

Main.java

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

/**
 * Main class - for accessing suffix tree applications
 * @author David ManLove
 */

public class Main {

    /**
     * The main method.
     * @param args the arguments
     */
    public static void main(String args[]) {
        String errorMessage = "Required syntax:\n";
        errorMessage += "  java Main SearchOne <filename> <query string> for Task 1\n";
        errorMessage += "  java Main SearchAll <filename> <query string> for Task 2\n";
        errorMessage += "  java Main LRS <filename> for Task 3\n";
        errorMessage += "  java Main LCS <filename1> <filename2> for Task 4";

        if (args.length < 2)
            System.out.println(errorMessage);
        else {
            // Get the command from the first argument
            String command = args[0];

            switch (command) {
                case "SearchOne": {
```

```
if (args.Length < 3) {
    System.out.println(errorMessage);
    break;
}

// Read filename and query string from arguments
String fileName = args[1];
String queryString = args[2];

// Read file content into byte array
FileInput fileInput = new FileInput(fileName);
byte[] text = fileInput.readFile();

// Build suffix tree for the text
SuffixTree tree = new SuffixTree(text);

// Create application object and perform search
SuffixTreeAppl appl = new SuffixTreeAppl(tree);
Task1Info result = appl.searchSuffixTree(queryString.getBytes());

// Display results
if (result.getPos() == -1) {
    System.out.println("Search string \"" + queryString +
        "\" not found in " + fileName);
} else {
    System.out.println("Search string \"" + queryString +
        "\" occurs at position " + result.getPos() +
        " of " + fileName);
}
break;
}

case "SearchAll": {
```

```
if (args.Length < 3) {
    System.out.println(errorMessage);
    break;
}

// Read filename and query string from arguments
String fileName = args[1];
String queryString = args[2];

// Read file content into byte array
FileInput fileInput = new FileInput(fileName);
byte[] text = fileInput.readFile();

// Build suffix tree for the text
SuffixTree tree = new SuffixTree(text);

// Create application object and find all occurrences
SuffixTreeAppl appl = new SuffixTreeAppl(tree);
Task2Info result = appl.allOccurrences(queryString.getBytes());

// Display results
LinkedList<Integer> positions = result.getPositions();
if (positions.isEmpty()) {
    System.out.println("The string \"" + queryString +
        "\" does not occur in " + fileName);
} else {
    System.out.println("The string \"" + queryString +
        "\" occurs in " + fileName + " at positions:");
    for (Integer pos : positions) {
        System.out.println(pos);
    }
    System.out.println("The total number of occurrences is " +
        positions.size());
}
```

```
        }

        break;
    }

    case "LRS": {
        // Read filename from arguments
        String fileName = args[1];

        // Read file content into byte array
        FileInputStream fileInput = new FileInputStream(fileName);
        byte[] text = fileInput.readAllBytes();

        // Build suffix tree for the text
        SuffixTree tree = new SuffixTree(text);

        // Create application object and find LRS
        SuffixTreeAppl appl = new SuffixTreeAppl(tree);
        Task3Info result = appl.traverseForLrs();

        // Display results
        if (result.getLen() == 0) {
            System.out.println("No repeated substring found in " + fileName);
        } else {
            // Extract the LRS from the original text
            byte[] s = tree.getString();
            int len = result.getLen();
            int pos1 = result.getPos1();
            int pos2 = result.getPos2();

            byte[] lrsBytes = new byte[len];
            System.arraycopy(s, pos1, lrsBytes, 0, len);
            String lrsString = new String(lrsBytes);
```

```
System.out.println("An LRS in " + fileName + " is \\" +  
    lrsString + "\");  
System.out.println("Its length is " + len);  
System.out.println("Starting position of one occurrence is " + pos1);  
System.out.println("Starting position of another occurrence is " + pos2);  
}  
break;  
}  
  
case "LCS": {  
    if (args.Length < 3) {  
        System.out.println(errorMessage);  
        break;  
    }  
  
    // Read two filenames from arguments  
    String fileName1 = args[1];  
    String fileName2 = args[2];  
  
    // Read both file contents into byte arrays  
    FileInputStream fileInput1 = new FileInputStream(fileName1);  
    byte[] text1 = fileInput1.readfile();  
  
    FileInputStream fileInput2 = new FileInputStream(fileName2);  
    byte[] text2 = fileInput2.readfile();  
  
    // Build generalized suffix tree for both texts  
    SuffixTree tree = new SuffixTree(text1, text2);  
  
    // Create application object and find LCS  
    SuffixTreeAppl appl = new SuffixTreeAppl(tree);  
    Task4Info result = appl.traverseForLcs(text1.length);
```

```
// Display results
if (result.getLen() == 0) {
    System.out.println("No common substring found between " +
        fileName1 + " and " + fileName2);
} else {
    // Extract the LCS (from text1)
    int len = result.getLen();
    int pos1 = result.getPos1();
    int pos2 = result.getPos2();

    byte[] lcsBytes = new byte[len];
    System.arraycopy(text1, pos1, lcsBytes, 0, len);
    String lcsString = new String(lcsBytes);

    System.out.println("An LCS of " + fileName1 + " and " +
        fileName2 + " is \"" + lcsString + "\"");
    System.out.println("Its length is " + len);
    System.out.println("Starting position in " + fileName1 +
        " is " + pos1);
    System.out.println("Starting position in " + fileName2 +
        " is " + pos2);
}
break;
}

default: System.out.println(errorMessage);
}
}
}
```

SuffixTree.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 edge-splitting.

 *

 * @author Ela Hunt, Rob Irving, David Manlove

 */

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) {  
  
    // Initialize root node  
  
    root = new SuffixTreeNode(null, null, 0, 0, -1);  
  
    // Calculate lengths  
  
    int Len1 = sInput1.Length;  
  
    int Len2 = sInput2.Length;  
  
    // Total length: s1 + '$' + s2 (without final '#')  
  
    stringLen = Len1 + Len2 + 1;  
  
    // Create combined string: s1 + '$' + s2 + '#'
```

```
s = new byte[stringLen + 1];

// Copy first string
System.arraycopy(sInput1, 0, s, 0, len1);

// Add first terminator '$'
s[Len1] = (byte) '$';

// Copy second string
System.arraycopy(sInput2, 0, s, Len1 + 1, Len2);

// Add second terminator '#'
s[stringLen] = (byte) '#';

// Build the suffix tree
buildSuffixTree();

}

/***
* Builds the suffix tree.
*/
```

```
*/  
  
private void buildSuffixTree() {  
  
    for (int i=0; i<= stringLen; i++) {  
  
        // for large files, the following line may be useful for  
  
        // indicating the progress of the suffix tree construction  
  
        // if (i % 10000==0) System.out.println(i);  
  
        insert(i); // insert suffix number i into tree  
  
    }  
  
}  
  
}
```

```
/**  
  
 * 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.getLeftLabel()] == 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 that its right label is

     * now j-1. Create two children of next: (1) corresponding to

     * suffix i, with left label k and right label s.length-1,

     * 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; }  
  
}
```

SuffixTreeAppl.java

```
package SuffixTreePackage;

import java.util.*;

/**
 * Class with methods for carrying out applications of suffix trees
 * @author David ManLove
 */

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 result = new Task1Info();

    // Handle empty query string
    if (x == null || x.Length == 0) {
        result.setPos(-1);
        return result;
    }

    // Get string and root node from suffix tree
    byte[] s = t.getString();
    SuffixTreeNode currentNode = t.getRoot();
    int xPos = 0; // Current position in query string x

    // Start matching from root
    while (xPos < x.Length) {
        // Look for child edge matching x[xPos]
        SuffixTreeNode childNode = currentNode.getChild();
        SuffixTreeNode matchedEdge = null;

        // Traverse sibling list to find matching edge
        while (childNode != null) {
            if (s[childNode.getLeftLabel()] == x[xPos]) {

```

```

        matchedEdge = childNode;
        break;
    }
    childNode = childNode.getSibling();
}

// No matching edge found - x is not a substring
if (matchedEdge == null) {
    result.setPos(-1);
    return result;
}

// Continue matching along the edge
int edgePos = matchedEdge.getLeftLabel();
int edgeEnd = matchedEdge.getRightLabel();

// Match characters on edge label
while (edgePos <= edgeEnd && xPos < x.Length) {
    if (s[edgePos] != x[xPos]) {
        // Character mismatch
        result.setPos(-1);
        return result;
    }
    edgePos++;
    xPos++;
}

// Check if we have matched all of x
if (xPos == x.Length) {
    // Find any Leaf node under matched node
    int position = findLeafSuffix(matchedEdge);
    result.setPos(position);
    result.setMatchNode(matchedEdge);
}

```

```
        return result;
    }

    // Move down to continue searching
    currentNode = matchedEdge;
}

result.setPos(-1);
return result;
}

/** 
 * Helper method to find a Leaf suffix in the subtree rooted at node.
 *
 * @param node the root of the subtree to search
 * @return the suffix number of a Leaf node, or -1 if not found
 */
private int findLeafSuffix(SuffixTreeNode node) {
    // Check if current node is a Leaf
    if (node.getSuffix() != -1) {
        return node.getSuffix();
    }

    // Recursively search children
    SuffixTreeNode child = node.getChild();
    while (child != null) {
        int suffix = findLeafSuffix(child);
        if (suffix != -1) {
            return suffix;
        }
        child = child.getSibling();
    }
}
```

```
    return -1;
}

/** 
 * Search suffix tree t representing string s for all occurrences of target 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) {
    Task2Info result = new Task2Info();

    // First use Task 1 to find if x exists and get the matched node
    Task1Info task1Result = searchSuffixTree(x);

    // If x is not found, return empty list
    if (task1Result.getPos() == -1) {
        return result;
    }

    // Collect all Leaf suffixes from the subtree of the matched node
    SuffixTreeNode matchNode = task1Result.getMatchNode();
    collectAllLeaves(matchNode, result);

    return result;
}
```

```

/**
 * Helper method to collect all Leaf suffix numbers in the subtree rooted at node.
 * Adds each suffix number to the result's position list.
 *
 * @param node the root of the subtree to traverse
 * @param result the Task2Info object to store positions in
 */
private void collectAllLeaves(SuffixTreeNode node, Task2Info result) {
    // If this is a Leaf node, add its suffix number
    if (node.getSuffix() != -1) {
        result.addEntry(node.getSuffix());
        return;
    }

    // Otherwise, recursively collect from all children
    SuffixTreeNode child = node.getChild();
    while (child != null) {
        collectAllLeaves(child, result);
        child = child.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 () {

```

```

Task3Info result = new Task3Info();

// Start DFS from root with initial depth 0
SuffixTreeNode root = t.getRoot();
findLongestRepeatedSubstring(root, 0, result);

return result;
}

/**
 * Helper method to recursively find the Longest repeated substring.
 * Uses DFS to traverse the tree and tracks string depth at each node.
 *
 * @param node the current node being visited
 * @param depth the string depth from root to this node
 * @param result the Task3Info object to store the best result
 */
private void findLongestRepeatedSubstring(SuffixTreeNode node, int depth, Task3Info result) {
    // Leaf nodes don't represent repeated substrings
    if (node.getSuffix() != -1) {
        return;
    }

    // Internal nodes (non-Leaf, non-root) represent repeated substrings
    // Check if this node gives us a Longer repeated substring
    if (node != t.getRoot() && depth > result.getLen()) {
        // Find two different Leaf nodes in this subtree
        int[] twoLeaves = findTwoDifferentLeaves(node);
        if (twoLeaves[0] != -1 && twoLeaves[1] != -1) {
            result.setLen(depth);
            result.setPos1(twoLeaves[0]);
            result.setPos2(twoLeaves[1]);
        }
    }
}

```

```

}

// Recursively process all children
SuffixTreeNode child = node.getChild();
while (child != null) {
    int edgeLength = child.getRightLabel() - child.getLeftLabel() + 1;
    findLongestRepeatedSubstring(child, depth + edgeLength, result);
    child = child.getSibling();
}
}

/**
* Helper method to find two different Leaf nodes in the subtree.
* Returns their suffix numbers as the two occurrence positions.
*
* @param node the root of the subtree
* @return array of two suffix numbers, or [-1, -1] if not found
*/
private int[] findTwoDifferentLeaves(SuffixTreeNode node) {
    int[] result = new int[] {-1, -1};
    LinkedList<Integer> Leaves = new LinkedList<>();
    collectLeafSuffixes(node, Leaves);

    if (Leaves.size() >= 2) {
        result[0] = Leaves.get(0);
        result[1] = Leaves.get(1);
    }

    return result;
}

/**
* Helper method to collect all Leaf suffix numbers in a subtree.

```

```

*
* @param node the root of the subtree
* @param Leaves the List to store suffix numbers in
*/
private void collectLeafSuffixes(SuffixTreeNode node, LinkedList<Integer> Leaves) {
    if (node.getSuffix() != -1) {
        Leaves.add(node.getSuffix());
        return;
    }

    SuffixTreeNode child = node.getChild();
    while (child != null) {
        collectLeafSuffixes(child, Leaves);
        child = child.getSibling();
    }
}

/**
 * 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) {
    Task4Info result = new Task4Info();

```

```

// First, mark all nodes with information about their Leaf descendants
SuffixTreeNode root = t.getRoot();
markLeafDescendants(root, s1Length);

// Then find the deepest node that has Leaves from both strings
findLongestCommonSubstring(root, 0, s1Length, result);

return result;
}

/** 
 * Helper method to mark each node with information about Leaf descendants.
 * Uses post-order traversal to propagate information from Leaves upward.
 *
 * @param node the current node being processed
 * @param s1Length the length of the first string
 * @return Task4Info containing information about this node's descendants
 */
private Task4Info markLeafDescendants(SuffixTreeNode node, int s1Length) {
    // If this is a Leaf node
    if (node.getSuffix() != -1) {
        int suffix = node.getSuffix();
        if (suffix < s1Length) {
            // Leaf from string 1
            node.setLeafNodeString1(true);
            node.setLeafNodeNumString1(suffix);
            return new Task4Info(0, suffix, -1, true, false);
        } else {
            // Leaf from string 2 (adjust position by subtracting s1Length + 1 for '$')
            node.setLeafNodeString2(true);
            node.setLeafNodeNumString2(suffix - s1Length - 1);
            return new Task4Info(0, -1, suffix - s1Length - 1, false, true);
        }
    }
}

```

```
    }

}

// Internal node: collect information from all children
boolean hasString1 = false;
boolean hasString2 = false;
int pos1 = -1;
int pos2 = -1;

SuffixTreeNode child = node.getChild();
while (child != null) {
    Task4Info childInfo = markLeafDescendants(child, s1Length);

    if (childInfo.getString1Leaf()) {
        hasString1 = true;
        if (pos1 == -1) {
            pos1 = childInfo.getPos1();
        }
    }

    if (childInfo.getString2Leaf()) {
        hasString2 = true;
        if (pos2 == -1) {
            pos2 = childInfo.getPos2();
        }
    }

    child = child.getSibling();
}

// Mark current node with collected information
node.setLeafNodeString1(hasString1);
node.setLeafNodeString2(hasString2);
```

```

if (hasString1) {
    node.setLeafNodeNumString1(pos1);
}
if (hasString2) {
    node.setLeafNodeNumString2(pos2);
}

return new Task4Info(0, pos1, pos2, hasString1, hasString2);
}

/***
 * Helper method to find the longest common substring.
 * Traverses the marked tree to find the deepest node with Leaves from both strings.
 *
 * @param node the current node being visited
 * @param depth the string depth from root to this node
 * @param s1Length the length of the first string
 * @param result the Task4Info object to store the best result
 */
private void findLongestCommonSubstring(SuffixTreeNode node, int depth,
                                       int s1Length, Task4Info result) {
    // If this node has Leaf descendants from both strings
    if (node.getLeafNodeString1() && node.getLeafNodeString2()) {
        // Check if this gives us a longer common substring
        if (depth > result.getLen()) {
            result.setLen(depth);
            result.setPos1(node.getLeafNodeNumString1());
            result.setPos2(node.getLeafNodeNumString2());
        }
    }

    // Recursively process all children
    SuffixTreeNode child = node.getChild();

```

```
        while (child != null) {
            int edgeLength = child.getRightLabel() - child.getLeftLabel() + 1;
            findLongestCommonSubstring(child, depth + edgeLength, s1Length, result);
            child = child.getSibling();
        }
    }
}
```