

```
1 import components.set.Set;
7
8 /**
9  * Utility class to support string reassembly from fragments.
10 *
11 * @author Obsa Temesgen
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
58 *   strSet and
59 *   1 <= k and k <= |s1| and k <= |s2|)
60 *   (not OVERLAPS(s1, s2, k))
61 * </pre>
62 */
63 public final class StringReassembly {
64
65     /**
66      * Private no-argument constructor to prevent instantiation
67      * of this utility
68      * class.
69      */
70     private StringReassembly() {
71     }
72
73     /**
74      * Reports the maximum length of a common suffix of {@code
75      * str1} and prefix
76      * of {@code str2}.
77      *
78      * @param str1
79      *           first string
80      * @param str2
81      *           second string
82      * @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
102        until 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,
str2, 0,
109            maxOverlap)) {
110            maxOverlap--;
111        }
112        return maxOverlap;
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         int index = str2.indexOf(str1.substring(str1.length() -
    overlap));
143
144         // Remove the overlapping part from str2 before
    concatenating
145         String nonOverlappingStr2 = str2.substring(index +
```

```
        overlap);
146
147        // Concatenate str1 with non-overlapping str2
148        return str1 + nonOverlappingStr2;
149    }
150
151    /**
152     * Adds {@code str} to {@code strSet} if and only if it is
    not a substring
153     * of any string already in {@code strSet}; and if it is
    added, also removes
154     * from {@code strSet} any string already in {@code strSet}
    that is a
155     * substring of {@code str}.
156     *
157     * @param strSet
158     *     set to consider adding to
159     * @param str
160     *     string to consider adding
161     * @updates strSet
162     * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
163     * @ensures <pre>
164     * if SUPERSTRINGS(#strSet, str) = {}
165     * then strSet = #strSet union {str} \ SUBSTRINGS(#strSet,
    str)
166     * else strSet = #strSet
167     * </pre>
168     */
169    public static void addToSetAvoidingSubstrings(Set<String>
    strSet,
170        String str) {
171        assert strSet != null : "Violation of: strSet is not
    null";
172        assert str != null : "Violation of: str is not null";
173        /*
174         * Note: Precondition not checked!
175         */
176
177        /*
```

```
178         * Hint: consider using contains (a String method)
179         */
180
181         // Check if str is a substring of any existing string
182         boolean shouldAdd = true;
183         Set<String> substringsToRemove = new Set2<>();
184         for (String existingStr : strSet) {
185             if (shouldAdd && existingStr.contains(str)) {
186                 shouldAdd = false;
187             }
188             if (!shouldAdd && str.contains(existingStr)) {
189                 substringsToRemove.add(existingStr);
190             }
191         }
192
193         for (String substring : substringsToRemove) {
194             strSet.remove(substring);
195         }
196
197         if (shouldAdd) {
198             strSet.add(str);
199         }
200     }
201 }
202
203 /**
204  * Returns the set of all individual lines read from {@code
input}, except
205  * that any line that is a substring of another is not in
the returned set.
206  *
207  * @param input
208  *         source of strings, one per line
209  * @return set of lines read from {@code input}
210  * @requires input.is_open
211  * @ensures <pre>
212  * input.is_open and input.content = <> and
213  * linesFromInput = [maximal set of lines from
#input.content such that
```

```
214     *
215     * </pre>
216     */
217     public static Set<String> linesFromInput(SimpleReader
218     input) {
219         assert input != null : "Violation of: input is not
220         null";
221         assert input.isOpen() : "Violation of: input.is_open";
222         // Instantiate Set2 to store the unique lines
223         Set<String> strSet = new Set2<>();
224         // Read each line from input
225         while (!input.atEOS()) { // boolean modifier to report
226             end of stream
227             String str = input.nextLine();
228             addToSetAvoidingSubstrings(strSet, str);
229         }
230         return strSet;
231     }
232 }
233
234 /**
235  * Returns the longest overlap between the suffix of one
236  * string and the
237  * prefix of another string in {@code strSet}, and
238  * identifies the two
239  * strings that achieve that overlap.
240  *
241  * @param strSet
242  *         the set of strings examined
243  * @param bestTwo
244  *         an array containing (upon return) the two
245  *         strings with the
246  *         largest such overlap between the suffix of
247  *         {@code bestTwo[0]}
248  *         and the prefix of {@code bestTwo[1]}
```

```

245     * @return the amount of overlap between those two strings
246     * @replaces bestTwo[0], bestTwo[1]
247     * @requires <pre>
248     * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
249     * bestTwo.length >= 2
250     * </pre>
251     * @ensures <pre>
252     * bestTwo[0] is in strSet  and
253     * bestTwo[1] is in strSet  and
254     * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap)  and
255     * for all str1, str2: string of character, overlap:
integer
256     *      where (str1 is in strSet  and  str2 is in strSet
and
257     *              OVERLAPS(str1, str2, overlap))
258     *      (overlap <= bestOverlap)
259     * </pre>
260     */
261     private static int bestOverlap(Set<String> strSet, String[]
bestTwo) {
262         assert strSet != null : "Violation of: strSet is not
null";
263         assert bestTwo != null : "Violation of: bestTwo is not
null";
264         assert bestTwo.length >= 2 : "Violation of:
bestTwo.length >= 2";
265         /*
266          * Note: Rest of precondition not checked!
267          */
268         int bestOverlap = 0;
269         Set<String> processed = strSet.newInstance();
270         while (strSet.size() > 0) {
271             /*
272              * Remove one string from strSet to check against
all others
273              */
274             String str0 = strSet.removeAny();
275             for (String str1 : strSet) {
276                 /*

```



```
277             * Check str0 and str1 for overlap first in one
                order...
278             */
279             int overlapFrom0To1 = overlap(str0, str1);
280             if (overlapFrom0To1 > bestOverlap) {
281                 /*
282                 * Update best overlap found so far, and
                the two strings
283                 * that produced it
284                 */
285                 bestOverlap = overlapFrom0To1;
286                 bestTwo[0] = str0;
287                 bestTwo[1] = str1;
288             }
289             /*
290             * ... and then in the other order
291             */
292             int overlapFrom1To0 = overlap(str1, str0);
293             if (overlapFrom1To0 > bestOverlap) {
294                 /*
295                 * Update best overlap found so far, and
                the two strings
296                 * that produced it
297                 */
298                 bestOverlap = overlapFrom1To0;
299                 bestTwo[0] = str1;
300                 bestTwo[1] = str0;
301             }
302         }
303         /*
304         * Record that str0 has been checked against every
                other string in
305         * strSet
306         */
307         processed.add(str0);
308     }
309     /*
310     * Restore strSet and return best overlap
311     */
```

```
312         strSet.transferFrom(processed);
313         return bestOverlap;
314     }
315
316     /**
317      * Combines strings in {@code strSet} as much as possible,
318      leaving in it
319      * only strings that have no overlap between a suffix of
320      one string and a
321      * prefix of another. Note: uses a "greedy approach" to
322      assembly, hence may
323      * not result in {@code strSet} being as small a set as
324      possible at the end.
325      *
326      * @param strSet
327      *         set of strings
328      * @updates strSet
329      * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
330      * @ensures <pre>
331      * ALL_SUPERSTRINGS(strSet) is subset of
332      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
340 null";
341
342         /*
343          * Note: Precondition not checked!
344          */
345         /*
346          * Combine strings as much possible, being greedy
347          */
348         boolean done = false;
349         while ((strSet.size() > 1) && !done) {
350             String[] bestTwo = new String[2];
351             int bestOverlap = bestOverlap(strSet, bestTwo);
```

```
345         if (bestOverlap == 0) {
346             /*
347             * No overlapping strings remain; can't do any
348             more
349             */
350             done = true;
351         } else {
352             /*
353             * Replace the two most-overlapping strings
354             with their
355             * combination; this can be done with add
356             rather than
357             * addToSetAvoidingSubstrings because the
358             latter would do the
359             * same thing (this claim requires
360             justification)
361             */
362             strSet.remove(bestTwo[0]);
363             strSet.remove(bestTwo[1]);
364             String overlapped = combination(bestTwo[0],
365             bestTwo[1],
366             bestOverlap);
367             strSet.add(overlapped);
368         }
369     }
370 }
371
372 /**
373  * Prints the string {@code text} to {@code out}, replacing
374  each '~' with a
375  * line separator.
376  *
377  * @param text
378  *         string to be output
379  * @param out
380  *         output stream
381  * @updates out
382  * @requires out.is_open
383  * @ensures <pre>
```

```
377     * out.is_open  and
378     * out.content = #out.content *
379     * [text with each '~' replaced by line separator]
380     * </pre>
381     */
382     public static void printWithLineSeparators(String text,
SimpleWriter out) {
383         assert text != null : "Violation of: text is not null";
384         assert out != null : "Violation of: out is not null";
385         assert out.isOpen() : "Violation of: out.is_open";
386
387         for (int i = 0; i < text.length(); i++) {
388             char ch = text.charAt(i);
389             if (ch == '~') {
390                 out.println(); // Print line separator
391             } else {
392                 out.print(ch); // Print the character as it is
393             }
394         }
395
396         // Ensure the last line is terminated with a newline
397         out.println();
398
399     }
400
401     /**
402     * Given a file name (relative to the path where the
application is running)
403     * that contains fragments of a single original source
text, one fragment
404     * per line, outputs to stdout the result of trying to
reassemble the
405     * original text from those fragments using a "greedy
assembler". The
406     * result, if reassembly is complete, might be the original
text; but this
407     * might not happen because a greedy assembler can make a
mistake and end up
408     * predicting the fragments were from a string other than
```

```
the true original
409     * source text. It can also end up with two or more
    fragments that are
410     * mutually non-overlapping, in which case it outputs the
    remaining
411     * fragments, appropriately labelled.
412     *
413     * @param args
414     *         Command-line arguments: not used
415     */
416     public static void main(String[] args) {
417         SimpleReader in = new SimpleReader1L();
418         SimpleWriter out = new SimpleWriter1L();
419         /*
420          * Get input file name
421          */
422         out.print("Input file (with fragments): ");
423         String inputFileName = in.nextLine();
424         SimpleReader inFile = new
SimpleReader1L(inputFileName);
425         /*
426          * Get initial fragments from input file
427          */
428         Set<String> fragments = linesFromInput(inFile);
429         /*
430          * Close inFile; we're done with it
431          */
432         inFile.close();
433         /*
434          * Assemble fragments as far as possible
435          */
436         assemble(fragments);
437         /*
438          * Output fully assembled text or remaining fragments
439          */
440         if (fragments.size() == 1) {
441             out.println();
442             String text = fragments.removeAny();
443             printWithLineSeparators(text, out);

```

```
444         } else {
445             int fragmentNumber = 0;
446             for (String str : fragments) {
447                 fragmentNumber++;
448                 out.println();
449                 out.println("-----");
450                 out.println("  -- Fragment #" + fragmentNumber
+ ": --");
451                 out.println("-----");
452                 printWithLineSeparators(str, out);
453             }
454         }
455         /*
456         * Close input and output streams
457         */
458         in.close();
459         out.close();
460     }
461
462 }
463
```