I. Describe and analyze an efficient algorithm to find strings in labeled rooted trees. Your input consists of a pattern string $P[1..m]$ and a rooted text tree $T$ with $n$ nodes, each labeled with a single character. Nodes in $T$ can have any number of children. A path in $T$ is called a downward path if every node on the path is a child (in $T$) of the previous node in the path. Your goal is to determine whether there is a downward path in $T$ whose sequence of labels matches the string $P$.

Solution (rolling hash): We use a variant of the Rabin-Karp rolling hash, as described in the lecture notes. For any string $w$ of length $m$, let $h(w)$ denote its hash value. The rolling hash satisfies the following key properties:

- We can compute the hash value of any string of length $m$ from scratch in $O(m)$ worst-case time.
- For any string $w[0..m]$, given the hash value $h(w[0..m-1])$ and the characters $w[0]$ and $w[m]$, we can compute the hash value $h(w[1..m])$ in $O(1)$ worst-case time.
- Assuming we choose the salt parameters of $h$ randomly from a large enough range, for any two strings $w \neq w'$, both of length $m$, we have $\Pr[h(w) = h(w')] \leq 1/m$.

Our algorithm assumes each node $v$ in the input tree $T$ stores its label and a pointer to its parent, in addition to its usual child pointers. To avoid annoying boundary cases, we (pretend to) add a path of $m-1$ nodes above the root of $T$, each labeled with a sentinel character @ that does not appear in the pattern string $P$.

For each node $v$ in the (original) input tree $T$, we actually compute three values:

- $v$.depth is the depth of $v$. This is used only to compute...
- $v$.start is the label of the starting node of the unique downward path of length $m$ ending at $v$. In particular, if $v$ has depth less than $m$, then $v$.start is the sentinel character @. This is used only to compute...
- $v$.hash is the rolling hash value of the label of the downward path of length $m$ ending at $v$.

The following algorithm computes $v$.hash for every node $v$, along with the hash value of the pattern, which we call phash. Then for every node $v$ such that $v$.hash = phash, the algorithm performs a brute force string comparison.

```
MATCH_TREE_PATH(P[1..m], T):
    Initialize rolling hash function $h$
    COMPUTE_hashes(T, m) \(\langle O(n) \rangle\)
    phash \(\leftarrow h(P)\) \(\langle O(m) \rangle\)
    for all vertices $v$ in $T$
        if phash = $v$.hash \(\langle \text{prob} < 1/m \rangle\)
            if EXACT_MATCH($P, v$) \(\langle O(m) \rangle\)
                return TRUE
    return FALSE
```
**ComputeHashes(T, m):**

for all nodes v in T in (depth-first) preorder

1. **(compute depth —)**
   - if v = T.root
     - v.depth ← 0
   - else
     - v.depth ← v.parent.depth + 1

2. **(compute start symbol —)**
   - if v.depth < m - 1
     - v.start ← !
   - else
     - v.start ← S[v.depth -(m - 1)]

3. **(compute rolling hash value —)**
   - if v = T.root
     - v.hash ← \( h(\hat{m}^{n-1} \cdot v.label) \) \( \langle O(m) \rangle \)
   - else
     - compute v.hash from v.parent.hash and v.label and v.start

**ExactMatch(P, v):**

for i ← m to 1

- if v = Null and P[i] ≠ @
  - return False
- if v ≠ Null and P[i] ≠ v.label
  - return False
- v ← v.parent
return True

This algorithm runs in \( O(n + Fm) \) time, where \( F \) is the number of false matches, that is, the number of nodes \( v \) where \( v.hash = phash \) but \( \text{ExactMatch}(P, v) = \text{FALSE} \). The probability of any false match is \( O(1/m) \), so \( E[F] = O(n/m) \). We conclude that the algorithm runs in \( O(n) \) expected time.

**Rubric:** 10 points = 2 for using rolling hash + 4 for other details + 2 for analysis of false matches + 2 for time analysis. This is not the only correct \( O(n) - \) time algorithm. Max 5 points for \( O(mn) - \) time algorithm (for example, brute force).
Solution (modified KMP, 8/10): Preprocess the pattern \( P \) for KMP string matching in \( O(m) \) time. Then perform a standard preorder traversal of \( T \). Then we effectively run KMP along each root-to-leaf path in \( T \), but using memoization to avoid scanning common prefixes of downward paths more than once. At each node \( v \), we compute an index \( v.j \) equal to the length of the longest prefix of \( P \) that is also the label of a downward path ending at \( v \).

\[
\text{TreeKMP}(P[1..m], T): \\
\text{fail} \leftarrow \text{ComputeFailure}(P) \\
\text{for all nodes } v \text{ in } T \text{ in preorder} \\
\quad \text{if } v = T.\text{root} \\
\quad \quad \quad v.j \leftarrow [v.label = P[1]] \\
\quad \quad \text{else} \\
\quad \quad \quad v.j \leftarrow v.\text{parent}.j + 1 \\
\quad \quad \quad \text{while } v.j > 0 \text{ and } v.label \neq P[v.j] \\
\quad \quad \quad \quad v.j \leftarrow \text{fail}[v.j] \\
\quad \quad \quad \text{if } v.j = m \\
\quad \quad \quad \quad \quad \text{return } \text{True} \\
\quad \quad \quad \text{return } \text{False}
\]

Unfortunately, this algorithm does not run in \( O(n) \) time; the amortized analysis of the number of failures doesn’t generalize to trees. It’s true that the number of failures along each root-to-leaf path is \( O(n) \), but the number of failures at each node is unbounded. The best time bound we can guarantee is \( O(mn) \), which is no better than brute force.

For example, suppose \( P \) consists of \( m \) As, and \( T \) consisting of a path of \( m - 1 \) As whose bottom node has \( n - m + 1 \) children, each labeled with a distinct symbol not equal to A. If we use the unoptimized KMP failure function, our algorithm fails \( m \) times at each leaf of \( T \), so the overall running time is \( \Theta(mn) \).

However, if we use the optimized failure function described in the lecture notes, the number of failures at each node is at most \( O(\log m) \), and therefore our algorithm runs in \( O(n \log m) \) time in the worst case.

The following proof of that the optimized failure function admits at most \( O(\log m) \) consecutive failures is adapted from Knuth, Morris, and Pratt’s paper.

Fix an arbitrary pattern \( P[1..m] \). To simplify notation, define \( \ell = \text{fail}(m) \) and \( k = \text{fail}(\ell) \), and consider the prefixes \( z = P[1..m - 1] \) and \( y = P[1..\ell - 1] \) and \( x = P[1..k - 1] \) of \( P \). The definition of \( \text{fail} \) implies that both \( x \) and \( y \) are borders of \( z \). Moreover, because the failure function is optimized, we have \( P[m] \neq P[\ell] \) and \( P[\ell] \neq P[k] \). (See the figure on the next page.)

For the sake of argument, suppose \( \Delta = k + \ell - m \geq 0 \). Then we can make two observations about the symbol \( P[\Delta + 1] \):

- There is an overlap of \( \Delta \) symbols between the prefix \( y \) and the suffix \( x \) of \( z \), so \( x[\Delta + 1] = P[\ell] \). String \( x \) is a prefix of \( P \), so \( P[\Delta + 1] = x[\Delta + 1] = P[\ell] \).
• There is an overlap of $\Delta$ symbols between the prefix $x$ and the suffix $y$ of $z$, so $y[\Delta + 1] = P[k]$. String $y$ is a prefix of $P$, so $P[\Delta + 1] = y[\Delta + 1] = P[k]$.

So apparently $P[\Delta + 1] = P[k] = P[\ell]$. But our optimization guarantees that $P[k] \neq P[\ell]$; we have reached a contradiction! We conclude that $m \geq k + \ell$.

If $\ell \leq m/2$, then $k < \ell \leq m/2$; on the other hand, if $\ell > m/2$, then $k < m - \ell < m/2$. In both cases, we have $k < m/2$. Because we applied our argument to arbitrary patterns $P$, it also applies to all prefixes of any pattern $P$. We conclude that $\text{fail}(%(\text{fail}(j)) < j/2$ for every index $j$. It follows immediately that the longest failure chain in $P$ has length at most $2[\lg m]$.\(^a\)

\(^a\)More careful arguments imply that the longest failure chain has length at most $\lceil \log_{\phi} m \rceil \approx 1.44 \lg m$, where $\phi$ is the golden ratio. This improved bound is actually tight for the Fibonacci strings.

**Rubric:** 8 points = 4 for correct algorithm + 2 for correct $O(mn)$-time analysis + 2 for correct $O(n \log m)$-time analysis with optimized fail function. 5 extra credit points for including a proof that the optimized fail function guarantees at most $O(\log m)$ consecutive failures. (I claimed this fact in class and in the notes, so a proof isn’t required for the 8 points of partial credit.)

I’m willing to believe that $O(n)$ worst-case time is possible, but I haven’t figured out how.
2. Suppose we want to design an algorithm to detect the subject of a fugue. We will assume a very simple representation as an array $F[1..n]$ of integers, each representing a note in the fugue as the number of half-steps above or below middle C.

(a) Describe an algorithm to find the length of the longest prefix of $F$ that reappears later as a substring of $F$. The prefix and its later repetition must not overlap.

**Solution:** We perform a binary search for the largest value of $L$ such that the input string contains a prefix of length $L$ that appears later in the string.

To answer the decision problem, we search for the “pattern” string $F[1..L]$ inside the “text” string $F[L+1..n]$, using any of the linear-time algorithms described in class. Because the decision algorithm for each value of $L$ runs in $O(n)$ time, the entire algorithm runs in $O(n \log n)$ time.

**Rubric:** 5 points: 1½ for binary search + 1½ for linear-time decision algorithm + 1 for enforcing disjointness + 1 for time analysis (if the algorithm is correct). This is not the only correct algorithm with this running time.

**Solution (extra credit):** Recall that a *border* of a string $w$ is any proper prefix of $w$ that is also a suffix of $w$. The Knuth-Morris-Pratt failure function is defined as follows for each index $i$:

$$\text{fail}(i) − 1$$

is the length of the longest border of $F[1..i−1]$. The preprocessing phase of KMP computes $\text{fail}(i)$ for every index $i$ in $O(n)$ time. To simplify notation (and avoid off-by-one errors), let $\text{border}(i) = \text{fail}(i + 1) − 1$ denote the length of the longest border $F[1..i]$.

To ensure that we compute $\text{border}(n) = \text{fail}(n + 1) − 1$, add an arbitrary sentinel character $F[n + 1]$ to the end of the input string.

If we didn’t require the prefix of $F$ and its repetition to be disjoint, we would be nearly done; the longest prefix that repeats later in the string (possibly with overlap) has length $\max_{1 \leq i \leq n} \text{border}(i) = \max_{2 \leq i \leq n+1} \text{fail}(i) − 1$. But enforcing the disjointness condition requires a bit more more work.

Let’s call a border *strict* if the equal prefix and suffix do not overlap. For example, ABAB is a strict border of ABABABAB, but ABABAB is not. For each index $i$, let $\text{border}^*(i)$ denote the length of the longest strict border of $F[1..i]$.

Fix an index $i$, let $b = \text{border}(i)$, and let $\delta = i − b$; then $F[1..b] = F[\delta+1..i]$ is the longest border of $F[1..i]$. Assume $b > \delta$, so the prefix $F[1..b]$ and the suffix $F[\delta+1..i]$ overlap; otherwise, we immediately have $\text{border}^*(i) = b$. Then the shorter prefix $F[1..\delta]$ is equal to $F[\delta+1..2\delta]$, and it follows by induction that $F[1..i]$ consists of $i/\delta$ concatenated copies of $F[1..\delta]$, followed by a copy of the prefix $F[1..i \mod \delta]$.

For example, consider the string

$$F[1..i] = \text{\underline{ABACABACABACABACABA}};$$

Here we have $i = 27$, $b = 23$, $\delta = 4$. The string consists of “$i/\delta = 5\frac{3}{4}$ copies” of the prefix ABAC, and its longest strict border has length $b − 3\delta = 11$. 

5
Repeatedly removing the first or last $\delta$ symbols of $F[1..b]$ yields another border of $F[1..i]$, and every border of $F[1..i]$ can be obtained this way. Thus, every border of $F[1..i]$ has length $b - k\delta$ for some integer $0 \leq k \leq b/\delta$, and we want the longest border whose length is at most $i/2$. The smallest integer $k$ such that $b - k\delta \leq i/2$ is $k = \lceil \frac{2b-i}{2\delta} \rceil$. We conclude that

$$\text{border}^*(i) = b - \left\lceil \frac{2b-i}{2i-2b} \right\rceil \cdot (i - b),$$

where $b = \text{border}(i)$. (The ceiling expression reduces to 0 when $b \leq i/2$, so the formula is correct even in that case.)

Our final algorithm has three stages: (1) Run the KMP preprocessing algorithm to compute the failure function in $O(n)$ time. (2) For each index $i$, compute $\text{border}^*(i)$ in $O(1)$ time. (3) Return $\max_i \text{border}^*(i)$. Altogether, our algorithm runs in $O(n)$ time.

Solution (extra credit): We can modify the \text{ComputeFailure} algorithm from Knuth-Morris-Pratt as follows:

```plaintext
LONGESTSUBJECT(F[1..n]):
    j ← 0
    j* ← 0
    longest ← 0
    for i ← 1 to n
        \(\langle\text{compute standard fail function}\rangle\)
        fail[i] ← j
        while $j > 0$ and $F[i] \neq F[j]$
            j ← fail[j]
        j ← j + 1
        \(\langle\text{compute strict fail function}\rangle\)
        fail*[i] ← j*  \(\langle\text{We don't actually need this line.}\rangle\)
        while $j^* > 0$ and ($F[i] \neq F[j^*]$ or $2j^* > i$)
            j* ← fail*[j*]  \(\langle\text{not fail}^*[j]\rangle\)
        longest ← max{longest, j*}
        j* ← j* + 1
    return longest
```

The lines in black are copied verbatim from \text{ComputeFailure}. The lines in blue compute a “strict failure” function $\text{fail}^*$ defined as follows:

$\text{fail}^*[i] - 1$ is the length of the longest strict border of $F[1..i-1]$.

The correctness of this algorithm follows from two characterizations of strict borders:

- $F[1..j]$ is a strict border of $F[1..i]$ if and only if $F[1..j-1]$ is a strict border of $F[1..i-1]$ and $F[i] = F[j]$ and $2j \leq i$.
- $F[1..j]$ is a strict border of $F[1..i]$ if and only if either $F[1..j]$ is the longest strict border of $F[1..i]$ or $F[1..j]$ is a border (not a strict border!) of the longest strict border of $F[1..i]$. 

6
In particular, just before the last line $j^* ← j^* + 1$ is executed, $F[1..j]$ is the longest strict border of $F[1..i]$. Thus, at the end of the algorithm $\text{longest}$ is the length of the longest prefix of $F$ that appears later in $F$ without overlapping.

The standard analysis of ComputeFailure implies that this modified algorithm runs in $O(n)$ time. Specifically: We increment $j$ at most $n$ times, and therefore we decrease $j$ by setting $j ← \text{fail}[j]$ at most $n$ times. Similarly, we increment $j^*$ at most $n$ times, and therefore we decrease $j^*$ by setting $j^* ← \text{fail}[j^*]$ at most $n$ times.

**Solution (extra credit):** Alternatively, we can leave ComputeFailure unchanged and modify the main loop of of Knuth-Morris-Pratt:

```plaintext
\text{KunstDerFuge}(F[1..n]):
\begin{align*}
\text{fail} & \leftarrow \text{ComputeFailure}(F) \\
 j & \leftarrow 1 \\
\text{longest} & \leftarrow 0 \\
\text{for } i & \leftarrow 1 \text{ to } n \\
& \quad \text{while } j > 0 \text{ and } (F[i] \neq F[j] \text{ or } 2j > i) \\
& \quad \quad j \leftarrow \text{fail}[j] \\
& \quad \quad \text{longest} \leftarrow \max\{\text{longest}, j\} \\
& \quad j \leftarrow j + 1
\end{align*}
return longest
```

This is the same as the previous solution, except that we’ve split the work into two for loops instead of just one. There are only three differences from Knuth-MorrisPratt:

- the lines in blue, which keep track of the maximum value of $j$;
- the clause “or $2j > i$” in red, which ensures that $F[1..j]$ is always a strict border of $F[1..i]$; and
- the missing return after the while loop, because we always have $j < i \leq n$.

**Rubric:** Max 7½ points (yes, out of 5) = 4 for correctly using or modifying KMP failure function + 2½ for enforcing disjointness + 1 for time analysis (if the algorithm is correct). These are not the only linear-time algorithms for this problem.
(b) Describe an algorithm to find the length of the longest prefix of $F$ that reappears later, possibly transposed, as a substring of $F$. Again, the prefix and its later repetition must not overlap.

**Solution:** First we build a new difference array $\Delta F[1..n-1]$, where $\Delta F[i] = F[i+1] - F[i]$ for each index $i$. Any repeated prefix of $F$ of length $k$, even after transposition, corresponds to a repeated prefix of $\Delta F$ with length $k-1$ without transposition. For example:

$$F = 3, 1, 4, 1, 5, 2, 6, 5, 3, 1, 4, 1, -1, 2, -1, 3, 7, 0, 1, 4, 2$$

$$\implies \Delta F = -2, 3, -3, 4, -7, 4, -1, -2, -2, 3, -3, -2, 3, -3, 4, -7, 1, 3, -2$$

So we almost have a reduction to part (a), but shifting from $F$ to $\Delta F$ changes the disjointness requirement. Now we want the longest prefix of $\Delta F$ that appears as a substring later, with a gap of at least one number between the prefix and its repetition. For example:

$$F = 1, 2, 1, 2, 3, 4, 3, 4, 5, 6$$

$$\implies \Delta F = +1, -1, +1, +1, +1, -1, +1, +1$$

Adapting our first solution to part (a) is straightforward. After building $\Delta F$, we perform a binary search for the largest value of $L$ such that $\Delta F[1..L-1]$ is a substring of $\Delta F[L+1..n-1]$, again using any linear-time string matching algorithm for the decision problem at each step. The resulting algorithm runs in $O(n \log n)$ time.

**Rubric:** 5 points = 2 for computing differences $\Delta F + 1$ for binary search (or using part (a) as a black box) + 1 for enforcing gap between prefix and suffix + 1 for time analysis (if the algorithm is correct). This is not the only correct algorithm with this running time.

**Solution:** As in the previous solution, let $\Delta F[i] = F[i+1] - F[i]$ for each index $i$. We need to find the longest prefix of $\Delta F$ that appears as a substring later, with a nontrivial gap between the prefix and its repetition.

Adapting our linear-time algorithm for part (a) requires some tweaking. For each index $i$, and let $b = \text{border}(i)$ be the length of the longest border of $\Delta F[1..i]$, and let $\text{border}^+(i)$ denote the length of the longest border of $\Delta F[1..i]$ with a nontrivial gap between prefix and suffix. There are three cases to consider.

- If $\text{border}(i) < i/2$, then $\text{border}^+(i) = \text{border}(i)$.
- If $\text{border}(i) = i/2$, then $\text{border}^+(i)$ is the length of the second longest border of $\Delta F[1..i]$, which is $\text{border}(\text{border}(i))$.

- If $\text{border}(i) > i/2$, then the prefix and suffix overlap, we can follow the structural analysis in our previous solution almost verbatim; the array $\Delta F[1..i]$ is a power of the prefix $\Delta F[1..\delta]$, where $\delta = b - i$. To ensure a gap of at least one symbol, we need to change the inequality $b - k\delta \leq i/2$ to $b - k\delta < i/2$. The smallest value of $k$ that satisfies this modified constraint is

$$k = \left\lceil \frac{2b - i + 2}{2\delta} \right\rceil.$$
Summarizing, we have

\[
border^+(i) = \begin{cases} 
\text{border}(i) & \text{if } border(i) < i/2 \\
\text{border}(\text{border}(i)) & \text{if } border(i) = i/2 \\
b - \left\lfloor \frac{2b + 2i - 2b}{2i - 2b} \right\rfloor (i - b) & \text{otherwise}
\end{cases}
\]

for each index \(i\), where \(b = \text{border}(i)\) in the third case.

Our final algorithm has four stages: (1) compute the difference array \(\Delta F\); (2) run the KMP preprocessing algorithm on \(\Delta F\); (3) for each index \(i\), compute \(\text{border}^+(i)\) in \(O(1)\) time; (4) return \(\max_i \text{border}^+(i)\). Again, the entire algorithm runs in \(O(n)\) time.

**Solution (extra credit):** As in the previous solutions, let \(\Delta F[i] = F[i+1] - F[i]\) for each index \(i\). We need to find the longest prefix of \(\Delta F\) that appears as a substring later, with a nontrivial gap between the prefix and its repetition.

```python
KunstDerFuge(F[1..n]):
    for i ← 1 to n - 1
        ΔF[i] ← F[i+1] - F[i]
    fail ← ComputeFailure(ΔF)
    j ← 1
    longest ← 0
    for i ← 1 to n
        while j > 0 and (ΔF[i] ≠ ΔF[j] or 2j ≥ i)
            j ← fail[j]
        longest ← max{longest, j}
    return longest
```

The standard analysis of KMP implies that this algorithm runs in \(O(n)\) time.

**Rubric:** 7½ points (yes, out of 5) = 2 for computing \(\Delta F\) + 2½ for correctly using the KMP failure function (or invoking part (a) as a black box) + 2 for enforcing gap + 1 for time analysis (if algorithm is correct). These are not the only linear-time algorithms for this problem.