```
import java.util.*;
import SuffixTreePackage.*;

/**
 * Main class – for accessing suffix tree applications
 * David Manlove, Jan 03.  Modified by David Manlove, Jan 07 and Jan 09.
 */

public class Main {

        /**
         * The main method.
         * @param args the arguments
         */
        static SuffixTreeAppl theTree;
        static byte[] treeBytes;
        static byte[] searchTerm;

        public static void displayIndexWarning(){
                System.out.println("NB. All indexes start counting from 0 upwards");
        }

        public static void main(String args[]) {
                Scanner standardInput = new Scanner(System.in);
                do {
                        System.out.println();
                        System.out.print("Enter the number of the task or type 'q' to quit: ");

                        String line = standardInput.nextLine();
                        System.out.println();
                        try {
                                // try to extract an integer from line if possib
le
                                int numTask = Integer.parseInt(line);


                                switch (numTask) {
                                case 1:
                                        System.out.print("What file would you like to sear
ch: ");
                                        treeBytes = new FileInput(standardInput.
nextLine()).readFile();
                                        System.out.print("What would you like to search t
he tree for: ");
                                        searchTerm = standardInput.nextLine().ge
tBytes();
                                        theTree = new SuffixTreeAppl(new SuffixT
ree(treeBytes));
                                        Task1Info result1 = theTree.searchSuffix
Tree(searchTerm);
                                        if (result1.getPos() == -1){
                                                System.out.println("The string \"" +
 new String(searchTerm) + "\" does not occur");
                                        }else{
                                                System.out.println("The string \"" +
 new String(searchTerm) + "\" occurs at position " + result1.getPos());
                                                displayIndexWarning();
                                        }
                                        break;
                                case 2:
                                        System.out.print("What file would you like to sear
ch: ");
```



```
                                        treeBytes = new FileInput(standardInput.
nextLine()).readFile();
                                        System.out.print("What would you like to search t
he tree for: ");
                                        searchTerm = standardInput.nextLine().ge
tBytes();
                                        theTree = new SuffixTreeAppl(new SuffixT
ree(treeBytes));
                                        //System.out.println(new String(searchTe
rm));
                                        Task2Info result2 = theTree.allOccurrenc
es(searchTerm);
                                        if (result2.getPositions().isEmpty()){
                                                System.out.println("The string \"" +
 new String(searchTerm) + "\" does not occur");
                                        }else{
                                                System.out.println("The string \"" +
 new String(searchTerm) + "\" occurs " + result2.getPositions().size() + " times at posit
ions: ");
                                                for (int x : result2.getPosition
s()){
                                                        System.out.println(x);
                                                }
                                                displayIndexWarning();
                                        }
                                        break;
                                case 3:
                                        System.out.print("What file would you like to sear
ch: ");
                                        treeBytes = new FileInput(standardInput.
nextLine()).readFile();
                                        theTree = new SuffixTreeAppl(new SuffixT
ree(treeBytes));
                                        Task3Info result3 = theTree.traverseForL
rs();
                                        String str = "";
                                        int pos1 = result3.getPos1(); int pos2 =
 result3.getPos2();
                                        int len = result3.getLen();
                                        if (len!=0){
                                                str = new String(theTree.getTree
().getString()).substring(pos1, pos1+len);
                                        }
                                        if (len == 0){
                                                System.out.println("There are no repe
ating substrings");
                                        }else{
                                                System.out.printf("Longest Repeating
Substring is: %s\nIts length is %d\nOne occurrence is at position %d\nAnother occurrence is at position %d\n", str
,len,pos1,pos2);
                                                displayIndexWarning();
                                        }
                                        break;
                                case 4:
                                        System.out.print("What is the first file would you l
ike to search: ");
                                        String file1Name = standardInput.nextLin
e();
                                        treeBytes = new FileInput(file1Name).rea
dFile();
                                        System.out.print("What is the second file would yo
u like to search: ");
```

```
                                        String file2Name = standardInput.nextLin
e();
                                        byte[] tree2Bytes = new FileInput(file2N
ame).readFile();
                                        theTree = new SuffixTreeAppl(new SuffixT
ree(treeBytes, tree2Bytes));
                                        Task4Info res = theTree.traverseForLcs(t
reeBytes.length);
                                        if (res.getLen() == 0){
                                                System.out.println("There are no com
mon substrings");
                                        }else{
                                                System.out.println("Longest Common
 Substring is: \"" + new String(tree2Bytes).substring(res.getPos1(), res.getLen() + re
s.getPos1()) + "\" \nIts length is " + res.getLen());
                                                System.out.printf("Occurring at positi
on %d in %s and position %d in %s\n",res.getPos2(), file1Name, res.getPos1(), file2Name);
                                                displayIndexWarning();
                                        }
                                        break;
                                /* replace the above four lines with code to dis
play relevant
                                 * output for each task
                    *
                                 * in the case of Tasks 1, 2 and 3, get the name
 of a text file
                                 * from standard input; in the case of Task 4, g
et the names of
                                 * two text files from standard input
                                 * then, in all cases, read the data from the te
xt file(s) using
                                 * the FileInput class and build the relevant su
ffix tree
                                 * in the case of Tasks 1 and 2, get a string fr
om standard input
                                 * and convert the string to bytes, with the rel
evant information
                                 * stored in the array of bytes from positions 0
 onwards
                                 * then call the relevant method from above to p
rocess the
                                 * information, and display the output using Sys
tem.out.print
                                 * and System.out.println */
                                default: throw new NumberFormatException();
                                }
                        }
                        catch (NumberFormatException e) {
                                if (line.length()==0 || line.charAt(0)!='q')
                                        System.out.println("You must enter either '1', '2'
, '3', '4' or 'q'.");
                                else
                                        break;
                        }
                } while (true);
                standardInput.close();
        }
```



```
}
```

```
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 c
haracter). */
	private int stringLen;

	/**
	 * Builds the suffix tree for a given string.
	 *
	 * @param sInput the string whose suffix tree is to be built
	 * - assumes that '$' does not occur as a character anywhere in sInput
	 * - assumes that characters of sInput occupy positions 0 onwards
	 */
	public SuffixTree (byte [] sInput) {
		root = new SuffixTreeNode(null, null, 0, 0, -1);  // create root
 node of suffix tree;
		stringLen = sInput.length;
		s = new byte[stringLen + 1]; // create longer byte array ready f
or termination character
		System.arraycopy(sInput, 0, s, 0, stringLen);
		s[stringLen] = (byte) '$';   // 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 sI
nput1 or sInput2
	 * - assumes that characters of sInput1 and sInput2 occupy positions 0 o
nwards
	 */
	public SuffixTree (byte[] sInput1, byte[] sInput2) {
	 // to be completed!
		root = new SuffixTreeNode(null, null, 0, 0, -1);
```



```
		s = new byte[sInput1.length + sInput2.length + 2];
		System.arraycopy(sInput1, 0, s, 0, sInput1.length);
		s[sInput1.length] = (byte) '#';
		System.arraycopy(sInput2, 0, s, sInput1.length + 1, sInput2.leng
th);
		s[sInput1.length + sInput2.length + 1] = (byte) '$';
		stringLen = sInput1.length + sInput2.length + 1;
		buildSuffixTree();
	}

	/**
	 * Builds the suffix tree.
	 */
	private void buildSuffixTree() {
		try {
			for (int i=0; i<= stringLen; i++) {
				// for large files, the following line may be us
eful for
				// indicating the progress of the suffix tree co
nstruction
				//if (i % 10000==0) System.out.println(i);

				// raise an exception if the text file contained
 a '$'
				if (s[i] == (byte) '$' && i < stringLen)
					throw new Exception();
				else
					insert(i);  // insert suffix number i of
 z into tree
			}
		}
		catch (Exception e) {
			System.out.println("Text file contains a $ character!");
			System.exit(-1);
		}
	}

	/**
	 * Given node nodeIn of suffix tree and character ch, search nodeIn,
	 * plus all sibling nodes of nodeIn, looking for a node whose left
	 * label x satisfies ch == s[x].
	 * - Assumes that characters of s occupy positions 0 onwards
	 *
	 * @param nodeIn a node of the suffix tree
	 * @param ch the character to match
	 *
	 * @return the matching suffix tree node (null if none exists)
	 */
	public SuffixTreeNode searchList (SuffixTreeNode nodeIn, byte ch) {

		SuffixTreeNode next = nodeIn;
		SuffixTreeNode nodeOut = null;

		while (next != null) {
			if (next.getLeftLabel() < stringLen && s[next.getLeftLab
el()] == ch)
			{
				nodeOut = next;
				next = null;
			}
			else
				next = next.getSibling();
```

```
                }
                return nodeOut;  // return matching node if successful, or null otherwise
        }

        /**
         * Inserts suffix number i of s into suffix tree.
     * - assumes that characters of s occupy positions 0 onwards
         *
         * @param i the suffix number of s to insert
         */
        private void insert(int i) {

                int pos, j, k;
                SuffixTreeNode current, next;
                pos = i;  // position in s
                current = root;

                while (true) {
                        // search for child of current with left label x such that s[x]==s[pos]
                        next = searchList(current.getChild(), s[pos]);

                        if (next == null) {
                                // current node has no such child, so add new one corresponding to
                                // positions pos onwards of s
                                current.addChild(pos, stringLen, i);
                                break;
                        }
                        else {
                                // try to match s[node.getLeftLabel()+1..node.getRightLabel()] with
                                // segment of s starting at position pos+1
                                j = next.getLeftLabel() + 1;
                                k = pos + 1;

                                while (j <= next.getRightLabel()) {
                                        if (s[j] == s[k]) {
                                                j++;
                                                k++;
                                        }
                                        else
                                                break;
                                }
                                if (j > next.getRightLabel()) {
                                        // succeeded in matching whole segment, so go further down tree
                                        pos = k;
                                        current = next;
                                }
                                else {
                                        /* succeeded in matching s[next.getLeftLabel()..j-1] with
                                         * s[pos..k-1].  Split the node next so
                                         * now j-1.  Create two children of next
                                         * suffix i, with left label k and right
                                         * and (2) with left label j and right label next.getRightLabel(),
```



```
                                         * whose children are those of next (if any), and whose suffix
                                         * number is equal to that of next. */

                                        SuffixTreeNode n1 = new SuffixTreeNode(null, null, k, stringLen, i);
                                        SuffixTreeNode n2 = new SuffixTreeNode(next.getChild(), n1,
                                                j, next.getRightLabel(), next.getSuffix());
                                        // now update next's right label, list of children and suffix number
                                        next.setRightLabel(j-1);
                                        next.setChild(n2);
                                        next.setSuffix(-1); // next is now an internal node
                                        break;
                                }
                        }
                }
        }

        /**
         * Gets the root node.
         *
         * @return the root node
         */
        public SuffixTreeNode getRoot() { return root; }

        /**
         * Sets the root node.
         *
         * @param node the new root node
         */
        public void setRoot(SuffixTreeNode node) { root = node; }

        /**
         * Gets the string represented by the suffix tree.
         *
         * @return the string represented by the suffix tree
         */
        public byte[] getString() { return s; }

        /**
         * Sets the string represented by the suffix tree.
         *
         * @param sInput the new string represented by the suffix tree
         */
        public void setString(byte [] sInput) { s = sInput; }

        /**
         * Gets the length of the string represented by the suffix tree.
         *
         * @return the length of the string represented by the suffix tree
         */
        public int getStringLen() { return stringLen; }

        /**
         * Sets the length of the string represented by the suffix tree.
         *
         * @param len the new length of the string represented by the suffix tree
```

```
         */
        public void setStringLen(int len) { stringLen = len; }
}
```



```
package SuffixTreePackage;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

/**
 * 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;
        private SuffixTreeNode t2Node = null;
        private LinkedList<SuffixTreeNode> occurrences = null;

        /**
         * Default constructor.
         */
        public SuffixTreeAppl () {
                t = null;
                //t2Node = null;
                //occurrences = new LinkedList<SuffixTreeNode>();
        }

        public SuffixTree getTree(){
                return t;
        }

        /**
         * 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 result = new Task1Info();
                boolean match = true;
                int startLocation = -1;
                int xIndex = 0;
                int nodeIndex = 0;
                int lengthAtNode;
                result.setPos(-2);
                SuffixTreeNode currentNode = t.getRoot().getChild();

                while (currentNode != null){
```

```
                        lengthAtNode = currentNode.getRightLabel() - currentNode
.getLeftLabel() + 1;
                        //checks x with all the values at the currentNode
                        nodeIndex = 0;
                        match = true;
                        while (nodeIndex < lengthAtNode && match && xIndex < x.l
ength){
                                int i = (currentNode.getLeftLabel()) + nodeIndex
;
                                if (x[xIndex] != t.getString()[i]){
                                        match = false;
                                        result.setPos(-1);
                                        startLocation = -1;
                                }else{
                                        if (startLocation == -1){
                                                startLocation = currentNode.getL
eftLabel();

                                                //needed for Task2Info, added he
re necessary for very small trees that have
                                                //very shallow depths.
                                                t2Node = currentNode;
                                        }
                                        xIndex++;
                                        nodeIndex++;
                                }
                                //nodeIndex++;
                                //xIndex++;
                        }
                        if (match){
                                if (currentNode.getChild() != null){
                                        /*t2Node is a global variable that is us
ed when calulating the Task2Info
                                         * this is because most of this code is
useful and the redundancy for Task2Info (ie. setting a pos)
                                         * does not leave a great overhead (imo
this is better than copying the code).
                                         */
                                        System.out.println("Here in code");
                                        if (currentNode != null){
                                                t2Node = currentNode;
                                        }
                                }
                                currentNode = currentNode.getChild();
                        }else{
                                currentNode = currentNode.getSibling();
                        }
                }
                if (match == true){
                        result.setPos(startLocation);
                }

                return result;
                        // if mismatch at current node check all of the siblings
 until no more siblings
                                //at which point set match to false.
                        // if a match at current node then set currentNode to be
 the child of that node (recurse)

        }

        /**
```



```
         * Search suffix tree t representing string s for all occurrences of tar
get x.
         * Stores in Task2Info.positions a linked list of all such occurrences.
         * Each occurrence is specified by a starting position index in s
         * (as in searchSuffixTree above).  The linked list is empty if there
         * are no occurrences of x in s.
         * - assumes that characters of s and x occupy positions 0 onwards
         *
         * @param x the target string to search for
         *
         * @return a Task2Info object
         */
        public Task2Info allOccurrences(byte[] x) {
                //update the t2Node
                searchSuffixTree(x);
                occurrences = new LinkedList<SuffixTreeNode>();
                if (t2Node == null){
                        //no occurrences.
                        return new Task2Info();

                //else if needed in case there is only 1 occurrence
                }else if(t2Node.getSuffix() != -1){
                        occurrences.add(t2Node);
                }
                //get all occurrences.
                getLeafDecendants(t2Node.getChild());
                Task2Info res = new Task2Info();
                for (SuffixTreeNode node: occurrences){
                        //add all occurrences
                        res.addEntry(node.getSuffix());
                }
                return res;
        }

        public void getLeafDecendants(SuffixTreeNode currentNode){
                while (currentNode != null){
                        //add suffix and go to sibling
                        if (currentNode.getChild() == null){
                                occurrences.add(currentNode);
                                currentNode = currentNode.getSibling();
                        //explore children for suffixes and then go to sibling
                        }else{
                                getLeafDecendants(currentNode.getChild());
                                currentNode = currentNode.getSibling();

                        }
                }
        }

        /**
         * Traverses suffix tree t representing string s and stores ln, p1 and
         * p2 in Task3Info.len, Task3Info.pos1 and Task3Info.pos2 respectively,
         * so that s[p1..p1+ln-1] = s[p2..p2+ln-1], with ln maximal;
         * i.e., finds two embeddings of a longest repeated substring of s
         * - assumes that characters of s occupy positions 0 onwards
         * so that p1 and p2 count from 0
         *
         * @return a Task3Info object
         */
        public Task3Info traverseForLrs () {
                SuffixTreeNode currentNode = t.getRoot().getChild();
                LinkedList<SuffixTreeNode> bestLsrNodes = new LinkedList<SuffixT
```

```
reeNode>();
                LinkedList<SuffixTreeNode> currentPath = new LinkedList<SuffixTr
eeNode>();
                //initialise the path equal to the furthest descendent
                currentPath.addLast(currentNode);

                while (currentPath != null){
                        //last item in the path (potential "vaild" node")
                        currentNode = currentPath.getLast();

                        if (getLength(currentPath) > getLength(bestLsrNodes) &&
isValidBranch(currentNode)){
                                //remove previous searches
                                bestLsrNodes.clear();
                                for (SuffixTreeNode n: currentPath){
                                        //update bestLsrNodes
                                        bestLsrNodes.addLast(n);
                                }
                        }
                        //update the path
                        currentPath = next(currentPath);
                }
                Task3Info result = new Task3Info();
                if (bestLsrNodes.isEmpty()){
                        //if no lrs exists...
                        return result;
                }else{
                        //otherwise set positions and return
                        int[] positions = getLeafSuffixes(bestLsrNodes.getLast()
);
                        result.setPos1(positions[0]);
                        result.setPos2(positions[1]);
                        result.setLen(getLength(bestLsrNodes));
                        return result;
                }
        }

        /*Gets 2 the leaf nodes of a branch node, should be checked with
        isValidBranch(x) first. (when immediate parent, check with isValidBranch
(x) first)*/
        private int[] getLeafSuffixes(SuffixTreeNode x){
                int[] result = new int[2];
                SuffixTreeNode currentNode = x.getChild();
                result[0] = currentNode.getSuffix();
                currentNode = currentNode.getSibling();
                result[1] = currentNode.getSuffix();
                return result;
        }

        //Gets the collective length of all of the nodes in a list.
        public int getLength(LinkedList<SuffixTreeNode> x){
                int len = 0;
                if (x.isEmpty()){
                        return 0;
                }
                for (SuffixTreeNode node: x){
                        len += getLengthOfNode(node);
                }
                return len;
        }

        /*
```



```
         * Iterates through the tree, returning a list representing the "childre
n"
         * with each the last node being the current node and the preceding node
s being the parents.
         * The iteration is done depth first.
         */
        public LinkedList<SuffixTreeNode> next(LinkedList<SuffixTreeNode> x){
                SuffixTreeNode currentNode = x.getLast();
                //necessary when visiting parents as it prevents them from immed
iately visiting the child.
                boolean visited = false;

                /*used as a loop escape, once the path is at an acceptable confi
guration,
                 * required as sometimes the parents must be visited several tim
es.
                 */
                boolean isBad = true;
                while (isBad || x.isEmpty()){

                        //Go to the child if not already visited.
                        if(currentNode.getChild() != null && !visited){
                                x.addLast(currentNode.getChild());
                                isBad = false;

                        //If not, then go to the sibling if possible
                        }else if (currentNode.getSibling() != null){
                                x.removeLast();
                                x.addLast(currentNode.getSibling());
                                isBad = false;

                        //If not, then go up the list as high as required.
                        }else{
                                if (!x.isEmpty()){
                                        //Go up one step
                                        x.removeLast();
                                        if (!x.isEmpty())
                                                currentNode = x.getLast();
                                        else{
                                                //stop looping if empty
                                                isBad = false;
                                        }
                                }else{
                                        /* break needed so that items are not continual
ly removed from the list
                                        after the list is empty, prevents NoSuch
ElementException */
                                        break;
                                }
                                //prevent from going to the child in next iterat
ion.
                                visited = true;
                        }
                }
                if (x.isEmpty()){
                        //returns null when no more paths are available
                        return null;
                }
                return x;
                /*NB. Multiple handlers of x.isEmpty() are necessary as items ar
e removed from x
                 * at various stages in its execution. All are to prevent either
```

```
                 * NoSuchElementExceptions or to escape the loop.
                 */
        }

        //returns true if a branch ONLY has leaf nodes (at least 2)
        public boolean isValidBranch(SuffixTreeNode x){
                //if the node is a leaf..
                boolean result = true;
                int count = 0;

                //if the node itself is a leaf then return false.
                if (x.getChild() == null){
                        return false;

                //if not a leaf.
                }else{
                        SuffixTreeNode currentNode = x.getChild();
                        while (currentNode != null){
                                //count the number of leaf nodes..
                                if (currentNode.getSuffix() != -1)
                                        count++;
                                currentNode = currentNode.getSibling();
                        }
                }
                //return true if there are at least 2 leaf nodes.
                if (count >= 2){
                        return true;
                }else{
                        return false;
                }
        }

        // simple helper to aid readability, gets the length of a node.
        public int getLengthOfNode(SuffixTreeNode x){
                return x.getRightLabel() - x.getLeftLabel() + 1;
        }



        /**
         * Traverse generalised suffix tree t representing strings s1 (of length
         * s1Length), and s2, and store ln, p1 and p2 in Task4Info.len,
         * Task4Info.pos1 and Task4Info.pos2 respectively, so that
         * s1[p1..p1+ln-1] = s2[p2..p2+ln-1], with len maximal;
         * i.e., finds embeddings in s1 and s2 of a longest common substring
         * of s1 and s2
         * - assumes that characters of s1 and s2 occupy positions 0 onwards
         * so that p1 and p2 count from 0
         *
         * @param s1Length the length of s1
         *
         * @return a Task4Info object
         */
        public Task4Info traverseForLcs (int s1Length) {
                //Very similar to Task3Info, only changed sections will be comme
nted.
                SuffixTreeNode currentNode = t.getRoot().getChild();
                LinkedList<SuffixTreeNode> bestLcrNodes = new LinkedList<SuffixT
reeNode>();
                LinkedList<SuffixTreeNode> currentPath = new LinkedList<SuffixTr
eeNode>();
```



```
                LinkedList<SuffixTreeNode> tempLeaves = new LinkedList<SuffixTre
eNode>();
                currentPath.addLast(currentNode);
                boolean isLast = false;
                boolean left = false;
                boolean right = false;
                while (currentPath != null || isLast){
                        currentNode = currentPath.getLast();
                        if (getLength(currentPath) > getLength(bestLcrNodes) &&
isValidBranch(currentNode)){
                                tempLeaves = getImmidiateLeafDescs(currentNode);
                                for (SuffixTreeNode node:tempLeaves){
                                        //if a node is in the first file set lef
t to true
                                        if (node.getSuffix() < s1Length){
                                                left = true;
                                        //if a node is in the second file set ri
ght to true
                                        }else if(node.getSuffix() > s1Length){
                                                right = true;
                                        }
                                }
                                //ONLY is subtring is present in BOTH the first
and second files.
                                if (left && right){
                                        //reset bestLcrNodes to empty
                                        bestLcrNodes.clear();
                                        //Update bestLcrNodes
                                        for (SuffixTreeNode n: currentPath){
                                                bestLcrNodes.addLast(n);
                                        };
                                }
                                //Reset left and right values as we move to a ne
w node.
                                left = false;
                                right = false;
                        }
                        currentPath = next(currentPath);
                }
                Task4Info result = new Task4Info();
                if (bestLcrNodes.isEmpty()){
                        return result;
                }else{
                        int[] positions = getLeafSuffixes(bestLcrNodes.getLast()
);
                        int which = -1;


                        //Index movements relative to the position in THEIR resp
ective files.
                        if (positions[0] > s1Length){
                                positions[0]-= (s1Length + 1);
                                which = 0;
                        }else if (positions[1] > s1Length){
                                positions[1]-= (s1Length + 1);
                                which = 1;
                        }

                        //Set the data for return
                        if (which == 0){
                                result.setPos1(positions[0]);
                                result.setPos2(positions[1]);
                        }else if (which == 1){
```

```
                        result.setPos1(positions[1]);
                        result.setPos2(positions[0]);
                }
                result.setLen(getLength(bestLcrNodes));
                return result;
        }
    }

        private LinkedList<SuffixTreeNode> getImmidiateLeafDescs(SuffixT
reeNode x){
                LinkedList<SuffixTreeNode> leaves = new LinkedList<Suffi
xTreeNode>();
                // if x itself is a lead return null
                if (x.getChild() == null){
                        return null;

                //if a branch...
                }else{
                        //examine all children and siblings of the branc
h
                        x = x.getChild();
                        while (x != null){
                                //if a leaf then add to the list
                                if (x.getChild() == null){
                                        leaves.add(x);
                                }
                                x = x.getSibling();
                        }
                        return leaves;
                }
        }
}
```