© 2020 Sudarshan S. Chawathe

The **packaging and submission procedure** for this assignment[1] is similar to that for the previous one, replacing `hw01` with `hw02`. As well, the **guidelines** on allowable use of external resources and other details remain unchanged. It is important to *use the class newsgroup* for clarification and elaboration.

The **focus** of this assignment is *lexicographic ordering*, with **tasks** concerned with implementing such ordering using relevant features of C++. Lexicographic ordering may be thought of as a way to lift a *strict total order* defined on a set to a strict total order on the set of sequences of elements drawn from that set. It is a widely used generalization of the way in which words are ordered in a conventional dictionary (e.g., *flower* $<$ *flume*) based on an underlying order on the alphabet (e.g., $a < b < c < \cdots < z$).

In more detail, recall[2] that if $<$ is a strict total order on set A then for any two elements $x, y \in A$, exactly one of $x < y$, $x = y$, and $y < x$ must hold. For example, the natural $<$ relation on integers is a strict total order but the usual $\subseteq$ relation is not a strict total order on collections of sets. Let us use the notation ε to denote the empty sequence which contains no elements and the notation $h \cdot t$ to denote a nonempty sequence that has h (head) as its first element and t (tail) as the remainder of the sequence (possibly empty). Then the lexicographic ordering on sequences may be defined using two rules:

- $\varepsilon < h \cdot t$.
- $g \cdot s < h \cdot t$ if $g < h$ or $g = h$ and (recursively) $s < t$.

We will use the *UTF-8* character encoding[3] and compare characters using their Unicode code points. (The ASCII and UTF-8 encodings of ASCII characters are the same.)

1. (20 pts.) Write a program that reads two strings on its standard input, one per line, and outputs a single integer to its standard output, followed by a newline. The integer that is output should be 0 if the two strings are equal, -1 if the first is lexicographically less than the second, and 1 otherwise. (In this simple input format, the strings cannot contain newlines, which are used as separators.) Use the string-comparison features built into C++ (including the STL) as much as possible to simplify the implementation.

2. (20 pts.) Write another program that behaves identically to the above but that does not use the string-comparison features built into C++; instead, the string comparison is implemented from scratch based on the earlier definition.

3. (20 pts.) Write a program to that reads a set of strings on standard input (one string per line) and writes that set to standard output (one string per line, just like the input) but in lexicographically sorted order. Use appropriate facilities of C++ and the STL to simplify the code.

[1]This version, dt. 2020-10-10, updates the one dt. 2020-10-09, by fixing the wording of Qs. 4 and 6.
[2]See, e.g., `https://en.wikipedia.org/wiki/Total_order`.
[3]`https://en.wikipedia.org/wiki/UTF-8`

4. (20 pts.) Write another program that behaves identically to the above (Question 3) but that does not use any string-comparison or sorting facilities of C++ or the STL and that is based on a from-scratch implementation of *selection sort*. See Question 7.

5. (20 pts.) Write a program to that reads a *collection of sets of strings* on standard input (one string per line, with a single blank line separating sets within the collection) and writes that collection to standard output (using the same format as the input) but in lexicographically sorted order. Use appropriate facilities of C++ and the STL to simplify the code.

6. (20 pts.) Write another program that behaves identically to the above (Question 5) but that does not use any string-comparison or sorting facilities of C++ or the STL and that is based on a from-scratch implementation of *selection sort*. See Question 7.

7. (20 pts.) Using the templating facilities of C++, implement a general sorting routine, based on selection sort, that can sort items of arbitrary types. Include demonstration code that sorts collections of at least three substantially different types using this implementation.

8. (30 pts.) Write a series of tests that may be used to check the correctness of code for the earlier questions. Using a combination of suitable comments and the associated report (below), document your tests and explain why they are both correct and interesting (i.e., test for likely errors, boundary conditions, tricky cases, etc.).

9. (30 pts.) Write a brief report (two pages suggested length, five pages maximum length) documenting your program and, in particular, highlighting the parts that are interesting, parts that are or incomplete or buggy, aspects that were easy, difficult, etc. (Documented bugs will diminish their negative impact on the score.)