Steven Tung
CSE548: Homework 1

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

To solve this problem in O(n) time, I am going to utilize the prefix function from the KMP algorithm. The prefix function, π[i], for a string S of length n is defined as the length of the longest proper prefix of S[0…i] which is also a suffix. Using this, we can get the smallest period p of the string as $p = n - \pi[n-1]$ if n is divisible by p. I will elaborate on why this works in the correctness proof.

The following is the pseudocode. The compute_prefix_function was created using inspiration from 'Version 2' from the lecture slides.

```
def compute_prefix_function(S): # O(n) Complexity
    n = len(S)
    pi = [0] * n
    b = 0
    for i in range(1, n):
        while b > 0 and S[i] != S[b]:
            b = pi[b - 1]
        if S[i] == S[b]:
            b += 1
            pi[i] = b
    return pi

def find_smallest_period(S): # O(1) Complexity
    n = len(S)
    pi = compute_prefix_function(S)
    smallest_period = n - pi[n - 1]
    if n % smallest_period == 0:
        return smallest_period
    else:
        return n  # no smaller period found, return n itself

# Test
S = "adadadad"
print(find_smallest_period(S))  # test: 2 because 'ad'
```

Correctness Proof:

- From the slides, both lemmas help to explain the correctness of the computation of pi. When computing π[i], I try to extend the borders of P[1…i-1] when a match is found. When there is a mismatch, the function backtracks to make sure we only check valid prefixes. The compute_prefix_function ensures that we have the correct value which will be used to find_smallest_period.
- The find_smallest_period function is correct because we first find π[n-1] which is the last element in the array. This gives us the longest proper prefix of S which is also a suffix. Now, we do $p = n - \pi[n-1]$ (length of S minus length of longest proper prefix) which is the length of the smallest period. Lastly, we must find if $n \bmod p = 0$. If it is then that means p is the smallest period. Otherwise, the string does not show a repeating structure in which case we return the length of the string (it is its own period).

Time Complexity Analysis:

- compute_prefix_function is O(n) complexity because we only traverse the string once for the outer for loop. The reason why it is not $O(n^2)$ is because the inner while loop only executes on a mismatch to update b. b always decreases on mismatch. Therefore, the number of times the while loop is entered is at most n - 1. This means that the total number of iterations from both loops is at most 2n.
- find_smallest_period is O(1) complexity. There are no iterations.
- Total Complexity: O(n)

Problem 2

Claim A: **TRUE**

I know that the prefix function $\pi_S[i]$ represents the length of the longest proper prefix of S that is also a suffix of S[1…i]. So, given $\pi_S[i] = k$, we have that the prefix of length k matches a suffix of S[1…i]. We also are given that $j \geq i$, which means that the longest prefix match in S[1…j] can be at most k (previous match) plus $(j - i)$ new characters.We know that we cannot extend the prefix match arbitrarily (only by 1 for each step). New characters should continue the existing match. Therefore, it holds that $\pi_S[j] \leq \pi_S[i] + (j - i)$ because at most each new character increases π_S by 1. $j - i$ is the number of character increases from i to j. If we took a different approach and thought about it logically, if a mismatch occurs at j, then $\pi_S[j] = 0$ and if a match occurs, then $\pi_S[j] = 1 + \pi_S[j-1]$. $\pi_S[j]$ will never become greater than $\pi_S[i] + (j - i)$.

~~Claim B: **TRUE**~~

~~We have that S is made of k number of copies of A. We also assume that A is primitive and irreducible (@8 on piazza). This means that the longest proper prefix that is also a suffix should contain all but one copy of A. When we go through the π function, we notice that the first copy~~

~~of A has 0 in their respective indices, the count only goes up after the first copy of A. In other words, the longest prefix that is also a suffix should be of length $m - |A|$. This is stored in $\pi_S[m]$. Since A is repeating, the prefix function captures this periodicity. Therefore it holds that $\pi_S[m] = m - |A|$.~~

Claim B: **FALSE**
Counterexample: Let A = "abab", k=3 which means S="abababababab"
We have:

- $\pi_S[12] = 10$
- $m - |A| = 12 - 4 = 8$

Notice $10 \neq 8$, therefore counterexample proves False.

Claim C: **FALSE**
Counterexample: Let $S = abcabc,\ c = 'a',\ and\ cS = 'aabcabc'$

We have:

- $\pi_S = [0, 0, 0, 1, 2, 3]$
- $\pi_{cS} = [0, 1, 0, 0, 1, 0, 0]$

Notice that $\pi_S[3] = \pi_{cS}[4] - 1$ does not equal. It gives $1 = 0$.

Problem 3

Part A:
In the naive approach, the computing of the fingerprint is the bottleneck because we are doing multiple repeated operations. The goal is to reduce a query from O(mn) time to O(1) time with a data structure of size O(n) space. My approach to this will be to precompute the powers of r and also compute prefix hashes. Both of these operations occur in O(n) time only once. Then we can query the hash of any substring in O(1) time by utilizing the values saved in the data structure.

We will define the following for the data structures in O(n) space:

- Array R of length n + 1 where $R[0] = 1,\ R[k] = (R[k-1] \cdot r)\ mod\ q$ for k = 1,2,...,n
  - $R[k]$ stores the value of $r^k mod\ q$
  - This will save time later on because we do all of the calculations now once.
- Array H of length n + 1 where $H[0] = 0,\ H[k] = (H[k-1] \cdot r + T[k-1])\ mod\ q$
  - $H[k]$ represents the hash of the prefix $T[0 \ldots k-1]$
  - We get $H[k]$ (hash of the first k characters) by taking $H[k-1]$ (hash of the first k-1 characters), multiply by r, and add the next character $T[k-1]$, mod q.

Now we want to query for the hash of any substring. We do this using the following algorithm:

- $\phi(i, j) = (H[j + 1] - H[i] \cdot R[j - i + 1]) \, mod \, q$
    - $j - i + 1$ is the length of the substring (assuming 0-based indexing)
    - $H[j + 1]$ is the hash of the prefix $T[0 \ldots j]$
    - $H[i]$ is the hash of the prefix $T[0 \ldots i - 1]$
    - $H[i] \cdot R[j - i + 1]$: this is for subtracting out the contribution of $T[0 \ldots i - 1]$ from $H[j + 1]$. We do this because we need to shift the prefix so that its highest power of r lines up with j.
    - The subtraction operation is for removing the prefix portion from the hash of $T[0 \ldots j]$. The remainder is the polynomial that corresponds to $T[i \ldots j]$.
    - Lastly, we mod operation

Space and Time Complexity Analysis

- The two arrays are of O(n) space each which is still O(n) space.
- The query function is of O(1) time as we just plug in the values of i and j.

Part B:

First, the preprocessing on the entire string T in O(n) to build the prefix hash array and power array will be done. Then each query will be answered in O(1) time by comparing the KR fingerprints of $T[i \ldots i + l + 1]$ and $T[j \ldots j + l + 1]$.

In my testing, no false positives occurred when I utilized the q value of somewhere around at least 30097. There are lower numbers that didn't run into false positives but that could just be due to chance as some numbers higher ran into false positives. However, for the sake of having a large prime number to minimize false positives, I coded into the program a value of 1000000021.

The following values are the parameters I used for this program:

- q = 1000000021
- r = 257

Python code with comments in them. I also attached the Python file.

```
# read from 'in.txt'
with open('in.txt', 'r') as f:
    T = f.readline()  # Long string
    k = int(f.readline())  # query Amount
    queries = []
    for l in range(k):
        line = f.readline()
        i, j, length = map(int, line.split())
        queries.append((i, j, length))
```

```
# params
MOD = 1000000021 # THIS IS q
BASE = 257 # THIS IS r
n = len(T)
H = [0] * (n + 1) # prefix hash array

# precompute powers of BASE
# R[i] = BASE^i mod MOD
R = [0] * (n + 1)
R[0] = 1
for i in range(1, n + 1):
    R[i] = (R[i - 1] * BASE) % MOD

# char to integers
for i in range(1, n + 1):
    H[i] = (H[i - 1] * BASE + (ord(T[i - 1]) - ord('a') + 1)) % MOD

# get the hash of substring TIME: O(1)
def get_hash(a, b):
    result = H[b] - (H[a - 1] * R[b - a + 1] % MOD) # defined in part A
    return result % MOD

# process each query and output
with open('out.txt', 'w') as out:
    for (i, j, length) in queries:
        if length == 0: # len == 0 should match
            out.write("YES\n")
            continue

        # calculate T[i..i+length-1] and T[j..j+length-1] then compare
        hash1 = get_hash(i, i + length - 1)
        hash2 = get_hash(j, j + length - 1)

        if hash1 == hash2:
            out.write("YES\n")
        else:
            out.write("NO\n")
```

Every query is calculated in O(1) time by calling the get_hash function to calculate the value of the hash. The get_hash function calls the two precomputed arrays and simply calculates the hash.

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.

Problem 4

The following code consists of 5 functions:

- compute_pi: computes the pi array for the kmp algorithm. Derived from the pseudo in the slides.
- kmp_search: performs the kmp search and returns an array for indices. Derived from the pseudo from the slides
- naive_search: performs the naive method of searching
- generate_random_string: generates a random string of length n based on given alphabet
- run_test: simply runs the test x number of times and compares the results of kmp_search and naive_search.

The code will be attached in a Python file.

Running Instructions:

1. You only need to call the run_test by passing it 3 parameters: number of tests, max length of text/pattern, and alphabet size. This function tests kmp and naive and does the comparisons.
2. Since there are comparisons made between the KMP and naive algorithms, it will catch mismatches (when you modify kmp_search to have bugs). When a mismatch is found, the test will stop at the mismatching test.
3. If you want to, you can remove some print functions to remove clutter and only show the mismatches.

```
import random
import string

# computes the Longest Prefix Suffix (pi) array for the pattern.
# this is derived from the Version 2 from the slides with some modifications
def compute_pi(pattern):
    pi = [0] * len(pattern)
    b = 0  # length of the previous longest prefix suffix
    for i in range(1, len(pattern)):
        while b > 0 and pattern[i] != pattern[b]:
            b = pi[b - 1]  # backtrack to the prev pi value
        if pattern[i] == pattern[b]:
            b += 1
            pi[i] = b
    return pi

# performs kmp algorithm.
# this algorithm is based on the pseudocode provided but modified to have one less
while loop.
def kmp_search(text, pattern):
    if not pattern:
```

```
        return list(range(len(text) + 1))

    lps = compute_pi(pattern)
    occurrences = []  # store indices

    i, l = 0, 0  # pointers for text and pattern

    while i < len(text):
        if text[i] == pattern[l]:
            i += 1
            l += 1
            if l == len(pattern):  # full pattern match
                occurrences.append(i - l)
                l = lps[l - 1]
        else:
            if l > 0:
                l = lps[l - 1]
            else:
                i += 1
    return occurrences

# naive method, returns array of indices
def naive_search(text, pattern):
    occurrences = [] # stores indices
    n, m = len(text), len(pattern)
    for start in range(n - m + 1):
        match = True
        for j in range(m):
            if text[start + j] != pattern[j]:
                match = False
                break
        if match:
            occurrences.append(start)
    return occurrences

# this function is just to create random strings using the alphabet
def generate_random_string(length, alphabet):
    return ''.join(random.choice(alphabet) for i in range(length))

# this function is for testing my kmp vs naive
# give it 3 params: number of tests, max length of text, and alphabet size
# will terminate test on mismatch!
def run_test(num_tests=10000000, max_length=10000, alphabet_size=4):
    alphabet = string.ascii_lowercase[:alphabet_size] # build alphabet

    for i in range(num_tests):
        text_len = random.randint(0, max_length)
```

```
        pattern_len = random.randint(2, 8)
        text = generate_random_string(text_len, alphabet)
        pattern = generate_random_string(pattern_len, alphabet)
        kmp_result = kmp_search(text, pattern)
        naive_result = naive_search(text, pattern)

        # comparing results
        if kmp_result != naive_result:
            print("Mismatch found!")
            print("Text:    ", text)
            print("Pattern: ", pattern)
            print("KMP:     ", kmp_result)
            print("Naive:   ", naive_result)
            return  # Stop on first mismatch

        print("text: ", text)
        print("pattern: ", pattern)
        print("kmp_result: ", kmp_result)
        print("naive_result: ", naive_result)
        print("------------------------")

    print("Test Passed!")

run_test(1000, 10000, 4)
```

Problem 5

To solve the LCE query in $O(logn)$ time, we can use binary search as binary search itself is $O(logn)$ time. Instead of incrementally checking every length between 0 and $n - max(i, j) + 1$, we will use binary search to reduce the search space based on the current length value.

The comments in the pseudocode provide a general overview of how the algorithm works.

Pseudocode

```
def LCE(i, j, n):
    low, high = 0, n - max(i, j) + 1
    while low < high: # ensure we stay in range
        mid = (low + high + 1) // 2 # change mid to the middle value of high and low
        if Eq(i, j, mid):  # check if first 'mid' characters match
            low = mid  # sets to mid which is the length. Will try to extend further
        else:
            high = mid - 1  # decrease search space since we know it has to be less
    return low # this is the longest common prefix length
```

Correctness Proof

- Since we can utilize the function $Eq(, i, j, l)$, we have a reliable (assumed correctly working) way to check substrings. This means that we can refine the search space accurately if we couple it will binary search.
- Also, we can guarantee that binary search correctly narrows down the maximum possible value because we know that if $Eq(, i, j, l)$ is true then $Eq(, i, j, l')$ is true for all $l' < l$ (monotonicity)

Complexity Analysis

- The algorithm performs binary search in O(logn) time. The range of values is 0 to n but we know that binary search reduces the search space by half on each iteration.
- We are given that $Eq(i, j, l)$ is O(1) time
- Overall complexity is O(logn)

Problem 6

First, we need to build a trie to store P. The following is a class definition. Each TrieNode has a dictionary that stores the children's char of the node, an appropriately generalized pi, and a list of any patterns that end at the current trie node.

```
class TrieNode:
    def __init__(self):
        # children dictionary that holds children chars
        # pi points to prefix-function link
        # output holds patterns that end at the node
        self.children, self.pi, self.output = {}, None, []

# builds trie by taking in a list of patterns
def build_trie(patterns):
    root = TrieNode()
    for p in patterns: # go through every pattern to add
        curr = root
        for c in p: # go through every char in a pattern
            if c not in curr.children:
                curr.children[c] = TrieNode()
            curr = curr.children[c]
        curr.output.append(p) # add at end of pattern
    return root
```

To remove redundant comparisons (to preserve linear runtime), we want to make sure that when a mismatch happens, we can just jump to the longest proper suffix of the current path (this is also a prefix of one of the given patterns). EXAMPLE: we have the text “sheep” and the pattern set = {“she”, “he”}. When the algorithm finds the pattern “she”, the pi link will point to “he” which will let the algorithm know that “he” occurs without having to rescan the text (the instructions

state text is streamed left to right with no reprocessing). The easiest way to do this is to use Breadth-First Search as it is a tree and we want to process pi link level-by-level to guarantee that when we get to a child, its parent is already computed.

```
# building pi links using BFS
def build_pi_links(root):
    queue = deque()
    root.pi = root

    # initialize for root's immediate children
    for node in root.children.values():
        node.pi = root
        queue.append(node)

    # going level-by-level, we will start at the pi link of the current node then
we will follow the links until a matching child node or until we get to root
    while queue:
        current = queue.popleft()
        for c, child in current.children.items():
            queue.append(child)
            state = current.pi
            while state != root and c not in state.children:
                state = state.pi
            # set the child's pi link then merge the output
            child.pi = state.children.get(c, root)
            child.output += child.pi.output
```

Lastly, we want to perform the search.

```
# searches text using trie and returns list of occurences
def perform_search(text, root):
    curr = root
    occurrences = []

    for i, c in enumerate(text):
        # if the char not child of the current node, follow pi links
        while curr != root and c not in curr.children:
            curr = curr.pi
        curr = curr.children.get(c, root)
        occurrences.extend((p, i - len(p) + 1, i) for p in curr.output)

    return occurrences # TUPLE return
```

Correctness

- Regarding the Trie: during the construction of the Trie, each pattern is represented by a

path from the root to some terminal node.

- It also handles cases where maybe one pattern is the prefix of another pattern (example: “dog” and “do”) by utilizing a list in the trie node that stores all patterns that terminate at that node.
- Generalizing the pi (links): By using BFS construction, it ensures that each node’s pi (fail) link correctly represents the longest possible suffix that is also in the trie. This is my attempt to use the KMP prefix function logic with the trie structure of patterns.
- Single Pass Searching: Since we cannot backtrack in the text, we will not. Instead, we use the pi links as explained earlier above. This ensures we have linear scanning.

Complexity Analysis:

- Building Trie: O(||P||)
- Building the pi (failure) links: O(||P||) as each node is enqueued/dequeued once and we have that each character is processed a constant number of times.
- Searching: O(n + occ) as each character in text leads to at most one or two fail steps. In addition, occurrences reporting takes occ time in total.
- Total Time: O(||P||) + O(||P||) + O(n + occ) = O(||P|| + n + | occ)