

```
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 Jeng Zhuang
12  *
13  * @mathdefinitions <pre>
14  *
15  * OVERLAPS (
16  *   s1: string of character,
17  *   s2: string of character,
18  *   k: integer
19  * ) : boolean is
20  *  $0 \leq k \text{ and } k \leq |s1| \text{ and } k \leq |s2| \text{ and }$ 
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 /= 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
first string
77      *

```

```
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
97             : "Violation of: " + "str1 is not substring of
    str2";
98         assert str1.indexOf(str2) < 0
99             : "Violation of: " + "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, maxOverlap)) {
107            maxOverlap--;
108        }
109        return maxOverlap;
110    }
111
```

```
112     /**
113      * Returns concatenation of {@code str1} and {@code str2} from
114      * which one of
115      * the two "copies" of the common string of {@code overlap}
116      * characters at
117      * the end of {@code str1} and the beginning of {@code str2}
118      * 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
132     overlap) {
133         assert str1 != null : "Violation of: str1 is not null";
134         assert str2 != null : "Violation of: str2 is not null";
135         assert 0 <= overlap && overlap <= str1.length() && overlap
136         <= str2.length()
137         && str1.regionMatches(str1.length() - overlap,
138         str2, 0, overlap)
139         : "" + "Violation of: OVERLAPS(str1, str2,
140         overlap)";
141         String combination;
142         // Combine str1 and str2 with overlap removed from str1
143         combination = str1.substring(0, str1.length() - overlap) +
144         str2;
145         return combination;
146     }
147     /**
148      * Adds {@code str} to {@code strSet} if and only if it is not
```

```
    a substring
144     * of any string already in {@code strSet}; and if it is added,
    also removes
145     * from {@code strSet} any string already in {@code strSet}
    that is a
146     * substring of {@code str}.
147     *
148     * @param strSet
149     *         set to consider adding to
150     * @param str
151     *         string to consider adding
152     * @updates strSet
153     * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
154     * @ensures <pre>
155     * if SUPERSTRINGS(#strSet, str) = {}
156     * then strSet = #strSet union {str} \ SUBSTRINGS(#strSet,
str)
157     * else strSet = #strSet
158     * </pre>
159     */
160     public static void addToSetAvoidingSubstrings(Set<String>
strSet, String str) {
161         assert strSet != null : "Violation of: strSet is not null";
162         assert str != null : "Violation of: str is not null";
163         /*
164          * Note: Precondition not checked!
165          */
166
167         /*
168          * Hint: consider using contains (a String method)
169          */
170
171         // Check if str is a substring of any existing element
172         boolean isSubstring = false;
173         for (String s : strSet) {
174             // Once isSbustring is true, we are done,
175             // str is a substring of existing element
176             if (s.contains(str) && !isSubstring) {
177                 isSubstring = true;
178             }
179         }
180     }
181 }
```

```
179         }
180         if (!isSubstring) {
181             // Collect elements that are str's substrings
182             Set<String> toRemove = strSet.newInstance();
183             for (String s : strSet) {
184                 if (str.contains(s)) {
185                     toRemove.add(s);
186                 }
187             }
188
189             // Remove the collected elements
190             for (String s : toRemove) {
191                 strSet.remove(s);
192             }
193
194             // Add the new str
195             strSet.add(str);
196         }
197     }
198
199     /**
200      * Returns the set of all individual lines read from {@code
input}, except
201      * that any line that is a substring of another is not in the
returned set.
202      *
203      * @param input
204      *      source of strings, one per line
205      * @return set of lines read from {@code input}
206      * @requires input.is_open
207      * @ensures <pre>
208      * input.is_open and input.content = <> and
209      * linesFromInput = [maximal set of lines from #input.content
such that
210      * CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
211      * </pre>
212      */
213     public static Set<String> linesFromInput(SimpleReader input) {
214         assert input != null : "Violation of: input is not null";
```

```
215         assert input.isOpen() : "Violation of: input.is_open";
216
217         Set<String> inputStr = new Set1L<String>();
218
219         // Process each line from input and add to the set while
avoiding substrings
220         while (!input.atEOS()) {
221             String line = input.nextLine();
222             // Use the method addToSetAvoidingSubstrings
223             addToSetAvoidingSubstrings(inputStr, line);
224         }
225         return inputStr;
226
227     }
228
229     /**
230      * Returns the longest overlap between the suffix of one string
and the
231      * prefix of another string in {@code strSet}, and identifies
the two
232      * strings that achieve that overlap.
233      *
234      * @param strSet
235      *         the set of strings examined
236      * @param bestTwo
237      *         an array containing (upon return) the two strings
with the
238      *         largest such overlap between the suffix of {@code
bestTwo[0]}
239      *         and the prefix of {@code bestTwo[1]}
240      * @return the amount of overlap between those two strings
241      * @replaces bestTwo[0], bestTwo[1]
242      * @requires <pre>
243      *     CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
244      *     bestTwo.length >= 2
245      * </pre>
246      * @ensures <pre>
247      *     bestTwo[0] is in strSet and
248      *     bestTwo[1] is in strSet and
249      *     OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap) and
```

```
250     * for all str1, str2: string of character, overlap: integer
251     *     where (str1 is in strSet and str2 is in strSet and
252     *         OVERLAPS(str1, str2, overlap))
253     *     (overlap <= bestOverlap)
254     * </pre>
255     */
256     private static int bestOverlap(Set<String> strSet, String[]
bestTwo) {
257         assert strSet != null : "Violation of: strSet is not null";
258         assert bestTwo != null : "Violation of: bestTwo is not
null";
259         assert bestTwo.length >= 2 : "Violation of: bestTwo.length
>= 2";
260         /*
261          * Note: Rest of precondition not checked!
262          */
263         int bestOverlap = 0;
264         Set<String> processed = strSet.newInstance();
265         while (strSet.size() > 0) {
266             /*
267              * Remove one string from strSet to check against all
others
268              */
269             String str0 = strSet.removeAny();
270             for (String str1 : strSet) {
271                 /*
272                  * Check str0 and str1 for overlap first in one
order...
273                  */
274                 int overlapFrom0To1 = overlap(str0, str1);
275                 if (overlapFrom0To1 > bestOverlap) {
276                     /*
277                      * Update best overlap found so far, and the
two strings
278                      * that produced it
279                      */
280                     bestOverlap = overlapFrom0To1;
281                     bestTwo[0] = str0;
282                     bestTwo[1] = str1;
283                 }
            }
        }
    }
}
```



```
284             /*
285             * ... and then in the other order
286             */
287             int overlapFrom1To0 = overlap(str1, str0);
288             if (overlapFrom1To0 > bestOverlap) {
289                 /*
290                 * Update best overlap found so far, and the
two strings
291                 * that produced it
292                 */
293                 bestOverlap = overlapFrom1To0;
294                 bestTwo[0] = str1;
295                 bestTwo[1] = str0;
296             }
297         }
298         /*
299         * Record that str0 has been checked against every
other string in
300         * strSet
301         */
302         processed.add(str0);
303     }
304     /*
305     * Restore strSet and return best overlap
306     */
307     strSet.transferFrom(processed);
308     return bestOverlap;
309 }
310
311 /**
312  * Combines strings in {@code strSet} as much as possible,
leaving in it
313  * only strings that have no overlap between a suffix of one
string and a
314  * prefix of another. Note: uses a "greedy approach" to
assembly, hence may
315  * not result in {@code strSet} being as small a set as
possible at the end.
316  *
317  * @param strSet
```

```
318     *           set of strings
319     * @updates strSet
320     * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
321     * @ensures <pre>
322     * ALL_SUPERSTRINGS(strSet) is subset of
  ALL_SUPERSTRINGS(#strSet) and
323     * |strSet| <= |#strSet| and
324     * CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
325     * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
326     * </pre>
327     */
328     public static void assemble(Set<String> strSet) {
329         assert strSet != null : "Violation of: strSet is not null";
330         /*
331          * Note: Precondition not checked!
332          */
333         /*
334          * Combine strings as much possible, being greedy
335          */
336         boolean done = false;
337         while ((strSet.size() > 1) && !done) {
338             String[] bestTwo = new String[2];
339             int bestOverlap = bestOverlap(strSet, bestTwo);
340             if (bestOverlap == 0) {
341                 /*
342                  * No overlapping strings remain; can't do any more
343                  */
344                 done = true;
345             } else {
346                 /*
347                 their
348                 * Replace the two most-overlapping strings with
349                 * combination; this can be done with add rather
350                 * addToSetAvoidingSubstrings because the latter
351                 * same thing (this claim requires justification)
352                 */
353                 strSet.remove(bestTwo[0]);
354                 strSet.remove(bestTwo[1]);
355             }
356         }
357     }
```

```
354         String overlapped = combination(bestTwo[0],
bestTwo[1], bestOverlap);
355         strSet.add(overlapped);
356     }
357 }
358 }
359
360 /**
361  * Prints the string {@code text} to {@code out}, replacing
each '~' with a
362  * line separator.
363  *
364  * @param text
365  *         string to be output
366  * @param out
367  *         output stream
368  * @updates out
369  * @requires out.is_open
370  * @ensures <pre>
371  * out.is_open and
372  * out.content = #out.content *
373  * [text with each '~' replaced by line separator]
374  * </pre>
375  */
376 public static void printWithLineSeparators(String text,
SimpleWriter out) {
377     assert text != null : "Violation of: text is not null";
378     assert out != null : "Violation of: out is not null";
379     assert out.isOpen() : "Violation of: out.is_open";
380
381     // Temporary variable to hold the input text
382     String tempText = text;
383
384     // Iterate through each character in the text
385     for (int i = 0; i < tempText.length(); i++) {
386         if (text.charAt(i) == '~') {
387             // Print a newline when encountering '~'
388             out.print("\n");
389         } else {
390             // Print the character as is
```

```
391         out.print(text.charAt(i));
392     }
393 }
394
395 }
396
397 /**
398  * Given a file name (relative to the path where the
399  * application is running)
400  * that contains fragments of a single original source text,
401  * one fragment
402  * per line, outputs to stdout the result of trying to
403  * reassemble the
404  * original text from those fragments using a "greedy
405  * assembler". The
406  * result, if reassembly is complete, might be the original
407  * text; but this
408  * might not happen because a greedy assembler can make a
409  * mistake and end up
410  * predicting the fragments were from a string other than the
411  * true original
412  * source text. It can also end up with two or more fragments
413  * that are
414  * mutually non-overlapping, in which case it outputs the
415  * remaining
416  * fragments, appropriately labelled.
417  *
418  * @param args
419  *         Command-line arguments: not used
420  */
421 public static void main(String[] args) {
422     SimpleReader in = new SimpleReader1L();
423     SimpleWriter out = new SimpleWriter1L();
424     /*
425      * Get input file name
426      */
427     out.print("Input file (with fragments): ");
428     String inputFileName = in.nextLine();
429     SimpleReader inFile = new SimpleReader1L(inputFileName);
430     /*
```

```
422      * Get initial fragments from input file
423      */
424      Set<String> fragments = linesFromInput(inFile);
425      /*
426      * Close inFile; we're done with it
427      */
428      inFile.close();
429      /*
430      * Assemble fragments as far as possible
431      */
432      assemble(fragments);
433      /*
434      * Output fully assembled text or remaining fragments
435      */
436      if (fragments.size() == 1) {
437          out.println();
438          String text = fragments.removeAny();
439          printWithLineSeparators(text, out);
440      } else {
441          int fragmentNumber = 0;
442          for (String str : fragments) {
443              fragmentNumber++;
444              out.println();
445              out.println("-----");
446              out.println("  -- Fragment #" + fragmentNumber + ":
--");
447              out.println("-----");
448              printWithLineSeparators(str, out);
449          }
450      }
451      /*
452      * Close input and output streams
453      */
454      in.close();
455      out.close();
456  }
457
458 }
459
```