CSE12 - Spring 2014 HW #5

Sorting and Running Time

(100 points)

Due 11:59pm 6 May 2014

In this assignment you will analyze the running time of sorting code and algorithms

This assignment is an individual assignment. You may ask Professors/TAs/Tutors for some guidance and help, but you can't copy code. You can, of course, discuss the assignment with your classmates, but don't look at or copy each others code or written answers.

The following files are provided for you and can be found on the HW page:

- SortTimer.java
- Sort12.java
- Queue12.java
- Insertion12.java
- Bubble12.java
- Quick12.java
- Merge12.java
- random-strings.txt
- random-strings-sorted.txt

You will submit the following file for this assignment:

- **Merge12.java**
- **HW5.txt**

HW5 will be a plain text file (NOT .docx or .pdf or anything else). You should use the ^ symbol to indicate exponentiation (e.g. n^2 should be written n^2) and log(n) where appropriate. At the top of your HW5.txt file, please include the following header:

```
CSE 12 Homework 5
Your Name
Your PID
Section [Your section] (A00 for Alvarado, B00 for Papadopoulos)
The date
```

Part 1 - Implementing Merge Sort (30 points)

You are to implement the merge sort algorithm discussed in lecture and described in detail in your textbook. Your implementation must be in the **Merge12.java** file. Make certain to turn in your *modified* Merge12.java file.

You may want to look at the other fully-implemented sort algorithms in Bubble12.java, Insertion12.java, and Quick12.java as guides or models. You should feel free to look at the code in the book but you should actually type in your modified code yourself. You will need to modify it for use in this assignment so that it implements the Sort12 interface. Specifically, the code described in the book is specific to integer arrays, the Sort12 interface requires the defined sort() method to handle any List of elements that implement the Comparable interface.

We have provided some very minimal skeleton code with the suggested methods mergeSortInternal and merge for you to write. You do not have to write and use these methods, though we strongly recommend it.

Notice that internalMergeSort and merge take ArrayLists as their lists to sort, while sort takes any List object, and it *modifies* the list object to be sorted at the end of the method. For efficiency, we suggest that you create a copy of the original list as an ArrayList to pass into your internalMergeSort method and then copy the sorted elements back into the original list after internalMergeSort completes. This ensures that you can get elements out of the list efficiently during the sorting process, no matter what kind of list the user originally passed in. **Be sure you copy the elements back into the *original object referenced by list*, instead of changing the reference `list` to point to a new object. (Please see the "testing merge sort below")**

When you look at the method headers, e.g.:

```
public  <T extends Comparable<? super T>> void  sort(List<T> list)
```

You will see the somewhat mysterious looking `<T extends Comparable<? super T>>`. Here T is what is called a *bounded type parameter*, while ? is called a *wildcard.* We will discuss both of these concepts more at the beginning of Week 6. You do not need to understand the details in order to implement Merge Sort. But in short what this header is saying is that the type T (which will be defined when the method is called based on what type the List contains) must be a descendant (is-a) of the Comparable type, and that the Comparable type it is descended from must know how to compare all T's and T's superclasses explicitly. In other words, T must know how to compare itself to other T's, or any other class from which T is derived, for the sort method to work.

Testing Merge Sort

The supplied timing code SortTimer.java is very similar to CollectionTimer.java used in HW3. It has been modified to allow you to sort a text file and print the results with the -p option. The -p option will print for each iteration, so you likely want to use for a single iteration, and a single timing repetition.

For example, sort the first 100 words of the file random-strings.txt using merge sort and print the resulting sorted list,, you would type in

```
% java SortTimer -p random-strings.txt 2 100 1 1 1
```

This will use sort algorithm “2”, sort the first 100 strings, extend each test by 1, do just 1 test, and 1 rep/iteration. For SortTimer, algorithm 2 is merge sort. You can use this to verify that your merge sort is sorting properly. By redirecting the output to a file and verifying that the result is properly sorted (We WILL test that your implemented merge sort actually sorts correctly).

Please make certain to properly indent, and adequately comment. Feel free to change variable names to reduce how much you type.

Part 2: Running Time of Various sort Algorithms (60 points)

In this section, you will test actual efficiency of each of the methods and attempt to empirically verify that their computational complexity is “reasonable” given what you have learned about the actual algorithms.

To help you, SortTimer has a usage statement if you invoke with no arguments

$ java SortTimer

```
Usage: java  SortTimer [-p] <document> <sortAlg> [start] [increment]
[steps] [reps]
      -p      - print the sorted list
      document  - text file to sort
      Algorithm - 0:bubble, 1:insertion: 2:merge, 3:quick
      start     - number of words read in from  document
      increment - number of words to add for each iteration
      steps         - number of iterations
      reps      - number of times to repeat for timing
```

There are four (4) sort algorithms: Bubble, a modified insertion sort, merge sort, and quicksort.

You should run with the defaults for each of the sort algorithms to get a feeling for how quickly they run using random-strings.txt as the document, e.g.,

```
 $ java SortTimer random-strings.txt 1
```

will perform a test with the modified insertion sort and would have output similar to

```
$ java SortTimer random-strings.txt 1
Document: random-strings.txt
 sortAlg: 1
=====================================
  1:    5000 words in       5 milliseconds
  2:   10000 words in      11 milliseconds
  3:   15000 words in      21 milliseconds
  4:   20000 words in      35 milliseconds
  5:   25000 words in      53 milliseconds
```

You will probably find that for the more efficient algorithms, that the default settings do not run long enough to give good results.

Note that you are working with a *modified* insertion sort in this homework, which you will explore in more detail in the last part of this assignment. But for this section, *do not be alarmed* that it will not necessarily have the n^2 behavior of standard insertion sort that we discussed in class (that's why it has been modified!)

Place the answers to the following in our HW5.txt file:

I. Answer following questions for each of the sorting algorithms (i.e. repeat the questions for each algorithm)

A. What testing parameters did you select so that you could gain insight as to how each algorithm performs as N increases. This is likely a different set of parameters for each sort algorithm. Explain briefly why you made your choices (one or two sentences)
B. Copy the output of your actual test run for the algorithm
C. Given the numbers you selected, What is the apparent complexity of each sorting algorithm. Which data points did you use to come to your conclusion?

(please section each of your answers as)

I. Bubble
    A. answer to part A
    B. answer to part B
    C. answer to part C
II. Insertion
    A. answer to part A
    B. answer to part B
    C. answer to part C
III. Merge
    A. answer to part A
    B. answer to part B

C. answer to part C

IV. Quick
   A. answer to part A
   B. answer to part B
   C. answer to part C

II. Repeat the above questions, but use the random-strings-sorted.txt as the document.
Use the same format as in part I for your answers

Please note: you may have to give java an argument to increase the stack size for quicksort. To do this: `java -Xss128m SortTimer …`
and would set the stack size to 128 MBytes

III. What do you notice about the behaviour of the various algorithms in the pre-sorted case?

Part 3: Examining Modified Insertion Sort (10 points)

In your **HW5.txt** file describe how modified insertion sort differs from classic insertion sort. Specifically,

- what does the method binsearch actually do?
- how is it used in the modified insertion sort?
- what is the space complexity of classic insertion sort? (in other words, how much additional (temporary) space is required to have insertion sort work)
- what is the space complexity of modified insertion sort? (how much additional (temporary) space is used by modified insertion sort).

Note: space complexity is very similar to time complexity, it just measures how much memory is required to run the algorithm Most often (but not always) this in term of how much auxilliary or extra space is needed for the algorithm to complete (ignoring the space required by the data structure itself, since it must be stored no matter what)

Turning in your assignment

For this assignment you will turn in your **HW5.txt** and **Merge12.java** files using the command **bundleHW5**.