```java
import components.set.Set;

/**
 * Utility class to support string reassembly from fragments.
 *
 * @author Feras Akileh
 *
 * @mathdefinitions <pre>
 *
 * OVERLAPS (
 *   s1: string of character,
 *   s2: string of character,
 *   k: integer
 *  ) : boolean is
 *  0 <= k  and  k <= |s1|  and  k <= |s2|  and
 *  s1[|s1|-k, |s1|) = s2[0, k)
 *
 * SUBSTRINGS (
 *   strSet: finite set of string of character,
 *   s: string of character
 *  ) : finite set of string of character is
 *  {t: string of character
 *    where (t is in strSet  and  t is substring of s)
 *   (t)}
 *
 * SUPERSTRINGS (
 *   strSet: finite set of string of character,
 *   s: string of character
 *  ) : finite set of string of character is
 *  {t: string of character
 *    where (t is in strSet  and  s is substring of t)
 *   (t)}
 *
 * 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 {

```

```
65      /**
66       * Private no-argument constructor to prevent instantiation of this utility
67       * class.
68       */
69      private StringReassembly() {
70      }
71
72      /**
73       * Reports the maximum length of a common suffix of {@code str1} and prefix
74       * of {@code str2}.
75       *
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         // gets a substring of the str1
139         int subEnd = str1.length() - overlap;
140         String subString = str1.substring(0, subEnd);
141
142         // concatenates the substring with string 2
143         String combination = subString.concat(str2);
144
145         // returns the new combined string
146         return combination;
147
148     }
149
150     /**
151      * Adds {@code str} to {@code strSet} if and only if it is not a substring
152      * of any string already in {@code strSet}; and if it is added, also removes
153      * from {@code strSet} any string already in {@code strSet} that is a
154      * substring of {@code str}.
155      *
156      * @param strSet
157      *            set to consider adding to
158      * @param str
159      *            string to consider adding
160      * @updates strSet
161      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
162      * @ensures <pre>
163      * if SUPERSTRINGS(#strSet, str) = {}
164      *  then strSet = #strSet union {str} \ SUBSTRINGS(#strSet, str)
165      *  else strSet = #strSet
166      * </pre>
167      */
168     public static void addToSetAvoidingSubstrings(Set<String> strSet,
169             String str) {
170         assert strSet != null : "Violation of: strSet is not null";
171         assert str != null : "Violation of: str is not null";
172
173         // checks strSet size
174         if (strSet.size() > 0) {
175             // removes a part of the set
176             String removed = strSet.removeAny();
177             if (removed.indexOf(str) == -1) {
178                 // calls addToSetAvoidingSubstrings
179                 addToSetAvoidingSubstrings(strSet, str);
180             }
181             if (str.indexOf(removed) == -1) {
182                 strSet.add(removed);
```

```
183             }
184         } else {
185             strSet.add(str);
186         }
187
188     }
189
190     /**
191      * Returns the set of all individual lines read from {@code input}, except
192      * that any line that is a substring of another is not in the returned set.
193      *
194      * @param input
195      *            source of strings, one per line
196      * @return set of lines read from {@code input}
197      * @requires input.is_open
198      * @ensures <pre>
199      * input.is_open  and  input.content = <>  and
200      * linesFromInput = [maximal set of lines from #input.content such that
201      *                   CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
202      * </pre>
203      */
204     public static Set<String> linesFromInput(SimpleReader input) {
205         assert input != null : "Violation of: input is not null";
206         assert input.isOpen() : "Violation of: input.is_open";
207
208         // creates new set to return
209         Set<String> stringSet = new Set2<>();
210
211         // enters loop until the end of the stream is reached
212         while (!input.atEOS()) {
213             // creates new string from the input stream
214             String streamLine = input.nextLine();
215
216             // calls addToSetAvoidingSubstrings
217             addToSetAvoidingSubstrings(stringSet, streamLine);
218         }
219
220         return stringSet;
221
222     }
223
224     /**
225      * Returns the longest overlap between the suffix of one string and the
226      * prefix of another string in {@code strSet}, and identifies the two
227      * strings that achieve that overlap.
228      *
229      * @param strSet
230      *            the set of strings examined
231      * @param bestTwo
232      *            an array containing (upon return) the two strings with the
233      *            largest such overlap between the suffix of {@code bestTwo[0]}
234      *            and the prefix of {@code bestTwo[1]}
235      * @return the amount of overlap between those two strings
236      * @replaces bestTwo[0], bestTwo[1]
237      * @requires <pre>
238      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
239      * bestTwo.length >= 2
240      * </pre>
241      * @ensures <pre>
```

```
242      * bestTwo[0] is in strSet  and
243      * bestTwo[1] is in strSet  and
244      * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap)  and
245      * for all str1, str2: string of character, overlap: integer
246      *     where (str1 is in strSet  and  str2 is in strSet  and
247      *            OVERLAPS(str1, str2, overlap))
248      *   (overlap <= bestOverlap)
249      * </pre>
250      */
251     private static int bestOverlap(Set<String> strSet, String[] bestTwo) {
252         assert strSet != null : "Violation of: strSet is not null";
253         assert bestTwo != null : "Violation of: bestTwo is not null";
254         assert bestTwo.length >= 2 : "Violation of: bestTwo.length >= 2";
255         /*
256          * Note: Rest of precondition not checked!
257          */
258         int bestOverlap = 0;
259         Set<String> processed = strSet.newInstance();
260         while (strSet.size() > 0) {
261             /*
262              * Remove one string from strSet to check against all others
263              */
264             String str0 = strSet.removeAny();
265             for (String str1 : strSet) {
266                 /*
267                  * Check str0 and str1 for overlap first in one order...
268                  */
269                 int overlapFrom0To1 = overlap(str0, str1);
270                 if (overlapFrom0To1 > bestOverlap) {
271                     /*
272                      * Update best overlap found so far, and the two strings
273                      * that produced it
274                      */
275                     bestOverlap = overlapFrom0To1;
276                     bestTwo[0] = str0;
277                     bestTwo[1] = str1;
278                 }
279                 /*
280                  * ... and then in the other order
281                  */
282                 int overlapFrom1To0 = overlap(str1, str0);
283                 if (overlapFrom1To0 > bestOverlap) {
284                     /*
285                      * Update best overlap found so far, and the two strings
286                      * that produced it
287                      */
288                     bestOverlap = overlapFrom1To0;
289                     bestTwo[0] = str1;
290                     bestTwo[1] = str0;
291                 }
292             }
293             /*
294              * Record that str0 has been checked against every other string in
295              * strSet
296              */
297             processed.add(str0);
298         }
299         /*
300          * Restore strSet and return best overlap
```

```
301          */
302         strSet.transferFrom(processed);
303         return bestOverlap;
304     }
305
306     /**
307      * Combines strings in {@code strSet} as much as possible, leaving in it
308      * only strings that have no overlap between a suffix of one string and a
309      * prefix of another. Note: uses a "greedy approach" to assembly, hence may
310      * not result in {@code strSet} being as small a set as possible at the end.
311      *
312      * @param strSet
313      *            set of strings
314      * @updates strSet
315      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
316      * @ensures <pre>
317      * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet)  and
318      * |strSet| <= |#strSet|  and
319      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
320      * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
321      * </pre>
322      */
323     public static void assemble(Set<String> strSet) {
324         assert strSet != null : "Violation of: strSet is not null";
325         /*
326          * Note: Precondition not checked!
327          */
328         /*
329          * Combine strings as much possible, being greedy
330          */
331         boolean done = false;
332         while ((strSet.size() > 1) && !done) {
333             String[] bestTwo = new String[2];
334             int bestOverlap = bestOverlap(strSet, bestTwo);
335             if (bestOverlap == 0) {
336                 /*
337                  * No overlapping strings remain; can't do any more
338                  */
339                 done = true;
340             } else {
341                 /*
342                  * Replace the two most-overlapping strings with their
343                  * combination; this can be done with add rather than
344                  * addToSetAvoidingSubstrings because the latter would do the
345                  * same thing (this claim requires justification)
346                  */
347                 strSet.remove(bestTwo[0]);
348                 strSet.remove(bestTwo[1]);
349                 String overlapped = combination(bestTwo[0], bestTwo[1],
350                         bestOverlap);
351                 strSet.add(overlapped);
352             }
353         }
354     }
355
356     /**
357      * Prints the string {@code text} to {@code out}, replacing each '~' with a
358      * line separator.
359      *
```

```
360      * @param text
361      *            string to be output
362      * @param out
363      *            output stream
364      * @updates out
365      * @requires out.is_open
366      * @ensures <pre>
367      * out.is_open  and
368      * out.content = #out.content *
369      *   [text with each '~' replaced by line separator]
370      * </pre>
371      */
372     public static void printWithLineSeparators(String text, SimpleWriter out) {
373         assert text != null : "Violation of: text is not null";
374         assert out != null : "Violation of: out is not null";
375         assert out.isOpen() : "Violation of: out.is_open";
376
377         // enters loop that adds a newline if the line separator is present
378         for (int i = 0; i < text.length(); i++) {
379             if (text.charAt(i) == '~') {
380                 out.println();
381             } else {
382                 // prints the normal char if it isn't the line separator
383                 out.print(text.charAt(i));
384             }
385         }
386
387     }
388
389     /**
390      * Given a file name (relative to the path where the application is running)
391      * that contains fragments of a single original source text, one fragment
392      * per line, outputs to stdout the result of trying to reassemble the
393      * original text from those fragments using a "greedy assembler". The
394      * result, if reassembly is complete, might be the original text; but this
395      * might not happen because a greedy assembler can make a mistake and end up
396      * predicting the fragments were from a string other than the true original
397      * source text. It can also end up with two or more fragments that are
398      * mutually non-overlapping, in which case it outputs the remaining
399      * fragments, appropriately labelled.
400      *
401      * @param args
402      *            Command-line arguments: not used
403      */
404     public static void main(String[] args) {
405         SimpleReader in = new SimpleReader1L();
406         SimpleWriter out = new SimpleWriter1L();
407         /*
408          * Get input file name
409          */
410         out.print("Input file (with fragments): ");
411         String inputFileName = in.nextLine();
412         SimpleReader inFile = new SimpleReader1L(inputFileName);
413         /*
414          * Get initial fragments from input file
415          */
416         Set<String> fragments = linesFromInput(inFile);
417         /*
418          * Close inFile; we're done with it
```

```
419          */
420         inFile.close();
421         /*
422          * Assemble fragments as far as possible
423          */
424         assemble(fragments);
425         /*
426          * Output fully assembled text or remaining fragments
427          */
428         if (fragments.size() == 1) {
429             out.println();
430             String text = fragments.removeAny();
431             printWithLineSeparators(text, out);
432         } else {
433             int fragmentNumber = 0;
434             for (String str : fragments) {
435                 fragmentNumber++;
436                 out.println();
437                 out.println("--------------------");
438                 out.println("  -- Fragment #" + fragmentNumber + ": --");
439                 out.println("--------------------");
440                 printWithLineSeparators(str, out);
441             }
442         }
443         /*
444          * Close input and output streams
445          */
446         in.close();
447         out.close();
448     }
449 
450 }
```