ECE 3790 Lab #1

Submitted by: Richard Constantine (#7686561)

Part 1 (Option 1)

The results of this section are based on the program shown in Appendix 1. The program is designed to compare 6 searches using the System.NanoTime() utility (provided by Java) to record and compare the run-times. Each sorting algorithm will be tested 50 and 1000 times, with each execution of the program using a list of numbers (either randomly or partly sorted) between 1 and 1,000,000. The program also checks the array after each trial to ensure the list is correctly sorted, as well as prints the average run-time, variance and standard deviation. Sample output is provided in Appendix 2 showing the program compare the average run-times of 50 trials on an input of 100 integers.

The algorithms/sorts that will be examined in particular are:

- Insertion Sort – worst case run-time $O(n^2)$
  - Works by sorting each element individually into a new list.
- Merge Sort – worst case run-time O(nlogn)
  - Works by dividing the list into individual elements, then merging/comparing each pair into sub-lists until the final, completed sub-list is acquired.
- Mutually Recursive Sort – worst case run-time $O(n^2)$
  - Utilizes partitioning from quicksort, but completes the sort by merging the elements in a similar fashion to merge sort.

These will be benchmarked against:

- Quicksort – worst case run-time $O(n^2)$
  - Works by recursively partitioning the list using pivot points selected by the median-of-three technique, then sorting the list around these pivots.
- Hybrid Quicksort – worst case run-time of $O(n^2)$
  - Utilizes quick sort on the large sections of the list, but begins to use insertion sort on the smaller sub-lists created by the partitions of the quicksort.

CPU: Intel Core i7-3770 @ 3.4 GHz – RAM: 8 GB – L1 Data/Instruction: 4 x 32kB – L2 Cache: 4 x 256kB – L3 Cache: 8MB

1) Quick sort is chosen as the benchmark algorithm because it sorts the list in place, requires less cache memory than other conventional sorting algorithms, and is "often the best practical choice for sorting because it is remarkably efficient on the average; …". [1]

2) Average Run-times using 50 Trials:

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
10 Random Integers	1.0×10^3	6.1×10^3	1.3×10^3	4.0×10^3	5.0×10^3	2.2×10^3

100 Random Integers	5.9×10^4	1.7×10^4	1.7×10^4	2.7×10^4	1.2×10^4	1.1×10^3
1,000 Random Integers	2.7×10^5	1.7×10^5	1.1×10^5	1.4×10^5	5.9×10^4	5.0×10^4
100,00 Random Integers	1.3×10^7	1.3×10^6	9.5×10^5	1.0×10^6	7.2×10^5	5.7×10^5

Average Run-times using 1000 Trials:

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
10 Random Integers	4.9×10^2	1.0×10^3	8.2×10^2	1.7×10^3	8.0×10^2	8.3×10^2
100 Random Integers	8.4×10^3	9.6×10^3	4.3×10^3	6.3×10^3	2.6×10^3	3.4×10^3
1,000 Random Integers	1.4×10^5	6.5×10^4	5.5×10^4	5.4×10^4	3.9×10^4	3.6×10^4
100,00 Random Integers	1.2×10^7	9.9×10^5	7.3×10^5	9.0×10^5	6.3×10^5	4.6×10^5

Plots for these results can be found in Appendix 3.

3) Variance & Standard Deviation using 50 Trials:

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
10 Random Integers	σ^2: 1.6×10^6 σ: 1.3×10^3	σ^2: 1.6×10^7 σ: 3.9×10^3	σ^2: 2.0×10^6 σ: 1.4×10^3	σ^2: 8.9×10^6 σ: 3.0×10^3	σ^2: 1.7×10^7 σ: 4.2×10^3	σ^2: 3.7×10^6 σ: 1.9×10^3
100 Random Integers	σ^2: 1.3×10^9 σ: 3.6×10^4	σ^2: 4.1×10^8 σ: 2.0×10^4	σ^2: 3.4×10^6 σ: 1.8×10^3	σ^2: 7.3×10^8 σ: 2.7×10^4	σ^2: 6.3×10^7 σ: 8.0×10^4	σ^2: 1.2×10^7 σ: 3.5×10^3
1,000 Random Integers	σ^2: 2.2×10^{11} σ: 4.7×10^5	σ^2: 8.2×10^9 σ: 9.1×10^4	σ^2: 2.5×10^8 σ: 1.5×10^4	σ^2: 6.6×10^9 σ: 8.1×10^4	σ^2: 7.8×10^8 σ: 2.8×10^5	σ^2: 2.7×10^8 σ: 1.6×10^4
100,00 Random Integers	σ^2: 1.7×10^{13} σ: 4.1×10^6	σ^2: 2.2×10^{11} σ: 4.7×10^5	σ^2: 4.6×10^{10} σ: 2.1×10^5	σ^2: 8.3×10^{10} σ: 2.9×10^5	σ^2: 3.6×10^9 σ: 6.0×10^4	σ^2: 5.8×10^9 σ: 7.6×10^5

Variance & Standard Deviation using 1000 Trials:

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
10 Random Integers	σ^2: 7.9×10^5 σ: 8.9×10^2	σ^2: 1.8×10^6 σ: 1.3×10^3	σ^2: 5.2×10^5 σ: 7.1×10^2	σ^2: 3.2×10^6 σ: 1.8×10^3	σ^2: 4.3×10^6 σ: 2.1×10^3	σ^2: 1.0×10^6 σ: 1.0×10^3
100 Random Integers	σ^2: 2.3×10^8 σ: 1.5×10^4	σ^2: 4.3×10^7 σ: 6.6×10^3	σ^2: 5.0×10^6 σ: 2.2×10^3	σ^2: 6.2×10^7 σ: 7.8×10^3	σ^2: 3.3×10^6 σ: 1.8×10^3	σ^2: 8.5×10^6 σ: 2.9×10^3
1,000 Random Integers	σ^2: 1.3×10^{10} σ: 1.2×10^5	σ^2: 8.3×10^8 σ: 2.9×10^4	σ^2: 2.7×10^8 σ: 1.6×10^4	σ^2: 7.8×10^8 σ: 2.9×10^4	σ^2: 1.0×10^8 σ: 1.0×10^4	σ^2: 8.9×10^7 σ: 9.4×10^3
100,00 Random Integers	σ^2: 1.1×10^{12} σ: 1.0×10^6	σ^2: 1.5×10^{10} σ: 1.2×10^5	σ^2: 7.6×10^9 σ: 8.7×10^5	σ^2: 9.5×10^9 σ: 9.7×10^5	σ^2: 1.9×10^9 σ: 4.4×10^5	σ^2: 9.0×10^8 σ: 3.0×10^5

It can be observed that increasing the number trials has little affect variance and slight effect on average run time.

This large variance in processing speed is very large due to many reasons:

- the measurements provided by the Java utility are quite precise however not very accurate
- processors do not execute instructions at the same exact speed
- dependant on background processes and other interferences
- dependant on hardware limitations (RAM/cache memory) and how much each algorithm utilizes the hardware

Since this lab is only to **approximate** the run-times of the algorithms and since the variance cannot be efficiently reduced without a more accurate measurement tool or far more trials (i.e. better hardware or timing equipment) we will say this variance/standard deviation is sufficient for our purposes of estimation and will rely on averaging the run times.

4) The almost sorted list is acquired by designating a random number (between 1 to 1,000,000) to every 5^{th} element in the list (therefore about 4/5 or 80% pre-sorted), otherwise, the remaining elements are in sorted order.

Almost sorted using 50 trials:

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
10 Random Integers	1.3×10^3	4.6×10^3	1.4×10^3	4.0×10^3	4.9×10^3	3.1×10^3

100 Random Integers	4.2×10^4	2.0×10^4	1.1×10^4	3.1×10^4	1.3×10^4	1.3×10^4
1,000 Random Integers	1.8×10^5	1.6×10^5	9.5×10^4	1.2×10^5	2.5×10^5	1.8×10^5
100,00 Random Integers	4.7×10^6	8.7×10^5	6.1×10^5	5.8×10^5	4.2×10^6	4.6×10^6

Almost sorted using 1000 trials:

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
10 Random Integers	4.4×10^2	1.2×10^3	1.0×10^3	2.0×10^3	6.0×10^2	8.1×10^2
100 Random Integers	8.4×10^3	7.9×10^3	6.0×10^3	7.3×10^3	6.0×10^3	5.3×10^3
1,000 Random Integers	5.7×10^4	4.9×10^4	4.2×10^4	4.6×10^4	2.0×10^5	2.0×10^5
100,00 Random Integers	4.4×10^6	6.2×10^5	4.5×10^5	4.6×10^5	2.2×10^6	2.0×10^6

Figures are located in Appendix 3.

Observations:

- When the list is almost sorted, quicksort runs much slower because this represents its worst case scenario.
- Merge sort now looks to be the fastest sort under these conditions.
- Appendix 4 shows that the quicksort algorithms run almost as slow as insertion sort under its worst condition (lots of data)

5) The almost sorted list has a large impact on the quicksort run time because this represents the worst case scenario for a quick sort (i.e. behaves like $O(n^2)$).

6) The run time/performance of each algorithm is estimated by recording the approximate time it takes for the particular algorithm to complete a single sort, then averaging these times over 50 or 1000 trials. Since we are only estimating the performance of each algorithm, a run-time versus data size plot is sufficient for our requirements.
However, since things like cache memory usage, CPU instructions and even the environment can affect the behavior of these algorithms, other methods (like counting number of comparisons/primitive operations or how much cache memory is used) can be useful metrics of performance.

7) A stable sort "breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array." [1] For example, if a hand of playing cards is to be sorted by numerical value, and two cards of equal value have different suits, the order of those cards (with respect to the suits) must be maintained.

8) Largest possible sorts using 50 trials

	Insertion Sort (Avg. Time in Nanoseconds)	Recursive Merge Sort (Avg. Time in Nanoseconds)	Iterative Merge Sort (Avg. Time in Nanoseconds)	Mutually Recursive Sort (Avg. Time in Nanoseconds)	**Quicksort (Avg. Time in Nanoseconds)**	**Hybrid Quicksort (Avg. Time in Nanoseconds)**
100,000 Random Integers (Used for SedgeSort analysis)	1.2×10^9	1.2×10^7	9.2×10^6	1.1×10^7	7.8×10^6	8.7×10^5
250,000 Random Integers	7.7×10^9	3.2×10^7	2.5×10^7	2.7×10^7	2.0×10^7	5.8×10^7

Using a stop watch, it took a total time of 6:35.85 minutes to finish executing the program on a sorting input of 250,000 integers. With better hardware or more time, possibly more could be accomplished – for our purposes this is sufficient.

When compared to Sedgesort, which was capable of sorting 100,000 elements in 4.7×10^{-3} seconds (minima) [2], our fastest sort, Quicksort, was only able to sort the list in 7.8×10^{-3} seconds.

Comparing Sedgesort in C, we can use the clock() function to estimate the runtime of the algorithm. When compared to Quicksort, we find that Sedgesort organizes the list of 100,000 elements in 78 local clock cycles (LCCs), however Quicksort takes a total of 235 LCCs (i.e. Sedgesort is again much faster).

Appendix 4 shows the C code used, along with a sample output.

Part 2

The code developed for this section along with examples of its execution are located in Appendices 5 and 6 respectively.

References

[1] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, Mass.: MIT Press, 2009.

[2] A. Faigon, "A library of internal sorting routines", *Yendor.com*, 2017. [Online]. Available: http://www.yendor.com/programming/sort/.

Appendix 1 – Part 1 Code

Source: Richard Constantine

Developed using examples given from Comp 2140 at the U. of M.

```
import java.util.Scanner;

public class CompareSorts {


  public static void main(String[] args) {

    final long BIG_NUMBER = Long.MAX_VALUE; // used to compare the quickest sort time

    int numTrials = 50; // number of trials (ie how many times each individual sort is executed)
    int breakpoint; // for use with hybrid quick sort
    long startTime, endTime, elapsedTime, sumTime, avgTime; // variables used in time calculations
    long quickestTime = BIG_NUMBER; // used to track the quickest average sort time
    long[] time = new long[numTrials]; // used to track the elapsed time of each trial
    double variance, stdDev; // used to store the variance and standard deviationof each sort
    String quickestSort = ""; // used to store the name of the quickest sort when identified

    // Create Unsorted List
**********************************************************************************

    int numInts = 250000; // number of integers to be sorted

    int[] arrayOrignal = new int[numInts]; // initalize array that will store the list of integers to be sorted
    int[] arrayCopy = new int[numInts]; // initialize array that will store a copy the list of integers to be sorted
    breakpoint = numInts/100; // define breakpoint as 1/100 the list size (arbitrary);

    Scanner keyboard = new Scanner(System.in); // read input via keyboard
    System.out.println("Is the list almost sorted? (type 'yes' if almost sorted, type anything else for 'no')");
    String input = keyboard.next();


    // almost sorted array
    if(input.equals("yes")){
      for(int i = 0; i < numInts; i++){
        if(i%5 == 0)
          arrayOrignal[i] = (int)(Math.random() * 1000000); // assign random number between 1 and 1,000,000
        else
          arrayOrignal[i] = i;
      }//for
    }//if

    // randomly sorted array
    else{
      for(int i = 0; i < numInts; i++)
        arrayOrignal[i] = (int)(Math.random() * 1000000); // assign random number between 1 and 1,000,000
    }//else

    System.out.println("Comparing multiple sorts.\n");
```

```
   // Perform Sorts
*********************************************************************************

   // insertion sort
*********************************************************************************

   sumTime = 0; // initialize sumTime
   for (int i = 0; i < numTrials; i++){;
    System.arraycopy(arrayOrignal, 0, arrayCopy, 0, numInts); // copy orignal array into the duplicate array (ie
reset)
    startTime = System.nanoTime(); // record start time
    insertionSort(arrayCopy);// call to insertion sort
    endTime = System.nanoTime(); // record stop time
    if (!isSorted(arrayCopy)) // check if the array is sorted
     System.out.println("ERROR - NOT SORTED"); // print statement if not sorted
    elapsedTime = endTime - startTime; // calculate elapsed time
    time[i] = elapsedTime; // record time
    sumTime += elapsedTime; // add the elapsed time to running sum
   }//for

   // mean
   avgTime = sumTime/numTrials; // average the sum run time over the number of trials

   // variance
   variance = 0; // used to store the variance of each sort
   for(int i = 0; i < numTrials; i++)
    variance += Math.pow((time[i] - avgTime), 2);

   variance = variance / numTrials; // final value for variance

   // standard deviation
   stdDev = Math.sqrt(variance);

   if (avgTime < quickestTime){ // checks if this is the quickest average sorting time so far
    quickestTime = avgTime; // if quickest, change the quickest time
    quickestSort = "insertion sort"; // store the name of the quickest sort
   }//if

   System.out.println("Average time for insertion sort: " + avgTime + " nanoseconds");
   System.out.println("Variance for insertion sort: " + variance + " nanoseconds");
   System.out.println("Standard Deviation for insertion sort: " + stdDev + " nanoseconds\n");

   // recursive merge sort
*********************************************************************************

   sumTime = 0; // initialize sumTime
   for (int i = 0; i < numTrials; i++){;
    System.arraycopy(arrayOrignal, 0, arrayCopy, 0, numInts); // copy orignal array into the duplicate array (ie
reset)
    startTime = System.nanoTime(); // record start time
    recursiveMergeSort(arrayCopy);// call to recursive merge sort
    endTime = System.nanoTime(); // record stop time
```

```
    if (!isSorted(arrayCopy)) // check if the array is sorted
      System.out.println("ERROR - NOT SORTED"); // print statement if not sorted

    elapsedTime = endTime - startTime; // calculate elapsed time
    time[i] = elapsedTime; // record time
    sumTime += elapsedTime; // add the elapsed time to running sum
   }//for

   // mean
   avgTime = sumTime/numTrials; // average the sum run time over the number of trials

   // variance
   variance = 0; // used to store the variance of each sort
   for(int i = 0; i < numTrials; i++)
    variance += Math.pow((time[i] - avgTime), 2);

   variance = variance / numTrials; // final value for variance

   // standard deviation
   stdDev = Math.sqrt(variance);

   if (avgTime < quickestTime){ // checks if this is the quickest average sorting time so far
    quickestTime = avgTime; // if quickest, change the quickest time
    quickestSort = "recursive merge sort"; // store the name of the quickest sort
   }//if

   System.out.println("Average time for recursive merge sort: " + avgTime + " nanoseconds");
   System.out.println("Variance for recursive merge sort: " + variance + " nanoseconds");
   System.out.println("Standard Deviation for recursive merge sort: " + stdDev + " nanoseconds\n");

   // iterative merge sort
*********************************************************************************

   sumTime = 0; // initialize sumTime
   for (int i = 0; i < numTrials; i++){;
    System.arraycopy(arrayOrignal, 0, arrayCopy, 0, numInts); // copy orignal array into the duplicate array (ie
reset)
    startTime = System.nanoTime(); // record start time
    iterativeMergeSort(arrayCopy);// call to iterative merge sort
    endTime = System.nanoTime(); // record stop time

    if (!isSorted(arrayCopy)) // check if the array is sorted
      System.out.println("ERROR - NOT SORTED"); // print statement if not sorted

    elapsedTime = endTime - startTime; // calculate elapsed time
    time[i] = elapsedTime; // record time
    sumTime += elapsedTime; // add the elapsed time to running sum
   }//for

   // mean
   avgTime = sumTime/numTrials; // average the sum run time over the number of trials
```

```
   // variance
   variance = 0; // used to store the variance of each sort
   for(int i = 0; i < numTrials; i++)
     variance += Math.pow((time[i] - avgTime), 2);

   variance = variance / numTrials; // final value for variance

   // standard deviation
   stdDev = Math.sqrt(variance);

   if (avgTime < quickestTime){ // checks if this is the quickest average sorting time so far
     quickestTime = avgTime; // if quickest, change the quickest time
     quickestSort = "iterative merge sort"; // store the name of the quickest sort
   }//if

   System.out.println("Average time for iterative merge sort: " + avgTime + " nanoseconds");
   System.out.println("Variance for iterative merge sort: " + variance + " nanoseconds");
   System.out.println("Standard Deviation for iterative merge sort: " + stdDev + " nanoseconds\n");

   //mutally recursive merge sort and quick sort
*********************************************************************************

   sumTime = 0; // initialize sumTime
   for (int i = 0; i < numTrials; i++){; // loop over and sort 50 times
     System.arraycopy(arrayOrignal, 0, arrayCopy, 0, numInts); // copy orignal array into the duplicate array (ie
reset)
     startTime = System.nanoTime(); // record start time
     mutuallyRecursiveSort(arrayCopy);// call to quick sort
     endTime = System.nanoTime(); // record stop time

     if (!isSorted(arrayCopy)) // check if the array is sorted
       System.out.println("ERROR - NOT SORTED"); // print statement if not sorted

     elapsedTime = endTime - startTime; // calculate elapsed time
     time[i] = elapsedTime; // record time
     sumTime += elapsedTime; // add the elapsed time to running sum
   }//for

   // mean
   avgTime = sumTime/numTrials; // average the sum run time over the number of trials

   // variance
   variance = 0; // used to store the variance of each sort
   for(int i = 0; i < numTrials; i++)
     variance += Math.pow((time[i] - avgTime), 2);

   variance = variance / numTrials; // final value for variance

   // standard deviation
   stdDev = Math.sqrt(variance);

   if (avgTime < quickestTime){ // checks if this is the quickest average sorting time so far
     quickestTime = avgTime; // if quickest, change the quickest time
```

```
    quickestSort = "mutally recursive sort"; // store the name of the quickest sort
   }//if

   System.out.println("Average time for mutually recursive sort: " + avgTime + " nanoseconds");
   System.out.println("Variance for mutually recursive sort: " + variance + " nanoseconds");
   System.out.println("Standard Deviation for mutually recursive sort: " + stdDev + " nanoseconds\n");

   // quick sort
*********************************************************************************

   sumTime = 0; // initialize sumTime
   for (int i = 0; i < numTrials; i++){;
    System.arraycopy(arrayOrignal, 0, arrayCopy, 0, numInts); // copy orignal array into the duplicate array (ie
reset)
    startTime = System.nanoTime(); // record start time
    quickSort(arrayCopy);// call to quick sort
    endTime = System.nanoTime(); // record stop time

    if (!isSorted(arrayCopy)) // check if the array is sorted
     System.out.println("ERROR - NOT SORTED"); // print statement if not sorted

    elapsedTime = endTime - startTime; // calculate elapsed time
    time[i] = elapsedTime; // record time
    sumTime += elapsedTime; // add the elapsed time to running sum
   }//for

   // mean
   avgTime = sumTime/numTrials; // average the sum run time over the number of trials

   // variance
   variance = 0; // used to store the variance of each sort
   for(int i = 0; i < numTrials; i++)
    variance += Math.pow((time[i] - avgTime), 2);


   variance = variance / numTrials; // final value for variance

   // standard deviation
   stdDev = Math.sqrt(variance);

   if (avgTime < quickestTime){ // checks if this is the quickest average sorting time so far
    quickestTime = avgTime; // if quickest, change the quickest time
    quickestSort = "quicksort"; // store the name of the quickest sort
   }//if

   System.out.println("Average time for quick sort: " + avgTime + " nanoseconds");
   System.out.println("Variance for quick sort: " + variance + " nanoseconds");
   System.out.println("Standard Deviation for quick sort: " + stdDev + " nanoseconds\n");

   // hybrid quick sort
*********************************************************************************

   sumTime = 0; // initialize sumTime
```

```
   for (int i = 0; i < numTrials; i++){; // loop over and sort 50 times
     System.arraycopy(arrayOrignal, 0, arrayCopy, 0, numInts); // copy orignal array into the duplicate array (ie
reset)
     startTime = System.nanoTime(); // record start time
     hybridQuickSort(arrayCopy, breakpoint);// call to quick sort
     endTime = System.nanoTime(); // record stop time

     if (!isSorted(arrayCopy)) // check if the array is sorted
       System.out.println("ERROR - NOT SORTED"); // print statement if not sorted

     elapsedTime = endTime - startTime; // calculate elapsed time
     time[i] = elapsedTime; // record time
     sumTime += elapsedTime; // add the elapsed time to running sum
   }//for

   // mean
   avgTime = sumTime/numTrials; // average the sum run time over the number of trials

   // variance
   variance = 0; // used to store the variance of each sort
   for(int i = 0; i < numTrials; i++)
     variance += Math.pow((time[i] - avgTime), 2);

   variance = variance / numTrials; // final value for variance

   // standard deviation
   stdDev = Math.sqrt(variance);

   if (avgTime < quickestTime){ // checks if this is the quickest average sorting time so far
     quickestTime = avgTime; // if quickest, change the quickest time
     quickestSort = "hybrid quicksort"; // store the name of the quickest sort
   }//if

   System.out.println("Average time for hybrid quick sort: " + avgTime + " nanoseconds");
   System.out.println("Variance for hybrid quick sort: " + variance + " nanoseconds");
   System.out.println("Standard Deviation for hybrid quick sort: " + stdDev + " nanoseconds\n");


System.out.println("\n*************************************************************************\
n");

   System.out.println("The quickest sorting method is " + quickestSort + ".");

   System.out.println("It sorted all " + numInts + " numbers in an average time of " + quickestTime +
                      " over " + numTrials + " trials.");


System.out.println("\n*************************************************************************\
n");

   System.out.println("Program Ends");
  }//main
```

```
 /*
  * [The main method reads a file containing integers (each on separate lines) and stores them into an array via
  *  a scanner, a try/catch statement, and a for loop. This array is then sorted by a variety of methods, in
  *  particular: insertion sort, recursive & iterative merge sort, quick sort, insertion/quick hybrid sort and
  *  the mutually recursive merge/quick sort. These sorts are then timed using the system clock (in
  *  nanoseconds) over 50 trials. The average time of each sort as well as the quickest sort are displayed to the
  *  user.]
  * [Input: Recieves arguments (as an aray of strings) when the program is executed]
  * [Output: Prints the average time taken by each sort as well which sort finished the quickest]
  *
  * @param [firstParam: string value containing arguments passed by the user]
  * @return [Void - n/a]
  */

 private static void insertionSort (int[] a, int start, int end){ // **Part 1**
   int siftVal; // item to be sifted
   int j; // a[j] is to be compared to siftVal

   for (int i = start; i < end; i++){ // this condition sorts a[start] to a[end-1]
     siftVal = a[i];
     j = i-1;

     while (j >= start && a[j]  > siftVal){ /* only shift item right if siftVal is less then the unmoved item or
      * if it is the last value to be moved (ie first value in the list)
      */
       a[j+1] = a[j]; // move item right
       j--;
     }//while
     a[j+1] = siftVal; // place the siftVal in the open position provided by the while loop
   }//for

 }//insertionSort

 /*
  * [Insertion sort uses 2  pointers (ie int i, j) as well as a temp value (siftVal) to sort the list. It operates by
  *  saving the an array element (starting with 2nd element) into siftVal. It then compares the previous elements to
  *  locate its postion in the sorted portion of the array (a[start] to a[i-1] and shifts right any values greater
  *  then siftVal. Finally it places siftVal into the correct position in the sorted part of the array.]
  * [Input: Receives the array to be sorted, as well as the starting and ending position of the sort from the driver
  *  method (or calling method) and acts to sort the given list]
  * [Output: The portion of the array passed to the method, and defined by start and end, will be sorted in
ascending
  *  order]
  *
  * @param [firstParam: the integer array to be sorted]
  * @param [secondParam; the starting postion within the array to begin sorting (usually start of the array)]
  * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
  *       array]
  * @return [Void - n/a]
  */

 public static void insertionSort (int[] a){
```

```
   insertionSort(a, 0, a.length); // call to insertionSort, providing correct parameters (entire array)

  }//insertionSort driver

  /*
   * [The insertion sort driver allows for easy and convienent calling of the insertionSort method from the main
method]
   * [Input: Receives the array to be sorted from the main method (or calling method) and acts to sort the given list]
   * [Output: The array is sorted via insertion sort]
   *
   * @param [firstParam: the integer array to be sorted]
   * @return [Void - n/a]
   */

  private static void recursiveMergeSort (int[] a, int start, int end, int[] temp){ // **Part 2**
    int mid; // tracks the middle position of the list

    if (1 < end - start){ // this condition handles the base case (ie < 2 items then do nothing)
      mid = start + (end-start)/2; // calculate mid point

      recursiveMergeSort(a, start, mid, temp); // sort lower half of array
      recursiveMergeSort(a, mid, end, temp); // sort upper half of array
      recursiveMerge(a, start, mid, end, temp); // merges/sorts the the current section of the array
    }//if

  }//recursiveMergeSort

  /*
   * [Recursive merge sort works by dividing the list into individual elements by recursively dividing the list in half
   *  multiple times. It then calls the recursiveMerge method (below) to compare/sort each element in pairs. These
   *  sorted pairs are then merged/sorted into a temporary array. It now begins comparing these newly formed
pairs and
   *  merges these into the temp array (which is now slightly more sorted). These merged pairs are then compared
and so
   *  on until the array finally broken into 2 sub groups. These are then sorted/merged and contents of the now
sorted
   *  temp array are copied back to provide the final sorted result.]
   * [Input: Receives the array to be sorted, the starting and ending position, as well as the temp array (of same size
   *  as the orignal array) from the driver method (or calling method) and acts to sort the given list]
   * [Output: The portion of the array passed to the method, and defined by start and end, will be sorted in
ascending
   *  order]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *       array]
   * @param [fourthParam; the temporary integer array that will be used to perform the intermediary sorts]
   * @return [Void - n/a]
   */
```

```
  private static void recursiveMerge (int[] a, int start, int mid, int end, int[] temp){
    int currL = start; // pointer to track position in lower/left section of array
    int currR = mid; // pointer to track position in upper/right section of array
    int currT;  // pointer to track position in temp array

    for (currT = start; currT < end; currT++){
      if (currL < mid && (currR >= end || a[currL] < a[currR])){ /* copy values into temp array from lower section
       * of the orignal array if that value is smaller
       * or no values remain in the upper section
       */
        temp[currT] = a[currL];
        currL++;
      }//if
      else{
        temp[currT] = a[currR]; // else copy the the right value
        currR++;
      }//else
    }//for
    for (currT = start; currT < end; currT++)
      a[currT] = temp[currT]; // hard copy the temp array back into original array

  }//recursiveMerge

  /*
   * [Recursive merge works by taken the sub-lists/pairs passed by recursiveMergeSort and sorts/merges them into
   *  the temp array.]
   * [Input: Receives the integer array to be sorted, the starting and ending position, as well as the temp array
   *  (of same size as the orignal array) from the driver method (or calling method) and acts to sort the given list]
   * [Output: The portion of the array passed to the method, and defined by start and end, will be sorted in
ascending
   *  order]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the middle postion within the array that defines the mid point between the sublists/pairs
   *       to be sorted/merged]
   * @param [fourthParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *       array]
   * @param [fifthParam; the temporary integer array that will be used to perform the intermediary sorts]
   * @return [Void - n/a]
   */

  public static void recursiveMergeSort (int[] a){

    int[] temp = new int[a.length]; // temp array used during the recursiveMerge method
    recursiveMergeSort(a, 0, a.length, temp); // call to mergeSort, providing correct parameters (entire array)

  }//public recursiveMergeSort driver

  /*
   * [The recursive merge sort driver allows for easy and convienent calling of the recursiveMergeSort method from
   *  the main method]
```

```
 * [Input: Receives the array to be sorted from the main method (or calling method) and acts to sort the given list]
 * [Output: The integer array is sorted via recursive merge sort]
 *
 * @param [firstParam: the integer array to be sorted]
 * @return [Void - n/a]
 */

public static void iterativeMergeSort (int[] a){ // **Part 3**
  int start, mid, end;
  int[] temp = new int[a.length]; // temp array used during the merge step

  for(int subListSize = 1; subListSize < a.length; subListSize *= 2){ // counts the sublist size
    for(int i = 0; i < a.length; i += (subListSize * 2)){ // moves through list and tracks start of each pair
      start = i;
      mid = i + subListSize;
      end = i + subListSize * 2;

      if(end > a.length) /* checks if the 2nd sublist is not multiple of subListSize and also accounts for an odd
       * number of sublists because recursiveMerge takes into account when mid (ie currR) >= end
       */
        end = a.length; // adjust the endpoint

      recursiveMerge(a, start, mid, end, temp);

    }//for
  }//for

}//iterativeMergeSort

/*
 * [Iterative merge sort works in almost the same fashion as the recursive merge sort, however it uses 2 for loops
(ie
 *  2 pointers) to define the sublists/pairs. The first for loop tracks the size of the sublist and the second
 *  tracks the starting position of each pair/sublist. These values can be used to identify the start, mid, and
 *  end values used by the recursiveMerge method which works to sort each sublist/pair passed to it.]
 * [Input: Receives the array to be sorted from the main method]
 * [Output: The array passed to it will be sorted]
 *
 * @param [firstParam: the integer array to be sorted]
 * @return [Void - n/a]
 */

private static void quickSort (int[] a, int start, int end ){ // **Part 4**
  int pivotPosn;

  if ((end - start) == 2 && a[start] > a[end-1]) // this condition handles list of only 2 items
    swap(a, start, end-1);

  else if (1 < (end - start)){ // this condition handles the base case (ie < 2 items then do nothing)
    choosePivot(a, start, end); // choose pivot position (using median of 3's)
    pivotPosn = partition(a, start, end); // partitions the array according the pivot
    quickSort(a, start, pivotPosn); // sort the smalls
    quickSort(a, pivotPosn + 1, end); // sort the bigs
```

```
   }//if
  }//quickSort

  /*
   * [Quick sort works by recursively choosing a pivot point, sorting all the values around this pivot point (by
   *  comparing to it), then placing the pivot into sorted postion.]
   * [Input: Receives the array to be sorted and the starting/ending position from the driver method (below) and
acts
   *  to sort the given list]
   * [Output: The portion of the array passed to the method, and defined by start and end, will be sorted in
ascending
   *  order]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *        array]
   * @return [Void - n/a]
   */

  private static int partition(int[] a, int start, int end){
    int bigStart = start + 1; //the bigs will start one after the pivot until smalls are swapped in between
    int pivot = a[start]; //pivot begins at start of the array

    for (int curr = start + 1; curr < end; curr++){
      if (a [curr] < pivot) { // belongs to smalls
        swap(a, curr, bigStart); // swap into correct position
        bigStart++;
      }//if
    }//for

    swap (a, start, bigStart - 1); // swap pivot into correct location
    return bigStart - 1; // returns the position of the pivot

  }//partition (for quickSort)

  /*
   * [The partition method works by sorting the array around the pivot chosen by the choosePivot method.]
   * [Input: Receives input from quick sort and acts to sort the arrray according to the pivot]
   * [Output: Returns the integer value of the position occupied by the pivot array element]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *        array]
   * @return [Integer value of the postion of the pivot]
   */

  private static void swap(int[] a, int  posn1, int posn2){
    int temp = a[posn1];
    a[posn1] = a[posn2];
```

```
   a[posn2] = temp;

  }//swap

  /*
   * [The swap method is a simple method that swaps 2 values in an array without data loss.]
   * [Input: Receives the array to be sorted, and the 2 postions within the array to be swapped]
   * [Output: The array passed to swap will have the element at posn1 switched with the element at posn2]
   *
   * @param [firstParam: the array to be sorted]
   * @param [secondParam; the first position of the array to swapped]
   * @param [thirdParam; the second position of the array to swapped
   * @return [Void - n/a]
   */

  private static void choosePivot(int[] a, int  start, int end){
    int mid = start + (end - start)/2; // calculate middle position
    if ((a[start] <= a[mid] && a[mid] <= a[end-1]) || (a[start] >= a[mid] && a[mid] >= a[end-1])) // check if a[mid] is
median
      swap(a, start, mid); // swap a[mid] (pivot) to start position
    else if ((a[mid] <= a[end-1] && a[end-1] <= a [start]) || (a[mid] >= a[end-1] || a[end-1] >= a[start])) // check if
a[end-1] is median
      swap(a, start, end-1); // swap a[end] (pivot) to start position
//  else if a[start] is median then do nothing

  }//choosePivot (for quickSort)

  /*
   * [The choosePivot method uses the median of three method to find the median value of first, middle and last
element
   *  of the array and use this element as the partition/pivot for the quick sort. Choosing a practical partition/pivot
   *  point speeds up the sort. It also swaps the chosen partition/pivot element into the first position of the
   *  array for use by the partition method.]
   * [Input: Receives input from quick sort acts to select a partition value using the median of three]
   * [Output: A partition/pivot is chosen and moved to the first element of the array]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array that defines where the method chooses the pivot]
   * @param [thirdParam; the ending position that defines where the method chooses the pivot]
   * @return [Void - n/a]
   */

  public static void quickSort (int[] a){

    quickSort(a, 0, a.length); // call to quickSort providing, correct parameters (entire array)

  }//quickSort driver

  /*
   * [The quick sort driver allows for easy and convienent calling of the quick sort method from the main method
   *  (or calling method).]
   * [Input: Receives the array to be sorted from the main method (or calling method) and acts to sort the given list]
   * [Output: The array is sorted via quick sort]
```

```
  *
  * @param [firstParam: the array to be sorted]
  * @return [Void - n/a]
  */

 private static void hybridQuickSort( int[] a, int start, int end, int breakpoint ){ // **Part 5**
  int pivotPosn;

  if ((end-start) == 2 && a[start] > a[end-1]) // this condition handles list of only 2 unsorted items
   swap(a, start, end-1);

  else if (2 < (end-start) && (end-start) <= breakpoint) /* this condtion handles where end-start <= breakpoint
however
   * contains more than 2 array elements
   */
   insertionSort(a, start, end);

  else if ((end-start) > breakpoint){ /* this condition handles the case where number of items > breakpoint
   * and will do nothing when the list has < 2 total items or the 2 items are
   * already sorted
   */
   choosePivot(a, start, end); // choose pivot position (using median of 3's)
   pivotPosn = partition(a, start, end); // partitions the array according the pivot
   hybridQuickSort(a, start, pivotPosn, breakpoint); // sort the smalls
   hybridQuickSort(a, pivotPosn + 1, end, breakpoint); // sort the bigs
  }//if

 }//hybridQuickSort

 /*
  * [The hybrid quick sort method uses the quick sort for the large parts of the array, but begins to use insertion
  *  sort once the number of elements in each partition are equal to or below the user specified breakpoint.]
  * [Input: Receives the array to be sorted from the driver method (below) and acts to sort the given list]
  * [Output: A partition is chosen and moved to the first element of the array]
  *
  * @param [firstParam: the array to be sorted]
  * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
  * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
  *       array]
  * @return [Void - n/a]
  */

 public static void hybridQuickSort ( int[] a, int breakpoint ){

  hybridQuickSort(a, 0, a.length, breakpoint); // call to hybridQuickSort, providing correct parameters (entire
array)

 }//public hybridQuickSort driver

 /*
  * [The hybrid quick sort driver allows for easy and convienent calling of the hybrid quick sort method from
  *  the main method (or calling method).]
```

```
  * [Input: Receives the array to be sorted and breakpoint from the main method and acts to sort the given list]
  * [Output: The array is sorted via the hybridQuickSort]
  *
  * @param [firstParam: the integer array to be sorted]
  * @return [Void - n/a]
  */

 private static void mutualQuickSort( int[] a, int start, int end, int[] temp ){ // **Part 6**
   int pivotPosn;

   if (1 < end - start){ // this condition handles the base case (ie < 2 items then do nothing)
     choosePivot(a, start, end); // choose pivot position (using median of 3's)
     pivotPosn = partition(a, start, end); // partitions the array according the pivot
     mutualMergeSort(a, start, pivotPosn, temp); // sort the smalls
     mutualMergeSort(a, pivotPosn + 1, end, temp); // sort the bigs
   }//if

 }//mutualQuickSort

 /*
   * [The mutual quick sort utilizes the skeleton of quick sort (including choosePivot and partition methods)
however
   *  calls mutualMergeSort to perform the sort via merging.]
   * [Input: Receives the array to be sorted and the starting/ending position from the driver method (below) and
acts
   *  to sort the given list]
   * [Output: The portion of the array passed to the method, and defined by start and end, will be sorted in
ascending
   *  order]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *        array]
   * @param [fourthParam; the temporary integer array that will be used to perform the intermediary sorts]
   * @return [Void - n/a]
   */

 private static void mutualMergeSort( int[] a, int start, int end, int[] temp ){
   int mid; // tracks the middle position of the list

   if (1 < end - start){ // this condition handles the base case (ie < 2 items then do nothing)
     mid = start + (end-start)/2; // calculate mid point

     mutualQuickSort(a, start, mid, temp); // sort lower half of array
     mutualQuickSort(a, mid, end, temp); // sort upper half of array
     recursiveMerge(a, start, mid, end, temp); // merges/sorts the the current section of the array
   }//if

 }//mutualMergeSort

 /*
```

```
   * [The mutual merge sort utilizes the skeleton of recursive merge sort (including the recuriveMerge method)
however
   *  calls mutualQuickSort to perform the partition and recursiveMerge to perform the sort.]
   * [Input: Receives the array to be sorted and the starting/ending position from the driver method (below) and
acts
   *  to sort the given list]
   * [Output: The portion of the array passed to the method, and defined by start and end, will be sorted in
ascending
   *  order]
   *
   * @param [firstParam: the integer array to be sorted]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *        array]
   * @param [fourthParam; the temporary integer array that will be used to perform the intermediary sorts]
   * @return [Void - n/a]
   */

  public static void mutuallyRecursiveSort(int[] a){

    int[] temp = new int[a.length]; // temp array used during the mutuallyRecursiveSort method
    mutualQuickSort(a, 0, a.length, temp); // call to mutalQuickSort, providing correct parameters (entire array)

  }//mutallyRecursiveSort driver

  /*
   * [The mutallyRecursiveSort driver allows for easy and convienent calling of the mutallyRecursiveSort method
from
   *  the main method (or calling method).]
   * [Input: Receives the array to be sorted and acts to sort the given list]
   * [Output: The array is sorted via the mutallyRecursiveSort]
   *
   * @param [firstParam: the integer array to be sorted]
   * @return [Void - n/a]
   */

  private static boolean isSorted( int[] a, int start, int end ){ // **Part 7**
    boolean result = true;

    for (int i = start+1; i < end && result; i++){ /* loop through the array while comparing each subsequent value
     * to the previous one - exit only if not sorted or the end is reached
     */
      boolean lessThan = a[i-1] <= a[i]; // this method ensures the array index is not out of bounds
      if (!lessThan)
        result = false; // return false if any part of the array is not sorted
    }//for

    return result;

  }//isSorted

  /*
```

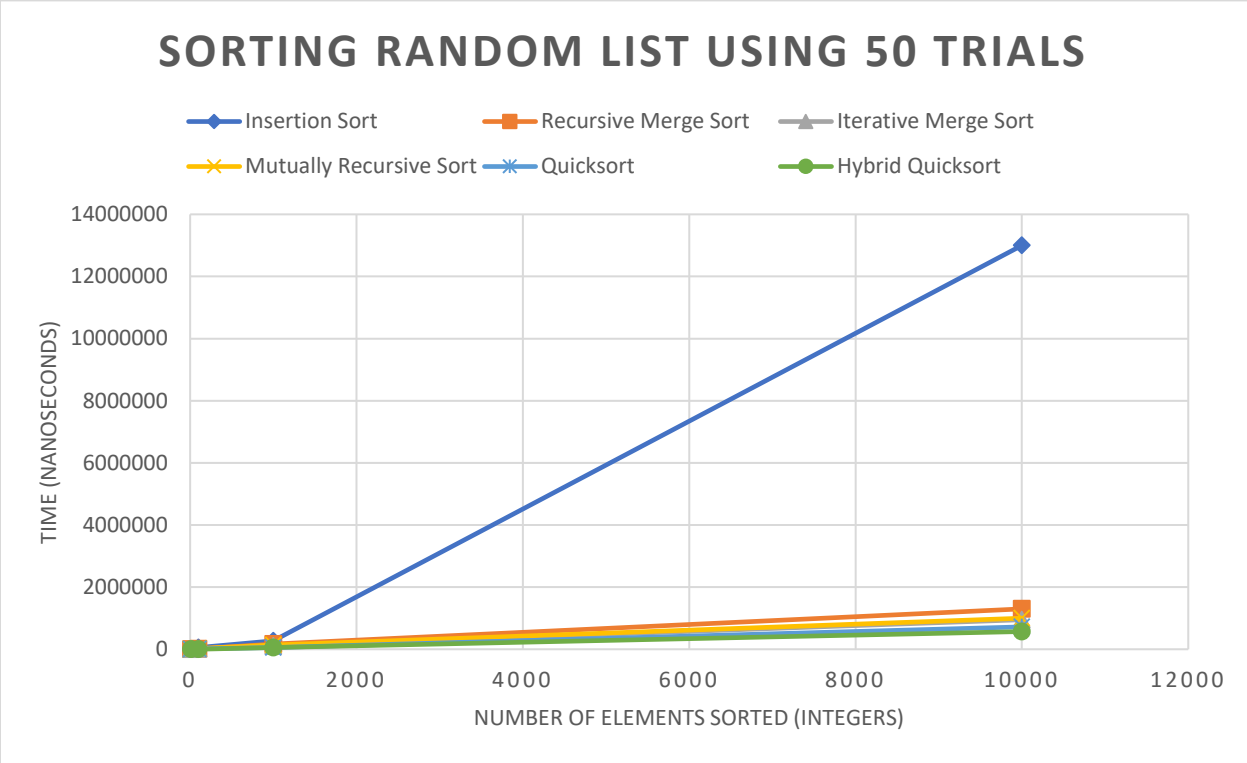
```
   * [isSorted performs a linear search through the array passed to it (between start and end), comparing each
value to
   *  ensure the array is sorted. It returns true if it is sorted and false otherwise.]
   * [Input: Receives the array to test as well as the start and end positions to perform the check from the driver
   *  method (or calling method)]
   * [Output: Outputs a boolean that states whether the array is sorted]
   *
   * @param [firstParam: the integer array to be checked]
   * @param [secondParam; the starting postion within array the to begin sorting (usually start of the array)]
   * @param [thirdParam; the ending position that defines where the method will stop sorting (usually the end of
the
   *        array]
   * @return [the boolean result stating true if sorted and false if not sorted]
   */

  public static boolean isSorted( int[] a ){

    return isSorted(a, 0, a.length); /* call to isSorted, providing correct parameters (entire array)
     * and returning its result as a boolean
     */
  }//isSorted driver

  /*
   * [The isSorted driver allows for easy and convienent calling of the isSorted method from the main method (or
   *  calling method).]
   * [Input: Receives the integer array to be checked]
   * [Output: Outputs a boolean stating whether the array is sorted]
   *
   * @param [firstParam: the integer array to be checked]
   * @return [the boolean result stating true if sorted and false if not sorted]
   */

}//CompareSorts
```

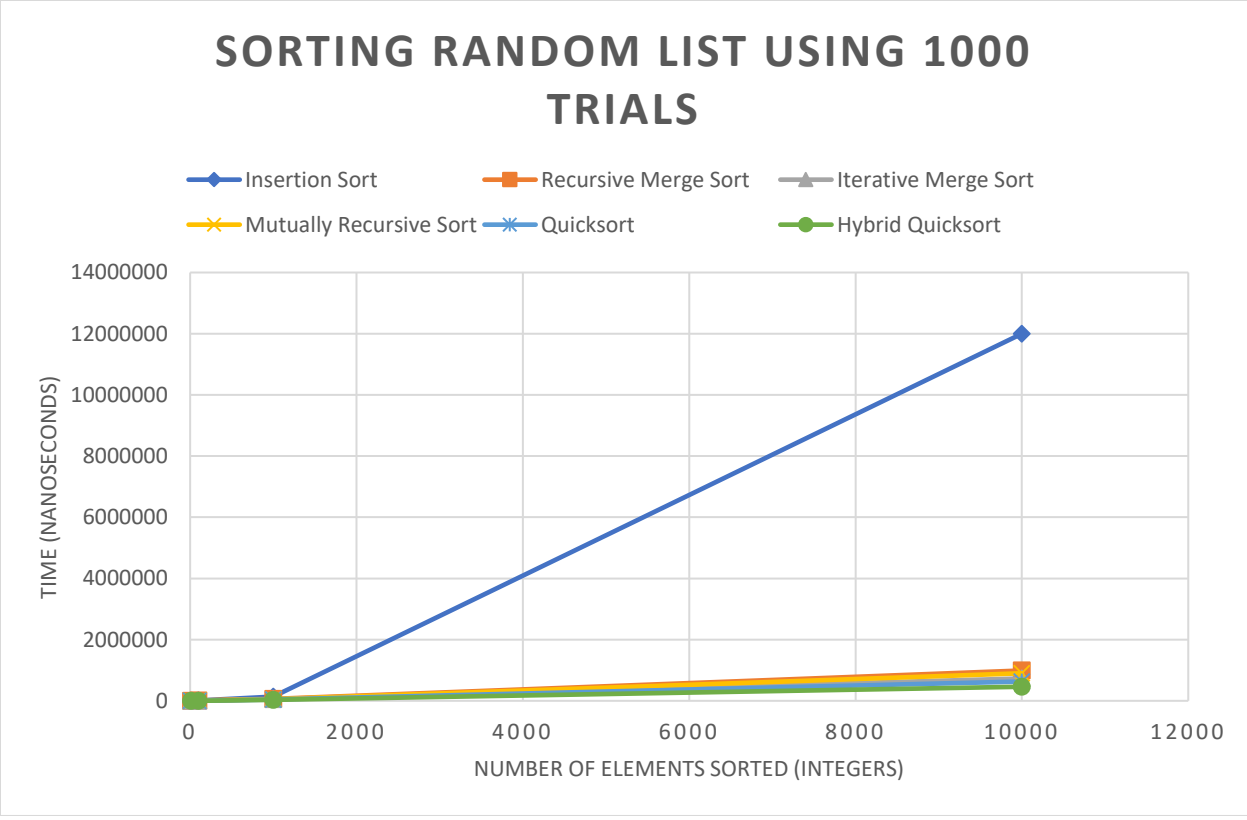
Appendix 2 – Part 1 Sample Output

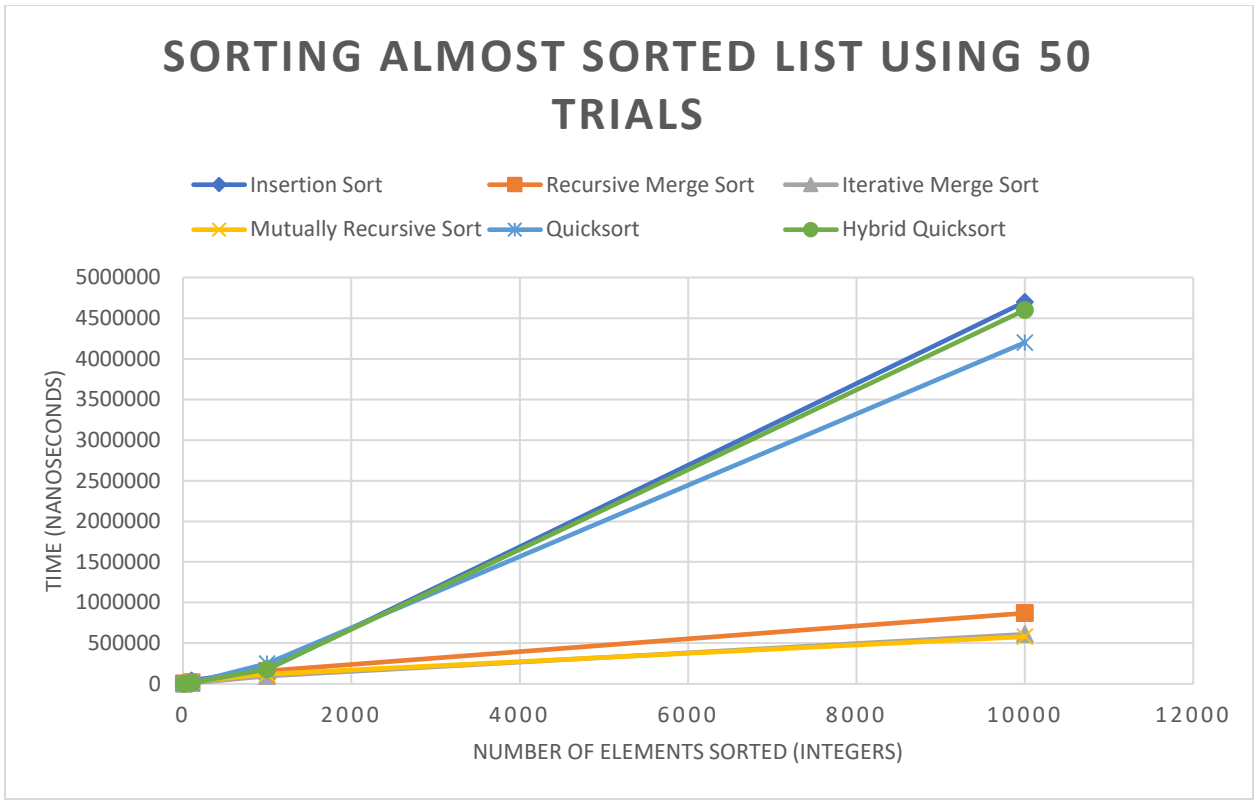
```
Is the list almost sorted? (type 'yes' if almost sorted, type anything else for 'no')
no
Comparing multiple sorts.

Average time for insertion sort: 59096 nanoseconds
Variance for insertion sort: 1.26913716436E9 nanoseconds
Standard Deviation for insertion sort: 35624.95142958092 nanoseconds

Average time for recursive merge sort: 17251 nanoseconds
Variance for recursive merge sort: 4.121491094E8 nanoseconds
Standard Deviation for recursive merge sort: 20301.455844347714 nanoseconds

Average time for iterative merge sort: 16648 nanoseconds
Variance for iterative merge sort: 3414216.56 nanoseconds
Standard Deviation for iterative merge sort: 1847.759876174391 nanoseconds

Average time for mutually recursive sort: 27093 nanoseconds
Variance for mutually recursive sort: 7.3409564978E8 nanoseconds
Standard Deviation for mutually recursive sort: 27094.199559684355 nanoseconds

Average time for quick sort: 12426 nanoseconds
Variance for quick sort: 6.3498091E7 nanoseconds
Standard Deviation for quick sort: 7968.568943041153 nanoseconds

Average time for hybrid quick sort: 10739 nanoseconds
Variance for hybrid quick sort: 1.21604694E7 nanoseconds
Standard Deviation for hybrid quick sort: 3487.1864590239506 nanoseconds


**********************************************************************

The quickest sorting method is hybrid quicksort.
It sorted all 100 numbers in an average time of 10739 over 50 trials.

**********************************************************************

Program Ends
```

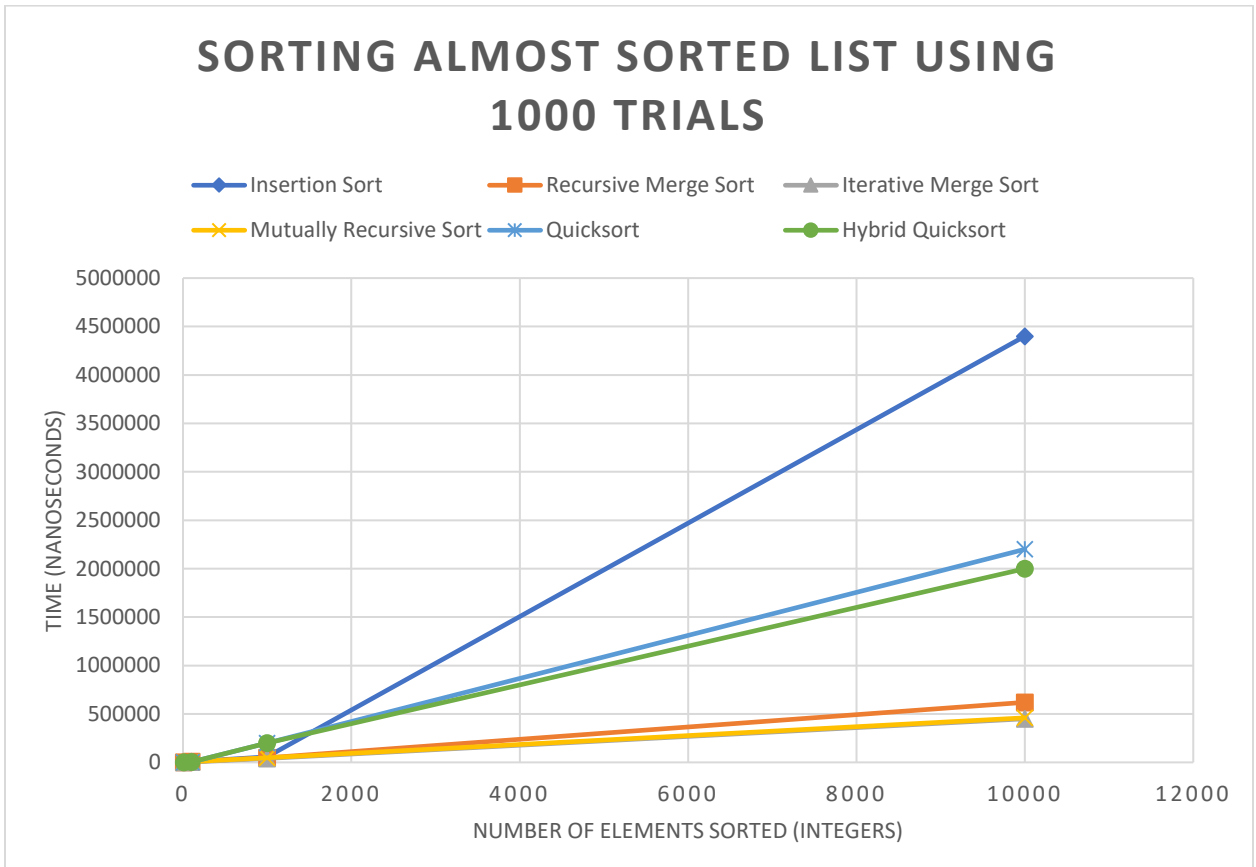
Appendix 3 – Part 1 Figures/Graphs

Source: Richard Constantine






SORTING ALMOST SORTED LIST USING 50 TRIALS
Insertion Sort
Recursive Merge Sort
Iterative Merge Sort
Mutually Recursive Sort
Quicksort
Hybrid Quicksort
TIME (NANOSECONDS)
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
2000
4000
6000
8000
10000
12000
NUMBER OF ELEMENTS SORTED (INTEGERS)


SORTING ALMOST SORTED LIST USING 1000 TRIALS
Insertion Sort
Recursive Merge Sort
Iterative Merge Sort
Mutually Recursive Sort
Quicksort
Hybrid Quicksort
TIME (NANOSECONDS)
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
2000
4000
6000
8000
10000
12000
NUMBER OF ELEMENTS SORTED (INTEGERS)

Appendix 4 – C code Sedgesort Comparison

Sources: http://www.comp.dit.ie/rlawlor/Alg_DS/sorting/quickSort.c
http://www.yendor.com/programming/sort/.

Lab Partner – Marc Payawal

```
/*
 *  A library of sorting functions
 *
 *  Written by:  Ariel Faigon,  1987
 *
 *  (C) Copyright by Ariel Faigon, 1996
 *  Released under the GNU GPL (General Public License) version 2
 *  or any later version (http://www.gnu.org/licenses/licenses.html)
 */
#include <stdio.h>
#include "sort.h"
#include <time.h>
#include        <stdlib.h>

/*-----------------------------------------------------------------
 *                      This file shouldn't be touched.
 *             For customizable parameters, see 'sort.h'
 *-----------------------------------------------------------------*/


/* Interesting tidbit:
 *
 * 15 has been found empirically as the optimal cutoff value in 1996
 *
 * 10 years later, in 2006, with computers being very different and
 * much faster, I revisited this constant and found the optimal value
 * to be a close tie between 15 and 16
 *
 * ~7 years later (end of 2013), on an Intel i7-4771 with hyper-threading,
 * larger caches, and a newer compiler (clang replacing gcc), technology
 * has finally made a difference and could make the switch to a higher
 * CUTOFF value.
 *
 * Here are the run times sorting a 100k element array:
 *  -DCUTOFF=10:   sedgesort:     5148 microsec.
 *  -DCUTOFF=15:   sedgesort:     5032 microsec.
 *  -DCUTOFF=16:   sedgesort:     4998 microsec.
 *  -DCUTOFF=20:   sedgesort:     4926 microsec.
 *  -DCUTOFF=25:   sedgesort:     4880 microsec.
 *  -DCUTOFF=30:   sedgesort:     4830 microsec.
 *  -DCUTOFF=50:   sedgesort:     4746 microsec. (minima)
 *  -DCUTOFF=60:   sedgesort:     4770 microsec.
 */
#ifndef CUTOFF
#  define CUTOFF 50
#endif

#define SIZE 1000000
```

```
void quickSort( int[], int, int);
int partition( int[], int, int);

/*
 |  void  partial_quickersort (array, lower, upper)
 |  KEY_T  array[];
 |  int    lower, upper;
 |
 |  Abstract:
 |          Sort array[lower..upper] into a partial order
 |          leaving segments which are CUTOFF elements long
 |          unsorted internally.
 |
 |  Efficiency:
 |          Could be made faster for _worst_ cases by selecting
 |          a pivot using median-of-3. I don't do it because
 |          in practical cases my pivot selection is arbitrary and
 |          thus pretty random, your mileage may vary.
 |
 |  Method:
 |          Partial Quicker-sort using a sentinel (ala Robert Sedgewick)
 |
 |  BIG NOTE:
 |          Precondition: array[upper+1] holds the maximum possible key.
 |          with a cutoff value of CUTOFF.
 */
void  insort (array, len)
register KEY_T  array[];
register int    len;
{
          register int          i, j;
          register KEY_T     temp;

          for (i = 1; i < len; i++) {
                    /* invariant:  array[0..i-1] is sorted */
                    j = i;
                    /* customization bug: SWAP is not used here */
                    temp = array[j];
                    while (j > 0 && GT(array[j-1], temp)) {
                              array[j] = array[j-1];
                              j--;
                    }
                    array[j] = temp;
          }
}


void  partial_quickersort (array, lower, upper)
register KEY_T  array[];
register int    lower, upper;
{
   register int       i, j;
```

```
   register KEY_T  temp, pivot;

   if (upper - lower > CUTOFF) {
           SWAP(array[lower], array[(upper+lower)/2]);
           i = lower;  j = upper + 1;  pivot = array[lower];
           while (1) {
              /*
               * ------------------------- NOTE --------------------------
               * ignoring BIG NOTE above may lead to an infinite loop here
               * ---------------------------------------------------------
               */
              do i++; while (LT(array[i], pivot));
              do j--; while (GT(array[j], pivot));
              if (j < i) break;
              SWAP(array[i], array[j]);
           }
           SWAP(array[lower], array[j]);
           partial_quickersort (array, lower, j - 1);
           partial_quickersort (array, i, upper);
   }
}


/*
 |  void  sedgesort (array, len)
 |  KEY_T  array[];
 |  int    len;
 |
 |  Abstract:
 |          Sort array[0..len-1] into increasing order.
 |
 |  Method:
 |          Use partial_quickersort() with a sentinel (ala Sedgewick)
 |          to reach a partial order, leave the unsorted segments of
 |          length == CUTOFF to a simpler low-overhead, insertion sort.
 |
 |          This method is the fastest in this collection according
 |          to my experiments.
 |
 |  BIG NOTE:
 |          precondition: array[len] must hold a sentinel (largest
 |          possible value) in order for this to work correctly.
 |          An easy way to do this is to declare an array that has
 |          len+1 elements [0..len], and assign MAXINT or some such
 |          to the last location before starting the sort (see sorttest.c)
 */
void  sedgesort (array,len)
register KEY_T  array[];
register int    len;
{
   /*
    * ------------------------- NOTE --------------------------
    * ignoring BIG NOTE above may lead to an infinite loop here
```

```
    * ------------------------------------------------------
    */
   partial_quickersort (array, 0, len - 1);
   insort (array, len);
}

float *getRandom() {
        static float list[SIZE];
        int i;

        srand((unsigned)time(NULL));

        for (i = 0; i<SIZE; i++) {
                list[i] = (float)rand();
        }
        list[SIZE] = 100000;

        return list;
}
int *intgetRandom() {
        static int list[SIZE];
        int i;

        srand((unsigned)time(NULL));

        for (i = 0; i<SIZE; i++) {
                list[i] = rand() % 10000;
        }

        return list;
}


// quick sort taken from http://www.comp.dit.ie/rlawlor/Alg_DS/sorting/quickSort.c
void quickSort( int a[], int l, int r)
{
  int j;

  if( l < r )
  {
        // divide and conquer
     j = partition( a, l, r);
     quickSort( a, l, j-1);
     quickSort( a, j+1, r);
  }

}


// partition taken from http://www.comp.dit.ie/rlawlor/Alg_DS/sorting/quickSort.c
int partition( int a[], int l, int r) {
  int pivot, i, j, t;
  pivot = a[l];
```

```
  i = l; j = r+1;

  while( 1)
  {
        do ++i; while( a[i] <= pivot && i <= r );
        do --j; while( a[j] > pivot );
        if( i >= j ) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
  }
  t = a[l]; a[l] = a[j]; a[j] = t;
  return j;
}


void printFloat(float list[]) {
        for (int i = 0; i < SIZE; i++)
                printf("%f, ", list[i]);
}


int main(int argc, char *argv[]) {
        long time;
        float *listA;
        int *listB;

        listA = getRandom();
        //printFloat(listA);
        time = clock();
        sedgesort(listA, SIZE);
        time = clock() - time;
        //printFloat(listA);
        printf("Done sedgeSort, time: %ld\n", time);

        listB = intgetRandom();
        time = clock();
        quickSort(listB, 0, SIZE);
        time = clock() - time;
        printf("Done quickSort, time: %ld\n", time);
        return 0;
}
```

Sample output

```
Done sedgeSort, time: 78
Done quickSort, time: 235

RUN SUCCESSFUL (total time: 2s)
```

Appendix 5 – Part 2 Code

Source: Richard Constantine

```
import java.util.Scanner;

public class MatrixGenerator {


  public static void main(String[] args) {

    int size, sparseness; // user selected parameters of size and sparseness
    int[][] adjacencyMatrix; // define the 2d matrix
    boolean twoParts; // user selected parameter that determines whether there are 2 distinct, connected parts

    try{

      Scanner keyboard = new Scanner(System.in); // read input from the keyboard

      System.out.println("Please enter the size of the adjacency matrix (n x n) as an integer.");
      size = keyboard.nextInt(); // save size parameter

      System.out.println("Please enter the sparseness of the adjacency matrix as an integer between " +
                  "0 and 100 (0 being full - 100 being totally sparse).");
      sparseness = keyboard.nextInt(); // determines sparseness of the matrix (0 means full - 100 means totally
sparse)

      System.out.println("Would you like 2 distinct & connected components? (type 'yes' or anything else for 'no')");
      String response = keyboard.next(); // determine whether 2 connected components

      if(response.equals("yes")) // determine user response
        twoParts = true;
      else
        twoParts = false;

      adjacencyMatrix = new int[size][size]; // declare matrix of dimensions size x size

      generateMatrix(size, twoParts, sparseness, adjacencyMatrix);

      System.out.println("\nProgram Ends");

    }//try

    catch (Exception InputMismatchException){
      System.out.println("Woops, wrong input.");
    }//catch

  }//main

  private static void generateMatrix(int size, boolean twoParts, int sparse, int[][] adjMatrix){

    int randomNum;

// fill top-right of matrix w/ weights between 0-9
```

```
    for(int i = 0; i < size; i++){
      for(int j = i+1; j < size; j++){
        randomNum = 1 + (int)(Math.random() * 100); // random number between 1 and 100
        if(sparse < randomNum)
          adjMatrix[i%j][j] = 1 + (int)(Math.random() * 9); // fill top right with weights between 0-9
      }//for
    }//for

// ************************ Two Connected Components *********************************
    if(twoParts){
      for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
          if(i <= size/2 && j > size/2)
            adjMatrix[i][j] = 0; // set appropriate elements to 0
        }//for
      }//for
    }//if

// copy top-right into bottom-left
    for(int i = 0; i < size; i++){
      for(int j = i+1; j < size; j++){
        adjMatrix[j][i] = adjMatrix[i][j] ;
      }//for
    }//for

   printMatrix(size, adjMatrix);

  }//generateMatrix

  private static void printMatrix(int size, int[][] adjMatrix){

   System.out.println("\nAdjacency Matrix");

   // print matrix
   for(int i = 0; i < size; i++){
    for(int j = 0; j < size; j++){
      System.out.print(adjMatrix[i][j] + " ");
    }//for
    System.out.println();
   }//for

  }//print

}//MatrixGenerator
```

Appendix 6 – Part 2 Sample Output

```
Please enter the size of the adjacency matrix (n x n) as an integer.
2
Please enter the sparseness of the adjacency matrix as an integer between 0 and 100 (0 being full - 100 being
totally sparse).
100
Would you like 2 distinct & connected components? (type 'yes' or anything else for 'no')
no

Adjacency Matrix
0 0
0 0

Program Ends
```

****

```
Please enter the size of the adjacency matrix (n x n) as an integer.
10
Please enter the sparseness of the adjacency matrix as an integer between 0 and 100 (0 being full - 100 being
totally sparse).
50
Would you like 2 distinct & connected components? (type 'yes' or anything else for 'no')
no

Adjacency Matrix
0 0 0 0 0 0 6 0 0 1
0 0 5 2 0 0 3 8 0 0
0 5 0 7 0 0 0 9 0 0
0 2 7 0 6 4 2 0 0 3
0 0 0 6 0 0 1 8 0 3
0 0 0 4 0 0 0 6 6 0
6 3 0 2 1 0 0 0 0 9
0 8 9 0 8 6 0 0 0 0
0 0 0 0 0 6 0 0 0 0
1 0 0 3 3 0 9 0 0 0

Program Ends
```

****

Please enter the size of the adjacency matrix (n x n) as an integer.
7
Please enter the sparseness of the adjacency matrix as an integer between 0 and 100 (0 being full - 100 being totally sparse).
0
Would you like 2 distinct & connected components? (type 'yes' or anything else for 'no')
yes

Adjacency Matrix
0 9 3 7 0 0 0
9 0 6 9 0 0 0
3 6 0 7 0 0 0
7 9 7 0 0 0 0
0 0 0 0 0 5 8
0 0 0 0 5 0 7
0 0 0 0 8 7 0

Program Ends