```java
import java.util.Iterator;

/**
 * Utility class to support string reassembly from fragments.
 *
 * @author Gage Farmer
 *
 * @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 {

```

```
 67     /**
 68      * Private no-argument constructor to prevent instantiation of this utility
 69      * class.
 70      */
 71     private StringReassembly() {
 72     }
 73
 75      * Reports the maximum length of a common suffix of {@code str1} and prefix
 95     public static int overlap(String str1, String str2) {
114
115     /**
116      * Returns concatenation of {@code str1} and {@code str2} from which one of
117      * the two "copies" of the common string of {@code overlap} characters at
118      * the end of {@code str1} and the beginning of {@code str2} has been
119      * removed.
120      *
121      * @param str1
122      *            first string
123      * @param str2
124      *            second string
125      * @param overlap
126      *            amount of overlap
127      * @return combination with one "copy" of overlap removed
128      * @requires OVERLAPS(str1, str2, overlap)
129      * @ensures combination = str1[0, |str1|-overlap) * str2
130      */
131     public static String combination(String str1, String str2, int overlap) {
132         assert str1 != null : "Violation of: str1 is not null";
133         assert str2 != null : "Violation of: str2 is not null";
134         assert 0 <= overlap && overlap <= str1.length()
135                 && overlap <= str2.length()
136                 && str1.regionMatches(str1.length() - overlap, str2, 0,
137                         overlap) : ""
138                                 + "Violation of: OVERLAPS(str1, str2, overlap)";
139         int diff1 = str1.length() - overlap;
140         String combo = "";
141
142         combo = str1.substring(0, diff1);
143         combo = combo + str2;
144
145         return combo;
146     }
147
149      * Adds {@code str} to {@code strSet} if and only if it is not a substring
166     public static void addToSetAvoidingSubstrings(Set<String> strSet,
196
198      * Returns the set of all individual lines read from {@code input}, except
211     public static Set<String> linesFromInput(SimpleReader input) {
212         assert input != null : "Violation of: input is not null";
213         assert input.isOpen() : "Violation of: input.is_open";
214
215         /*
216          * I don't know if I misinterpreted the contract or not, but something
217          * doesn't feel right with this.
218          */
219
220         Set<String> set = new Set1L<>();
221         boolean moreFile = true;
222         String nextString = "";
```

```
223         try {
224             nextString = input.nextLine();
225         } catch (AssertionError e) {
226             moreFile = false;
227         }
228 
229         while (moreFile) {
230             addToSetAvoidingSubstrings(set, nextString);
231             try {
232                 nextString = input.nextLine();
233             } catch (AssertionError e) {
234                 moreFile = false;
235             }
236         }
237 
238         return set;
239     }
240 
242      * Returns the longest overlap between the suffix of one string and the
268     private static int bestOverlap(Set<String> strSet, String[] bestTwo) {
322 
323     /**
324      * Combines strings in {@code strSet} as much as possible, leaving in it
325      * only strings that have no overlap between a suffix of one string and a
326      * prefix of another. Note: uses a "greedy approach" to assembly, hence may
327      * not result in {@code strSet} being as small a set as possible at the end.
328      *
329      * @param strSet
330      *            set of strings
331      * @updates strSet
332      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
333      * @ensures <pre>
334      * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet)  and
335      * |strSet| <= |#strSet|  and
336      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
337      * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
338      * </pre>
339      */
340     public static void assemble(Set<String> strSet) {
341         assert strSet != null : "Violation of: strSet is not null";
342         /*
343          * Note: Precondition not checked!
344          */
345         /*
346          * Combine strings as much possible, being greedy
347          */
348         boolean done = false;
349         while ((strSet.size() > 1) && !done) {
350             String[] bestTwo = new String[2];
351             int bestOverlap = bestOverlap(strSet, bestTwo);
352             if (bestOverlap == 0) {
353                 /*
354                  * No overlapping strings remain; can't do any more
355                  */
356                 done = true;
357             } else {
358                 /*
359                  * Replace the two most-overlapping strings with their
360                  * combination; this can be done with add rather than
```

```
361                  * addToSetAvoidingSubstrings because the latter would do the
362                  * same thing (this claim requires justification)
363                  */
364                 strSet.remove(bestTwo[0]);
365                 strSet.remove(bestTwo[1]);
366                 String overlapped = combination(bestTwo[0], bestTwo[1],
367                         bestOverlap);
368                 strSet.add(overlapped);
369             }
370         }
371     }
372
373     /**
374      * Prints the string {@code text} to {@code out}, replacing each '~' with a
375      * line separator.
376      *
377      * @param text
378      *            string to be output
379      * @param out
380      *            output stream
381      * @updates out
382      * @requires out.is_open
383      * @ensures <pre>
384      * out.is_open  and
385      * out.content = #out.content *
386      *   [text with each '~' replaced by line separator]
387      * </pre>
388      */
389     public static void printWithLineSeparators(String text, SimpleWriter out) {
390         assert text != null : "Violation of: text is not null";
391         assert out != null : "Violation of: out is not null";
392         assert out.isOpen() : "Violation of: out.is_open";
393
394         String accum = "";
395         int i = 0;
396
397         while (i < text.length()) {
398             if (text.charAt(i) == '~') {
399                 out.println(accum);
400                 accum = "";
401             } else {
402                 accum = accum + text.charAt(i);
403             }
404             i++;
405         }
406
407     }
408
409     /**
410      * Generates the set of characters in the given {@code String} into the
411      * given {@code Set}.
412      *
413      * @param str
414      *            the given {@code String}
415      * @param set
416      *            the {@code Set} to be replaced
417      * @replaces set
418      * @ensures set = entries(str)
419      */
```

```
420     private static void generateSet(String str, Set<String> set) {
421         int i = 0;
422         String accum = "";
423 
424         while (i < str.length() - 1) {
425             if (str.charAt(i) == '~') {
426                 set.add(accum);
427                 accum = "";
428             } else {
429                 accum = accum + str.charAt(i);
430             }
431             i++;
432         }
433     }
434 
436      * Returns the first "word" (maximal length string of characters not in
467     private static String nextWordOrSeparator(String text, int position,
485 
486     /**
487      * Given a file name (relative to the path where the application is running)
488      * that contains fragments of a single original source text, one fragment
489      * per line, outputs to stdout the result of trying to reassemble the
490      * original text from those fragments using a "greedy assembler". The
491      * result, if reassembly is complete, might be the original text; but this
492      * might not happen because a greedy assembler can make a mistake and end up
493      * predicting the fragments were from a string other than the true original
494      * source text. It can also end up with two or more fragments that are
495      * mutually non-overlapping, in which case it outputs the remaining
496      * fragments, appropriately labelled.
497      *
498      * @param args
499      *            Command-line arguments: not used
500      */
501     public static void main(String[] args) {
502         SimpleReader in = new SimpleReader1L();
503         SimpleWriter out = new SimpleWriter1L();
504         /*
505          * Get input file name
506          */
507         out.print("Input file (with fragments): ");
508         String inputFileName = in.nextLine();
509         SimpleReader inFile = new SimpleReader1L(inputFileName);
510         /*
511          * Get initial fragments from input file
512          */
513         Set<String> fragments = linesFromInput(inFile);
514         /*
515          * Close inFile; we're done with it
516          */
517         inFile.close();
518         /*
519          * Assemble fragments as far as possible
520          */
521         assemble(fragments);
522         /*
523          * Output fully assembled text or remaining fragments
524          */
525         if (fragments.size() == 1) {
526             out.println();
```

```
527             String text = fragments.removeAny();
528             printWithLineSeparators(text, out);
529         } else {
530             int fragmentNumber = 0;
531             for (String str : fragments) {
532                 fragmentNumber++;
533                 out.println();
534                 out.println("--------------------");
535                 out.println("  -- Fragment #" + fragmentNumber + ": --");
536                 out.println("--------------------");
537                 printWithLineSeparators(str, out);
538             }
539         }
540         /*
541          * Close input and output streams
542          */
543         in.close();
544         out.close();
545     }
546
547 }
```