```java
import java.util.Comparator;

/**
 * HomeWork#18
 *
 * @author Sam Espanioly
 */
public final class HW18 {

    /**
     * Default constructor--private to prevent instantiation.
     */
    private HW18() {
        // no code needed here
    }

    /**
     * Main method.
     *
     * @param args
     *            the command line arguments; unused here
     */

    /**
     * Removes and returns the minimum value from {@code q} according to the
     * ordering provided by the {@code compare} method from {@code order}.
     *
     * @param q
     *            the queue
     * @param order
     *            ordering by which to compare entries
```

```
35      * @return the minimum value from {@code q}
36      * @updates q
37      * @requires <pre>
38      * q /= empty_string  and
39      *  [the relation computed by order.compare is a total
  preorder]
40      * </pre>
41      * @ensures <pre>
42      * perms(q * <removeMin>, #q)  and
43      *  for all x: string of character
44      *      where (x is in entries (q))
45      *    ([relation computed by order.compare method]
  (removeMin, x))
46      * </pre>
47      */
48     //use compare do not use sort
49     private static String removeMin(Queue<String> q,
  Comparator<String> order) {
50
51         int length = q.length();
52
53         String elements[] = new String[length];
54         for (int i = 0; i < length; i++) { //loop to fill
  array
55             elements[i] = q.dequeue();
56         } //we pick the first value in the queue assuming
  there is one
57         String min = elements[0];
58         for (int i = 0; i < length; i++) {
59             if (order.compare(elements[i], min) < 0) {
60                 String temp = elements[i]; //temporary
  value
61                 //we are looking the minimum value
62                 if (min.length() > temp.length()) {
```

```
63                      min = temp;
64                  }
65
66              }
67          }
68          return min;
69      }
70
71      /**
72       * Sorts {@code q} according to the ordering provided
  by the {@code compare}
73       * method from {@code order}.
74       *
75       * @param q
76       *              the queue
77       * @param order
78       *              ordering by which to sort
79       * @updates q
80       * @requires [the relation computed by order.compare
  is a total preorder]
81       * @ensures q = [#q ordered by the relation computed
  by order.compare]
82       */
83      public static void sort(Queue<String> q,
  Comparator<String> order) {
84          if (q.length() > 0) {
85
86              Queue<String> temp = new Queue1L<>();
87              temp.enqueue(removeMin(q, order)); //
  temporary order
88              q.append(temp); // the updated version of it
89              while (q.length() > 0) { //recursion
90                  sort(q, order);
91              }
```

```
 92         }
 93     }
 94
 95 //    public static void main(String[] args) {
 96 //        SimpleWriter out = new SimpleWriter1L();
 97 //        out.println("Hello World!");
 98 //        out.close();
 99 //    }
100
101 }
102
```