Steven Tung
CSE548: Homework 2

Note: Pseudocode was written in Python style because formatting symbols on Google Docs looks ugly and painful so I just copy pasted it from my compiler and formatted it accordingly.

Problem 1

The best approach to this is to use a generalized suffix tree. I will assume a working O(n) function called build_gst. Another assumption to be made is that each node has the following:

- children: list of child nodes
- parent: pointer to its parent or None if root
- sdepth: integer that is the length of substring from root to this node
- color_set: set of which strings occur in its subtree
- is_leaf(): True if no children (trivial, self explanatory)
- leaf_label: which string this leaf corresponds to if leaf

```
def build_gst(strings):
    """
    builds the Generalized Suffix Tree for the given strings
    time complexity: O(n) construction for k = O(1)
    construction of structure is excluded since its a structure in the lecture.

    Assumption:
    each node has children, parent, sdepth, color_set, leaf_label, and is_leaf()
    """
    return GSTroot # returns root of the tree

def compute_color_sets(node, result):
    """
    compute the set of strings that occur in each node's subtree
    and count amount of new distinct substrings contributed by nodes
    that have color_set size 2.

    parameters
    node: current node of the GST
    result_container: global count of substrings
    """
    # leaf belongs to exactly one string which is its leaf_label
    if node.is_leaf():
        node.color_set = {node.leaf_label}
    # not leaf: union of children's color sets
    else:
        node.color_set = set()
        for child in node.children:
```

```
            compute_color_sets(child, result)
            node.color_set |= child.color_set

    # num of strings in which this node's root-to-node substring occurs
    if node.parent is not None:  # skip root
        if len(node.color_set) == 2:
            # add number of new distinct substrings introduced by extending the
parent's substring to this node
            edge_length = node.sdepth - node.parent.sdepth
            result[0] += edge_length

def count_substrings(strings):
    # appends a terminator char like $, #, &, etc
    add_terminals = []
    for i, s in enumerate(strings):
        # could just use a list of terminators but this works for any size set
        add_terminals.append(s + chr(0xF000 + i))
    strings = add_terminals

    # build Generalized Suffix Tree and DFS to compute sets and count
    root = build_gst(strings)
    result = [0]
    compute_color_sets(root, result)
    return result[0]

# example
S = ["abaabb", "abba", "bbaaa"]
print(count_substrings(S)) # should print 5
```

Correctness Analysis

We start by combining all the k strings into a single generalized suffix tree (with every string having its unique terminator symbol), ensuring no suffix of one string continues into another's suffixes. We also have that each leaf corresponds to exactly one suffix of one string (its color set is the string itself). Each internal node's color set is the union of its children's set, indicating which of the k strings appears in that node's subtrees.

Now, we have that a node, v, at string-depth sdepth(v) contributes sdepth(v) - sdepth(parent(v)) new substrings (those that are formed by extending the parent's substring down to v). Only add that contribution to the total count if exactly two distinct strings appear in v's subtree (|color_set(v)|=2) to ensure that only substrings appearing in exactly two distinct strings are counted.

Why |color_set(v)|=2? Because if a substring has exactly 2 strings, any node whose path is that substring can only have children from those two strings. We also want to make sure we count

correctly, which is why we use a suffix tree. Each node's parent/child edge contributes only the newly introduced substring extension. This ensures it does not overlap with previously counted substrings at any node. Therefore, the post traversal of color sets and the incremental counting of the new substrings ensure the algorithm's correctness.

Complexity Analysis

- Building the Generalized Suffix Tree is O(n) time for k = O(1)
- The compute_color_sets function, which does the DFS traversal takes O(n) time
    - Also, when we do union operations on small sets, it is O(1) when k = O(1), therefore the total remains O(n)
    - When counting new substrings, we count if |color_set(v)=2|. This takes O(1) per node, which is O(n) overall.
- Therefore, the total complexity of the algorithm is O(n) time

Problem 2

Correctness Proof and my approach: My approach to this is to use a stack for the construction of the suffix tree using the suffix array and LCP in O(n) time. Start by initializing the root node at depth 0. Next, have a stack of nodes representing the current path from the root to the current node. Now while iterating over suffixes in the order they appear in the suffix array:

- Use LCP value to decide how many nodes on the stack match the new suffix's prefix
    - Because LCP array tells us how many characters the current suffix shares with the previous suffix
- Pop node if depth exceeds current LCP
- Also, if I have to, create a new internal node with depth that is the current LCP. This is the branching point in the tree.
- Add a new leaf node with depth being the length of the suffix for the current suffix. Also, store SA[i] in the index(leaf) so you can track which suffix this leaf corresponds to.
- Iterate until done

Doing this ensures that each node is created once. My goal is O(n) time construction so using the stack helps with making the operations efficient.

```
class Node:
    def __init__(self):
        self.depth = 0         # number of characters from the root
        self.parent = None     # pointer to the parent node
        self.children = {}     # dictionary mapping edge label (first char) to
child node
        self.index = -1        # for leaves: the starting index of the suffix

def build_suffix_tree(T, SA, LCP):
    n = len(T)
```

```
    root = Node()
    root.depth, root.parent = 0, None
    stack, last = [root], None

    for i in range(n):
        # LCP[0] is 0, otherwise its LCP[i]
        current_lcp = LCP[i] if i > 0 else 0

        # pop until the top node's depth is <= current_lcp.
        while stack[-1].depth > current_lcp:
            last = stack.pop()

        # if top node's depth is less than current_lcp, split
        if stack[-1].depth < current_lcp:
            # create new node with depth = current_lcp.
            v = Node()
            v.depth, v.parent = current_lcp, stack[-1]

            # compute key for link from parent's side
            # use previous suffix (at i-1) and parent depth
            parent_edge = (T[SA[i-1] + stack[-1].depth] if SA[i-1] +
stack[-1].depth < n else '$')
            # from the new node, the edge leading to the node last is labeled using
current_lcp
            child_edge = (T[SA[i-1] + current_lcp] if SA[i-1] + current_lcp < n
else '$')
            v.children[child_edge] = last
            if last is not None:
                last.parent = v

            # link the new node into the tree
            stack[-1].children[parent_edge], last = v, None
            stack.append(v)

        # create a leaf for current suffix
        leaf = Node()
        leaf.depth, leaf.index, leaf.parent = n - SA[i], SA[i], stack[-1]
        # determine the edge label for the leaf: from the parent's depth onward.
        leaf_edge = (T[SA[i] + stack[-1].depth]
                     if SA[i] + stack[-1].depth < n else '$')
        stack[-1].children[leaf_edge] = leaf
        stack.append(leaf)

    # clear stack except root
    while len(stack) > 1:
        stack.pop()
    return root
```

```
# test case
T = "banana$"
SA = [6, 5, 3, 1, 0, 4, 2]
LCP = [0, 0, 1, 3, 0, 0, 2]
root = build_suffix_tree(T, SA, LCP)
```

Complexity Analysis:

- The for loop is O(n) and the inner while loop is simply popping the stack so there isn't any major added complexity. Still O(n).
- Stack operations are O(1)
- Therefore, the time complexity is O(n)

Problem 3

Part 1

LCP implementation was inspired by lecture slides.

```
import numpy as np
# create suffix array in O(n)
def build_suffix_array(T):
    suffix_array = list(range(len(T)))
    suffix_array.sort(key=lambda i: T[i:])  # lexicographically sorted using SUFFIX
ARRAY INDUCED SORTING (SA-IS) which is O(n) complexity
    return suffix_array

# compute LCP in O(n)
def compute_LCP(T, SA):
    n = len(T)
    LCP, rank = [0] * n, [0] * n
    # rank array stores position of suffixes in SA
    for i in range(n):
        rank[SA[i]] = i

    h = 0
    # 2 loops but still O(n), observe the while loop sliding window
    for i in range(n):
        if rank[i] > 0:
            j = SA[rank[i] - 1]  # prev suffix in SA
            while (i + h < n) and (j + h < n) and (T[i + h] == T[j + h]):
                h += 1
            LCP[rank[i]] = h
            if h > 0:
                h -= 1  # reduce LCP for next iteration
    return LCP
```

```
# compute LPF in O(n)
def compute_LPF(T):
    n = len(T)
    suffix_array = build_suffix_array(T)
    lcp = compute_LCP(T, suffix_array)
    lpf = np.zeros(n, dtype=int)

    # Mapping suffix array positions to their LPF values
    for i in range(1, n):
        pos1 = suffix_array[i]  # current suffix index
        lpf[pos1] = max(lpf[pos1], lcp[i])  # LPF is the max LCP before this index

    return lpf

# test case
T = "banana$"
LPF_result = compute_LPF(T)
print(LPF_result)
```

Correctness Analysis

We have the definition of LPF given in the instructions, so I built the program based on that.

- Suffix array ranks all suffixes in lexicographical order. The implementation is trivial to be honest. All the function is doing is sorting via the lambda function to correctly order the suffix indices.
- LCP array measures the longest common prefix between consecutive suffixes in the suffix array. In my implementation, I built the rank array such that rank[i] gives the position of the suffix starting at index i in the suffix array. Then the LCP value between two consecutive suffixes in SA is calculated using a sliding window. We can see that the LCP length, which is defined by h, is reduced in each step.
- LPF array simply stores the maximum previous factor overlap for each suffix as defined in the problem. I traverse the suffix array and map LCP values to compute LPF[i]. This ensures that LPF[i] stores the longest match for prefix i in an earlier suffix.

Complexity Analysis

- Suffix array computation is O(n) complexity if we use the suffix array induced sorting algorithm to compute the suffix array.
- LCP Computation is O(n) because rank[i] computation is O(n) as it is one loop. The second loop iterates through each suffix once, and the while loop runs in amortized O(n) time, as we use a sliding window. This ensures that h only increases and decreases, leading to total O(n) complexity.
- LPF computation is updated using a single pass through the suffix and LCP array. It is O(n) as it's just one for loop
- Total Complexity: O(n) + O(n) + O(n) = O(n).

Part 2

Givens:

- LPF[i] represents the LCE between T[i] and some previous occurrence in T[1..i]
    - Describes longest proper prefix of T[i..n] that occurs earlier in T[1..i]
    - LPF values correspond to the longest match of a factor in the text at some position
- LCP[i] represents the LCP of two consecutive suffixes in the suffix array of T
    - Describes similarity between adjacent suffixes in the sorted suffix order
    - LCP values describe overlap between consecutive suffixes in the suffix array

Idea: We have that every substring that appears at least twice in T contributes one entry into the LPF array and one entry into the LCP array. Matching this up shows the same collection of lengths appears in both arrays.

Each repeated substring contributes exactly once to LPF. Consider a substring T[i..i+l] of length l that appears at least twice. Suppose a second occurrence starts a position j > i. We know that this means $LPF[j] \geq l$ because T[j..] matches T[i..] for at least l characters. Therefore, each repeated substring of length l contributes at least l to some LPF[j] and conversely any nonzero LPF[j] indicates a repeated substring of length LPF[j]. Hence, each repeated substring of length l shows up exactly once in the LPF array (at the second occurrence position).

Each repeated substring contributes exactly once to LCP. In the suffix array, two occurrences T[i..] and T[j..] are neighbors in sorted order at some point. LCP of these adjacent suffixes is at least l. Conversely, if LCP[k] = l > 0, it means the two suffixes in the sorted list share a prefix of length l, so that prefix repeats. Therefore, each repeated substring of length l appears exactly once in the LCP array (when two suffixes are neighbors)

Since each repeated substring of length l contributes one occurrence of l to LPF and one occurrence of l to LCP, LPF[1..n] and LCP[1..n] multiset values coincide.

Permutation Property, Recap Analysis, and One-to-One Correspondence:

- You can see that every match counted in the LPF array also contributes to a longest common prefix when suffixes are sorted
- The total count of values in LPF and LCP are both n-1. This means they must be related in a one-to-one correspondence (combining what was mentioned above)
- Every value of LPF corresponds to an occurrence of a repeated substring that appears elsewhere in the text
- When constructing the suffix array, these repeated substrings influence LCP values

Since the set of all such occurrences must match exactly in count and value, LPF values must be a permutation of the LCP values

Problem 4

Part 1

Given the substrings $T[i... i + l]$ and $T[j... j + l]$ of length l. I am going to define the following:

- $k = floor(log(l))$
- $p = 2^k$

We have that $p \leq l$ and $2p = 2^{k+1} > l$. My approach is to split each substring into two overlapping pieces, each of length p. So we have the following. Let || denote concatenation.

- $T[i..\, i + l] \rightarrow T[i..\, i + p) \,||\, T[i + l - p..\, i + l]$
- $T[j..\, j + l] \rightarrow T[j..\, j + p) \,||\, T[j + l - p..\, j + l]$

The idea is that if both pairs of length-p pieces are identical, the entire length-l substrings are identical. This also means that if one of the pieces differs, then they are not identical, obviously.

To compare the two pieces in O(1) time, we will compare ranks. Rank is the value that shows the lexicographical position of a string among all substrings of the same length. It'll be a useful unique identifier that reflects its order. The KMR algorithm already assigns a rank to every length-p substring starting at each position. This is simply a retrieval in O(1) time from A_k. Comparing the ranks $A_k[i]\ vs\ A_k[j]$ which represent $T[i..\, i + p)$ and $T[j..\, j + p)$ respectively. If they are not equal, then we know which of the two is lexicographically smaller and that they are not equal. If they are equal, then also compare $A_k[i + l - p]\ vs\ A_k[j + l - p]$ which represents $T[i + l - p..\, i + l]$ and $T[j + l - p..\, j + l]$ respectively. Again, same as before, if they differ, we know which substring is lexicographically smaller and they are not equal. We also know if the entire length-l substrings are equal. Therefore, we have that one pair of array lookups and comparisons is enough to see whether the two substrings are equal or not. Since each rank lookup is O(1), the overall comparison of the two substrings is also O(1).

In terms of data structure, we just need the arrays $A_0,\ A_1,\ \ldots,\ A_{logn}$ of length n. $A_k[i]$ is a rank value which uniquely identifies the substring $T[i..\, i + 2^k]$. This allows the comparison of any two substrings of the original string in constant time by just comparing just two pairs of ranks from the appropriate array.

Part 2

The following code is also in the attached Python file.

The implementation of the KMR algorithm, data structure, and comparisons is included. Essentially, the explanation of the code is the same as the previous part but I also put a few clarifying comments.

```
# compute floor(log2(x)) for x = 1 to n in O(n)
# returns a list where log_table[x] = floor(log2(x))
def build_log_table(n):
    log_table = [0] * (n + 1)
    for i in range(2, n + 1):
        log_table[i] = log_table[i // 2] + 1
    return log_table

# build KMR arrays for string T in O(nlogn)
# returns a 2d list where kmr_array[k][i] is rank of substring T[i..i+2^k]
def build_kmr(T, log_array):
    n = len(T)
    max_log = log_array[n]

    # 2D array with dimension (max_log+1) x n
    # kmr_array[k][i] represents rank of the substring from i with length 2^k
    kmr_array = [[0] * n for m in range(max_log + 1)]

    # k = 0, rank of a single character
    for i in range(n):
        kmr_array[0][i] = ord(T[i]) - ord('a')  # map a..z to 0..25

    # temp array for sorting
    temp = [((0, 0), 0)] * n
    # get ranks for substrings of lengths 2^k by doubling the length each iteration
    length, k = 1, 0
    while length < n:
        # creating pairs (current rank, next rank) for each index.
        for i in range(n):
            temp[i] = ((kmr_array[k][i], kmr_array[k][i + length] if i + length < n
else -1), i)

        # sort indices by the pair (first, second).
        temp.sort(key=lambda x: x[0])

        # assign new ranks for A[k+1] using sorted order
        new_rank = 0
        kmr_array[k + 1][temp[0][1]] = 0
        for i in range(1, n):
            # if pair differs from the previous, increase rank
            if temp[i][0] != temp[i - 1][0]:
                new_rank += 1
```

```
            kmr_array[k + 1][temp[i][1]] = new_rank
        k += 1
        length *= 2
    return kmr_array

# compare T[i..i+L) and T[j..j+L) in O(1) using KMR array
# returns True is equal, false if not
# params: KMR array, log table, i and j (starting indicies), length (substring
lengths to compare), n (total T length)
def substr_equal(KMR_array, log_array, i, j, length, n):
    # two empty substrings -> True or identical starting positons -> True
    if length == 0 or i == j:
        return True
    # bad counts -> False
    if i + length > n or j + length > n:
        return False

    # find largest power of 2 that is <= L.
    k = log_array[length]
    p = 2 ** k

    # compare rank of the first 2^k block.
    if KMR_array[k][i] != KMR_array[k][j]:
        return False
    # compare rank of the last 2^k block.
    if KMR_array[k][i + length - p] != KMR_array[k][j + length - p]:
        return False
    return True


# read from "in.txt"
with open("in.txt", "r") as f:
    T = f.readline().strip()
    q = int(f.readline().strip())
    queries = []
    for m in range(q):
        line = f.readline().strip()
        i, j, length = map(int, line.split())
        queries.append((i, j, length))

n = len(T)
# create logarithm table for KMR and queries
log_array = build_log_table(n)
# build the KMR data structure
A = build_kmr(T, log_array)

# process each query and output
```

```
with open("out.txt", "w") as out:
    for (i, j, L) in queries:
        i -= 1
        j -= 1
        # compare substrings
        if substr_equal(A, log_array, i, j, L, n):
            out.write("YES\n")
        else:
            out.write("NO\n")
```

The goal was to achieve O(1) time complexity for the substring comparisons, which is shown by the fact that there is no looping done in the substr_equal() function.

The implementation passed the test when compared using diff. All lines matched. A picture of the diff command is attached. No output means no differences.