```
 1 import java.util.*;
 3
 4 /**
 5  * Main class - for accessing suffix tree applications
 6  * David Manlove, Jan 03.  Modified by David Manlove, Jan 07 and Jan 09.
 7  */
 8
 9 public class Main {
10
11     /**
12      * The main method.
13      * @param args the arguments
14      */
15     public static void main(String args[]) {
16
17         Scanner standardInput = new Scanner(System.in);
18         do {
19             // display prompt for user
20             System.out.println();
21             System.out.print("Enter the number of the task or type 'q' to quit: ");
22
23             // read in a line from standard input
24             String line = standardInput.nextLine();
25             System.out.println();
26
27             try {
28                 // try to extract an integer from line if possible
29                 int numTask = Integer.parseInt(line);
30
31                 // variables used in multiple cases
32                 String text, searchWord;
33                 SuffixTree suffixTree;
34                 SuffixTreeAppl suffixTreeAppl;
35                 FileInput newInput;
36                 byte [] readInput;
37
38                 switch (numTask) {
39                 case 1:
40                     // get file name to be found
41                     System.out.print("Enter the name of the text file: ");
42                     text = standardInput.nextLine();
43                     System.out.println();
44
45                     // get string to be located
46                     System.out.print("Enter the string to search for: ");
47                     searchWord = standardInput.nextLine();
48
49                     // read suffix tree into a byte string
50                     newInput = new FileInput(text);
51                     readInput = newInput.readFile();
52
53                     // turn file into suffixTreeAppl instance
54                     suffixTree = new SuffixTree(readInput);
55                     suffixTreeAppl = new SuffixTreeAppl(suffixTree);
56
57                     // search for string and return result
58                     Task1Info task1 = suffixTreeAppl.searchSuffixTree(searchWord.getBytes
   ());
59                     if (task1.getPos() == (-1)) {
60                         System.out.println("Search string " + '"' + searchWord + '"'
61                                 + " not found in " + text);
62                     } else {
```

```
63                      System.out.println("Search string " + '"' + searchWord + '"'
64                              + " occurs at position " + task1.getPos() + " of " + text);
65                  }
66                  break;
67
68              case 2:
69                  // get file name to be found
70                  System.out.print("Enter the name of the text file: ");
71                  text = standardInput.nextLine();
72                  System.out.println();
73
74                  // get string to be located
75                  System.out.print("Enter the string to search for: ");
76                  searchWord = standardInput.nextLine();
77
78                  // read suffix tree into a byte string
79                  newInput = new FileInput(text);
80                  readInput = newInput.readFile();
81
82                  // turn file into suffixTreeAppl instance
83                  suffixTree = new SuffixTree(readInput);
84                  suffixTreeAppl = new SuffixTreeAppl(suffixTree);
85
86                  //Search for occurrences and return result
87                  Task2Info task2 = suffixTreeAppl.allOccurrences(searchWord.getBytes());
88                  if ( task2.getPositions().isEmpty()) {
89                      System.out.println("Search string " + '"' + searchWord + '"'
90                              + " does not occur in " + text);
91                  } else {
92                      int len = task2.getPositions().size();
93                      System.out.println("Search string " + '"' + searchWord + '"'
94                              + " occurs in " + text + " at positions:");
95
96                      // display each occurrence location
97                      int i = 0;
98                      while ( i < task2.getPositions().size() ) {
99                          System.out.println(task2.getPositions().get(i));
100                         i++;
101                     }
102                     System.out.println("The total number of occurences is " + len);
103                 }
104                 break;
105
106             case 3:
107                 // get file name to be found
108                 System.out.print("Enter the name of the text file: ");
109                 text = standardInput.nextLine();
110                 System.out.println();
111
112                 // read suffix tree into a byte string
113                 newInput = new FileInput(text);
114                 readInput = newInput.readFile();
115
116                 // turn file into suffixTreeAppl instance
117                 suffixTree = new SuffixTree(readInput);
118                 suffixTreeAppl = new SuffixTreeAppl(suffixTree);
119
120                 // find lrs
121                 Task3Info task3 = suffixTreeAppl.traverseForLrs();
122
123                 // check there was a repeated substring
124                 if (task3.getLen() == 0) {
```

```
125                     System.out.println("There are no repeated substrings in " + text);
126                 } else {
127                     // get the longest string Found
128                     String lrs = new String(readInput).substring(task3.getPos1(),
129                             task3.getPos1()+task3.getLen());
130
131                     // display results
132                     System.out.println("An LRS in " + text + " is " + '"' + lrs + '"');
133                     System.out.println("Its Length is " + task3.getLen());
134                     System.out.println("Starting position of one occurence is "
135                             + task3.getPos1());
136                     System.out.println("Starting position of another occurence is "
137                             + task3.getPos2());
138                 }
139                 break;
140
141             case 4:
142                 // get file names to be found
143                 System.out.print("Enter the name of the first text file: ");
144                 text = standardInput.nextLine();
145                 System.out.println();
146                 System.out.print("Enter the name of the second text file: ");
147                 String text2 = standardInput.nextLine();
148                 System.out.println();
149
150                 // read suffix tree into a byte string
151                 newInput = new FileInput(text);
152                 readInput = newInput.readFile();
153
154                 newInput = new FileInput(text2);
155                 byte [] readInput2 = newInput.readFile();
156
157                 // turn file into suffixTreeAppl instance
158                 suffixTree = new SuffixTree(readInput, readInput2);
159                 suffixTreeAppl = new SuffixTreeAppl(suffixTree);
160
161                 Task4Info task4 = suffixTreeAppl.traverseForLcs(readInput.length);
162
163                 if (task4.getLen() == 0) {
164                     System.out.println("There are no common substrings between "
165                             + text + " and " + text2);
166                 } else {
167                     // display results
168                     String lcs = new String(readInput).substring(task4.getPos1(),
169                             task4.getPos1()+task4.getLen());
170                     System.out.println("An LCS of " + text + " and " + text2 +
171                             " is " + '"' + lcs + '"');
172                     System.out.println("Its Length is " + task4.getLen());
173                     System.out.println("Starting position of one occurence is "
174                             + task4.getPos1());
175                     System.out.println("Starting position of another occurence is "
176                             + task4.getPos2());
177                 }
178                 break;
179
180             default: throw new NumberFormatException();
181             }
182         }
183         catch (NumberFormatException e) {
184             if (line.length()==0 || line.charAt(0)!='q')
185                 System.out.println("You must enter either '1', '2', '3', '4' or 'q'.");
186             else
```

```
187                     break;
188                 }
189             } while (true);
190             standardInput.close();
191         }
192 }
```

```java
package SuffixTreePackage;

/**
 * Class for construction and manipulation of suffix trees based on a list
 * of children at each node.
 *
 * Includes naive O(n^2) suffix tree construction algorithm based on
 * repeated insertion of suffixes and node-splitting.
 *
 * Modifies Ada implementation of naive suffix tree construction algorithm
 * due to Rob Irving, Jan 00.
 *
 * Also incorporates Java code for naive suffix tree construction algorithm
 * due to Ela Hunt, Jan 01.
 *
 * Modifications by David Manlove, Apr 02, Jan 03,Jan 07 and Jan 09.
 */

public class SuffixTree {

    /** Root node of the suffix tree. */
    private SuffixTreeNode root;

    /** String (byte array) corresponding to suffix tree. */
    private byte [] s;

    /** Length of string corresponding to suffix tree (without termination character). */
    private int stringLen;

    /**
     * Builds the suffix tree for a given string.
     *
     * @param sInput the string whose suffix tree is to be built
     * - assumes that '$' does not occur as a character anywhere in sInput
     * - assumes that characters of sInput occupy positions 0 onwards
     */
    public SuffixTree (byte [] sInput) {
        root = new SuffixTreeNode(null, null, 0, 0, -1);  // create root node of suffix tree;
        stringLen = sInput.length;
        s = new byte[stringLen + 1]; // create longer byte array ready for termination character
        System.arraycopy(sInput, 0, s, 0, stringLen);
        s[stringLen] = (byte) '$';   // append termination character to original string
        buildSuffixTree();           // build the suffix tree
    }

    /**
     * Builds a generalised suffix tree for two given strings.
     *
     * @param sInput1 the first string
     * @param sInput2 the second string
     * - assumes that '$' and '#' do not occur as a character anywhere in sInput1 or sInput2
     * - assumes that characters of sInput1 and sInput2 occupy positions 0 onwards
     */
    public SuffixTree (byte[] sInput1, byte[] sInput2) {
        root = new SuffixTreeNode(null, null, 0, 0, -1); // create root node of suffix tree
        int len1 = sInput1.length;
        int len2 = sInput2.length;
        stringLen = len1+len2+1;
        s = new byte [len1+len2+2]; // create a byte array to hold both texts, # and $
```

```
60          System.arraycopy(sInput1, 0, s, 0, len1);
61          s[len1] = (byte) '#';
62          System.arraycopy(sInput2, 0, s, len1+1, len2);
63          s[len1+len2+1] = (byte) '$';
64          buildSuffixTree();
65      }
66
67      /**
68       * Builds the suffix tree.
69       */
70      private void buildSuffixTree() {
71          try {
72              for (int i=0; i<= stringLen; i++) {
73                  // for large files, the following line may be useful for
74                  // indicating the progress of the suffix tree construction
75                  if (i % 10000==0) System.out.println(i);
76
77                  // raise an exception if the text file contained a '$'
78                  if (s[i] == (byte) '$' && i < stringLen)
79                      throw new Exception();
80                  else
81                      insert(i);  // insert suffix number i of z into tree
82              }
83          }
84          catch (Exception e) {
85              System.out.println("Text file contains a $ character!");
86              System.exit(-1);
87          }
88      }
89
90      /**
91       * Given node nodeIn of suffix tree and character ch, search nodeIn,
92       * plus all sibling nodes of nodeIn, looking for a node whose left
93       * label x satisfies ch == s[x].
94       * - Assumes that characters of s occupy positions 0 onwards
95       *
96       * @param nodeIn a node of the suffix tree
97       * @param ch the character to match
98       *
99       * @return the matching suffix tree node (null if none exists)
100      */
101     public SuffixTreeNode searchList (SuffixTreeNode nodeIn, byte ch) {
102
103         SuffixTreeNode next = nodeIn;
104         SuffixTreeNode nodeOut = null;
105
106         while (next != null) {
107             if (next.getLeftLabel() < stringLen && s[next.getLeftLabel()] == ch)
108             {
109                 nodeOut = next;
110                 next = null;
111             }
112             else
113                 next = next.getSibling();
114         }
115         return nodeOut;  // return matching node if successful, or null otherwise
116     }
117
118     /**
119      * Inserts suffix number i of s into suffix tree.
120      * - assumes that characters of s occupy positions 0 onwards
121      *
```

```
122      * @param i the suffix number of s to insert
123      */
124     private void insert(int i) {
125 
126         int pos, j, k;
127         SuffixTreeNode current, next;
128         pos = i;  // position in s
129         current = root;
130 
131         while (true) {
132             // search for child of current with left label x such that s[x]==s[pos]
133             next = searchList(current.getChild(), s[pos]);
134 
135             if (next == null) {
136                 // current node has no such child, so add new one corresponding to
137                 // positions pos onwards of s
138                 current.addChild(pos, stringLen, i);
139                 break;
140             }
141             else {
142                 // try to match s[node.getLeftLabel()+1..node.getRightLabel()] with
143                 // segment of s starting at position pos+1
144                 j = next.getLeftLabel() + 1;
145                 k = pos + 1;
146 
147                 while (j <= next.getRightLabel()) {
148                     if (s[j] == s[k]) {
149                         j++;
150                         k++;
151                     }
152                     else
153                         break;
154                 }
155                 if (j > next.getRightLabel()) {
156                     // succeeded in matching whole segment, so go further down tree
157                     pos = k;
158                     current = next;
159                 }
160                 else {
161                     /* succeeded in matching s[next.getLeftLabel()..j-1] with
162                      * s[pos..k-1].  Split the node next so that its right label is
163                      * now j-1.  Create two children of next: (1) corresponding to
164                      * suffix i, with left label k and right label s.length-1,
165                      * and (2) with left label j and right label next.getRightLabel(),
166                      * whose children are those of next (if any), and whose suffix
167                      * number is equal to that of next. */
168 
169                     SuffixTreeNode n1 = new SuffixTreeNode(null, null, k, stringLen, i);
170                     SuffixTreeNode n2 = new SuffixTreeNode(next.getChild(), n1,
171                                                            j, next.getRightLabel(),
    next.getSuffix());
172                     // now update next's right label, list of children and suffix number
173                     next.setRightLabel(j-1);
174                     next.setChild(n2);
175                     next.setSuffix(-1); // next is now an internal node
176                     break;
177                 }
178             }
179         }
180     }
181 
182     /**
```

```
183      * Gets the root node.
184      *
185      * @return the root node
186      */
187     public SuffixTreeNode getRoot() { return root; }
188 
189     /**
190      * Sets the root node.
191      *
192      * @param node the new root node
193      */
194     public void setRoot(SuffixTreeNode node) { root = node; }
195 
196     /**
197      * Gets the string represented by the suffix tree.
198      *
199      * @return the string represented by the suffix tree
200      */
201     public byte[] getString() { return s; }
202 
203     /**
204      * Sets the string represented by the suffix tree.
205      *
206      * @param sInput the new string represented by the suffix tree
207      */
208     public void setString(byte [] sInput) { s = sInput; }
209 
210     /**
211      * Gets the length of the string represented by the suffix tree.
212      *
213      * @return the length of the string represented by the suffix tree
214      */
215     public int getStringLen() { return stringLen; }
216 
217     /**
218      * Sets the length of the string represented by the suffix tree.
219      *
220      * @param len the new length of the string represented by the suffix tree
221      */
222     public void setStringLen(int len) { stringLen = len; }
223 }
224 
```

```java
package SuffixTreePackage;

/**
 * Class with methods for carrying out applications of suffix trees
 * David Manlove, Jan 03.  Modified by David Manlove, Jan 07 and Jan 09.
 */

public class SuffixTreeAppl {

    /** The suffix tree */
    private SuffixTree t;

    /**
     * Default constructor.
     */
    public SuffixTreeAppl () {
        t = null;
    }

    /**
     * Constructor with parameter.
     *
     * @param tree the suffix tree
     */
    public SuffixTreeAppl (SuffixTree tree) {
        t = tree;
    }

    /**
     * Search the suffix tree t representing string s for a target x.
     * Stores -1 in Task1Info.pos if x is not a substring of s,
     * otherwise stores p in Task1Info.pos such that x occurs in s
     * starting at s[p] (p counts from 0)
     * - assumes that characters of s and x occupy positions 0 onwards
     *
     * @param x the target string to search for
     *
     * @return a Task1Info object
     */
    public Task1Info searchSuffixTree(byte[] x) {

        Task1Info task1 = new Task1Info();
        int pos, j, len;
        SuffixTreeNode current;
        pos = 0;  // position in x
        current = t.getRoot();
        len = x.length-1;

        while (true) {
            // search for child which has left edge label equal to our current position in x
            current = t.searchList(current.getChild(), x[pos]);

            // if no matches are found terminate unsuccessfully
            // otherwise if all characters have been matched terminate successfully
            // otherwise continue to check values between left edge label of next node
            if (current == null) {
                task1.setPos(-1);
                return task1;
            } else if (pos == len) {
                task1.setPos(current.getLeftLabel()-len);
                task1.setMatchNode(current);
```

```
62                  return task1;
63              }
64              else {
65                  // move to next character and label to continue checking edge
66                  j = current.getLeftLabel() + 1;
67                  pos++;
68
69                  // enters only if the edge above node has multiple characters
70                  // terminates when string found or edge label matches fully
71                  while (j <= current.getRightLabel()) {
72                      if (t.getString()[j] == x[pos]) {
73                          if (pos == len) {
74                              task1.setMatchNode(current);
75                              task1.setPos(j-len);
76                              return task1;
77                          } else {
78                              j++;
79                              pos++;
80                          }
81                      }
82                      else {
83                          task1.setPos(-1);
84                          return task1;
85                      }
86                  }
87                  // succeeded in matching whole edge, go further down tree
88              }
89          }
90      }
91
92      /**
93       * Search suffix tree t representing string s for all occurrences of target x.
94       * Stores in Task2Info.positions a linked list of all such occurrences.
95       * Each occurrence is specified by a starting position index in s
96       * (as in searchSuffixTree above).  The linked list is empty if there
97       * are no occurrences of x in s.
98       * - assumes that characters of s and x occupy positions 0 onwards
99       *
100      * @param x the target string to search for
101      *
102      * @return a Task2Info object
103      */
104     public Task2Info allOccurrences(byte[] x) {
105
106         SuffixTreeAppl suffixTreeAppl = new SuffixTreeAppl(t);
107         Task1Info task1 = suffixTreeAppl.searchSuffixTree(x);
108
109         // matching suffix node means all leaf nodes below are matches
110         SuffixTreeNode start = task1.getMatchNode();
111
112         Task2Info task2 = new Task2Info();
113         if (start == null) {
114             // if no match exists
115             return task2;
116         } else if (start.getSuffix() == -1) {
117             // if multiple instances exist call recursive function to count it
118             task2 = recLeafCount(start.getChild(), task2);
119         } else {
120             // if string is unique return its position
121             task2.addEntry(start.getSuffix());
122         }
123         return task2;
```

```
124     }
125
126     // return a task 2 object with positions of every occurrence of a leaf node
127     public Task2Info recLeafCount(SuffixTreeNode current, Task2Info task2) {
128         Task2Info empty = new Task2Info();
129         // return an empty object to the previous call if no node exists
130         // otherwise is node is a non leaf so we check its child
131         // otherwise add an entry
132         if (current == null) {
133             return empty;
134         } else if (current.getSuffix() == -1 ) {
135             recLeafCount(current.getChild(), task2);
136         } else {
137             task2.addEntry(current.getSuffix());
138         }
139         // repeat process on sibling node
140         recLeafCount(current.getSibling(), task2);
141         return task2;
142     }
143
144     /**
145      * Traverses suffix tree t representing string s and stores ln, p1 and
146      * p2 in Task3Info.len, Task3Info.pos1 and Task3Info.pos2 respectively,
147      * so that s[p1..p1+ln-1] = s[p2..p2+ln-1], with ln maximal;
148      * i.e., finds two embeddings of a longest repeated substring of s
149      * - assumes that characters of s occupy positions 0 onwards
150      * so that p1 and p2 count from 0
151      *
152      * @return a Task3Info object
153      */
154     public Task3Info traverseForLrs () {
155
156         Task3Info task3 = new Task3Info();
157         SuffixTreeNode root = t.getRoot();
158
159         // if 0 children from root
160         if (root.getChild() == null) {
161             return task3;
162         }
163         task3 = recFindLrs(task3, root, 0);
164         return task3;
165     }
166
167     // recursively check for a node with 2 or more children with maximum depth
168     public Task3Info recFindLrs(Task3Info task3, SuffixTreeNode current, int depth) {
169         SuffixTreeNode child = current.getChild();
170
171         // case for only 1 child leaf
172         if (child.getSuffix() != -1 && child.getSibling() == null ) {
173             // not a repeated substring so we return
174             return task3;
175         // case for 1 child non leaf
176         } else if (child.getSuffix() == -1 && child.getSibling() == null ){
177             // node could have a repeated substring below it so continue downward
178             depth += current.getRightLabel() - current.getLeftLabel() + 1;
179             task3 = recFindLrs(task3, child, depth);
180         } else {
181             // if more than 1 child update depth
182             depth += current.getRightLabel() - current.getLeftLabel() + 1;
183
184             // if child is non leaf continue downward
185             // otherwise check its sibling
```

```
186             while (child != null) {
187                 if (child.getSuffix() == -1 ) {
188                     task3 = recFindLrs(task3, child, depth);
189                 }
190                 child = child.getSibling();
191 
192             }
193             // check if depth is more than current length
194             // if so replace length and occurrence locations
195             if (task3.getLen() < depth) {
196                 task3.setLen(depth-1);
197                 task3.setPos1(current.getChild().getLeftLabel()-task3.getLen());
198                 task3.setPos2(current.getChild().getSibling().getLeftLabel()-task3.getLen
    ());
199             }
200 
201         }
202         return task3;
203     }
204 
205     /**
206      * Traverse generalised suffix tree t representing strings s1 (of length
207      * s1Length), and s2, and store ln, p1 and p2 in Task4Info.len,
208      * Task4Info.pos1 and Task4Info.pos2 respectively, so that
209      * s1[p1..p1+ln-1] = s2[p2..p2+ln-1], with len maximal;
210      * i.e., finds embeddings in s1 and s2 of a longest common substring
211          * of s1 and s2
212      * - assumes that characters of s1 and s2 occupy positions 0 onwards
213      * so that p1 and p2 count from 0
214      *
215      * @param s1Length the length of s1
216      *
217      * @return a Task4Info object
218      */
219     public Task4Info traverseForLcs (int s1Length) {
220         Task4Info task4 = new Task4Info();
221         SuffixTreeNode root = t.getRoot();
222 
223         // if 0 children from root
224         if (root.getChild() == null) {
225             return task4;
226         }
227         task4 = recFindLcs(task4, root, 0, s1Length+1);
228         return task4;
229     }
230 
231     // recursively check for a node with 2 or more children from opposing strings with
    maximum depth
232     public Task4Info recFindLcs(Task4Info task4, SuffixTreeNode current, int depth, int
    splitPoint) {
233         SuffixTreeNode child = current.getChild();
234 
235         // case for only 1 child leaf
236         if (child.getSuffix() != -1 && child.getSibling() == null ) {
237             // not a repeated substring so we return
238             return task4;
239         // case for 1 child non leaf
240         } else if (child.getSuffix() == -1 && child.getSibling() == null ){
241             // could have a substring below it so continue downward
242             depth += current.getRightLabel() - current.getLeftLabel() + 1;
243             task4 = recFindLcs(task4, child, depth, splitPoint);
244         } else {
```

```
245             // if more than 1 child update depth
246             depth += current.getRightLabel() - current.getLeftLabel() + 1;
247             task4.setString1Leaf(false);
248             task4.setString2Leaf(false);
249             int s1Location = 0;
250             int s2Location = 0;
251 
252             // if child is non leaf continue downward
253             // otherwise check its sibling
254             while (child != null) {
255                 if (child.getSuffix() == -1 ) {
256                     task4 = recFindLcs(task4, child, depth, splitPoint);
257                 }
258                 // update booleans to keep track of the node
259                 //and also note the edge for which it holds true
260                 if (child.getLeftLabel() <= splitPoint) {
261                     task4.setString1Leaf(true);
262                     s1Location = child.getLeftLabel();
263                 } else {
264                     task4.setString2Leaf(true);
265                     s2Location = child.getLeftLabel();
266                 }
267                 child = child.getSibling();
268 
269             }
270             // check if depth is more than current length
271             // if so replace length and occurrence locations
272             if (task4.getString1Leaf() == true && task4.getString2Leaf() == true &&
    task4.getLen() < depth) {
273                 task4.setLen(depth-1);
274                 task4.setPos1(s1Location-task4.getLen());
275                 task4.setPos2(s2Location-task4.getLen()-splitPoint);
276             }
277 
278         }
279         return task4;
280     }
281 
282 }
283 
```