Claremont McKenna College
Computer Science

CS 62 **Handout 4: PS 4** **October 4, 2015**

This problem set is due **11:55pm on Tuesday, October 13**.

Be sure to include a comment at the top of each file submitted that gives your name and email address.

Submit the following files to the **Assignments** area on **Sakai**. Multiple submissions are allowed within the time limit. Please do **not** submit `.class` files or Eclipse project files.

```
Answers.pdf
Selection.java
MergeObject.java
Quick.java
ExternalMergeSort.java
input.txt (that works with your implementation)
readme.txt (if appropriate)
any other file that we will need to grade your work.
```

Problem 1

In the [given] folder you will see an incomplete version of a selection sort implementation. We want to complete the implementation and analyze the running time complexity. Do the following:

1. Complete the implementation.
2. Set up a recurrence relation for the running time complexity of `sSortRec`, $\Theta(n)$, where n is the number of elements in the array being sorted.
3. Solve the recurrence relation to get the $\Theta(n)$. N squared

Submit your completed `Selection.java`. Also include your answers for parts 2 and 3 in a file named `Answers.pdf` and submit it. Organize your answers for readability.

~~Problem 2~~

Let a and b be positive constants. Show that if $f(n)$ is $\Theta(log_a n)$, then $f(n)$ is also $\Theta(log_b n)$. Hint: Use the definition of Θ and an appropriate fundamental property of logarithm.

~~Problem 3~~

Rewrite `Merge.java` with a new name `MergeObject.java` so that it sorts an array of `Objects` rather than an array of `ints`. That is, the signature of this method must be:

```
public static void sort(Object[] a)
```

It sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array are objects from classes that have implemented the `Comparable` interface. Furthermore, all elements in the array are assumed to be mutually comparable (that is, `e1.compareTo(e2)` would not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

Submit `MergeObject.java`.

~~Problem 4~~

We will study quick sort on Tuesday in class. In the [given] folder, you will find an incomplete version of `Quick.java`. Your task in the problem is to complete the given file. Submit your completed `Quick.java`.

State what the running time complexity of your algorithm in the best case is in Θ notation. Prove your claim. Include your statement and proof in a file named `Answers.pdf`. (Hint: When you prove this, it will be quite similar to the one we will do in class for merge sort.)

Now, state what the running time complexity of your algorithm for the worst case is in Θ notation. Prove your claim. Include your statement and proof in the file `Answers.pdf`. (Hint: When you prove this, it will be quite similar to the one you did for selection sort in Problem 1.)

~~Problem 5~~

String external sort will be uploaded

As we discussed in class on Thursday (10/1) merge sort can be used to sort a file of items on disk. It is useful when the file contains too much data to fit in memory.

In this problem you will sort a file containing words (strings) first and modify it to sort integers. Assume that the file is too large to fit in memory to sort in memory. In the [given] folder you will see five files provided for this problem: `ExternalMergeSort.java` (an incomplete version), `ExternalMergeSort.tar` (a tar file containing an executable), `input1.txt` (a sample input file I used), `input2.txt` (another sample input file I used. This one shows that a line can contain multiple items.), and `tempFiles.png` (a screen shot showing the temporary files generated in one of my runs). Your task is to complete `ExternalMergeSort.java`. First untar the tar file and run the executable like this:

```
> java ExternalMergeSort input1.txt 3
or
> java ExternalMergeSort input2.txt 3
```

using the sample input files given. Here 3 is the chunk (buffer) size as you divide the large file into smaller chunks. It will generate a bunch of temporary files in the same folder where the executable is located. Run it several times with an integer as small as 1 to as large as the number of items in the input file to see what kinds of temporary files it generates. You will want to delete the temporary files between runs to see the new ones being generated in each run.

As you will see in the given Java file, you will divide the input file into two smaller files and each of those two into yet two more smaller files continuing this way until the size of the divided files are small enough to sort in memory using one of the sorting algorithms that we have studied. As instructed in `ExternalMergeSort.java` as comments, you will incorporate quick sort algorithm into the file. You may use the one we completed in Problem 4 above. If you have not completed Problem 4 yet, use `Arrays.sort(...)` from `java.util.*` until yours is ready.

For the `mergeFiles` method, you should *not* read the entire files into memory, merge them, and write the result out to a temporary file. Remember that your input file was too large to fit in memory? This design was necessary to deal with that situation in a memory efficient manner. You should open both input files and read one item at a time from either the first or the second file and output it directly to the output file being generated in the desired order.

When you are done making it work with word (string) files, try to make it work with a file containing numbers (integers). This would be simple changes.

Delete the unnecessary temporary files as one of the last steps in `main`. You will want to remember the temporary file names being generated in a data structure such as an array list and delete them when you don't need them any longer.

Submit all the Java files that we will need to run your program along with an input file that works with your implementation. The size of the input file that you submit does not have to be that large ("input1.txt" or "input2.txt" that I gave you could be used). We will use a large file of our own to test your code. Sakai would not be very happy with large files - nor would we be as we pass around files to graders. If you did the integer version as well, name your Java files to reflect that so that we can easily tell which is which, and also include an input file for that version as well. Provide a readme file describing how to run your program if it is not obvious.