CSC172 LAB 14

ALL SORTS OF SORTS

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming.

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Not all sorts are created equal

Sorting algorithms are of enormous interest and importance. Nearly every computerized application employs sorting somewhere behind the scenes to help users manage the many complex tasks that are enabled by modern computations. Since everyone has an intuitive understanding of sorting, it makes an ideal area for beginning the study of algorithm design.

> *“Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being*

> *invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide-and-conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds."*
>
> - http://en.wikipedia.org/wiki/Sorting_algorithms

In this lab, you will implement three basic sorting algorithms and compare their performance.

1. Begin this lab by compiling and testing the code below, which sorts an array using `bubblesort`.

```
import java.util.Arrays;
public class SortTest {
    private static int count;

    public static void main(String args[]) {
        long startTime, endTime, elapsedTime;
        int size = Integer.parseInt(args[0]);

        Integer [] a = new Integer[size];
        Integer [] b = new Integer[size];

        for (int i = 0; i < size; i++) {
            a[i] = b[i] = (int)(Math.random() * 100);
        }



        System.out.println(Arrays.toString(a));
        count = 0;
        startTime = System.currentTimeMillis();
        bubblesort(a);
        endTime = System.currentTimeMillis();
        elapsedTime = endTime - startTime;
        System.out.println(Arrays.toString(a));
        System.out.println("bubblesort took "+ count + " moves to sort "
            + size + " items");
              System.out.println("\t in : "+ elapsedTime + " millesec");
```

```
        // Reset count and array
        count = 0;
        for (int i = 0; i < size; i++)
            a[i] = b[i];
    }

    public static <AnyType extends Comparable<? super AnyType>>
    void bubblesort(AnyType[] a) {
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length - 1; j++) {
                if (a[j].compareTo(a[j + 1]) > 0) {
                    AnyType tmp = a[j];
                    count++;
                    a[j] = a[j + 1];
                    count++;
                    a[j + 1] = tmp;
                    count++;
                }
            }
        }
    }
}
```

2. Run bubble sort on arrays of increasing size and plot the results. Generate a plot with your favorite plotting software. At each size, run the algorithm three times on three different random arrays of a given size and take the average of the results. How closely do your measured results match the expected theoretical results for bubble sort? (Write a sentence or two to answer this question and include it as a caption on your plot). If you encounter an `OutOfMemoryError` when running your program, you can increase the size of the memory of your java run with the `-Xmx` option (look up this option by typing “java -Xmx option” into Google).
3. Add a method to the code above for the `insertionSort` algorithm. You can use the code from your text book, but you will need to add some instructions in order to count the number of exchanges. Test your algorithm before proceeding.
4. As you did with bubble sort run insertion sort on arrays of different sizes and plot the results. Make a statement about the measured and theoretically expected results.
5. Add a method to the code above for the `shellSort` algorithm. You can use the code from your text book, but alter the code to use Hibbard's increments. You will also need to add some instructions in order to count the number of exchanges. Test your algorithm before proceeding. As you did with bubble sort and insertion sort, run shell sort on arrays of different sizes and plot the results. Make a statement about the measured and theoretically expected results.

6. Use the library `sort()` method found in `java.util.Arrays` to sort the array. You will not be able to count operations, but you can still time the execution of the method. As you did with previous sorts, run `sort()` on arrays of different sizes and plot the results. Make a conjecture about the nature of this algorithm based on your results.
7. For extra credit, implement the `quickSort` algorithm, test it to confirm it works correctly, and include a plot of its performance in your writeup. How does quick sort compare to the other sorting methods?

3 Hand In

Hand in the source code from this lab at the appropriate location on the BlackBoard system at my.rochester.edu. You should hand in a single zip file (compressed archive) containing your source code, README, plots, and OUTPUT files, as described below.

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.), and one sentence explaining the contents of all the other files you hand in.
2. A PDF file containing the plots of your sorting algorithms' performance, along with captions describing their expected (Big-oh) and actual runtimes and any other observations you have made.
3. Java source code file(s) representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.
4. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4 Grading

Each section (1-6) accounts for 15% of the lab grade (total 90%)

README file counts for the remaining 10%

15% extra credit possible for implementing, plotting, and analyzing quick sort.