

```

1 import components.set.Set;
2 import components.set.Set1L;
3 import components.simplereader.SimpleReader;
4 import components.simplereader.SimpleReader1L;
5 import components.simplewriter.SimpleWriter;
6 import components.simplewriter.SimpleWriter1L;
7
8 /**
9  * Utility class to support string reassembly from fragments.
10  *
11  * @author David Park
12  *
13  * @mathdefinitions <pre>
14  *
15  * OVERLAPS (
16  *   s1: string of character,
17  *   s2: string of character,
18  *   k: integer
19  * ) : boolean is
20  *  $0 \leq k$  and  $k \leq |s1|$  and  $k \leq |s2|$  and
21  *  $s1[|s1|-k, |s1|) = s2[0, k)$ 
22  *
23  * SUBSTRINGS (
24  *   strSet: finite set of string of character,
25  *   s: string of character
26  * ) : finite set of string of character is
27  * {t: string of character
28  *   where (t is in strSet and t is substring of s)
29  *   (t)}
30  *
31  * SUPERSTRINGS (
32  *   strSet: finite set of string of character,
33  *   s: string of character
34  * ) : finite set of string of character is
35  * {t: string of character
36  *   where (t is in strSet and s is substring of t)
37  *   (t)}
38  *
39  * CONTAINS_NO_SUBSTRING_PAIRS (
40  *   strSet: finite set of string of character
41  * ) : boolean is
42  * for all t: string of character
43  *   where (t is in strSet)
44  *   (SUBSTRINGS(strSet \ {t}, t) = {})
45  *
46  * ALL_SUPERSTRINGS (
47  *   strSet: finite set of string of character
48  * ) : set of string of character is
49  * {t: string of character
50  *   where (SUBSTRINGS(strSet, t) = strSet)
51  *   (t)}
52  *
53  * CONTAINS_NO_OVERLAPPING_PAIRS (
54  *   strSet: finite set of string of character
55  * ) : boolean is
56  * for all t1, t2: string of character, k: integer
57  *   where (t1  $\neq$  t2 and t1 is in strSet and t2 is in strSet and

```

```

58 *      1 <= k and k <= |s1| and k <= |s2|)
59 *      (not OVERLAPS(s1, s2, k))
60 *
61 * </pre>
62 */
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         /*
139          * Hint: consider using substring (a String method)
140          */
141
142         String x = str1.substring(0, str1.length() - overlap);
143         x = x.concat(str2);
144
145         /*
146          * This line added just to make the program compilable. Should be
147          * replaced with appropriate return statement.
148          */
149         return x;
150     }
151
152     /**
153     * Adds {@code str} to {@code strSet} if and only if it is not a substring
154     * of any string already in {@code strSet}; and if it is added, also removes
155     * from {@code strSet} any string already in {@code strSet} that is a
156     * substring of {@code str}.
157     *
158     * @param strSet
159     *         set to consider adding to
160     * @param str
161     *         string to consider adding
162     * @updates strSet
163     * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
164     * @ensures <pre>
165     * if SUPERSTRINGS(#strSet, str) = {}
166     * then strSet = #strSet union {str} \ SUBSTRINGS(#strSet, str)
167     * else strSet = #strSet
168     * </pre>
169     */
170     public static void addToSetAvoidingSubstrings(Set<String> strSet,
171         String str) {

```

```

172     assert strSet != null : "Violation of: strSet is not null";
173     assert str != null : "Violation of: str is not null";
174     /*
175      * Note: Precondition not checked!
176      */
177
178     /*
179      * Hint: consider using contains (a String method)
180      */
181
182     boolean isSubstring = false;
183     Set<String> substringsToRemove = new Set1L<>();
184
185     for (String existingStr : strSet) {
186         if (str.contains(existingStr)) {
187             // If str is a superstring of an existing string, then remove
188             substringsToRemove.add(existingStr);
189         } else if (existingStr.contains(str)) {
190             // If str is a substring of an existing string, don't add str.
191             isSubstring = true;
192             break;
193         }
194     }
195
196     // Remove substrings from the set.
197     for (String toRemove : substringsToRemove) {
198         strSet.remove(toRemove);
199     }
200
201     // Add the new string if it's not a substring of any existing string.
202     if (!isSubstring) {
203         strSet.add(str);
204     }
205 }
206
207 /**
208  * Returns the set of all individual lines read from {@code input}, except
209  * that any line that is a substring of another is not in the returned set.
210  *
211  * @param input
212  *         source of strings, one per line
213  * @return set of lines read from {@code input}
214  * @requires input.is_open
215  * @ensures <pre>
216  * input.is_open and input.content = <> and
217  * linesFromInput = [maximal set of lines from #input.content such that
218  *                     CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
219  * </pre>
220  */
221 public static Set<String> linesFromInput(SimpleReader input) {
222     assert input != null : "Violation of: input is not null";
223     assert input.isOpen() : "Violation of: input.is_open";
224
225     Set<String> temp = new Set1L<>();
226     String x = "";
227     while (!input.atEOS()) {
228         x = input.nextLine();

```

```

229         addToSetAvoidingSubstrings(temp, x);
230     }
231
232     /*
233     * This line added just to make the program compilable. Should be
234     * replaced with appropriate return statement.
235     */
236     return temp;
237 }
238
239 /**
240  * Returns the longest overlap between the suffix of one string and the
241  * prefix of another string in {@code strSet}, and identifies the two
242  * strings that achieve that overlap.
243  *
244  * @param strSet
245  *     the set of strings examined
246  * @param bestTwo
247  *     an array containing (upon return) the two strings with the
248  *     largest such overlap between the suffix of {@code bestTwo[0]}
249  *     and the prefix of {@code bestTwo[1]}
250  * @return the amount of overlap between those two strings
251  * @replaces bestTwo[0], bestTwo[1]
252  * @requires <pre>
253  * CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
254  * bestTwo.length >= 2
255  * </pre>
256  * @ensures <pre>
257  * bestTwo[0] is in strSet and
258  * bestTwo[1] is in strSet and
259  * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap) and
260  * for all str1, str2: string of character, overlap: integer
261  *     where (str1 is in strSet and str2 is in strSet and
262  *         OVERLAPS(str1, str2, overlap))
263  *     (overlap <= bestOverlap)
264  * </pre>
265  */
266 private static int bestOverlap(Set<String> strSet, String[] bestTwo) {
267     assert strSet != null : "Violation of: strSet is not null";
268     assert bestTwo != null : "Violation of: bestTwo is not null";
269     assert bestTwo.length >= 2 : "Violation of: bestTwo.length >= 2";
270     /*
271     * Note: Rest of precondition not checked!
272     */
273     int bestOverlap = 0;
274     Set<String> processed = strSet.newInstance();
275     while (strSet.size() > 0) {
276         /*
277         * Remove one string from strSet to check against all others
278         */
279         String str0 = strSet.removeAny();
280         for (String str1 : strSet) {
281             /*
282             * Check str0 and str1 for overlap first in one order...
283             */
284             int overlapFrom0To1 = overlap(str0, str1);
285             if (overlapFrom0To1 > bestOverlap) {

```

```

286         /*
287         * Update best overlap found so far, and the two strings
288         * that produced it
289         */
290         bestOverlap = overlapFrom0To1;
291         bestTwo[0] = str0;
292         bestTwo[1] = str1;
293     }
294     /*
295     * ... and then in the other order
296     */
297     int overlapFrom1To0 = overlap(str1, str0);
298     if (overlapFrom1To0 > bestOverlap) {
299         /*
300         * Update best overlap found so far, and the two strings
301         * that produced it
302         */
303         bestOverlap = overlapFrom1To0;
304         bestTwo[0] = str1;
305         bestTwo[1] = str0;
306     }
307 }
308 /*
309 * Record that str0 has been checked against every other string in
310 * strSet
311 */
312 processed.add(str0);
313 }
314 /*
315 * Restore strSet and return best overlap
316 */
317 strSet.transferFrom(processed);
318 return bestOverlap;
319 }
320
321 /**
322 * Combines strings in {@code strSet} as much as possible, leaving in it
323 * only strings that have no overlap between a suffix of one string and a
324 * prefix of another. Note: uses a "greedy approach" to assembly, hence may
325 * not result in {@code strSet} being as small a set as possible at the end.
326 *
327 * @param strSet
328 *      set of strings
329 * @updates strSet
330 * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
331 * @ensures <pre>
332 * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet) and
333 * |strSet| <= |#strSet| and
334 * CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
335 * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
336 * </pre>
337 */
338 public static void assemble(Set<String> strSet) {
339     assert strSet != null : "Violation of: strSet is not null";
340     /*
341     * Note: Precondition not checked!
342     */

```

```

343     /*
344     * Combine strings as much possible, being greedy
345     */
346     boolean done = false;
347     while ((strSet.size() > 1) && !done) {
348         String[] bestTwo = new String[2];
349         int bestOverlap = bestOverlap(strSet, bestTwo);
350         if (bestOverlap == 0) {
351             /*
352             * No overlapping strings remain; can't do any more
353             */
354             done = true;
355         } else {
356             /*
357             * Replace the two most-overlapping strings with their
358             * combination; this can be done with add rather than
359             * addToSetAvoidingSubstrings because the latter would do the
360             * same thing (this claim requires justification)
361             */
362             strSet.remove(bestTwo[0]);
363             strSet.remove(bestTwo[1]);
364             String overlapped = combination(bestTwo[0], bestTwo[1],
365                 bestOverlap);
366             strSet.add(overlapped);
367         }
368     }
369 }
370
371 /**
372  * Prints the string {@code text} to {@code out}, replacing each '~' with a
373  * line separator.
374  *
375  * @param text
376  *         string to be output
377  * @param out
378  *         output stream
379  * @updates out
380  * @requires out.is_open
381  * @ensures <pre>
382  * out.is_open  and
383  * out.content = #out.content *
384  * [text with each '~' replaced by line separator]
385  * </pre>
386  */
387 public static void printWithLineSeparators(String text, SimpleWriter out) {
388     assert text != null : "Violation of: text is not null";
389     assert out != null : "Violation of: out is not null";
390     assert out.isOpen() : "Violation of: out.is_open";
391
392     for (int i = 0; i < text.length(); i++) {
393         if (text.charAt(i) == '~') {
394             out.println();
395         } else {
396             out.print(text.charAt(i));
397         }
398     }
399 }

```

```

400     }
401
402     /**
403      * Given a file name (relative to the path where the application is running)
404      * that contains fragments of a single original source text, one fragment
405      * per line, outputs to stdout the result of trying to reassemble the
406      * original text from those fragments using a "greedy assembler". The
407      * result, if reassembly is complete, might be the original text; but this
408      * might not happen because a greedy assembler can make a mistake and end up
409      * predicting the fragments were from a string other than the true original
410      * source text. It can also end up with two or more fragments that are
411      * mutually non-overlapping, in which case it outputs the remaining
412      * fragments, appropriately labelled.
413      *
414      * @param args
415      *         Command-line arguments: not used
416      */
417     public static void main(String[] args) {
418         SimpleReader in = new SimpleReader1L();
419         SimpleWriter out = new SimpleWriter1L();
420         /*
421          * Get input file name
422          */
423         out.print("Input file (with fragments): ");
424         String inputFileName = in.nextLine();
425         SimpleReader inFile = new SimpleReader1L(inputFileName);
426         /*
427          * Get initial fragments from input file
428          */
429         Set<String> fragments = LinesFromInput(inFile);
430         /*
431          * Close inFile; we're done with it
432          */
433         inFile.close();
434         /*
435          * Assemble fragments as far as possible
436          */
437         assemble(fragments);
438         /*
439          * Output fully assembled text or remaining fragments
440          */
441         if (fragments.size() == 1) {
442             out.println();
443             String text = fragments.removeAny();
444             printWithLineSeparators(text, out);
445         } else {
446             int fragmentNumber = 0;
447             for (String str : fragments) {
448                 fragmentNumber++;
449                 out.println();
450                 out.println("-----");
451                 out.println("  -- Fragment #" + fragmentNumber + ": --");
452                 out.println("-----");
453                 printWithLineSeparators(str, out);
454             }
455         }
456         /*

```



```
457         * Close input and output streams
458         */
459         in.close();
460         out.close();
461     }
462
463 }
```