1 Exercise

There is no programming in this lab, all the code is given to you. On the MyCLasses page for this assignment there is a file `SortTests.java` that contains all of the needed code. Create a new java project, download the `SortTests.java` file and copy the file into the project. At this point you are ready to go.

This exercise is a timing analysis of the three sorting algorithms we discussed in class, the bubble sort, insertion sort and selection sort. In addition, I included another sorting algorithm, the merge sort, which belongs to a different class of sorting algorithms. The merge sort is one of several divide-and-conquer algorithms. You do not need to know how it works but if you go on to Computer Science II you will learn how it works and analyze it along with other algorithms of the same type. You may find it interesting to look at the code, the MergeSort method calls itself. Methods that call themselves are called recursive methods, although it may seem strange to do something like that, the ability to do this is a very powerful technique.

When you analyze and compare algorithms what you usually do is count the number of operations that the algorithm does. For example, how many arithmetic operations are done (additions, subtractions, multiplications, or divisions), how many array entries are moved or changed, or how many comparisons are done (conditional statements that need to be evaluated). In other words, you look at the portions of the code that take most time and ask how often that portion of code is run to complete the algorithm. We will not do a strict mathematical analysis of these algorithms here, but if you go further into computer science you will learn the techniques for this analysis.

What we are doing in this lab is a timing analysis and comparison of the sorting algorithms. So we will create a random array and time how long it takes for each sorting algorithm to sort the array. Timing analysis is not as precise as operation counting but it still gives a good indication of the work done to complete an algorithm. There are a few things you need to be careful with in a timing analysis. First, a timing analysis is dependent on the hardware you are using. If you time a process on a slow computer and then do the same timing on a fast computer the results will clearly be different. **So when you do this lab you must do all of the runs on the same computer.**

Another downfall with timing analysis is that with modern operating systems where are many processes running in the background so your program does not have 100% access to your CPU. For the most part this is out of your control, but one thing you can do is keep a minimum amount of programs open when you do your timings. **So when doing your timings you will want to shut down all of the programs on your computer except for Eclipse and a spreadsheet.** In addition, you should do each timing run multiple times and average the results to help reduce the effect of background processes. I will not require you to do that here but you should run the same tests multiple times to see if the results are consistent before using the timing values.

When analyzing an algorithm you want to answer three questions,

1. How well does it run in the best case scenario?
2. How well does it run in the worst case scenario?
3. How well does it run on average?

For sorting an array,

1. The best case scenario is usually (but not always) how long does it take the algorithm to sort an array that is already sorted? If the array is already sorted then there should be no movement of array entries and in general the algorithm should do the least amount of work.

2. The worst case scenario is usually (but not always) how long does it take the algorithm to sort an array that is in reverse order? If the array is in reverse order then there should be a lot of movement of array entries and in general the algorithm should do the most amount of work.

3. The average case scenario is usually (but not always) how long does it take the algorithm to sort an array that is random? In this case, it is more informative to run several random arrays through the algorithms and take an average of the times.

Before we get started on timings and analyze the algorithms lets look at a couple segments of the program. At the top we have the three sorting algorithms discussed in class and some support functions, like coping the array and printing the array (which we do not use). We also included the merge sort code, again just remember that is is another sorting algorithm. In the main, there is a timer that is around each call to the sorting algorithm, `System.nanoTime()`. This command will return the number of nanoseconds the program has been running. So to time a process we call `System.nanoTime()` and store the value in some variable like `startTime` then we do the lengthy process and make another call to the timer and store its result in another variable, like `endTime`. Then the difference between the two times is how long the process took to run. Note that the return value of `System.nanoTime()` is a long. So the code,

```
startTime = System.nanoTime();
BubbleSort(A);
endTime = System.nanoTime();
System.out.printf("   Bubble Sort Time: %13.9f sec.\n", (endTime - startTime) / 1000000000.0);
```

will calculate, and print to the screen, the amount of time the bubble sort took to run. Recall that a nanosecond is one one-billionth of a second, hence we divide by 1,000,000,000 to convert the runtime into seconds.

Another block of code we need to discuss is the following,

```
for (int i = 0; i < A.length; i++) {
    // A[i] = i; // Best Case
    // A[i] = A.length - i; // Worst Case
    A[i] = (int) (Math.random() * 1000000) + 1; // Average Case (Random)
}
```

This is populating the array and it is set up to create our best, worst, and average cases. The first commented line creates an array that is already sorted, the second commented line creates an array that is in reverse order, and the last line in the for block creates a random array.

So when we are examining the best case scenario we will uncomment the first line and comment the other two, the same goes for examining the worst and average cases. Remember to run all timings on the same computer and to shut down all of your programs except for Eclipse and a spreadsheet.

1.1 Analysis of Bubble, Insertion, and Selection Sorts

Do the following for each scenario, best, worst, and average cases.

Show all of your work and answers to the following questions in either a Microsoft Word doc file or LibreOffice Writer odt file that contains all of the charts, graphs, and your explanations. As usual, you will submit this report file through MyClasses. Make sure the report contains all of the charts and graphs you produce.

1. Run the program on array sizes, 2,500, 5,000, 10,000, 20,000, 40,000, 80,000, and 160,000 elements. Note that the array sizes go up by a factor of 2 each time, we did this for a reason.

2. Make a chart from the data that has the array size as the first column, the bubble sort times as the second column, the insertion sort times as the third column, and the selection sort times as the fourth column.

3. Create a line graph of sort times verses array size, have the array size on the horizontal axis and the sort times on the vertical axis. So the graph should have three lines all on the same chart, one for the bubble sort, one for the insertion sort, and one for the selection sort. In most spreadsheet applications this is done through an XY-Scatter plot with the points connected with lines. For example,

4. Give a detailed answer to each of the following questions that are to be based on your chart and graph from the above exercise.

   (a) From the time analysis, which algorithm is consistently better than the others? Which algorithm is consistently worse than the others? Do any of the lines cross? This would indicate that one algorithm is better for smaller arrays but another is better for larger arrays.

   From the timing data, does there seem to be any consistencies between the algorithms? As an example, is one consistently 10 times slower than another one, are two fairly close to each other on every run, ...?

   (b) From the time analysis, does the line graph appear to be straight or curved? If the line is straight then this would imply a linear relationship and hence as the array size grows the time to sort it grows at the same rate. Specifically, if the array size doubles the amount of time to sort it will also double. If the line graph is not straight then there is a different relationship. Does the line graph suggest a linear relationship?

   (c) Let's expand on the last question. From the time analysis chart you created, look at the times of the array sizes. Each array size is double the previous size. Calculate the multiple increase in times between each array size. For example, if the bubble sort took 2 seconds with 20,000 and 10 seconds with 40,000 the multiple increase would be $\frac{10}{2} = 5$. That is, it took 5 times as long to sort the 40,000 element array than it did to sort the 20,000 element array.

      i. An algorithm is said to run in *linear time* if we double the size of the array and it takes twice as long to sort the array.

      ii. An algorithm is said to run in *quadratic time* if when we double the size of the array it takes about $2^2 = 4$ times as long to sort the array.

      iii. An algorithm is said to run in *cubic time* if we double the size of the array it takes about $2^3 = 8$ times as long to sort the array.

      iv. An algorithm is said to run in *quartic time* if we double the size of the array it takes about $2^4 = 16$ times as long to sort the array.

      v. An algorithm is said to run in *exponential time* if the timing factors from doubling the size of the arrays increase by yet another factor. For example, if going from 10,000 to 20.000 was a factor of 2, then 20,000 to 40,000 was a factor of 4, then 40,000 to 80,000 was a factor of 16, and so on.

   Calculate the ratios between every two consecutive times for each sort from your chart and answer the following questions for each sort. Note that when array sizes are small the sort times are effected by background processes more than when the array sizes are larger. So you will want to concentrate on the larger array sizes.

- What are these numbers close to for each sort?
- Are these values different between the three sorts or are they about the same?
- Does your data suggest a linear, quadratic, cubic, quartic, or exponential relationship? If so, which one?

1.2 Analysis of the Merge Sort

Do the following for just the average case scenario.

You may have noticed that the times for the merge sort in the above runs were significantly lower than any of the other three algorithms. The merge sort is far better than any of the methods we discussed in class for large arrays.

1. Comment out the sections of code that run the bubble, insertion, and selection sorts. So the only sort that is run is the merge sort.
2. Run the program on array sizes, 2,500, 5,000, 10,000, 20,000, 40,000, 80,000, 160,000, ... Keep doubling the array size until the time to sort the array exceeds 5 seconds or you run out of memory.
3. Make a chart from the data that has the array size as the first column, and the sort times as the second column.
4. Create a line graph of sort times verses array size.
5. From the time analysis, does the line graph appear to suggest a linear, quadratic, cubic, quartic, or exponential relationship? Why?

1.3 SortTests.java

```
import java.util.Scanner;

/*-
 * Program that allows the user to test the timings of the three sorts we discussed
 * in class: bubble, insertion, and selection.  It also includes a more advanced sort,
 * called the merge sort.
 * Written by: Don Spickler
 * Date: 4/26/2018
 */

public class SortTests {

    public static int[] CopyintArray(int A[]) {
        int[] B = new int[A.length];

        for (int i = 0; i < A.length; i++)
            B[i] = A[i];

        return B;
    }

    public static void PrintIntArray(int A[]) {
        for (int i = 0; i < A.length; i++) {
            System.out.print(A[i] + "  ");
        }
        System.out.println();
    }

    public static void BubbleSort(int A[]) {
        for (int i = A.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (A[j] > A[j + 1]) {
                    int temp = A[j];
                    A[j] = A[j + 1];
                    A[j + 1] = temp;
                }
            }
        }
    }

    public static void insertionSort(int[] A) {
        for (int itemsSorted = 1; itemsSorted < A.length; itemsSorted++) {
            int temp = A[itemsSorted];
            int loc = itemsSorted - 1;
            while (loc >= 0 && A[loc] > temp) {
                A[loc + 1] = A[loc];
                loc = loc - 1;
            }
            A[loc + 1] = temp;
        }
    }

    public static void selectionSort(int[] A) {
        for (int lastPlace = A.length - 1; lastPlace > 0; lastPlace--) {
            int maxLoc = 0;
            for (int j = 1; j <= lastPlace; j++)
                if (A[j] > A[maxLoc]) {
                    maxLoc = j;
                }

            int temp = A[maxLoc];
            A[maxLoc] = A[lastPlace];
            A[lastPlace] = temp;
        }
    }

    private static void MergeSort(int[] A, int[] temp, int low, int high) {
        if (low < high) {
            int middle = (high + low) / 2;
            MergeSort(A, temp, low, middle);
            MergeSort(A, temp, middle + 1, high);
            Merge(A, temp, low, middle, high);
        }
    }
    private static void Merge(int[] A, int[] temp, int low, int middle, int high) {
        for (int i = low; i <= high; i++) {
            temp[i] = A[i];
        }

        int i = low;
        int j = middle + 1;
        int k = low;
        while (i <= middle && j <= high) {
            if (temp[i] <= temp[j]) {
                A[k] = temp[i];
                i++;
            } else {
                A[k] = temp[j];
                j++;
            }
            k++;
        }

        while (i <= middle) {
            A[k] = temp[i];
            k++;
            i++;
        }
    }

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Input the array size: ");
        int arraySize = keyboard.nextInt();
        System.out.println();

        int A[] = new int[arraySize];
        int B[];
        long startTime = 0;
        long endTime = 0;

        for (int i = 0; i < A.length; i++) {
            // A[i] = i; // Best Case
            // A[i] = A.length - i; // Worst Case
            A[i] = (int) (Math.random() * 1000000) + 1; // Average Case (Random)
        }

        B = CopyintArray(A);

        startTime = System.nanoTime();
        BubbleSort(A);
        endTime = System.nanoTime();
        System.out.printf("   Bubble Sort Time: %13.9f sec.\n", (endTime - startTime) / 1000000000.0);

        A = CopyintArray(B);

        startTime = System.nanoTime();
        insertionSort(A);
        endTime = System.nanoTime();
        System.out.printf("Insertion Sort Time: %13.9f sec.\n", (endTime - startTime) / 1000000000.0);

        A = CopyintArray(B);

        startTime = System.nanoTime();
        selectionSort(A);
        endTime = System.nanoTime();
        System.out.printf("Selection Sort Time: %13.9f sec.\n", (endTime - startTime) / 1000000000.0);

        A = CopyintArray(B);

        startTime = System.nanoTime();
        int[] temp = new int[A.length];
        MergeSort(A, temp, 0, A.length - 1);
        endTime = System.nanoTime();
        System.out.printf("    Merge Sort Time: %13.9f sec.\n", (endTime - startTime) / 1000000000.0);
    }
}
```