```java
import components.set.Set;
import components.set.Set1L;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;

/**
 * Utility class to support string reassembly from fragments.
 *
 * @author Robert Frenken
 *
 * @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.
     */
    private StringReassembly() {
    }

    /**
     * Reports the maximum length of a common suffix of {@code str1} and prefix
     * of {@code str2}.
     *
     * @param str1
     *            first string
     * @param str2
     *            second string
     * @return maximum overlap between right end of {@code str1} and left end of
     *         {@code str2}
     * @requires <pre>
     * str1 is not substring of str2  and
     * str2 is not substring of str1
     * </pre>
     * @ensures <pre>
     * OVERLAPS(str1, str2, overlap)  and
     * for all k: integer
     *     where (overlap < k  and  k <= |str1|  and  k <= |str2|)
     *  (not OVERLAPS(str1, str2, k))
     * </pre>
     */
    public static int overlap(String str1, String str2) {
        assert str1 != null : "Violation of: str1 is not null";
        assert str2 != null : "Violation of: str2 is not null";
        assert str2.indexOf(str1) < 0 : "Violation of: "
                + "str1 is not substring of str2";
        assert str1.indexOf(str2) < 0 : "Violation of: "
                + "str2 is not substring of str1";
        /*
         * Start with maximum possible overlap and work down until a match is
         * found; think about it and try it on some examples to see why
         * iterating in the other direction doesn't work
         */
        int maxOverlap = str2.length() - 1;
        while (!str1.regionMatches(str1.length() - maxOverlap, str2, 0,
                maxOverlap)) {
            maxOverlap--;
        }
        return maxOverlap;
    }

    /**
     * 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         // start with the start of string 2, and the end of string 1
139         int i = str1.length() - overlap;
140         // take all of string one, and the difference of string 2 and the overlap attached to
    the end
141         String combination = str1.substring(0, i) + str2;
142         return combination;
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          * Note: Precondition not checked!
169          */
170         boolean isSubstring = false;
```

```
171         Set<String> newSet = new Set1L<>();
172         for (String x : strSet) {
173             if (x.indexOf(str) != -1) {
174                 isSubstring = true;
175             }
176         }
177         if (!isSubstring) {
178 
179             // loop whole set except for last entry, which was new string, and remove
    substrings of addition
180             while (strSet.size() != 0) {
181                 String single = strSet.removeAny();
182                 if (str.indexOf(single) == -1) {
183                     newSet.add(single); // add if not a substring
184                 }
185             }
186             strSet.add(str); // add string
187             strSet.add(newSet); // add entries that are not substring
188         }
189     }
190 
191     /**
192      * Returns the set of all individual lines read from {@code input}, except
193      * that any line that is a substring of another is not in the returned set.
194      *
195      * @param input
196      *            source of strings, one per line
197      * @return set of lines read from {@code input}
198      * @requires input.is_open
199      * @ensures <pre>
200      * input.is_open  and  input.content = <>  and
201      * linesFromInput = [maximal set of lines from #input.content such that
202      *                   CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
203      * </pre>
204      */
205     public static Set<String> linesFromInput(SimpleReader input) {
206         assert input != null : "Violation of: input is not null";
207         assert input.isOpen() : "Violation of: input.is_open";
208 
209         Set<String> set = new Set1L<>();
210         // add all lines to the set
211         while (!input.atEOS()) {
212             String str = input.nextLine();
213             addToSetAvoidingSubstrings(set, str);
214         }
215 
216         return set;
217     }
218 
219     /**
220      * Returns the longest overlap between the suffix of one string and the
221      * prefix of another string in {@code strSet}, and identifies the two
222      * strings that achieve that overlap.
223      *
224      * @param strSet
225      *            the set of strings examined
226      * @param bestTwo
```

```
227      *            an array containing (upon return) the two strings with the
228      *            largest such overlap between the suffix of {@code bestTwo[0]}
229      *            and the prefix of {@code bestTwo[1]}
230      * @return the amount of overlap between those two strings
231      * @replaces bestTwo[0], bestTwo[1]
232      * @requires <pre>
233      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
234      * bestTwo.length >= 2
235      * </pre>
236      * @ensures <pre>
237      * bestTwo[0] is in strSet  and
238      * bestTwo[1] is in strSet  and
239      * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap)  and
240      * for all str1, str2: string of character, overlap: integer
241      *     where (str1 is in strSet  and  str2 is in strSet  and
242      *            OVERLAPS(str1, str2, overlap))
243      *   (overlap <= bestOverlap)
244      * </pre>
245      */
246     private static int bestOverlap(Set<String> strSet, String[] bestTwo) {
247         assert strSet != null : "Violation of: strSet is not null";
248         assert bestTwo != null : "Violation of: bestTwo is not null";
249         assert bestTwo.length >= 2 : "Violation of: bestTwo.length >= 2";
250         /*
251          * Note: Rest of precondition not checked!
252          */
253         int bestOverlap = 0;
254         Set<String> processed = strSet.newInstance();
255         while (strSet.size() > 0) {
256             /*
257              * Remove one string from strSet to check against all others
258              */
259             String str0 = strSet.removeAny();
260             for (String str1 : strSet) {
261                 /*
262                  * Check str0 and str1 for overlap first in one order...
263                  */
264                 int overlapFrom0To1 = overlap(str0, str1);
265                 if (overlapFrom0To1 > bestOverlap) {
266                     /*
267                      * Update best overlap found so far, and the two strings
268                      * that produced it
269                      */
270                     bestOverlap = overlapFrom0To1;
271                     bestTwo[0] = str0;
272                     bestTwo[1] = str1;
273                 }
274                 /*
275                  * ... and then in the other order
276                  */
277                 int overlapFrom1To0 = overlap(str1, str0);
278                 if (overlapFrom1To0 > bestOverlap) {
279                     /*
280                      * Update best overlap found so far, and the two strings
281                      * that produced it
282                      */
283                     bestOverlap = overlapFrom1To0;
```

```
284                     bestTwo[0] = str1;
285                     bestTwo[1] = str0;
286                 }
287             }
288             /*
289              * Record that str0 has been checked against every other string in
290              * strSet
291              */
292             processed.add(str0);
293         }
294         /*
295          * Restore strSet and return best overlap
296          */
297         strSet.transferFrom(processed);
298         return bestOverlap;
299     }
300
301     /**
302      * Combines strings in {@code strSet} as much as possible, leaving in it
303      * only strings that have no overlap between a suffix of one string and a
304      * prefix of another. Note: uses a "greedy approach" to assembly, hence may
305      * not result in {@code strSet} being as small a set as possible at the end.
306      *
307      * @param strSet
308      *            set of strings
309      * @updates strSet
310      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
311      * @ensures <pre>
312      * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet)  and
313      * |strSet| <= |#strSet|  and
314      * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
315      * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
316      * </pre>
317      */
318     public static void assemble(Set<String> strSet) {
319         assert strSet != null : "Violation of: strSet is not null";
320         /*
321          * Note: Precondition not checked!
322          */
323         /*
324          * Combine strings as much possible, being greedy
325          */
326         boolean done = false;
327         while ((strSet.size() > 1) && !done) {
328             String[] bestTwo = new String[2];
329             int bestOverlap = bestOverlap(strSet, bestTwo);
330             if (bestOverlap == 0) {
331                 /*
332                  * No overlapping strings remain; can't do any more
333                  */
334                 done = true;
335             } else {
336                 /*
337                  * Replace the two most-overlapping strings with their
338                  * combination; this can be done with add rather than
339                  * addToSetAvoidingSubstrings because the latter would do the
340                  * same thing (this claim requires justification)
```

```
341                  */
342                 strSet.remove(bestTwo[0]);
343                 strSet.remove(bestTwo[1]);
344                 String overlapped = combination(bestTwo[0], bestTwo[1],
345                         bestOverlap);
346                 strSet.add(overlapped);
347             }
348         }
349     }
350
351     /**
352      * Prints the string {@code text} to {@code out}, replacing each '~' with a
353      * line separator.
354      *
355      * @param text
356      *            string to be output
357      * @param out
358      *            output stream
359      * @updates out
360      * @requires out.is_open
361      * @ensures <pre>
362      * out.is_open  and
363      * out.content = #out.content *
364      *   [text with each '~' replaced by line separator]
365      * </pre>
366      */
367     public static void printWithLineSeparators(String text, SimpleWriter out) {
368         assert text != null : "Violation of: text is not null";
369         assert out != null : "Violation of: out is not null";
370         assert out.isOpen() : "Violation of: out.is_open";
371
372         for (int i = 0; i < text.length(); i++) {
373             char c = text.charAt(i);
374             if (c != '~') {
375                 out.print(c);
376             } else {
377                 out.println();
378             }
379         }
380
381     }
382
383     /**
384      * Given a file name (relative to the path where the application is running)
385      * that contains fragments of a single original source text, one fragment
386      * per line, outputs to stdout the result of trying to reassemble the
387      * original text from those fragments using a "greedy assembler". The
388      * result, if reassembly is complete, might be the original text; but this
389      * might not happen because a greedy assembler can make a mistake and end up
390      * predicting the fragments were from a string other than the true original
391      * source text. It can also end up with two or more fragments that are
392      * mutually non-overlapping, in which case it outputs the remaining
393      * fragments, appropriately labelled.
394      *
395      * @param args
396      *            Command-line arguments: not used
397      */
```

```
398     public static void main(String[] args) {
399         SimpleReader in = new SimpleReader1L();
400         SimpleWriter out = new SimpleWriter1L();
401         /*
402          * Get input file name
403          */
404         out.print("Input file (with fragments): ");
405         String inputFileName = in.nextLine();
406         SimpleReader inFile = new SimpleReader1L(inputFileName);
407         /*
408          * Get initial fragments from input file
409          */
410         Set<String> fragments = linesFromInput(inFile);
411         /*
412          * Close inFile; we're done with it
413          */
414         inFile.close();
415         /*
416          * Assemble fragments as far as possible
417          */
418         assemble(fragments);
419         /*
420          * Output fully assembled text or remaining fragments
421          */
422         if (fragments.size() == 1) {
423             out.println();
424             String text = fragments.removeAny();
425             printWithLineSeparators(text, out);
426         } else {
427             int fragmentNumber = 0;
428             for (String str : fragments) {
429                 fragmentNumber++;
430                 out.println();
431                 out.println("--------------------");
432                 out.println("  -- Fragment #" + fragmentNumber + ": --");
433                 out.println("--------------------");
434                 printWithLineSeparators(str, out);
435             }
436         }
437         /*
438          * Close input and output streams
439          */
440         in.close();
441         out.close();
442     }
443
444 }
```