```java
import components.set.Set;

/**
 * Utility class to support string reassembly from fragments.
 *
 * @author Put your name here
 *
 * @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>
 */
```

```
 63 public final class StringReassembly {
 64
 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         int intersect = str1.length() - overlap;
139         String subString = str1.substring(0, intersect);
140         str1 = subString + str2;
141
142         return str1;
143     }
144
145     /**
146      * Adds {@code str} to {@code strSet} if and only if it is not a substring
147      * of any string already in {@code strSet}; and if it is added, also removes
148      * from {@code strSet} any string already in {@code strSet} that is a
149      * substring of {@code str}.
150      *
151      * @param strSet
152      *            set to consider adding to
153      * @param str
154      *            string to consider adding
155      * @updates strSet
156      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
157      * @ensures <pre>
158      * if SUPERSTRINGS(#strSet, str) = {}
159      *  then strSet = #strSet union {str} \ SUBSTRINGS(#strSet, str)
160      *  else strSet = #strSet
161      * </pre>
162      */
163     public static void addToSetAvoidingSubstrings(Set<String> strSet,
164             String str) {
165         assert strSet != null : "Violation of: strSet is not null";
166         assert str != null : "Violation of: str is not null";
167
168         boolean found = false;
169
170         for (String x : strSet) {
171             if (x.contains(str)) {
172                 found = true;
173             }
174         }
175
176         if (!found) {
```

```
177             Set<String> temp = strSet.newInstance();
178
179             for (String x : strSet) {
180                 if (!str.contains(x)) {
181                     temp.add(x);
182                 }
183             }
184             temp.add(str);
185             strSet.transferFrom(temp);
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         Set<String> set = new Set1L<>();
209
210         while (!input.atEOS()) {
211             String str = input.nextLine();
212             addToSetAvoidingSubstrings(set, str);
213         }
214
215         return set;
216     }
217
218     /**
219      * Returns the longest overlap between the suffix of one string and the
220      * prefix of another string in {@code strSet}, and identifies the two
221      * strings that achieve that overlap.
222      *
223      * @param strSet
224      *            the set of strings examined
225      * @param bestTwo
226      *            an array containing (upon return) the two strings with the
227      *            largest such overlap between the suffix of {@code bestTwo[0]}
228      *            and the prefix of {@code bestTwo[1]}
229      * @return the amount of overlap between those two strings
230      * @replaces bestTwo[0], bestTwo[1]
231      * @requires <pre>
232      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
233      * bestTwo.length >= 2
```

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

```
291             processed.add(str0);
292         }
293         /*
294          * Restore strSet and return best overlap
295          */
296         strSet.transferFrom(processed);
297         return bestOverlap;
298     }
299
300     /**
301      * Combines strings in {@code strSet} as much as possible, leaving in it
302      * only strings that have no overlap between a suffix of one string and a
303      * prefix of another. Note: uses a "greedy approach" to assembly, hence may
304      * not result in {@code strSet} being as small a set as possible at the end.
305      *
306      * @param strSet
307      *            set of strings
308      * @updates strSet
309      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
310      * @ensures <pre>
311      * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet)  and
312      * |strSet| <= |#strSet|  and
313      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
314      * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
315      * </pre>
316      */
317     public static void assemble(Set<String> strSet) {
318         assert strSet != null : "Violation of: strSet is not null";
319         /*
320          * Note: Precondition not checked!
321          */
322         /*
323          * Combine strings as much possible, being greedy
324          */
325         boolean done = false;
326         while ((strSet.size() > 1) && !done) {
327             String[] bestTwo = new String[2];
328             int bestOverlap = bestOverlap(strSet, bestTwo);
329             if (bestOverlap == 0) {
330                 /*
331                  * No overlapping strings remain; can't do any more
332                  */
333                 done = true;
334             } else {
335                 /*
336                  * Replace the two most-overlapping strings with their
337                  * combination; this can be done with add rather than
338                  * addToSetAvoidingSubstrings because the latter would do the
339                  * same thing (this claim requires justification)
340                  */
341                 strSet.remove(bestTwo[0]);
342                 strSet.remove(bestTwo[1]);
343                 String overlapped = combination(bestTwo[0], bestTwo[1],
344                         bestOverlap);
345                 strSet.add(overlapped);
346             }
347         }
```

```
348     }
349
350     /**
351      * Prints the string {@code text} to {@code out}, replacing each '~' with a
352      * line separator.
353      *
354      * @param text
355      *            string to be output
356      * @param out
357      *            output stream
358      * @updates out
359      * @requires out.is_open
360      * @ensures <pre>
361      * out.is_open  and
362      * out.content = #out.content *
363      *   [text with each '~' replaced by line separator]
364      * </pre>
365      */
366     public static void printWithLineSeparators String text, SimpleWriter out) {
367         assert text != null : "Violation of: text is not null";
368         assert out != null : "Violation of: out is not null";
369         assert out.isOpen() : "Violation of: out.is_open";
370
371         for (int i = 0; i < text.length(); i++) {
372             if (text.charAt(i) == '~') {
373                 out.println();
374
375             } else {
376                 out.print(text.charAt(i));
377             }
378         }
379
380     }
381
382     /**
383      * Given a file name (relative to the path where the application is running)
384      * that contains fragments of a single original source text, one fragment
385      * per line, outputs to stdout the result of trying to reassemble the
386      * original text from those fragments using a "greedy assembler". The
387      * result, if reassembly is complete, might be the original text; but this
388      * might not happen because a greedy assembler can make a mistake and end up
389      * predicting the fragments were from a string other than the true original
390      * source text. It can also end up with two or more fragments that are
391      * mutually non-overlapping, in which case it outputs the remaining
392      * fragments, appropriately labelled.
393      *
394      * @param args
395      *            Command-line arguments: not used
396      */
397     public static void main String[] args) {
398         SimpleReader in = new SimpleReader1L();
399         SimpleWriter out = new SimpleWriter1L();
400         /*
401          * Get input file name
402          */
403         out.print("Input file (with fragments): ");
404         String inputFileName = in.nextLine();
```

```
405         SimpleReader inFile = new SimpleReader1L(inputFileName);
406         /*
407          * Get initial fragments from input file
408          */
409         Set<String> fragments = linesFromInput(inFile);
410         /*
411          * Close inFile; we're done with it
412          */
413         inFile.close();
414         /*
415          * Assemble fragments as far as possible
416          */
417         assemble(fragments);
418         /*
419          * Output fully assembled text or remaining fragments
420          */
421         if (fragments.size() == 1) {
422             out.println();
423             String text = fragments.removeAny();
424             printWithLineSeparators(text, out);
425         } else {
426             int fragmentNumber = 0;
427             for (String str : fragments) {
428                 fragmentNumber++;
429                 out.println();
430                 out.println("--------------------");
431                 out.println("  -- Fragment #" + fragmentNumber + ": --");
432                 out.println("--------------------");
433                 printWithLineSeparators(str, out);
434             }
435         }
436         /*
437          * Close input and output streams
438          */
439         in.close();
440         out.close();
441     }
442
443 }
```