

```
1 import components.set.Set;
7
8 /**
9  * Utility class to support string reassembly from fragments.
10 *
11 * @author Shyam Sai Bethina
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 /= 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
102        a match is
103        * found; think about it and try it on some examples to
104        see why
105        * iterating in the other direction doesn't work
106        */
107        int maxOverlap = str2.length() - 1;
108        while (!str1.regionMatches(str1.length() - maxOverlap,
109            str2, 0,
110                maxOverlap)) {
111            maxOverlap--;
112        }
113        return maxOverlap;
114    }
115    /**
116     * Returns concatenation of {@code str1} and {@code str2} from
117     which one of
118     * the two "copies" of the common string of {@code overlap}
119     characters at
120     * the end of {@code str1} and the beginning of {@code str2}
121     has been
122     * removed.
123     *
124     * @param str1
125     *           first string
126     * @param str2
127     *           second string
128     * @param overlap
129     *           amount of overlap
130     * @return combination with one "copy" of overlap removed
131     * @requires OVERLAPS(str1, str2, overlap)
132     * @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         //Gets the substring of str1 without the overlap portion
142         String result = str1.substring(0, str1.length() -
overlap);
143
144         //returns str1 without the overlap portion added with str2
145         return result + str2;
146     }
147
148     /**
149      * Adds {@code str} to {@code strSet} if and only if it is not
a substring
150      * of any string already in {@code strSet}; and if it is
added, also removes
151      * from {@code strSet} any string already in {@code strSet}
that is a
152      * substring of {@code str}.
153      *
154      * @param strSet
155      *     set to consider adding to
156      * @param str
157      *     string to consider adding
158      * @updates strSet
159      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
160      * @ensures <pre>
161      * if SUPERSTRINGS(#strSet, str) = {}
162      * then strSet = #strSet union {str} \ SUBSTRINGS(#strSet,
str)
163      * else strSet = #strSet

```

```
164     * </pre>
165     */
166     public static void addToSetAvoidingSubstrings(Set<String>
    strSet,
167         String str) {
168         assert strSet != null : "Violation of: strSet is not
    null";
169         assert str != null : "Violation of: str is not null";
170         /*
171          * Note: Precondition not checked!
172          */
173
174         /*
175          * Hint: consider using contains (a String method)
176          */
177
178         /*
179          * If str is a substring of a string in the set, then we
    end the method
180          * and return nothing
181          */
182         for (String s : strSet) {
183             if (s.indexOf(str) != -1) {
184                 return;
185             }
186         }
187
188         /*
189          * If there is no string within the set that has a
    substring equal to
190          * str, then we add str to the set
191          */
192         strSet.add(str);
193
194         //Creates a new temporary set
195         Set<String> temp = strSet.newInstance();
196
197         //Clears strSet and transfers elements to temporary set
    for later use
198         temp.transferFrom(strSet);
199
200         /*
201          * Goes through each element in the temporary set and
    makes sure to add
```

```
202         * str to the empty strSet. If it is already added, it
    then checks if
203         * str has a substring equal to each element in s, and if
    not, it adds
204         * it to strSet.
205         */
206         for (String s : temp) {
207             if (s.equals(str)) {
208                 strSet.add(s);
209             } else {
210                 if (!str.contains(s)) {
211                     strSet.add(s);
212                 }
213             }
214         }
215     }
216
217     /**
218     * Returns the set of all individual lines read from {@code
    input}, except
219     * that any line that is a substring of another is not in the
    returned set.
220     *
221     * @param input
222     *         source of strings, one per line
223     * @return set of lines read from {@code input}
224     * @requires input.is_open
225     * @ensures <pre>
226     * input.is_open and input.content = <> and
227     * linesFromInput = [maximal set of lines from #input.content
    such that
228     *
    CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
229     * </pre>
230     */
231     public static Set<String> linesFromInput(SimpleReader input) {
232         assert input != null : "Violation of: input is not null";
233         assert input.isOpen() : "Violation of: input.is_open";
234
235         //Creates empty set to add lines to
236         Set<String> result = new Set1L<>();
237
238         //Checks if the input is not at the end
239         while (!input.atEOS()) {
```

```
240         //if input is not at the end, then we get the next
        line
241         String nextLine = input.nextLine();
242         //adds the next line to the resulting set if there are
        no substrings
243         addToSetAvoidingSubstrings(result, nextLine);
244     }
245
246     return result;
247 }
248
249 /**
250  * Returns the longest overlap between the suffix of one
        string and the
251  * prefix of another string in {@code strSet}, and identifies
        the two
252  * strings that achieve that overlap.
253  *
254  * @param strSet
255  *         the set of strings examined
256  * @param bestTwo
257  *         an array containing (upon return) the two
        strings with the
258  *         largest such overlap between the suffix of
        {@code bestTwo[0]}
259  *         and the prefix of {@code bestTwo[1]}
260  * @return the amount of overlap between those two strings
261  * @replaces bestTwo[0], bestTwo[1]
262  * @requires <pre>
263  * CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
264  * bestTwo.length >= 2
265  * </pre>
266  * @ensures <pre>
267  * bestTwo[0] is in strSet and
268  * bestTwo[1] is in strSet and
269  * OVERLAPS(bestTwo[0], bestTwo[1], best0overlap) and
270  * for all str1, str2: string of character, overlap: integer
271  *     where (str1 is in strSet and str2 is in strSet and
272  *         OVERLAPS(str1, str2, overlap))
273  *     (overlap <= best0overlap)
274  * </pre>
275  */
276     private static int bestOverlap(Set<String> strSet, String[]
        bestTwo) {
```

```
277     assert strSet != null : "Violation of: strSet is not
    null";
278     assert bestTwo != null : "Violation of: bestTwo is not
    null";
279     assert bestTwo.length >= 2 : "Violation of: bestTwo.length
    >= 2";
280     /*
281      * Note: Rest of precondition not checked!
282      */
283     int bestOverlap = 0;
284     Set<String> processed = strSet.newInstance();
285     while (strSet.size() > 0) {
286         /*
287         others
288             * Remove one string from strSet to check against all
289             *
290             String str0 = strSet.removeAny();
291             for (String str1 : strSet) {
292                 /*
293                 order...
294                     * Check str0 and str1 for overlap first in one
295                     *
296                     int overlapFrom0To1 = overlap(str0, str1);
297                     if (overlapFrom0To1 > bestOverlap) {
298                         /*
299                         two strings
300                             * Update best overlap found so far, and the
301                             * that produced it
302                             */
303                             bestOverlap = overlapFrom0To1;
304                             bestTwo[0] = str0;
305                             bestTwo[1] = str1;
306                         }
307                         /*
308                         * ... and then in the other order
309                         */
310                         int overlapFrom1To0 = overlap(str1, str0);
311                         if (overlapFrom1To0 > bestOverlap) {
312                             /*
313                             two strings
314                                 * Update best overlap found so far, and the
315                                 * that produced it
316                                 */
317                                 bestOverlap = overlapFrom1To0;
```



```
314             bestTwo[0] = str1;
315             bestTwo[1] = str0;
316         }
317     }
318     /*
319     * Record that str0 has been checked against every
other string in
320     * strSet
321     */
322     processed.add(str0);
323 }
324 /*
325 * Restore strSet and return best overlap
326 */
327 strSet.transferFrom(processed);
328 return bestOverlap;
329 }
330
331 /**
332 * Combines strings in {@code strSet} as much as possible,
leaving in it
333 * only strings that have no overlap between a suffix of one
string and a
334 * prefix of another. Note: uses a "greedy approach" to
assembly, hence may
335 * not result in {@code strSet} being as small a set as
possible at the end.
336 *
337 * @param strSet
338 *         set of strings
339 * @updates strSet
340 * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
341 * @ensures <pre>
342 * ALL_SUPERSTRINGS(strSet) is subset of
ALL_SUPERSTRINGS(#strSet) and
343 * |strSet| <= |#strSet| and
344 * CONTAINS_NO_SUBSTRING_PAIRS(strSet) and
345 * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
346 * </pre>
347 */
348 public static void assemble(Set<String> strSet) {
349     assert strSet != null : "Violation of: strSet is not
null";
350     /*
```

```
351      * Note: Precondition not checked!
352      */
353      /*
354      * Combine strings as much possible, being greedy
355      */
356      boolean done = false;
357      while ((strSet.size() > 1) && !done) {
358          String[] bestTwo = new String[2];
359          int bestOverlap = bestOverlap(strSet, bestTwo);
360          if (bestOverlap == 0) {
361              /*
362              * No overlapping strings remain; can't do any
363              more
364              */
365              done = true;
366          } else {
367              /*
368              * Replace the two most-overlapping strings with
369              their
370              * combination; this can be done with add rather
371              than
372              * addToSetAvoidingSubstrings because the latter
373              would do the
374              * same thing (this claim requires justification)
375              */
376              strSet.remove(bestTwo[0]);
377              strSet.remove(bestTwo[1]);
378              String overlapped = combination(bestTwo[0],
379              bestTwo[1],
380              bestOverlap);
381              strSet.add(overlapped);
382          }
383      }
384  }
385  /**
386   * Prints the string {@code text} to {@code out}, replacing
387   each '~' with a
388   * line separator.
389   *
390   * @param text
391   *           string to be output
392   * @param out
393   *           output stream
```

```
389     * @updates out
390     * @requires out.is_open
391     * @ensures <pre>
392     * out.is_open  and
393     * out.content = #out.content *
394     *   [text with each '~' replaced by line separator]
395     * </pre>
396     */
397     public static void printWithLineSeparators(String text,
SimpleWriter out) {
398         assert text != null : "Violation of: text is not null";
399         assert out != null : "Violation of: out is not null";
400         assert out.isOpen() : "Violation of: out.is_open";
401
402         /*
403         * Replaces all instances of "~" with "\n" which is the
new line
404         * separator. Then prints out the resulting string
405         */
406         String result = text.replaceAll("~", "\n");
407         out.print(result);
408
409     }
410
411     /**
412     * Given a file name (relative to the path where the
application is running)
413     * that contains fragments of a single original source text,
one fragment
414     * per line, outputs to stdout the result of trying to
reassemble the
415     * original text from those fragments using a "greedy
assembler". The
416     * result, if reassembly is complete, might be the original
text; but this
417     * might not happen because a greedy assembler can make a
mistake and end up
418     * predicting the fragments were from a string other than the
true original
419     * source text. It can also end up with two or more fragments
that are
420     * mutually non-overlapping, in which case it outputs the
remaining
421     * fragments, appropriately labelled.
```

```
422     *
423     * @param args
424     *         Command-line arguments: not used
425     */
426     public static void main(String[] args) {
427         SimpleReader in = new SimpleReader1L();
428         SimpleWriter out = new SimpleWriter1L();
429         /*
430          * Get input file name
431          */
432         out.print("Input file (with fragments): ");
433         String inputFileName = in.nextLine();
434         SimpleReader inFile = new SimpleReader1L(inputFileName);
435         /*
436          * Get initial fragments from input file
437          */
438         Set<String> fragments = linesFromInput(inFile);
439
440         /*
441          * Close inFile; we're done with it
442          */
443         inFile.close();
444         /*
445          * Assemble fragments as far as possible
446          */
447         assemble(fragments);
448         /*
449          * Output fully assembled text or remaining fragments
450          */
451         if (fragments.size() == 1) {
452             out.println();
453             String text = fragments.removeAny();
454             printWithLineSeparators(text, out);
455         } else {
456             int fragmentNumber = 0;
457             for (String str : fragments) {
458                 fragmentNumber++;
459                 out.println();
460                 out.println("-----");
461                 out.println("  -- Fragment #" + fragmentNumber +
":  --");
462                 out.println("-----");
463                 printWithLineSeparators(str, out);
464             }
```

```
465     }
466     /*
467     * Close input and output streams
468     */
469     in.close();
470     out.close();
471 }
472
473 }
474
```