Algorithmics II (H)

Assessed Exercise 2025-26

Implementation Report

Student Name: KunQi Zhang
Student ID: 2733883Z
Date: 6 November 2025

STATUS REPORT

The program compiles and runs successfully without errors. All four tasks produce the correct output format.

I completed this exercise independently using course materials. No external sources or AI tools were used.

IMPLEMENTATION REPORT

Task 1: Searching for a Substring

Algorithm:
The algorithm traverses the suffix tree from the root, matching the query string character by character along edge labels. When a complete match is found, a helper method getAnyLeaf() finds any leaf node in the matched subtree and returns its suffix number as the occurrance position.

Key Implementation:
Starting from t.getRoot(), I use getChild() and getSibling() to traverse nodes. Edge labels are accessed by getLeftLabel() and getRightLabel() into the string obtained from t.getString(). The recursive helper method searches for any leaf where getSuffix() != -1.

Designs Choice:
Returning the first leaf foand rather than the leftmost occurrence simplifies the code while meeting the specification.

Complexity: O(m) where m is the query string length.

Task 2: Finding All Occurrences

Algorithm:
This extends Task 1 by collecting all leaf nodes in the matched subtrees. After using searchSuffixTree() to find the match point, a helper method getAllLeaves() performs DFS to collect all leaf suffix numbers.

Key Implementation:
The method recursively visits all nodes in the subtree. When getSuffix() != -1, the position is added to the result using Task2Info.addEntry().

Design Choice:
Positions are returned in traversal order without sorting, as the specification doesn't require ordering.

Complexity: O(m + k) where k is the number of occurrences.

Task 3: Longest Repeated Substring

Algorithm:
Internal nodes represent repeated substrings since they have multiple suffix descendants. The algorithm performs DFS tracking string depth (accumulated edge lengths). The deepest internal node represents the longest repeated substring.

Key Implementation:
Starting from root, I traverse all ndes checking getSuffix() == -1 for internal nodes. Edge length is calculated as getRightLabel() - getLeftLabel() + 1. When a deeper internal node is found, getTwoLeaves() collect leaves to provide two occurrence positions.

Design Choice:
The root is excluded (depth 0). Cllecting all leaves and selecting the first two is simpler than optimizing for just two.

Complexity: O(n) where n is text length.

Task 4: Longest Common Substring

Algorithm:
A generalized suffix tree is built by concatenating strings as s1$s2#. The algorithm uses two phases: marking nodes with boolean flags indicating which string(s) their leaves come from, then finding the deepest node with leaves from both strings.

Key Implementation:
The constructor uses System.arraycopy() to build the combined string with terminators. In markNodes(), leaf origin is determined by comparing getSuffix() with s1Length. For string 2 leaves, positions are adjusted by subtracting (s1Length + 1). The findLCS() method identifies nodes where both getLeafNodeString1() and getLeafNodeString2() are true.

Design Choice:
Two-phase approach separates concerns clearly. The distinct terminators '$' and '#' prevent suffix confusion. The buildSuffixTree() method was modified to allow '$' at intermediate positions.

Complexity: O(n1 + n2) where n1 and n2 are string lengths.