```java
import components.set.Set;
import components.set.Set1L;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;

/**
 * Utility class to support string reassembly from fragments.
 *
 * @author David Park
 *
 * @mathdefinitions <pre>
 *
 * OVERLAPS (
 *   s1: string of character,
 *   s2: string of character,
 *   k: integer
 *  ) : boolean is
 *  0 <= k  and  k <= |s1|  and  k <= |s2|  and
 *  s1[|s1|-k, |s1|) = s2[0, k)
 *
 * SUBSTRINGS (
 *   strSet: finite set of string of character,
 *   s: string of character
 *  ) : finite set of string of character is
 *  {t: string of character
 *    where (t is in strSet  and  t is substring of s)
 *   (t)}
 *
 * SUPERSTRINGS (
 *   strSet: finite set of string of character,
 *   s: string of character
 *  ) : finite set of string of character is
 *  {t: string of character
 *    where (t is in strSet  and  s is substring of t)
 *   (t)}
 *
 * CONTAINS_NO_SUBSTRING_PAIRS (
 *   strSet: finite set of string of character
 *  ) : boolean is
 *  for all t: string of character
 *    where (t is in strSet)
 *   (SUBSTRINGS(strSet \ {t}, t) = {})
 *
 * ALL_SUPERSTRINGS (
 *   strSet: finite set of string of character
 *  ) : set of string of character is
 *  {t: string of character
 *    where (SUBSTRINGS(strSet, t) = strSet)
 *   (t)}
 *
 * CONTAINS_NO_OVERLAPPING_PAIRS (
 *   strSet: finite set of string of character
 *  ) : boolean is
 *  for all t1, t2: string of character, k: integer
 *    where (t1 /= t2  and  t1 is in strSet  and  t2 is in strSet  and
 *           1 <= k  and  k <= |s1|  and  k <= |s2|)
 *   (not OVERLAPS(s1, s2, k))
 *
 * </pre>
 */
public final class StringReassembly {

    /**
     * Private no-argument constructor to prevent instantiation of this utility
     * class.
     */
    private StringReassembly() {
    }

    /**
     * Reports the maximum length of a common suffix of {@code str1} and prefix
     * of {@code str2}.
     *
     * @param str1
     *            first string
     * @param str2
     *            second string
     * @return maximum overlap between right end of {@code str1} and left end of
     *         {@code str2}
     * @requires <pre>
     * str1 is not substring of str2  and
     * str2 is not substring of str1
     * </pre>
     * @ensures <pre>
     * OVERLAPS(str1, str2, overlap)  and
     * for all k: integer
     *     where (overlap < k  and  k <= |str1|  and  k <= |str2|)
     *  (not OVERLAPS(str1, str2, k))
     * </pre>
     */
    public static int overlap(String str1, String str2) {
        assert str1 != null : "Violation of: str1 is not null";
        assert str2 != null : "Violation of: str2 is not null";
        assert str2.indexOf(str1) < 0 : "Violation of: "
                + "str1 is not substring of str2";
        assert str1.indexOf(str2) < 0 : "Violation of: "
                + "str2 is not substring of str1";
        /*
         * Start with maximum possible overlap and work down until a match is
         * found; think about it and try it on some examples to see why
         * iterating in the other direction doesn't work
         */
        int maxOverlap = str2.length() - 1;
        while (!str1.regionMatches(str1.length() - maxOverlap, str2, 0,
                maxOverlap)) {
            maxOverlap--;
        }
        return maxOverlap;
    }

    /**
     * Returns concatenation of {@code str1} and {@code str2} from which one of
     * the two "copies" of the common string of {@code overlap} characters at
     * the end of {@code str1} and the beginning of {@code str2} has been
     * removed.
     *
     * @param str1
     *            first string
     * @param str2
     *            second string
     * @param overlap
     *            amount of overlap
     * @return combination with one "copy" of overlap removed
     * @requires OVERLAPS(str1, str2, overlap)
     * @ensures combination = str1[0, |str1|-overlap) * str2
     */
    public static String combination(String str1, String str2, int overlap) {
        assert str1 != null : "Violation of: str1 is not null";
        assert str2 != null : "Violation of: str2 is not null";
        assert 0 <= overlap && overlap <= str1.length()
                && overlap <= str2.length()
                && str1.regionMatches(str1.length() - overlap, str2, 0,
                        overlap) : ""
                                + "Violation of: OVERLAPS(str1, str2, overlap)";

        /*
         * Hint: consider using substring (a String method)
         */

        String x = str1.substring(0, str1.length() - overlap);
        x = x.concat(str2);

        /*
         * This line added just to make the program compilable. Should be
         * replaced with appropriate return statement.
         */
        return x;
    }

    /**
     * Adds {@code str} to {@code strSet} if and only if it is not a substring
     * of any string already in {@code strSet}; and if it is added, also removes
     * from {@code strSet} any string already in {@code strSet} that is a
     * substring of {@code str}.
     *
     * @param strSet
     *            set to consider adding to
     * @param str
     *            string to consider adding
     * @updates strSet
     * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
     * @ensures <pre>
     * if SUPERSTRINGS(#strSet, str) = {}
     *  then strSet = #strSet union {str} \ SUBSTRINGS(#strSet, str)
     *  else strSet = #strSet
     * </pre>
     */
    public static void addToSetAvoidingSubstrings(Set<String> strSet,
            String str) {
```

```java
        assert strSet != null : "Violation of: strSet is not null";
        assert str != null : "Violation of: str is not null";
        /*
         * Note: Precondition not checked!
         */

        /*
         * Hint: consider using contains (a String method)
         */

        boolean isSubstring = false;
        Set<String> substringsToRemove = new Set1L<>();

        for (String existingStr : strSet) {
            if (str.contains(existingStr)) {
                // If str is a superstring of an existing string, then remove
                substringsToRemove.add(existingStr);
            } else if (existingStr.contains(str)) {
                // If str is a substring of an existing string, don't add str.
                isSubstring = true;
                break;
            }
        }

        // Remove substrings from the set.
        for (String toRemove : substringsToRemove) {
            strSet.remove(toRemove);
        }

        // Add the new string if it's not a substring of any existing string.
        if (!isSubstring) {
            strSet.add(str);
        }
    }

    /**
     * Returns the set of all individual lines read from {@code input}, except
     * that any line that is a substring of another is not in the returned set.
     *
     * @param input
     *            source of strings, one per line
     * @return set of lines read from {@code input}
     * @requires input.is_open
     * @ensures <pre>
     * input.is_open  and  input.content = <>  and
     * linesFromInput = [maximal set of lines from #input.content such that
     *                   CONTAINS_NO_SUBSTRING_PAIRS(linesFromInput)]
     * </pre>
     */
    public static Set<String> linesFromInput(SimpleReader input) {
        assert input != null : "Violation of: input is not null";
        assert input.isOpen() : "Violation of: input.is_open";

        Set<String> temp = new Set1L<>();
        String x = "";
        while (!input.atEOS()) {
            x = input.nextLine();
            addToSetAvoidingSubstrings(temp, x);
        }

        /*
         * This line added just to make the program compilable. Should be
         * replaced with appropriate return statement.
         */
        return temp;
    }

    /**
     * Returns the longest overlap between the suffix of one string and the
     * prefix of another string in {@code strSet}, and identifies the two
     * strings that achieve that overlap.
     *
     * @param strSet
     *            the set of strings examined
     * @param bestTwo
     *            an array containing (upon return) the two strings with the
     *            largest such overlap between the suffix of {@code bestTwo[0]}
     *            and the prefix of {@code bestTwo[1]}
     * @return the amount of overlap between those two strings
     * @replaces bestTwo[0], bestTwo[1]
     * @requires <pre>
     * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
     * bestTwo.length >= 2
     * </pre>
     * @ensures <pre>
     * bestTwo[0] is in strSet  and
     * bestTwo[1] is in strSet  and
     * OVERLAPS(bestTwo[0], bestTwo[1], bestOverlap)  and
     * for all str1, str2: string of character, overlap: integer
     *     where (str1 is in strSet  and  str2 is in strSet  and
     *            OVERLAPS(str1, str2, overlap))
     *   (overlap <= bestOverlap)
     * </pre>
     */
    private static int bestOverlap(Set<String> strSet, String[] bestTwo) {
        assert strSet != null : "Violation of: strSet is not null";
        assert bestTwo != null : "Violation of: bestTwo is not null";
        assert bestTwo.length >= 2 : "Violation of: bestTwo.length >= 2";
        /*
         * Note: Rest of precondition not checked!
         */
        int bestOverlap = 0;
        Set<String> processed = strSet.newInstance();
        while (strSet.size() > 0) {
            /*
             * Remove one string from strSet to check against all others
             */
            String str0 = strSet.removeAny();
            for (String str1 : strSet) {
                /*
                 * Check str0 and str1 for overlap first in one order...
                 */
                int overlapFrom0To1 = overlap(str0, str1);
                if (overlapFrom0To1 > bestOverlap) {
                    /*
                     * Update best overlap found so far, and the two strings
                     * that produced it
                     */
                    bestOverlap = overlapFrom0To1;
                    bestTwo[0] = str0;
                    bestTwo[1] = str1;
                }
                /*
                 * ... and then in the other order
                 */
                int overlapFrom1To0 = overlap(str1, str0);
                if (overlapFrom1To0 > bestOverlap) {
                    /*
                     * Update best overlap found so far, and the two strings
                     * that produced it
                     */
                    bestOverlap = overlapFrom1To0;
                    bestTwo[0] = str1;
                    bestTwo[1] = str0;
                }
            }
            /*
             * Record that str0 has been checked against every other string in
             * strSet
             */
            processed.add(str0);
        }
        /*
         * Restore strSet and return best overlap
         */
        strSet.transferFrom(processed);
        return bestOverlap;
    }

    /**
     * Combines strings in {@code strSet} as much as possible, leaving in it
     * only strings that have no overlap between a suffix of one string and a
     * prefix of another. Note: uses a "greedy approach" to assembly, hence may
     * not result in {@code strSet} being as small a set as possible at the end.
     *
     * @param strSet
     *            set of strings
     * @updates strSet
     * @requires CONTAINS_NO_SUBSTRING_PAIRS(strSet)
     * @ensures <pre>
     * ALL_SUPERSTRINGS(strSet) is subset of ALL_SUPERSTRINGS(#strSet)  and
     * |strSet| <= |#strSet|  and
     * CONTAINS_NO_SUBSTRING_PAIRS(strSet)  and
     * CONTAINS_NO_OVERLAPPING_PAIRS(strSet)
     * </pre>
     */
    public static void assemble(Set<String> strSet) {
        assert strSet != null : "Violation of: strSet is not null";
        /*
         * Note: Precondition not checked!
         */
```

```java
        /*
         * Combine strings as much possible, being greedy
         */
        boolean done = false;
        while ((strSet.size() > 1) && !done) {
            String[] bestTwo = new String[2];
            int bestOverlap = bestOverlap(strSet, bestTwo);
            if (bestOverlap == 0) {
                /*
                 * No overlapping strings remain; can't do any more
                 */
                done = true;
            } else {
                /*
                 * Replace the two most-overlapping strings with their
                 * combination; this can be done with add rather than
                 * addToSetAvoidingSubstrings because the latter would do the
                 * same thing (this claim requires justification)
                 */
                strSet.remove(bestTwo[0]);
                strSet.remove(bestTwo[1]);
                String overlapped = combination(bestTwo[0], bestTwo[1],
                        bestOverlap);
                strSet.add(overlapped);
            }
        }
    }

    /**
     * Prints the string {@code text} to {@code out}, replacing each '~' with a
     * line separator.
     *
     * @param text
     *            string to be output
     * @param out
     *            output stream
     * @updates out
     * @requires out.is_open
     * @ensures <pre>
     * out.is_open  and
     * out.content = #out.content *
     *   [text with each '~' replaced by line separator]
     * </pre>
     */
    public static void printWithLineSeparators(String text, SimpleWriter out) {
        assert text != null : "Violation of: text is not null";
        assert out != null : "Violation of: out is not null";
        assert out.isOpen() : "Violation of: out.is_open";

        for (int i = 0; i < text.length(); i++) {
            if (text.charAt(i) == '~') {
                out.println();
            } else {
                out.print(text.charAt(i));
            }
        }

    }

    /**
     * Given a file name (relative to the path where the application is running)
     * that contains fragments of a single original source text, one fragment
     * per line, outputs to stdout the result of trying to reassemble the
     * original text from those fragments using a "greedy assembler". The
     * result, if reassembly is complete, might be the original text; but this
     * might not happen because a greedy assembler can make a mistake and end up
     * predicting the fragments were from a string other than the true original
     * source text. It can also end up with two or more fragments that are
     * mutually non-overlapping, in which case it outputs the remaining
     * fragments, appropriately labelled.
     *
     * @param args
     *            Command-line arguments: not used
     */
    public static void main(String[] args) {
        SimpleReader in = new SimpleReader1L();
        SimpleWriter out = new SimpleWriter1L();
        /*
         * Get input file name
         */
        out.print("Input file (with fragments): ");
        String inputFileName = in.nextLine();
        SimpleReader inFile = new SimpleReader1L(inputFileName);
        /*
         * Get initial fragments from input file
         */
        Set<String> fragments = linesFromInput(inFile);
        /*
         * Close inFile; we're done with it
         */
        inFile.close();
        /*
         * Assemble fragments as far as possible
         */
        assemble(fragments);
        /*
         * Output fully assembled text or remaining fragments
         */
        if (fragments.size() == 1) {
            out.println();
            String text = fragments.removeAny();
            printWithLineSeparators(text, out);
        } else {
            int fragmentNumber = 0;
            for (String str : fragments) {
                fragmentNumber++;
                out.println();
                out.println("--------------------");
                out.println("  -- Fragment #" + fragmentNumber + ": --");
                out.println("--------------------");
                printWithLineSeparators(str, out);
            }
        }
        /*
```

```
         * Close input and output streams
         */
        in.close();
        out.close();
    }

}
```