

```

1 import components.set.Set;
2
3 /**
4  * Utility class to support string reassembly from fragments.
5  *
6  * @author Put your name here
7  *
8  * @mathdefinitions <pre>
9  *
10 * OVERLAPS (
11 *   s1: string of character,
12 *   s2: string of character,
13 *   k: integer
14 * ) : boolean is
15 *  $\emptyset \leq k \text{ and } k \leq |s1| \text{ and } k \leq |s2| \text{ and }$ 
16 *  $s1[|s1|-k, |s1|) = s2[0, k)$ 
17 *
18 * SUBSTRINGS (
19 *   strSet: finite set of string of character,
20 *   s: string of character
21 * ) : finite set of string of character is
22 * {t: string of character
23 *   where (t is in strSet and t is substring of s)
24 *   (t)}
25 *
26 * SUPERSTRINGS (
27 *   strSet: finite set of string of character,
28 *   s: string of character
29 * ) : finite set of string of character is
30 * {t: string of character
31 *   where (t is in strSet and s is substring of t)
32 *   (t)}
33 *
34 * CONTAINS_NO_SUBSTRING_PAIRS (
35 *   strSet: finite set of string of character
36 * ) : boolean is
37 * for all t: string of character
38 *   where (t is in strSet)
39 *   (SUBSTRINGS(strSet \ {t}, t) = {})
40 *
41 * ALL_SUPERSTRINGS (
42 *   strSet: finite set of string of character
43 * ) : set of string of character is
44 * {t: string of character
45 *   where (SUBSTRINGS(strSet, t) = strSet)
46 *   (t)}
47 *
48 * CONTAINS_NO_OVERLAPPING_PAIRS (
49 *   strSet: finite set of string of character
50 * ) : boolean is
51 * for all t1, t2: string of character, k: integer
52 *   where (t1  $\neq$  t2 and t1 is in strSet and t2 is in strSet and
53 *          $1 \leq k \text{ and } k \leq |s1| \text{ and } k \leq |s2|$ )
54 *   (not OVERLAPS(s1, s2, k))
55 *
56 * </pre>
57 */

```

```

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         /**
102          * Start with maximum possible overlap and work down until a match is
103          * found; think about it and try it on some examples to see why
104          * iterating in the other direction doesn't work
105          */
106         int maxOverlap = str2.length() - 1;
107         while (!str1.regionMatches(str1.length() - maxOverlap, str2, 0,
108             maxOverlap)) {
109             maxOverlap--;
110         }
111         return maxOverlap;
112     }
113
114     /**
115      * Returns concatenation of {@code str1} and {@code str2} from which one of
116      * the two "copies" of the common string of {@code overlap} characters at
117      * the end of {@code str1} and the beginning of {@code str2} has been
118      * removed.
119      *
120      * @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  * Returns the set of all individual lines read from {@code input}, except
191  * that any line that is a substring of another is not in the returned set.
192  *
193  * @param input
194  *     source of strings, one per line
195  * @return set of lines read from {@code input}
196  * @requires input.is_open
197  * @ensures <pre>
198  *     input.is_open and input.content = <> and
199  *     linesFromInput = [maximal set of lines from #input.content such that
200  *         CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
201  * </pre>
202  */
203 public static Set<String> linesFromInput (SimpleReader input) {
204     assert input != null : "Violation of: input is not null";
205     assert input.isOpen() : "Violation of: input.is_open";
206
207     Set<String> set = new Set1L<>();
208
209     while (!input.atEOS()) {
210         String str = input.nextLine();
211         addToSetAvoidingSubstrings(set, str);
212     }
213
214     return set;
215 }
216
217 /**
218  * Returns the longest overlap between the suffix of one string and the
219  * prefix of another string in {@code strSet}, and identifies the two
220  * strings that achieve that overlap.
221  *
222  * @param strSet
223  *     the set of strings examined
224  * @param bestTwo
225  *     an array containing (upon return) the two strings with the
226  *     largest such overlap between the suffix of {@code bestTwo[0]}
227  *     and the prefix of {@code bestTwo[1]}
228  * @return the amount of overlap between those two strings
229  * @replaces bestTwo[0], bestTwo[1]
230  * @requires <pre>
231  *     CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
232  *     bestTwo.length >= 2
233  * </pre>

```

```

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

```

```

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

```

```

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

```

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