**CSC205AB**

**MinilabSorting**

This program will exercise your knowledge of sorting – specifically bubbleSort, mergeSort, and quickSort. You will “benchmark,” or measure the efficiency of the 3 sorting algorithms. If you do this program in class, you can work together on it; otherwise, it must be done individually.

**Part1 – Big O notation:**

# Put your name (names if you are working together) in the top row of the table at the bottom of the document. Then fill in the column labeled “Efficiency.” Just fill in the white cells so that you show the “Big O notation” efficiency for each of the 3 sorting algorithms (be sure to put in the O, as in O(n2). You can do this by typing in the answer (if you are going to submit this paper via Canvas) or by hand.

# **Part2 – counting exchanges and comparisons:**

***You don’t have to do this part since the link to the animation that was used is broken – skip filling out the middle 2 columns in the table below…***

**Part3 – an efficiency experiment on your machine:**

First, close all running programs on your machine except for Textpad (as we will be timing the algorithms). Use the Sort.java program you are given - it is the one we have been working with and it implements all three sorting algorithms. It also has a main method which creates an array of 25 random integers and calls one of the sorting methods to sort it. Look over the code and then:

1. Make sure that the EXPLAIN constant is set to false. This will turn off the explanations as the code runs (it will be too much, as we will be sorting large arrays). You do not have to do this, but…to make your benchmarks the most accurate, you could actually comment out the code in the

if (EXPLAIN)

blocks (instead of just turning off the flag); that way, the code will not keep checking to see if explain is turned on.

1. Change the code in main by doing the following:

* Be sure the parts that actually print the array are commented out, since we will generate HUGE arrays and will not want them to print.
* Add code to save the system time immediately before and after the call to the sorting method. After the sort is completed, subtract them in order to tell how long it took to do the sort. Have the program print the result. You can get the system time in nanoseconds by doing the following:

long startTime = System.nanoTime();

<the call to the sort method>

long endTime = System.nanoTime();

System.out.println(“it took: “ + (endTime-startTime)/1000000000. + “ seconds”);

* The size of the randomly-generated array is currently set to be 200,000. Currently, it is filled with random numbers between 1 and 10,000. We would like the same set of random numbers to be used for each sorting algorithm, so we need to put a “seed” in the random number generator. So change the = new Random(); part to have a seed which is any large number > 0 you want, like this = new Random(<your number>);
* Now that you have a huge array and a way of measuring how long it takes to sort it, have main call bubbleSort to sort the array. This will take about 65 seconds, so don’t give up on it whe it run. Write how long it took in the last column of the table below.
* Change main so that it calls mergeSort instead. Compile the program, run it, and type or write in how long it took to run in the last column of the table below.
* Finally, change main so that it calls quickSort instead. Compile and run the program. Type or write in how long it took to run in the last column of the table below.

**Please submit**

The following table (with the white cells filled in), either on paper or via Canvas.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME(S): |  | Don’t do this part since  the applet link is broken | Don’t do this part since  the applet link is broken |  |
| Algorithm | Efficiency (in Big O notation) (Part 1: theoretical) | Number of compares (Part2: from applet run) | Number of exchanges (Part2: from applet run) | Time (milliseconds) (Part3: from sort.java) |
| **Bubblesort** |  |  |  |  |
| n=500 |  |  |  |  |
| n=2000 |  |  |  |  |
| n=200000 |  |  |  |  |
| **Mergesort** |  |  |  |  |
| n=500 |  |  |  |  |
| n=2000 |  |  |  |  |
| n=200000 |  |  |  |  |
| **Quicksort** |  |  |  |  |
| n=500 |  |  |  |  |
| n=2000 |  |  |  |  |
| n=200000 |  |  |  |  |