# CSE12 - Spring 2018 PR #4

*Sorting and Running Time*

(100 points)

**Due 11:59pm 19 May 2018**

In this assignment you will implement merge sort and then analyze the running time of sorting codes 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 other's 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**
- **PR4.txt**

PR4 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 PR4.txt file, please include the following header:

```
CSE 12 Program 4
Your Name
Your PID
The date
```

# Part 1 - Implementing Merge Sort (50 points)

You are to implement the merge sort algorithm discussed in lecture and described 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 may look at code in your textbook but you should actually type in your modified code yourself. There are MANY implementations of merge sort out there, it best if you simply start from the principles of how merge sort is supposed to work and code it yourself**. Your code MUST implement the Sort12 interface**. The defined sort() method must handle any List of elements that implement the Comparable interface. You don't know what kind of List is actually passed to you or what kind of backing store it uses. Do not make assumptions (more on this a bit later).

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 changing the reference `list` to point to a new object. (Please see the "testing merge sort below").**

**A note of advice:** some of you are still confused/unclear about object references and really understanding the meaning of the bold statement above. Ask yourself the question, why is it NOT OK to return a reference to a sorted ArrayList? NOW is the time to clear this up for yourself if the above statements are at all confusing or incomprehensible. Talk to TAs, tutors, or the professor if you still need help understanding the above.

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 in lecture. 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 allows 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 correctly sorts input).

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

**Performance of Merge Sort (10/50 points)**

Your performance of Merge Sort should be O(nLogn). We will test with Lists that have O(1) insertion performance, but accessing the nth element of the List passed to your sort algorithm might be O(n). The performance of merge sort should NOT be sensitive to the underlying backing store or implementation of the List passed to the sort() method.

You need to understand the comments above about COPYING the unsorted list to an ArrayList and WHY this is important for predictable performance. You can assume for this assignment that the clear() method of the Collection interface is a valid operation for the List instances you might be passed. (you might also want to review the addAll(Collection c) method) Look at the Collection and List interfaces and figure out which methods you can take advantage to simplify your code.

How can you TEST your code? Read some sortable information in and store in an ArrayList and test your merge sort. Then read that same sortable information in and store in a LinkedList. Your unsorted data needs to be large enough that you get long-enough runtimes to compare. Your resulting tests should have similar runtimes. If one is significantly faster than the other, then something is incorrect in your implementation. In our testing overall runtimes did not differ more than about 25% when the unsorted list was stored in ArrayList vs. LinkedList. (please do not ask if a 27% or 31% difference is acceptable, it is. If one is 2X or more faster/slower than the other then your code IS wrong). We're not grading you on absolute performance – meaning

your MergeSort might faster or slower than ours on the same machine with identical input. Your MergeSort should exhibit O(nLogn) complexity when timing different sized inputs.

## Part 2: Running Time of Various sort Algorithms (40 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.  You need larger input to adequately test this algorithms**

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 PR4.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? You might want to graph time vs. input size see the general shape of how a particular algorithm is performing. Graphs are not required, they just might help you better understand what is going on.

(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 **PR4.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 **PR4.txt** and **Merge12.java** via gradesource. An announcement on Piazza will be made when it is ready.