COMP2402: Data Structures

Assignment 1: Using the JCF

New: The grading server now runs `java` with the `–encoding utf8` argument so that any non-ASCII characters are treated as utf8 encoded. This fixes a problem where source code that included comments copy/pasted from the assignment was not compiling on the server but would probably have compiled anywhere else.

The server will now also show you your compilation errors.

Grading

This assignment will be tested and graded by a computer program (and you can submit as many times as you like). For this to work, there are some important rules you must follow:

- Keep the directory structure of the provided zip file. If you find a file in the subdirectory `comp2402a1` leave it there.
- Keep the package structure of the provided zip file. If you find a `package comp2402a1;` directive at the top of a file, leave it there.
- Do not rename or change the visibility of any methods already present. If a method or class is `public` leave it that way.
- Do not change the `main(String)` method of any class. Some of these are setup to read command line arguments and/or open input and output files. Don't change this behaviour. Instead, learn how to use command-line arguments to do your own testing.
- Submit early and often. The submission server compiles and runs your code, and gives you a mark. You can submit as often as you like and only your latest submission will count. There is no excuse for submitting code that does not compile or does not pass tests.
- Write efficient code. The submission server places a limit on how much time it will spend

executing your code, even on inputs with a million lines. For some questions it also places a limit on how much memory your code can use. If you choose and use your data structures correctly, your code will easily execute within the time limit. Choose the wrong data structure, or use it the wrong way, and your code will be too slow for the submission server to grade (resulting in a grade of 0).

To get a very rough idea of how fast your code should be, here is a sample solution running on a 1 million line input file:

```
morin@lauteschwein:~/comp2402$ java comp2402a1.Part8 1M.in 1M.out
Execution time: 1.379501858s
```

This is on a fairly fast computer, a slower computer might take 3-5 seconds, which is ok. Taking 30 seconds, minutes, hours, or days is not ok.

The skeleton code in the zip file compiles fine. Here's what it looks like when you unzip, compile, and run Part0 from the command line:

```
morin@lauteschwein:~/assn$ unzip comp2402a1.zip
Archive:  comp2402a1.zip
  inflating: comp2402a1/Part0.java
  inflating: comp2402a1/Part10.java
  inflating: comp2402a1/Part1.java
  inflating: comp2402a1/Part2.java
  inflating: comp2402a1/Part3.java
  inflating: comp2402a1/Part4.java
  inflating: comp2402a1/Part5.java
  inflating: comp2402a1/Part6.java
  inflating: comp2402a1/Part7.java
  inflating: comp2402a1/Part8.java
  inflating: comp2402a1/Part9.java
morin@lauteschwein:~/assn$ javac comp2402a1/*.java
morin@lauteschwein:~/assn$ ls comp2402a1/
Part0.class   Part1.class  Part3.class  Part5.class  Part7.class  Part9.cla
Part0.java    Part1.java   Part3.java   Part5.java   Part7.java   Part9.jav
Part10.class  Part2.class  Part4.class  Part6.class  Part8.class
Part10.java   Part2.java   Part4.java   Part6.java   Part8.java
morin@lauteschwein:~/assn$ java comp2402a1.Part0
apple
plum
peach
pie
```

```
[I hit Ctrl-d to end the input here]
plum
apple
peach
pie
Execution time: 9.437427395
morin@lauteschwein:~/teaching/2402/assn$
```

Submitting and Testing

The submission server is now accepting submissions. Please try it out.

The Assignment

Start by downloading the Assignment 1 Zip File, which contains a skeleton of the code you need to write. The file `Part0.java` in the zip file actually does something. You can use its `doit()` method as a starting point for your solutions. Compile it. Run it. Test out the various ways of inputting and outputting data.

This assignment is about using the Java Collections Framework to accomplish some basic text-processing tasks. These questions involve choosing the right abstraction (`Collection`, `Set`, `List`, `Queue`, `Deque`, `SortedSet`, `Map`, or `SortedMap`) to efficiently accomplish the task at hand. The best way to do these is to read the question and then think about what type of Collection is best to use to solve it. There are only a few lines of code you need to write to solve each of them.

Here is a directory containing sample input and output files for each question. You can also download all these files as a zip file. If you find that a particular question is unclear, you can probably clarify it by looking at the sample files for that question.

Unless specified otherwise, "sorted order" refers to the natural sorted order on Strings, as defined by String.compareTo(s).

Caution: It is always better to use `c.isEmpty()` than `c.size()==0` to check if a collection is empty. In some cases (and one that you may encounter in this assignment) `c.size()` is slow but `c.isEmpty()` is always fast.

1. [5 marks] Read the input one line at a time until you have read all lines. Now output these lines in the opposite order from which they were read.
2. [5 marks] Read the input one line at a time and output the current line if and only if it is larger than any other line read so far. (Here, smaller is with respect to the usual order on Strings, as defined by `String.compareTo()`.
3. [5 marks] Read the input one line at a time and output only the last 5000 lines in the order they appear. If there are fewer than 5000 lines, output them all. For full marks, your code should be fast and should never store more than 5001 lines.
4. [5 marks] Read the input one line at a time and output the current line if and only if it is different from *every* previous line.
5. [5 marks] Read the input one line at a time. When you are done, output all the lines in reverse sorted order (so that "zebra" will appear before "apple").
6. [5 marks] For this question, you may assume that every input line is distinct. Read the entire input input one line at time. If the input has $n < 4001$ lines then don't output anything. Otherwise, output the line ℓ that has exactly 4000 lines smaller than ℓ. (Again, greater than and less than are with respect to the ordering defined by `String.compareTo()`.) For full marks, you should do this without ever storing more than 4001 lines.
7. [5 marks] The input is broken into chunks of consecutive lines, where each pair of consecutive chunks is separated by a line containing "----snip----". Read the entire input and break it into chunks $C_1, \ldots, C_k$. Then output the chunks in reverse order $C_k, \ldots, C_1$ but preserving the order of the lines within each chunk.
8. [5 marks] For this question, you are going to make a *permutation array*. Read the entire input file. You may assume that each input line is distinct. If the file has n lines, treat them as if they are numbered $0, \ldots, n-1$. Now output a permutation $\pi_0, \ldots, \pi_{n-1}$ of $\{0, \ldots, n-1\}$ so that π_0 is the number of the smallest line, π_1 is the number of the second smallest line, ... , and π_{n-1} is the number of the largest line. (Again, smaller and larger refer to the natural ordering on strings). Your output should consist of n lines, where line i contains (the string representation of) π_i , for each $i \in \{0, \ldots, n-1\}$.
9. [5 marks] Read the input one line at a time and output the current line if and only if it is not a prefix of some previous line. For example, if the some line is "zoodles" and some subsequent line is "zoo", then the line containing "zoo" should not be output.
10. [5 marks] For this problem you may assume that every input line is distinct. Your output

should begin with the largest line ℓ_0 in the input. Next should be the largest line ℓ_1 that appears after ℓ_0 in the input. In general, line ℓ_i of your output should be the largest line that appears after ℓ_{i-1} in the input. (Hint: You can do this very efficiently and easily with an `ArrayList`. Think about your solution before you start coding.)