COMP 2402 AB- Fall 2023
Assignment #1

Due: Wednesday, September 27, 23:59

Submit early and often. Late submissions (up to 12 hours) will be accepted.

Academic Integrity

You may:

- Discuss general approaches with course staff and your classmates,
- Use code and/or ideas from the textbook,
- Use a search engine / the internet to look up basic Java syntax.

You may not:

- Send or otherwise share code or code snippets with classmates,
- Use code not written by you, unless it is code from the textbook (and you should cite it in comments),
- Use a search engine / the internet to look up approaches to the assignment,
- Use code from previous iterations of the course, unless it was solely written by you,
- Use the internet to find source code or videos that give solutions to the assignment.

If you ever have any questions about what is or is not allowable regarding academic integrity, please do not hesitate to reach out to course staff. We will be happy to answer. Sometimes it is difficult to determine the exact line, but if you cross it the punishment is severe and out of our hands. Any student caught violating academic integrity, whether intentionally or not, will be reported to the Dean and be penalized. Please see Carleton University's Academic Integrity page.

Grading

This assignment will be tested and graded by a computer program (and **you can submit as many times as you like, your highest grade is recorded**). 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).

Submitting and Testing

The submission server is ready: here. If you have issues, please post to Discord to the teaching team (or the class) and we'll see if we can help.

When you submit your code, the server runs tests on your code. These are bigger and better tests than the small number of tests provided in the "Local Tests" section further down this document. They are obfuscated from you, because you should try to find exhaustive tests of your own code. This can be frustrating because you are still learning to think critically about your own code, to figure out where its weaknesses are. But have patience with yourself and make sure you read the question carefully to understand its edge cases.

Warning: Do not wait until the last minute to submit your assignment. If the server is heavily loaded, borderline tests may start to fail. **You can submit multiple times and your best score is recorded.**

Start by downloading and decompressing the Assignment 1 Zip File (comp2402a1.zip), which contains a skeleton of the code you need to write. You should have the files: Part0.java, Part1.java, Part2.java, Part3.java, Part4.java, Part5.java, Part6.java, Part7.java, Part8.java, Part9.java, and Part10.java.

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:

```
alina@euclid:~$ unzip comp2402a1.zip
Archive:  comp2402a1.zip
  inflating: comp2402a1/Part0.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
  inflating: comp2402a1/Part10.java
alina@euclid:~$ javac comp2402a1/*.java
alina@euclid:~$ java comp2402a1.Part0
blue
red
yellow     } type these
violet
[Hit Ctrl-d or Ctrl-z to end the input here]
blue
red
yellow
violet
Execution time: 8.128651
alina@euclid:~$
```

If you are having trouble running these programs, figure this out first before attempting to do the assignment. If you are stuck, ask on Discord, and course staff or another student will likely help you fairly quickly.

The Assignment

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.

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. You should not need to modify `Part0.java`, it is a template.

You can download some sample input and output files for each question as a zip file (`a1-io.zip`). If you find that a particular question is unclear, you can probably clarify it by looking at the sample files for that question. Once you get a sense for how to test your code with these input files, write your own tests that test your program more thoroughly (**the provided tests are not exhaustive!**)

Assignment 1

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. [10 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, but with consecutive duplicates removed.
2. [10 marks] Read the input one line at a time and output the current line if and only if it is strictly larger than all other lines read so far or strictly smaller than the previously **outputted** line. If this is the first nonempty line you read, it is considered larger than anything you've read so far. (Here, larger is with respect to the usual order on Strings, as defined by String.compareTo()).
3. [10 marks] Read the input one line at a time. If you read less than 1000 lines, then do not output anything. If you read less than 2402 lines, then output the 1000th line in sorted order of the input lines. If you read 2402 or more lines, then consider only the last 2402 lines, and output the 1000th line in sorted order from the last 2402 lines. For full marks, your code should be fast and should never store more than 2403 lines.
4. [10 marks] Read the input one line at a time and output the current line ℓ if and only if none of the previous lines starts with ℓ.
5. [10 marks] Read the input one line at a time until you have read all lines. Output all the lines in sorted order (as defined by String.compareTo(s)).
6. [10 marks] For this question, you may assume that every input line is distinct. Read the entire input one line at time. If the input has less than 901 lines, then do not output anything. Otherwise, output the line ℓ that has exactly 900 lines greater 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 901 lines.
7. [10 marks] The input contains special lines: "***reset***". Read the entire input and break it into blocks of consecutive lines $B_1, \dots, B_k$, where B_1 is a block that contains one line, B_2 – two lines, B_3 – three lines, … B_k contains k lines (or less). When you read the "reset" line, you add it to the current block and reset the size of the next block to 1. So that following strings are broken into blocks of size $1, 2, 3, \dots$ lines, until you read another "reset" string and reset the size again. And so on. When you are done reading all the strings, output the blocks in reverse order $B_k, B_{k-1}, \dots, B_3, B_2, B_1$ but preserving the order of the lines within each block.
8. [10 marks] Assume your input consists of n lines. Read the input one line at a time until you have read all lines. Treat the lines as if they are numbered $0, \dots, n-1$. Note, there can be duplicates. Output a permutation $\pi_0, \pi_1, \dots, \pi_{n-1}$ of $\{0, \dots, n-1\}$ 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 $i \in \{0, \dots, n-1\}$. The numbers for duplicates should be outputted in the same order in which the lines were read.

9. [10 marks] Read the input one line at a time and output the current line if and only it has appeared at least 3 times before.
10. [10 marks] For this problem you may assume that every input line is distinct. Read the input one line at a time and assume that the lines are numbered $0, \dots, n-1$. Your output should begin with the largest line in the input. Assume its number is i. Next, output the largest line among the lines numbered $i+1, \dots, n-1$. Assume its number is j, where $j > i$. Next, output the largest line among the lines numbered $j+1, \dots, n-1$. And so on. The last line of your output will be the last line of the input. (Here, largest is with respect to the usual order on Strings, as defined by String.compareTo()). (Hint: Do not store all the lines. This problem can be solved very efficiently with an ArrayList. Think about your solution before you start coding.)

Tips, Tricks, and FAQs

How should I approach each problem?

- Make sure you understand it. Construct **small** examples, and compute (by hand) the expected output. If you aren't sure what the output should be, go no further until you get clarification.
- Now that you understand what you are supposed to output, and you've been able to solve it by hand, think about how you solved it and whether you could explain it to someone. How about explaining it to a computer?
- If it still seems challenging, what about a simpler case? Can you solve a similar or simplified problem? Maybe a special case? If you were allowed to make certain assumptions, could you do it then? Try constructing your code incrementally, solving the smaller or simpler problems, then, only expanding scope once you're sure your simplified problems are solved.

How should I test my code?

- You should be testing your code as you go along.
- Use small tests first so that you can compute the correct solution by hand.
- For testing, replace big numbers (like 1000 or 2402 lines in Part 3) with small (3 or 5). Don't forget to change them back before submitting to the server.
- Think about edge cases, e.g. empty lists, lists of different lengths, duplicates, etc.
- Think about large inputs, random inputs.
- Test for speed.