```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>
 */
public final class StringReassembly {

    /**
     * Private no-argument constructor to prevent instantiation of this utility
     * 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         /*
139          * Hint: consider using substring (a String method)
140          */
141 
142         String combine = str1 + str2.substring(overlap);
143 
144         /*
145          * This line added just to make the program compilable. Should be
146          * replaced with appropriate return statement.
147          */
148         return combine;
149     }
150 
151     /**
152      * Adds {@code str} to {@code strSet} if and only if it is not a substring
153      * of any string already in {@code strSet}; and if it is added, also removes
154      * from {@code strSet} any string already in {@code strSet} that is a
155      * substring of {@code str}.
156      *
157      * @param strSet
158      *            set to consider adding to
159      * @param str
160      *            string to consider adding
161      * @updates strSet
162      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
163      * @ensures <pre>
164      * if SUPERSTRINGS(#strSet, str) = {}
165      *  then strSet = #strSet union {str} \ SUBSTRINGS(#strSet, str)
166      *  else strSet = #strSet
167      * </pre>
168      */
169     public static void addToSetAvoidingSubstrings(Set<String> strSet,
170             String str) {
171         assert strSet != null : "Violation of: strSet is not null";
172         assert str != null : "Violation of: str is not null";
173         /*
174          * Note: Precondition not checked!
175          */
176 
177         Set<String> temp = strSet.newInstance();
178         //use include to find whether the strSet contains str or not.
179         boolean include = false;
180         for (String n : strSet) {
181             if (n.contains(str)) {
182                 include = true;
183             }
184         }
185         //construct a new set for the strset
186         if (!include) {
187             while (strSet.size() > 0) {
188                 String word = strSet.removeAny();
189                 if (!str.contains(word)) {
190                     temp.add(word);
191                 }
```

```
192             }
193             temp.add(str);
194             //put the temp to the strSet
195             strSet.transferFrom(temp);
196         }
197         /*
198          * Hint: consider using contains (a String method)
199          */
200
201     }
202
203     /**
204      * Returns the set of all individual lines read from {@code input}, except
205      * that any line that is a substring of another is not in the returned set.
206      *
207      * @param input
208      *            source of strings, one per line
209      * @return set of lines read from {@code input}
210      * @requires input.is_open
211      * @ensures <pre>
212      * input.is_open  and  input.content = <>  and
213      * linesFromInput = [maximal set of lines from #input.content such that
214      *                   CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
215      * </pre>
216      */
217     public static Set<String> linesFromInput(SimpleReader input) {
218         assert input != null : "Violation of: input is not null";
219         assert input.isOpen() : "Violation of: input.is_open";
220         Set<String> sSet = new Set1L<>();
221         //if the input could not be used, the nextLine would stop.
222         while (!input.atEOS()) {
223             String temp = input.nextLine();
224             addToSetAvoidingSubstrings(sSet, temp);
225         }
226         /*
227          * This line added just to make the program compilable. Should be
228          * replaced with appropriate return statement.
229          */
230         return sSet;
231     }
232
233     /**
234      * Returns the longest overlap between the suffix of one string and the
235      * prefix of another string in {@code strSet}, and identifies the two
236      * strings that achieve that overlap.
237      *
238      * @param strSet
239      *            the set of strings examined
240      * @param bestTwo
241      *            an array containing (upon return) the two strings with the
242      *            largest such overlap between the suffix of {@code bestTwo[0]}
243      *            and the prefix of {@code bestTwo[1]}
244      * @return the amount of overlap between those two strings
245      * @replaces bestTwo[0], bestTwo[1]
246      * @requires <pre>
247      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
248      * bestTwo.length >= 2
249      * </pre>
250      * @ensures <pre>
251      * bestTwo[0] is in strSet  and
252      * bestTwo[1] is in strSet  and
253      * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap)  and
```

```
254      * for all str1, str2: string of character, overlap: integer
255      *     where (str1 is in strSet  and  str2 is in strSet  and
256      *            OVERLAPS(str1, str2, overlap))
257      *   (overlap <= bestOverlap)
258      * </pre>
259      */
260     public static int bestOverlap(Set<String> strSet, String[] bestTwo) {
261         assert strSet != null : "Violation of: strSet is not null";
262         assert bestTwo != null : "Violation of: bestTwo is not null";
263         assert bestTwo.length >= 2 : "Violation of: bestTwo.length >= 2";
264         /*
265          * Note: Rest of precondition not checked!
266          */
267         int bestOverlap = 0;
268         Set<String> processed = strSet.newInstance();
269         while (strSet.size() > 0) {
270             /*
271              * Remove one string from strSet to check against all others
272              */
273             String str0 = strSet.removeAny();
274             for (String str1 : strSet) {
275                 /*
276                  * Check str0 and str1 for overlap first in one order...
277                  */
278                 int overlapFrom0To1 = overlap(str0, str1);
279                 if (overlapFrom0To1 > bestOverlap) {
280                     /*
281                      * Update best overlap found so far, and the two strings
282                      * that produced it
283                      */
284                     bestOverlap = overlapFrom0To1;
285                     bestTwo[0] = str0;
286                     bestTwo[1] = str1;
287                 }
288                 /*
289                  * ... and then in the other order
290                  */
291                 int overlapFrom1To0 = overlap(str1, str0);
292                 if (overlapFrom1To0 > bestOverlap) {
293                     /*
294                      * Update best overlap found so far, and the two strings
295                      * that produced it
296                      */
297                     bestOverlap = overlapFrom1To0;
298                     bestTwo[0] = str1;
299                     bestTwo[1] = str0;
300                 }
301             }
302             /*
303              * Record that str0 has been checked against every other string in
304              * strSet
305              */
306             processed.add(str0);
307         }
308         /*
309          * Restore strSet and return best overlap
310          */
311         strSet.transferFrom(processed);
312         return bestOverlap;
313     }
314
315     /**
```

```
316      * Combines strings in {@code strSet} as much as possible, leaving in it
317      * only strings that have no overlap between a suffix of one string and a
318      * prefix of another. Note: uses a "greedy approach" to assembly, hence may
319      * not result in {@code strSet} being as small a set as possible at the end.
320      *
321      * @param strSet
322      *            set of strings
323      * @updates strSet
324      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
325      * @ensures <pre>
326      * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet)  and
327      * |strSet| <= |#strSet|  and
328      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
329      * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
330      * </pre>
331      */
332     public static void assemble(Set<String> strSet) {
333         assert strSet != null : "Violation of: strSet is not null";
334         /*
335          * Note: Precondition not checked!
336          */
337         /*
338          * Combine strings as much possible, being greedy
339          */
340         boolean done = false;
341         while ((strSet.size() > 1) && !done) {
342             String[] bestTwo = new String[2];
343             int bestOverlap = bestOverlap(strSet, bestTwo);
344             if (bestOverlap == 0) {
345                 /*
346                  * No overlapping strings remain; can't do any more
347                  */
348                 done = true;
349             } else {
350                 /*
351                  * Replace the two most-overlapping strings with their
352                  * combination; this can be done with add rather than
353                  * addToSetAvoidingSubstrings because the latter would do the
354                  * same thing (this claim requires justification)
355                  */
356                 strSet.remove(bestTwo[0]);
357                 strSet.remove(bestTwo[1]);
358                 String overlapped = combination(bestTwo[0], bestTwo[1],
359                         bestOverlap);
360                 strSet.add(overlapped);
361             }
362         }
363     }
364
365     /**
366      * Prints the string {@code text} to {@code out}, replacing each '~' with a
367      * line separator.
368      *
369      * @param text
370      *            string to be output
371      * @param out
372      *            output stream
373      * @updates out
374      * @requires out.is_open
375      * @ensures <pre>
376      * out.is_open  and
377      * out.content = #out.content *
```

```
378      *   [text with each '~' replaced by line separator]
379      * </pre>
380      */
381     public static void printWithLineSeparators(String text, SimpleWriter out) {
382         assert text != null : "Violation of: text is not null";
383         assert out != null : "Violation of: out is not null";
384         assert out.isOpen() : "Violation of: out.is_open";
385         //replace the corresponding elements.
386         out.println(text.replaceAll("~", "\n"));
387 
388     }
389 
390     /**
391      * Given a file name (relative to the path where the application is running)
392      * that contains fragments of a single original source text, one fragment
393      * per line, outputs to stdout the result of trying to reassemble the
394      * original text from those fragments using a "greedy assembler". The
395      * result, if reassembly is complete, might be the original text; but this
396      * might not happen because a greedy assembler can make a mistake and end up
397      * predicting the fragments were from a string other than the true original
398      * source text. It can also end up with two or more fragments that are
399      * mutually non-overlapping, in which case it outputs the remaining
400      * fragments, appropriately labelled.
401      *
402      * @param args
403      *            Command-line arguments: not used
404      */
405     public static void main(String[] args) {
406         SimpleReader in = new SimpleReader1L();
407         SimpleWriter out = new SimpleWriter1L();
408         /*
409          * Get input file name
410          */
411         out.print("Input file (with fragments): ");
412         String inputFileName = in.nextLine();
413         SimpleReader inFile = new SimpleReader1L(inputFileName);
414         /*
415          * Get initial fragments from input file
416          */
417         Set<String> fragments = linesFromInput(inFile);
418         /*
419          * Close inFile; we're done with it
420          */
421         inFile.close();
422         /*
423          * Assemble fragments as far as possible
424          */
425         assemble(fragments);
426         /*
427          * Output fully assembled text or remaining fragments
428          */
429         if (fragments.size() == 1) {
430             out.println();
431             String text = fragments.removeAny();
432             printWithLineSeparators(text, out);
433         } else {
434             int fragmentNumber = 0;
435             for (String str : fragments) {
436                 fragmentNumber++;
437                 out.println();
438                 out.println("--------------------");
439                 out.println("  -- Fragment #" + fragmentNumber + ": --");
```

```
440                 out.println("--------------------");
441                 printWithLineSeparators(str, out);
442             }
443         }
444         /*
445          * Close input and output streams
446          */
447         in.close();
448         out.close();
449     }
450 
451 }
452 
```