted in increasing order
 *
 */
public Comparable[] SortInsertion(Comparable[] startArray) {
    //confirming preconditions
    if(debug)
        assert(startArray[0] != null);

    for (int j = 1; j < startArray.length; j++)
    {
        //confirming invariant
        if (debug)
            assert(this.IsSorted(startArray));

        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

    //confirming postconditions
    if(debug)
        assert(this.IsSorted(startArray));

    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;
}


/**
 * This method confirms that an array is in sorted order.
 *
 * @param Array Array, the array who is to be confirmed
 * @return boolean sorted. True if the array is sorted, false
    if not
 */

public boolean IsSorted(Comparable[] Array){
    boolean sorted = true;
    for (int i = 0; i < Array.length-1; i++){
        if (Array[i].compareTo(Array[i+1]) == 1){
            sorted = false;
            break;
        }
    }
    return sorted;
}


/**
 * runs the comparison analysis and prints results to the
    console
 */
public void run(){
    int n = 0;
    while (n <= 27400){

    //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 random cases
 */

/*
 * Make the array used for random case and fill the
     average array
 * with n random objects.
 */
Random generator = new Random();

for(int i = 0; i < n; i++)
     average[i] = generator.nextInt(n);


/*
 * The following code uses InsertionSort and its methods
 * to find the best case scenario. InsertionSort will
 * sort the array items in an increasing order. it will
 * make a new array with the same elements but in
     increasing order
 */

increasing = Arrays.copyOf(average,n);
increasing = this.SortInsertion(increasing);


/*
 * The following code uses InsertionSortDecreasing and its
      methods
 * to find the worst case scenario.
     InsertionSortDecreasing will
 * sort the array items in decreasing order. It will
 * make a new array with the same elements but in
     decreasing order
 */

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

```
        decreasing = insertionSortDecreasing.SortDecretion(
           decreasing);


        /**
         * Will print out the time as csv of the average, best,
            and worst cases
         */

        System.out.println(n + ”;” + this.TimeToSort(Arrays.copyOf
           (average,n)) + ”;” + this.TimeToSort(Arrays.copyOf(
           increasing,n)) + ”;” +this.TimeToSort(Arrays.copyOf(
           decreasing,n)));

        n = n+50;
        }
    }


}
```

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.

```java
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 objct]
 *               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**
     *  @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[] SortDecretion(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 C: MergeSort Class

The following is the Java class written for Merge Sort. This class provides a method that uses the Merge 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 MergeSort method, this class provides a second method, which times the execution process.

```
import java.util.Arrays;
import java.util.Random;



/**
 * COMP 215: Design and Analysis of Algorithms: Programming
   Assignment<br>
 *
 * The <code>MergeSort</code> class provides an algorithm that
   will sort a given
 * array into increasing order using the MergeSort 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 code will be used to compare the worst case, best case,
   and average case run times
 * for Merge Sort
 * <br> <br>
 * Created: <br>
 *     [02,01,2016], [Melany Diaz] [assistance from http://algs4.
   cs.princeton.edu/22mergesort/Merge.java.html]<br>
 * Modifications: <br>
 *
 * @author [Melany Diaz]
 */

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

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

   boolean debug = true;

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


   static InsertionSortDecreasing insertionSortDecreasing = new
      InsertionSortDecreasing();

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


   // Methods

   /**
    * This method uses merge sort to sort an array in increasing
       order
    *
    * @param startArray
    * @param lowerIndex
    * @param higherIndex
    */
   @SuppressWarnings("rawtypes")
   public void SortMerge( Comparable[] startArray, int lowerIndex
      , int higherIndex) {
       //precondition
       if (debug){
           assert startArray[0] != null;
       }
       if (lowerIndex < higherIndex){
           int middle = lowerIndex + (higherIndex -lowerIndex)/2;
           SortMerge (startArray, lowerIndex, middle);
           SortMerge (startArray,middle + 1, higherIndex);
           Merge(startArray, lowerIndex, middle, higherIndex);
       }

       //postcondition
       if(debug){
           assert isSorted(startArray, lowerIndex, higherIndex);
       }

   }



   /**
    * This method uses Merge Sort to sort in increasing order
    * @param a, array to be sorted
    * @param lo, lower index of the array
    * @param mid, middle index of the array
    * @param hi, higher index of the array
    */
   public void Merge(Comparable[] a,  int lo, int mid, int hi) {
       // precondition: a[lo .. mid] and a[mid+1 .. hi] are
          sorted subarrays
       if(debug){
           assert isSorted(a, lo, mid);
           assert isSorted(a, mid+1, hi);
       }
       Comparable[] aux = new Comparable[a.length];

       // copy to aux[]
       for (int k = lo; k <= hi; k++) {
           aux[k] = a[k];
       }

       // merge back to a[]
       int i = lo, j = mid+1;
       for (int k = lo; k <= hi; k++) {
           //invariant: a[low.. hi] is sorted
           if(debug){
               assert(isSorted(a, lo, k));
           }
           if      (i > mid)              a[k] = aux[j++];
           else if (j > hi)               a[k] = aux[i++];
           else if (less(aux[j], aux[i])) a[k] = aux[j++];
           else                           a[k] = aux[i++];
       }
       // postcondition: a[lo .. hi] is sorted
       if(debug){
           assert isSorted(a, lo, hi);
       }
   }


   /**
    * This method times how long it takes to run merge 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){
       int lowIndex = 0;
       int highIndex = Array.length -1;
       long start= System.nanoTime();
       this.SortMerge(Array, lowIndex, highIndex);
       long end = System.nanoTime();
       long duration = end - start;
       return duration;
   }

   /************************************
    *  Helper sorting functions.
    */

    // is v < w ?
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    // exchange a[i] and a[j]
    private static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }
```

```
/**
 * confirms that an array is in sorted order.
 *
 * @param Array Array, the array who is to be confirmed
 * @return boolean sorted. True if the array is sorted, false
    if not
 */
private static boolean isSorted(Comparable[] a) {
    return isSorted(a, 0, a.length - 1);
 }

 private static boolean isSorted(Comparable[] a, int lo, int
    hi) {
     for (int i = lo + 1; i <= hi; i++)
         if (less(a[i], a[i-1])) return false;
     return true;
 }

 /******************************************
  * runs the comparison analysis and prints results to the
     console
  */
 public void run(){
    int n = 0;
    while (n <= 27400){

     //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 random cases
     */

    /*
     * Make the array used for random case and fill the
        average array
     * with n random objects.
     */
    Random generator = new Random();
```

```
            for(int i = 0; i < n; i++)
            average[i] = generator.nextInt(n);


             /*
              * The following code finds the best case scenario.
             */
            InsertionSort i = new InsertionSort();
            increasing = Arrays.copyOf(average,n);
            increasing = i.SortInsertion(increasing);

            /*
              * The following code finds the worst case scenario.
              */
            InsertionSortDecreasing j = new InsertionSortDecreasing();
            decreasing = Arrays.copyOf(average,n);
            decreasing = j.SortDecretion(decreasing);

            /**
             * Will print out the time as csv of the average, best,
                 and worst cases
             */

            System.out.println(n + ";" + this.TimeToSort(Arrays.copyOf
                (average,n)) + ";" + this.TimeToSort(Arrays.copyOf(
                increasing,n)) + ";" + this.TimeToSort(Arrays.copyOf(
                decreasing,n)));

            n = n+50;
            }
        }
}
```

Appendix D: HeapSort Class

The following is the Java class written for Heap Sort. This class provides a method that uses the Heap 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 MergeSort method, this class provides a second method, which times the execution process.

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

/**
 * COMP 215: Design and Analysis of Algorithms: Programming
   Assignment<br>
 *
 * The <code>HeapSort</code> class provides an algorithm that
   will sort a given
 * array into increasing order using the HeapSort algorithm.
 *
 * This is made for a program that validates the run-time
 * complexity and asymptotic run-time complexity for different
   kinds of sort method.
 *
 * This code will be used to compare the worst case, best case,
   and average case run times
 * for HeapSort.
 * <br> <br>
 * Created: <br>
 *     [02,05,2016], [Melany Diaz]<br>
 * Modifications: <br>
 *
 *
 * @author [Melany Diaz]
 */
public class HeapSort
{
    // instance variables
   Comparable[] startArray;

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

   boolean debug = true;

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


   static InsertionSortDecreasing insertionSortDecreasing = new
      InsertionSortDecreasing();

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


   // Methods

   /**
    * Finds parent node
    */
   public int Parent(int i){
       return Math.floorDiv(i, 2);
   }

   /**
    * Finds left node
    */
   public int Left(int i){
       return 2*i;
   }
```

```
/**
 * Finds right node
 */
public int Right(int i){
    return (2*i)+1;
}

/**
 * makes a heap out of an array
 * @param startArray
 * @param i
 */
public void MaxHeapify(Comparable[] startArray, int i){
    int largest;
    int l = Left(i);
    int r = Right(i);
    if (l <= startArray.length && startArray[l].compareTo(
       startArray[i]) == 1)
        largest = 1;
    else largest = i;
    if (r <= startArray.length && startArray[r].compareTo(
       startArray[largest]) == 1)
        largest = r;
    if (largest != i){
        this.exch(startArray, i, largest);
        this.MaxHeapify(startArray, largest);
    }
}

/**
 * Builds a max heap
 * @param startArray
 */
@SuppressWarnings("rawtypes")
public void BuildMaxHeap( Comparable[] startArray){
    for(int i = Math.floorDiv(this.largest(startArray), 2); i
       ==1; i--){
        this.MaxHeapify(startArray, i);
    }
}



/**
 * Finds the largest element in a list
 * @param startArray
 * @return the element
 */
public int largest(Comparable[] startArray){
    int largest = 0;
    for (int i = 0; i < startArray.length-1; i++){
        if (startArray[i].compareTo(startArray[i+1]) == -1)
            largest = (int) startArray[i+1];
    }
    return largest;
}


/*
   ****************************************************************
 * This method uses heap sort to sort an array in increasing
    order
 *
 * @param startArray, the array needed to be sorted
 *
 ****************************************************************
    */
public void SortHeap(Comparable[] startArray) {
    //confirming preconditions
    if(debug)
        assert(startArray[0] != null);
    this.BuildMaxHeap(startArray);
    for (int i = startArray.length; i == 2; i--){
        //confirming invariant
        if (debug)
            assert(this.IsSorted(startArray));
        this.exch(startArray, 1, i);
        int n = startArray.length;
        n = startArray.length - 1;
        this.MaxHeapify(startArray, 1);
    }
    //confirming postconditions
    if(debug)
        assert(this.IsSorted(startArray));
}

 /**
  * exchange a[i] and a[j]
  * @param a, and array
```

```
      * @param i, index
      * @param j, index
      */
        private void exch(Comparable[] a, int i, int j) {
            Comparable swap = a[i];
            a[i] = a[j];
            a[j] = swap;
        }

    /**
     * This method times how long it takes to run heap 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.SortHeap(Array);
        long end = System.nanoTime();
        long duration = end - start;
        return duration;
    }

    /**
     * This method confirms that an array is in sorted order.
     *
     * @param Array Array, the array who is to be confirmed
     * @return boolean sorted. True if the array is sorted, false
        if not
     */

    public boolean IsSorted(Comparable[] Array){
        boolean sorted = true;
        for (int i = 0; i < Array.length -1; i++){
            if (Array[i].compareTo(Array[i+1]) == 1){
                sorted = false;
                break;
            }
        }
        return sorted;
    }
    /**
     * runs the comparison analysis
     */

    public void run(){
        int n = 0;
        while (n <= 27400){

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

       Random generator = new Random();
       for(int i = 0; i < n; i++)
        average[i] = generator.nextInt(n);


        /**
         * Will print out the time as csv of the average, best,
            and worst cases
         */

        System.out.println(n + ”;” + this.TimeToSort(Arrays.copyOf
           (average,n)) + ”;” + this.TimeToSort(Arrays.copyOf(
           average,n)) + ”;” + this.TimeToSort(Arrays.copyOf(
           average,n)));

        n = n+50;
        }
    }
}
```

Appendix E: QuickSort Class

The following is the Java class written for Quick Sort. This class provides a method that uses the Quick 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 MergeSort method, this class provides a second method, which times the execution process.

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


/**
 *  COMP 215: Design and Analysis of Algorithms: Programming
    Assignment<br>
 *
 *  The <code>QuickSort</code> class provides an algorithm that
    will sort a given
 *  array into increasing order using the QuickSort 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 code will be used to compare the worst case, best case,
    and average case run times
 *  for Quick Sort
 *  <br> <br>
 *  Created: <br>
 *     [02,05,2016], [Melany Diaz]<br>
 *  Modifications: <br>
 *
 *
 *  @author [Melany Diaz]
 */

public class QuickSort
```

```
{
    // instance variables
    Comparable[] startArray;

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

    boolean debug = true;

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

    static Random generator = new Random();

    static InsertionSortDecreasing insertionSortDecreasing = new InsertionSortDecreasing();

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


    // Methods

    /**
     * This method uses Quick sort to sort an array in increasing order
     *
     * @param startArray, the array needed to be sorted
     * @param int p
     * @param int r
     *
     * @return startArray, a permutation of the original
     * array but sorted in increasing order
     *
     */
    @SuppressWarnings(”rawtypes”)
    public void RandomizedQuicksort( Comparable[] startArray, int
       p, int r) {
         // precondition: a[lo .. mid] and a[mid+1 .. hi] are
            sorted subarrays
        if(debug){
            assert isSorted(startArray, p, r);
        }
        if (p < r){
//          int q = this.partition(startArray, p, r);
//          this.SortQuick(startArray, p, q-1);
//          this.SortQuick(startArray, q+1, r);
            int q = this.RandomizedPartition(startArray, p, r);
            this.RandomizedQuicksort(startArray, p, q-1);
            this.RandomizedQuicksort(startArray, q+1, r);
        }
        // postcondition: a[lo .. hi] is sorted
        if(debug){
            assert isSorted(startArray, p, r);
        }
    }

    /**
     * Partitions the array
     *
     * @param startArray
     * @param p
     * @param r
     * @return
     */
    private int partition(Comparable[] startArray, int p, int r) {
        int x = (int) startArray[r];
        int i = p-1;
        for ( int j = p; j<=r-1; j++){
            int y = (int) startArray[j];
            if(y <= x){
                i = i+1;
                this.exch(startArray, i, j);
            }
        }
        this.exch(startArray, i+1, r);
        return i+1;
    }
```

```
/**
  * exchange a[i] and a[j]
  * @param a, and array
  * @param i, index
  * @param j, index
  */

/**
 *  Helper sorting functions.
 */

 // is v < w ?
 private static boolean less(Comparable v, Comparable w) {
     return v.compareTo(w) < 0;
 }

 private void exch(Comparable[] a, int i, int j) {
     Comparable swap = a[i];
     a[i] = a[j];
     a[j] = swap;
    }

/**
 * This method times how long it takes to run Quick 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){
    int lowIndex = 0;
    int highIndex = Array.length -1;
    long start= System.nanoTime();
    this.RandomizedQuicksort(Array, lowIndex, highIndex);
    long end = System.nanoTime();
    long duration = end - start;
    return duration;
}

/**
 * confirms that an array is in sorted order.
 *
 * @param Array Array, the array who is to be confirmed
 * @return boolean sorted. True if the array is sorted, false
    if not
 */
private static boolean isSorted(Comparable[] a) {
    return isSorted(a, 0, a.length - 1);
 }

 private static boolean isSorted(Comparable[] a, int lo, int
    hi) {
     for (int i = lo + 1; i <= hi; i++)
         if (less(a[i], a[i-1])) return false;
     return true;
 }

 public int RandomizedPartition(Comparable[] A, int p,int r){
   int i = generator.nextInt((r - p) + 1) + p;
   this.exch(A, r, i);
   return this.partition(A, p, r);
 }

/**
 * runs the comparison analysis
 *
 */
public void run(){
    int n = 5;
    while (n <= 27400){

    //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 random cases
     */

    /*
     * Make the array used for random case and fill the
        average array
     * with n random objects.
     */
```

```
        for(int i = 0; i < n; i++)
         average[i] = generator.nextInt(n);

        this.RandomizedPartition(average, 0, average.length −1);


         /*
          * The following code finds the best case scenario.
          */

         increasing = Arrays.copyOf(average,n);


         /*
          * The following code finds the worst case scenario.
          */

         decreasing = Arrays.copyOf(average,n);

         /**
          * Will print out the time as csv of the average, best,
              and worst cases
          */

         System.out.println(n + ”;” + this.TimeToSort(Arrays.copyOf
            (average,n)) + ”;” + this.TimeToSort(Arrays.copyOf(
            increasing,n)) + ”;” + this.TimeToSort(Arrays.copyOf(
            decreasing,n)));

         n = n+200;
         }

    }


}
```

REFERENCES

[1]Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

[2]Heap sort. (n.d.). Retrieved February 14, 2016, from http://www.slideshare.net/saraeida/heap-sort-use-jing

[3]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/.

[4]Merge.java. (n.d.). Retrieved February 14, 2016, from http://algs4.cs.princeton.edu/22mergesort/Me

[5]"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.

[6]Quiles Luis. merge Sort. Florida Institute of Technology.