```
 1 import components.set.Set;
 2 import components.set.Set1L;
 3 import components.simplereader.SimpleReader;
 4 import components.simplereader.SimpleReader1L;
 5 import components.simplewriter.SimpleWriter;
 6 import components.simplewriter.SimpleWriter1L;
 7
 8 /**
 9  * Utility class to support string reassembly from fragments.
10  *
11  * @author Ansh Pachauri
12  *
13  * @mathdefinitions <pre>
14  *
15  * OVERLAPS (
16  *   s1: string of character,
17  *   s2: string of character,
18  *   k: integer
19  *  ) : boolean is
20  *  0 <= k  and  k <= |s1|  and  k <= |s2|  and
21  *  s1[|s1|-k, |s1|) = s2[0, k)
22  *
23  * SUBSTRINGS (
24  *   strSet: finite set of string of character,
25  *   s: string of character
26  *  ) : finite set of string of character is
27  *  {t: string of character
28  *    where (t is in strSet  and  t is substring of s)
29  *   (t)}
30  *
31  * SUPERSTRINGS (
32  *   strSet: finite set of string of character,
33  *   s: string of character
34  *  ) : finite set of string of character is
35  *  {t: string of character
36  *    where (t is in strSet  and  s is substring of t)
37  *   (t)}
38  *
39  * CONTAINS_NO_SUBSTRING_PAIRS (
```

```
 *   strSet: finite set of string of character
 *  ) : boolean is
 *  for all t: string of character
 *    where (t is in strSet)
 *   (SUBSTRINGS(strSet \ {t}, t) = {})
 *
 * ALL_SUPERSTRINGS (
 *   strSet: finite set of string of character
 *  ) : set of string of character is
 *  {t: string of character
 *    where (SUBSTRINGS(strSet, t) = strSet)
 *   (t)}
 *
 * CONTAINS_NO_OVERLAPPING_PAIRS (
 *   strSet: finite set of string of character
 *  ) : boolean is
 *  for all t1, t2: string of character, k: integer
 *    where (t1 /= t2  and  t1 is in strSet  and  t2 is in strSet  and
 *           1 <= k  and  k <= |s1|  and  k <= |s2|)
 *   (not OVERLAPS(s1, s2, k))
 *
 * </pre>
 */
public final class StringReassembly {

    /**
     * Private no-argument constructor to prevent instantiation of this utility
     * class.
     */
    private StringReassembly() {
    }

    /**
     * Reports the maximum length of a common suffix of {@code str1} and prefix
     * of {@code str2}.
     *
```

```
 76     * @param str1
 77     *            first string
 78     * @param str2
 79     *            second string
 80     * @return maximum overlap between right end of {@code
    str1} and left end of
 81     *         {@code str2}
 82     * @requires <pre>
 83     * str1 is not substring of str2  and
 84     * str2 is not substring of str1
 85     * </pre>
 86     * @ensures <pre>
 87     * OVERLAPS(str1, str2, overlap)  and
 88     * for all k: integer
 89     *     where (overlap < k  and  k <= |str1|  and  k <= |
    str2|)
 90     *  (not OVERLAPS(str1, str2, k))
 91     * </pre>
 92     */
 93    public static int overlap(String str1, String str2) {
 94        assert str1 != null : "Violation of: str1 is not null";
 95        assert str2 != null : "Violation of: str2 is not null";
 96        assert str2.indexOf(str1) < 0 : "Violation of: "
 97                + "str1 is not substring of str2";
 98        assert str1.indexOf(str2) < 0 : "Violation of: "
 99                + "str2 is not substring of str1";
100        /*
101         * Start with maximum possible overlap and work down
    until a match is
102         * found; think about it and try it on some examples to
    see why
103         * iterating in the other direction doesn't work
104         */
105        int maxOverlap = str2.length() - 1;
106        while (!str1.regionMatches(str1.length() - maxOverlap,
    str2, 0,
107                maxOverlap)) {
108            maxOverlap--;
109        }
```

```
110         return maxOverlap;
111     }
112 
113     /**
114      * Returns concatenation of {@code str1} and {@code str2}
    from which one of
115      * the two "copies" of the common string of {@code overlap}
    characters at
116      * the end of {@code str1} and the beginning of {@code
    str2} has been
117      * removed.
118      *
119      * @param str1
120      *             first string
121      * @param str2
122      *             second string
123      * @param overlap
124      *             amount of overlap
125      * @return combination with one "copy" of overlap removed
126      * @requires OVERLAPS(str1, str2, overlap)
127      * @ensures combination = str1[0, |str1|-overlap) * str2
128      */
129     public static String combination(String str1, String str2,
    int overlap) {
130         assert str1 != null : "Violation of: str1 is not null";
131         assert str2 != null : "Violation of: str2 is not null";
132         assert 0 <= overlap && overlap <= str1.length()
133                 && overlap <= str2.length()
134                 && str1.regionMatches(str1.length() - overlap,
    str2, 0,
135                         overlap) : ""
136                                 + "Violation of: OVERLAPS(str1,
    str2, overlap)";
137 
138         /*
139          * Hint: consider using substring (a String method)
140          */
141 
142         // TODO: fill in body
```

```
143         String strTemp = str2.substring(overlap,
    str2.length());
144         return str1 + strTemp;
145     }
146 
147     /**
148      * Adds {@code str} to {@code strSet} if and only if it is
    not a substring
149      * of any string already in {@code strSet}; and if it is
    added, also removes
150      * from {@code strSet} any string already in {@code strSet}
    that is a
151      * substring of {@code str}.
152      *
153      * @param strSet
154      *             set to consider adding to
155      * @param str
156      *             string to consider adding
157      * @updates strSet
158      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
159      * @ensures <pre>
160      * if SUPERSTRINGS(#strSet, str) = {}
161      *  then strSet = #strSet union {str} \ SUBSTRINGS(#strSet,
    str)
162      *  else strSet = #strSet
163      * </pre>
164      */
165     public static void addToSetAvoidingSubstrings(Set<String>
    strSet,
166             String str) {
167         assert strSet != null : "Violation of: strSet is not
    null";
168         assert str != null : "Violation of: str is not null";
169         /*
170          * Note: Precondition not checked!
171          */
172 
173         /*
174          * Hint: consider using contains (a String method)
```

```
175          */
176
177         // TODO: fill in body
178         boolean subCheck = true;
179         Set<String> tmp = new Set1L<>();
180         for (String x : strSet) {
181             if (x.contains(str)) {
182                 subCheck = false;
183                 if (!tmp.contains(str)) {
184                     tmp.add(str);
185                 }
186             } else if (str.contains(x)) {
187                 if (!tmp.contains(str)) {
188                     tmp.add(x);
189                 }
190             }
191         }
192         if (subCheck) {
193             strSet.add(str);
194         }
195         strSet.remove(tmp);
196     }
197
198     /**
199      * Returns the set of all individual lines read from {@code
    input}, except
200      * that any line that is a substring of another is not in
    the returned set.
201      *
202      * @param input
203      *            source of strings, one per line
204      * @return set of lines read from {@code input}
205      * @requires input.is_open
206      * @ensures <pre>
207      * input.is_open  and  input.content = <>  and
208      * linesFromInput = [maximal set of lines from
    #input.content such that
209      *
    CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
```

```
210      * </pre>
211      */
212     public static Set<String> linesFromInput(SimpleReader
    input) {
213         assert input != null : "Violation of: input is not
    null";
214         assert input.isOpen() : "Violation of: input.is_open";
215
216         // TODO: fill in body
217
218         Set<String> ans = new Set1L();
219         while (!input.atEOS()) {
220             String temp = input.nextLine();
221             if (ans.size() == 0) {
222                 ans.add(temp);
223             }
224             if (!ans.contains(temp)) {
225                 addToSetAvoidingSubstrings(ans, temp);
226             }
227         }
228         return ans;
229     }
230
231     /**
232      * Returns the longest overlap between the suffix of one
    string and the
233      * prefix of another string in {@code strSet}, and
    identifies the two
234      * strings that achieve that overlap.
235      *
236      * @param strSet
237      *             the set of strings examined
238      * @param bestTwo
239      *             an array containing (upon return) the two
    strings with the
240      *             largest such overlap between the suffix of
    {@code bestTwo[0]}
241      *             and the prefix of {@code bestTwo[1]}
242      * @return the amount of overlap between those two strings
```

```
243      * @replaces bestTwo[0], bestTwo[1]
244      * @requires <pre>
245      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
246      * bestTwo.length >= 2
247      * </pre>
248      * @ensures <pre>
249      * bestTwo[0] is in strSet  and
250      * bestTwo[1] is in strSet  and
251      * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap)  and
252      * for all str1, str2: string of character, overlap:
    integer
253      *     where (str1 is in strSet  and  str2 is in strSet
    and
254      *            OVERLAPS(str1, str2, overlap))
255      *   (overlap <= bestOverlap)
256      * </pre>
257      */
258     private static int bestOverlap(Set<String> strSet, String[]
    bestTwo) {
259         assert strSet != null : "Violation of: strSet is not
    null";
260         assert bestTwo != null : "Violation of: bestTwo is not
    null";
261         assert bestTwo.length >= 2 : "Violation of:
    bestTwo.length >= 2";
262         /*
263          * Note: Rest of precondition not checked!
264          */
265         int bestOverlap = 0;
266         Set<String> processed = strSet.newInstance();
267         while (strSet.size() > 0) {
268             /*
269              * Remove one string from strSet to check against
    all others
270              */
271             String str0 = strSet.removeAny();
272             for (String str1 : strSet) {
273                 /*
274                  * Check str0 and str1 for overlap first in one
```

```
    order...
275                  */
276                 int overlapFrom0To1 = overlap(str0, str1);
277                 if (overlapFrom0To1 > bestOverlap) {
278                     /*
279                      * Update best overlap found so far, and
    the two strings
280                      * that produced it
281                      */
282                     bestOverlap = overlapFrom0To1;
283                     bestTwo[0] = str0;
284                     bestTwo[1] = str1;
285                 }
286                 /*
287                  * ... and then in the other order
288                  */
289                 int overlapFrom1To0 = overlap(str1, str0);
290                 if (overlapFrom1To0 > bestOverlap) {
291                     /*
292                      * Update best overlap found so far, and
    the two strings
293                      * that produced it
294                      */
295                     bestOverlap = overlapFrom1To0;
296                     bestTwo[0] = str1;
297                     bestTwo[1] = str0;
298                 }
299             }
300             /*
301              * Record that str0 has been checked against every
    other string in
302              * strSet
303              */
304             processed.add(str0);
305         }
306         /*
307          * Restore strSet and return best overlap
308          */
309         strSet.transferFrom(processed);
```

```
310         return bestOverlap;
311     }
312 
313     /**
314      * Combines strings in {@code strSet} as much as possible,
    leaving in it
315      * only strings that have no overlap between a suffix of
    one string and a
316      * prefix of another. Note: uses a "greedy approach" to
    assembly, hence may
317      * not result in {@code strSet} being as small a set as
    possible at the end.
318      *
319      * @param strSet
320      *            set of strings
321      * @updates strSet
322      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
323      * @ensures <pre>
324      * ALL_SUPERSTRINGS(strSet) is subset of
    ALL_SUPERSTRINGS(#strSet)  and
325      * |strSet| <= |#strSet|  and
326      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
327      * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
328      * </pre>
329      */
330     public static void assemble(Set<String> strSet) {
331         assert strSet != null : "Violation of: strSet is not
    null";
332         /*
333          * Note: Precondition not checked!
334          */
335         /*
336          * Combine strings as much possible, being greedy
337          */
338         boolean done = false;
339         while ((strSet.size() > 1) && !done) {
340             String[] bestTwo = new String[2];
341             int bestOverlap = bestOverlap(strSet, bestTwo);
342             if (bestOverlap == 0) {
```

```
343                 /*
344                  * No overlapping strings remain; can't do any
    more
345                  */
346                 done = true;
347             } else {
348                 /*
349                  * Replace the two most-overlapping strings
    with their
350                  * combination; this can be done with add
    rather than
351                  * addToSetAvoidingSubstrings because the
    latter would do the
352                  * same thing (this claim requires
    justification)
353                  */
354                 strSet.remove(bestTwo[0]);
355                 strSet.remove(bestTwo[1]);
356                 String overlapped = combination(bestTwo[0],
    bestTwo[1],
357                         bestOverlap);
358                 strSet.add(overlapped);
359             }
360         }
361     }
362
363     /**
364      * Prints the string {@code text} to {@code out}, replacing
    each '~' with a
365      * line separator.
366      *
367      * @param text
368      *             string to be output
369      * @param out
370      *             output stream
371      * @updates out
372      * @requires out.is_open
373      * @ensures <pre>
374      * out.is_open  and
```

```
375     * out.content = #out.content *
376     *   [text with each '~' replaced by line separator]
377     * </pre>
378     */
379    public static void printWithLineSeparators(String text,
   SimpleWriter out) {
380        assert text != null : "Violation of: text is not null";
381        assert out != null : "Violation of: out is not null";
382        assert out.isOpen() : "Violation of: out.is_open";
383
384        // TODO: fill in body
385        for (int i = 0; i < text.length(); i++) {
386            if (text.charAt(i) == '~') {
387                out.println();
388            } else {
389                out.print(text.charAt(i));
390            }
391        }
392    }
393
394    /**
395     * Given a file name (relative to the path where the
   application is running)
396     * that contains fragments of a single original source
   text, one fragment
397     * per line, outputs to stdout the result of trying to
   reassemble the
398     * original text from those fragments using a "greedy
   assembler". The
399     * result, if reassembly is complete, might be the original
   text; but this
400     * might not happen because a greedy assembler can make a
   mistake and end up
401     * predicting the fragments were from a string other than
   the true original
402     * source text. It can also end up with two or more
   fragments that are
403     * mutually non-overlapping, in which case it outputs the
   remaining
```

```
404      * fragments, appropriately labelled.
405      *
406      * @param args
407      *             Command-line arguments: not used
408      */
409     public static void main(String[] args) {
410         SimpleReader in = new SimpleReader1L();
411         SimpleWriter out = new SimpleWriter1L();
412         /*
413          * Get input file name
414          */
415         out.print("Input file (with fragments): ");
416         String inputFileName = in.nextLine();
417         SimpleReader inFile = new
    SimpleReader1L(inputFileName);
418         /*
419          * Get initial fragments from input file
420          */
421         Set<String> fragments = linesFromInput(inFile);
422         /*
423          * Close inFile; we're done with it
424          */
425         inFile.close();
426         /*
427          * Assemble fragments as far as possible
428          */
429         assemble(fragments);
430         /*
431          * Output fully assembled text or remaining fragments
432          */
433         if (fragments.size() == 1) {
434             out.println();
435             String text = fragments.removeAny();
436             printWithLineSeparators(text, out);
437         } else {
438             int fragmentNumber = 0;
439             for (String str : fragments) {
440                 fragmentNumber++;
441                 out.println();
```

```
442                 out.println("--------------------");
443                 out.println("  -- Fragment #" + fragmentNumber
    + ": --");
444                 out.println("--------------------");
445                 printWithLineSeparators(str, out);
446             }
447         }
448         /*
449          * Close input and output streams
450          */
451         in.close();
452         out.close();
453     }
454
455 }
456
```