Computing Longest (Common) Lyndon Subsequences

Hideo Bannai[1][0000−0002−6856−5185], Tomohiro I[2][0000−0001−9106−6192], Tomasz Kociumaka[3][0000−0002−2477−1702], Dominik Köppl[1][0000−0002−8721−4444], and Simon J. Puglisi[4][0000−0001−7668−7636]

[1] M&D Data Science Center, Tokyo Medical and Dental University, Japan
{hdbn,koeppl}.dsc@tmd.ac.jp
[2] Department of Artificial Intelligence, Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp
[3] University of California, Berkeley, United States, kociumaka@berkeley.edu
[4] Department of Computer Science, Helsinki University, Finland
simon.puglisi@helsinki.fi

Abstract. Given a string T with length n whose characters are drawn from an ordered alphabet of size σ, its longest Lyndon subsequence is a longest subsequence of T that is a Lyndon word. We propose algorithms for finding such a subsequence in $\mathcal{O}(n^3)$ time with $\mathcal{O}(n)$ space, or *online* in $\mathcal{O}(n^3\sigma)$ space and time. Our first result can be extended to find the longest common Lyndon subsequence of two strings of length n in $\mathcal{O}(n^4\sigma)$ time using $\mathcal{O}(n^3)$ space.

Keywords: Lyndon word, subsequence, dynamic programming

1 Introduction

A recent theme in the study of combinatorics on words has been the generalization of regularity properties from substrings to subsequences. For example, given a string T over an ordered alphabet, the longest increasing subsequence problem is to find the longest subsequence of increasing symbols in T [2,25]. Several variants of this problem have been proposed [20,10]. These problems generalize to the task of finding such a subsequence that is not only present in one string, but common in two given strings [23,26,15], which can also be viewed as a specialization of the longest common subsequence problem [27,19,17].

More recently, the problem of computing the longest square word that is a subsequence [22], the longest palindrome that is a subsequence [6,18], the lexicographically smallest absent subsequence [21], and longest rollercoasters [12,4,11] have been considered.

Here, we focus on subsequences that are Lyndon, i.e., strings that are lexicographically smaller than any of its non-empty proper suffixes [24]. Lyndon words are objects of longstanding combinatorial interest (see e.g., [13]), and have also proved to be useful algorithmic tools in various contexts (see, e.g., [1]).

The longest Lyndon *substring* of a string is the longest factor of the Lyndon factorization of the string [5], and can be computed in linear time [9]. The longest Lyndon *subsequence* of a unary string is just one letter, which is also the only Lyndon subsequence of a unary string. A (naive) solution to find the longest Lyndon subsequence is to enumerate all distinct Lyndon subsequences, and pick the longest one. However, the number of distinct Lyndon subsequences can be as large as 2^n considering a string of increasing numbers $T = 1 \cdots n$. In fact, there are no bounds known (except when $\sigma = 1$) that bring this number in a polynomial relation with the text length n and the alphabet size σ [16], and thus deriving the longest Lyndon subsequence from all distinct Lyndon subsequences can be infeasible. In this paper, we focus on the algorithmic aspects of computing this longest Lyndon subsequence in polynomial time without the need to consider all Lyndon subsequences. In detail, we study the problems of computing

1. the lexicographically smallest (common) subsequence for each length online, cf. Section 3, and
2. the longest subsequence that is Lyndon, cf. Section 4, with two variations considering the computation as online, or the restriction that this subsequence has to be common among two given strings.

The first problem serves as an appetizer. Although the notions of *Lyndon* and *lexicographically smallest* share common traits, our solutions to the two problems are independent, but we will reuse some tools for the online computation.

2 Preliminaries

Let Σ denote a totally ordered set of symbols called *the alphabet*. An element of Σ^* is called a string. The alphabet Σ induces the *lexicographic order* $\prec$ on the set of strings Σ^*. Given a string $S \in \Sigma^*$, we denote its length with $|S|$, its i-th symbol with $S[i]$ for $i \in [1..|S|]$. Further, we write $S[i..j] = S[i] \cdots S[j]$, and we write $S[i..] = S[i..|S|]$ for the suffix of S starting at position i. A *subsequence* of a string S with length ℓ is a string $S[i_1] \cdots S[i_\ell]$ with $i_1 < \ldots < i_\ell$.

Let $\perp$ be the empty string. We stipulate that $\perp$ is lexicographically larger than every string of Σ^+. For a string S, appending $\perp$ to S yields S.

A string $S \in \Sigma^*$ is a *Lyndon word* [24] if S is lexicographically smaller than all its non-empty proper suffixes. Equivalently, a string S is a Lyndon word if and only if it is smaller than all its proper cyclic rotations.

The algorithms we present in the following may apply techniques limited to integer alphabets. However, since the final space and running times are not better than $\mathcal{O}(n)$ space and $\mathcal{O}(n \lg n)$ time, respectively, we can reduce the alphabet of T to an integer alphabet by sorting the characters in T with a comparison based sorting algorithm taking $\mathcal{O}(n \lg n)$ time and $\mathcal{O}(n)$ space, removing duplicate characters, and finally assigning each distinct character a unique rank within $[1..n]$. Hence, we assume in the following that T has an alphabet of size $\sigma \leq n$.

Algorithm 1: Computing the lexicographically smallest subsequence $D[i, \ell]$ in $T[1..i]$ of length ℓ.

```
1 D[0,1] ← ⊥
2 for i = 1 to n do                                   ▷ Initialize D[·,1]
3 |  D[i,1] ← min_{j∈[1..i]} T[j] = min(D[i−1,1], T[i])   ▷ O(1) time per
  |   entry
4 for ℓ = 2 to n do                                   ▷ Induce D[·,ℓ] from D[·,ℓ−1]
5 |  for i = 2 to i do                                 ▷ Induce D[i,ℓ]
6 |  |  if ℓ < i then D[i,ℓ] ← ⊥
7 |  |  else D[i,ℓ] ← min(D[i−1,ℓ], D[i−1,ℓ−1]T[i])
```

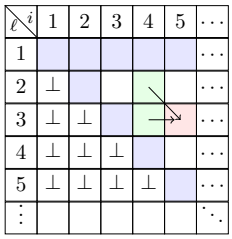

Fig. 1. Sketch of the proof of Lemma 1. We can fill the fields shaded in blue (the first row and the diagonal) in a precomputation step. Further, we know that entries left of the diagonal are all empty. A cell to the right of it (red) is based on its left-preceding and diagonal-preceding cell (green).

3 Lexicographically Smallest Subsequence

As a starter, we propose a solution for the following related problem: Compute the lexicographically smallest subsequence of T for each length $\ell \in [1..n]$ online.

3.1 Dynamic Programming Approach

The idea is to apply dynamic programming dependent on the length ℓ and the length of the prefix $T[1..i]$ in which we compute the lexicographically smallest subsequence of length ℓ. We show that the lexicographically smallest subsequence of $T[1..i]$ length ℓ, denoted by $D[i, \ell]$ is $D[i-1, \ell]$ or $D[i-1, \ell-1]T[i]$, where $D[0, \cdot] = D[\cdot, 0] = \bot$ is the empty word. See Algorithm 1 for a pseudo code.

Lemma 1. *Algorithm 1 correctly computes $D[i, \ell]$, the lexicographically smallest subsequence of $T[1..i]$ with length ℓ.*

Proof. The proof is done by induction over the length ℓ and the prefix $T[1..i]$. We observe that $D[i, \ell] = \bot$ for $i < \ell$ and $D[i, i] = T[1..i]$ since $T[1..i]$ has only one subsequence of length i. Hence, for (a) $\ell = 1$ as well as for (b) $i \leq \ell$, the claim holds. See Fig. 1 for a sketch.

Now assume that the claim holds for $D[i', \ell']$ with (a) $\ell' < \ell$ and all $i \in [1..n]$, as well as (b) $\ell' = \ell$ and all $i' \in [1..i-1]$. In what follows, we show that the claim also holds for $D[i, \ell]$ with $i > \ell > 1$. For that, let us assume that $T[1..i]$ has a subsequence L of length ℓ with $L \prec D[i, \ell]$.

If $L[\ell] \neq T[i]$, then L is a subsequence of $T[1..i-1]$, and therefore $D[i-1, \ell] \preceq L$ according to the induction hypothesis. But $D[i, \ell] \preceq D[i-1, \ell]$, a contradiction.

If $L[\ell] = T[i]$, then $L[1..\ell-1]$ is a subsequence of $T[1..i-1]$, and therefore $D[i-1, \ell-1] \preceq L[1..\ell-1]$ according to the induction hypothesis. But $D[i, \ell] \preceq D[i-1, \ell-1]T[i] \preceq L[1..\ell-1]T[i] = L$, a contradiction. Hence, $D[i, \ell]$ is the lexicographically smallest subsequence of $T[1..i]$ of length ℓ.

Unfortunately, the lexicographically smallest subsequence of a given length is not a Lyndon word in general, so this dynamic programming approach does not solve our problem finding the longest Lyndon subsequence. In fact, if T has a longest Lyndon subsequence of length ℓ, then there can be a lexicographically smaller subsequence of the same length. For instance, with $T = \texttt{aba}$, we have the longest Lyndon subsequence `ab`, while the lexicographically smallest length-2 subsequence is `aa`.

Analyzing the complexity bounds of Algorithm 1, we need $\mathcal{O}(n^2)$ space for storing the two-dimensional table $D[1..n, 1..n]$. Its initialization costs us $\mathcal{O}(n^2)$ time. Line 7 is executed $\mathcal{O}(n^2)$ time. There, we compute the lexicographical minimum of two subsequences. If we evaluate this computation with naive character comparisons, for which we need to check $\mathcal{O}(n)$ characters, we pay $\mathcal{O}(n^3)$ time in total, which is also the bottleneck of this algorithm.

Lemma 2. *We can compute the lexicographically smallest substring of T for each length ℓ online in $\mathcal{O}(n^3)$ time with $\mathcal{O}(n^2)$ space.*

3.2 Speeding Up String Comparisons

Below, we improve the time bound of Lemma 2 by representing each cell of $D[1..n, 1..n]$ with a node in a trie, which supports the following methods:

- $\mathsf{insert}(v, c)$: adds a new leaf to a node v with an edge labeled with character c, and returns a handle to the created leaf.
- $\mathsf{precedes}(u, v)$: returns true if the string represented by the node u is lexicographically smaller than the string represented by the node v.

Each cell of D stores a handle to its respective trie node. The root node of the trie represents the empty string $\perp$, and we associate $D[0, \ell] = \perp$ with the root node for all ℓ. A node representing $D[i-1, \ell-1]$ has a child representing $D[i, \ell]$ connected with an edge labeled with c if $D[i, \ell] = D[i-1, \ell-1]c$, which is a concept similar to the LZ78 trie. If $D[i, \ell] = D[i-1, \ell]$, then both strings are represented by the same trie node. Since each node stores a constant number of words and an array storing its children, the trie takes $\mathcal{O}(n^2)$ space.

Insert. A particularity of our trie is that it stores the children of a node in the order of their creation, i.e., we always make a new leaf the last among its siblings. This allows us to perform insert in constant time by representing the pointers to the children of a node by a plain dynamic array. When working with the trie, we

assure that we do not insert edges into the same node with the same character label (to prevent duplicates).

We add leaves to the trie as follows: Suppose that we compute $D[i, \ell]$. If we can copy $D[i-1, \ell]$ to $D[i, \ell]$ (Line 7), we just copy the handle of $D[i-1, \ell]$ pointing to its respective trie node to $D[i, \ell]$. Otherwise, we create a new trie leaf, where we create a new entry of D by selecting a new character ($\ell = 1$), or appending a character to one of the existing strings in D. We do not create duplicate edges since we prioritize copying to the creation of a new trie node: For an entry $D[i, \ell]$, we first default to the previous occurrence $D[i-1, \ell]$, and only create a new string $D[i-1, \ell-1]T[i]$ if $D[i-1, \ell-1]T[i] \prec D[i-1, \ell]$. $D[i-1, \ell-1]T[i]$ cannot have an occurrence represented in the trie. To see that, we observe that D obeys the invariants that (a) $D[i, \ell] = \min_{j \in [1..i]} D[j, \ell]$ (where min selects the lexicographically minimal string) and (b) all pairs of rows $D[\cdot, \ell]$ and $D[\cdot, \ell']$ with $\ell \neq \ell'$ have different entries. Since Algorithm 1 fills the entries in $D[\cdot, \ell]$ in a lexicographically non-decreasing order for each length ℓ, we cannot create duplicates (otherwise, an earlier computed entry would be lexicographically smaller than a later computed entry having the same length). The string comparison $D[i-1, \ell-1]T[i] \prec D[i-1, \ell]$ is done by calling precedes, which works as follows:

Precedes. We can implement the function precedes efficiently by augmenting our trie with the dynamic data structure of [7] supporting lowest common ancestor (LCA) queries in constant time and the dynamic data structure of [8] supporting level ancestor queries level-anc(u, d) returning the ancestor of a node u on depth d in amortized constant time. Both data structures conform with our definition of insert that only supports the insertion of *leaves*. With these data structures, we can implement precedes(u, v), by first computing the lowest ancestor w of u and v, selecting the children u' and v' of w on the paths downwards to u and v, respectively, by two level ancestor queries level-anc$(u, \text{depth}(w) + 1)$ and level-anc$(v, \text{depth}(w) + 1)$, and finally returning true if the label of the edge (w, u') is smaller than of (w, v').

We use precedes as follows for deciding whether $D[i-1, \ell-1]T[i] \prec D[i-1, \ell]$ holds: Since we know that $D[i-1, \ell-1]$ and $D[i-1, \ell]$ are represented by nodes u and v in the trie, respectively, we first check whether u is a child of v. In that case, we only have to compare $T[i]$ with $D[i-1, \ell][\ell]$. If not, then we know that $D[i-1, \ell-1]$ cannot be a prefix of $D[i-1, \ell]$, and precedes(u, v) determines whether $D[i-1, \ell-1]$ or the $\ell-1$-th prefix of $D[i-1, \ell]$ is lexicographically smaller.

Theorem 3. *We can compute the table $D[1..n, 1..n]$ in $\mathcal{O}(n^2)$ time using $\mathcal{O}(n^2)$ words of space.*

Fig. 2. Longest Lyndon subsequences of prefixes of a text T. The i-th row of bars below T depicts the selection of characters forming a Lyndon sequence. In particular, the i-th row corresponds to the longest subsequence of $T[1..9]$ for $i = 1$ (green), $T[1..11]$ for $i = 2$ (blue), and of $T[1..12]$ for $i = 3$ (red). The first row (green) corresponds also to a longest Lyndon subsequence of $T[1..10]$ and $T[1..11]$ (by extending it with $T[11]$). Extending the second Lyndon subsequence with $T[12]$ gives also a Lyndon subsequence, but is shorter than the third Lyndon subsequence (red). Having only the information of the Lyndon subsequences in $T[1..i]$ at hand seems not to give us a solution for $T[1..i+1]$.

3.3 Most Competitive Subsequence

If we want to find only the lexicographically smallest subsequence for a fixed length ℓ, this problem is also called to *Find the Most Competitive Subsequence*[5]. For that problem, there are linear-time solutions using a stack S storing the lexicographically smallest subsequence of length ℓ for any prefix $T[1..i]$ with $\ell \leq i$. Let top denote the top element of S. The idea is to scan T from left to right linearly. Given we are at a text position i, we recursively pop top as long as (a) S is not empty, (b) $T[\mathsf{top}] > T[i]$, and (c) $n - i \geq (\ell - |\mathsf{S}|)$. The last condition ensures that when we are near the end of the text, we still have enough positions in S to fill up S with the remaining positions to obtain a sequence of ℓ text positions. Finally, we put $T[i]$ on top of S if $|\mathsf{S}| < \ell$. Since a text position gets inserted into S and removed from S at most once, the algorithm runs in linear time. Consequently, if the whole text T is given (i.e., not online), this solution solves our problem in the same time and space bounds by running the algorithm for each ℓ separately.

Given $T = \mathtt{cba}$ as an example, for $\ell = 3$, we push all three characters of T onto S and output $\mathtt{cba}$. For $\ell = 2$, we first push $T[1] = \mathtt{c}$ onto S, but then pop it and push $\mathtt{b}$ onto S. Finally, although $T[3] < T[2]$, we do not discard $T[2] = \mathtt{b}$ stored on S since we need to produce a subsequence of length $\ell = 2$.

3.4 Lexicographically Smallest Common Subsequence

Another variation is to ask for the lexicographically smallest subsequence of each distinct length that is common with two strings X and Y. Luckily, our ideas of Sections 3.1 and 3.2 can be straightforwardly translated. For that, our matrix D becomes a cube $D_3[1..L, 1..|X|, 1..|Y|]$ with $L := \min(|X|, |Y|)$, and we set

$$D_3[\ell, x+1, y+1] = \min \begin{cases} D_3[\ell-1, x, y]X[x+1] \text{ if } X[x+1] = Y[y+1], \\ D_3[\ell, x, y+1], \\ D_3[\ell, x+1, y], \end{cases}$$

[5] https://leetcode.com/problems/find-the-most-competitive-subsequence/

with $D_3[0,\cdot,\cdot] = D_3[\ell,x,y] = \bot$ for all ℓ, x, y with $\mathrm{LCS}(X[1..x], Y[1..y]) < \ell$, where LCS denotes the length of a longest common subsequence of X and Y. This gives us an induction basis similar to the one used in the proof of Lemma 1, such that we can use its induction step analogously. The table D_3 has $\mathcal{O}(n^3)$ cells, and filling each cell can be done in constant time by representing each cell as a pointer to a node in the trie data structure proposed in Section 3.2. For that, we ensure that we never insert a subsequence of D_3 into the trie twice. To see that, let $L \in \Sigma^+$ be a subsequence computed in D_3, and let $D_3[\ell,x,y] = L$ be the entry at which we called insert to create a trie node for L (for the first time). Then $\ell = |L|$, and $X[1..x]$ and $Y[1..y]$ are the shortest prefixes of X and Y, respectively, containing L as a subsequence. Since $D_3[\ell,x,y] = \min_{x' \in [1..x], y' \in [1..y]} D_3[\ell,x',y']$, all other entries $D_3[\ell,x',y'] = L$ satisfy $D_3[\ell,x'-1,y'] = L$ or $D_3[\ell,x',y'-1] = L$, so we copy the trie node handle representing L instead of calling insert when filling out $D_3[\ell,x',y']$.

Theorem 4. *Given two strings X, Y of length n, we can compute the lexicographically smallest common subsequence for each length $\ell \in [1..n]$ in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n^3)$ space.*

4 Computing the Longest Lyndon Subsequence

In the following, we want to compute the longest Lyndon subsequence of T. See Fig. 2 for examples of longest Lyndon subsequences. Compared to the former introduced dynamic programming approach for the lexicographically smallest subsequences, we follow the sketched solution for the most competitive subsequence using a stack, which here simulates a traversal of the trie τ storing all pre-Lyndon subsequences. A *pre-Lyndon subsequence* is a subsequence that is Lyndon or can be extended with characters at its right end to become Lyndon. τ is a subgraph of the trie storing all subsequences, sharing the same root. This subgraph is connected since, by definition, there is no string S such that WS forms a pre-Lyndon word for a non-pre-Lyndon word W (otherwise, we could extend WS to a Lyndon word, and so W, too). We say that the *string label* of a node v is the string read from the edges on the path from root to v. We associate the label c of each edge of the trie with the leftmost possible position such that the string label V of v is associated with the sequence of text positions $i_1 < i_2 < \cdots < i_{|V|}$ and $T[i_1]T[i_2]\cdots T[i_{|V|}] = V$.

4.1 Basic Trie Traversal

Problems already emerge when considering the construction of τ since there are texts like $T = 1 \cdots n$ for which τ has $\Theta(2^n)$ nodes. Instead of building τ, we simulate a preorder traversal on it. With simulation we mean that we enumerate the pre-Lyndon subsequences of T in lexicographic order. For that, we maintain a stack S storing the text positions $(i_1, \ldots, i_\ell)$ with $i_1 < \cdots < i_\ell$ associated with the path from the root to the node v we currently visit i.e., $i_1, \ldots, i_\ell$ are

the smallest positions with $T[i_1]\cdots T[i_\ell]$ being the string label of v, which is a pre-Lyndon word. When walking down, we select the next text position $i_{\ell+1}$ such that $T[i_1]\cdots T[i_\ell]T[i_{\ell+1}]$ is a pre-Lyndon word. If such a text position does not exist, we backtrack by popping i_ℓ from S, and push the smallest text position $i'_\ell > i_{\ell-1}$ with $T[i'_\ell] > T[i_\ell]$ onto S and recurse. Finally, we check at each state of S storing the text positions $(i_1, \ldots, i_\ell)$ whether $T[i_1]\cdots T[i_\ell]$ is a Lyndon word. For that, we make use of the following facts:

Facts about Lyndon Words. A Lyndon word cannot have a *border*, that is, a non-empty proper prefix that is also a suffix of the string [9, Prop. 1.1]. A *pre-Lyndon word* is a (not necessarily proper) prefix of a Lyndon word. Given a string S of length n, an integer $p \in [1..n]$ is a *period* of S if $S[i] = S[i+p]$ for all $i \in [1..n-p]$. The length of a string is always one of its periods. We use the following facts:

(Fact 1) Only the length $|S|$ is the period of a Lyndon word S.
(Fact 2) The prefix $S[1..|p|]$ of a pre-Lyndon word S with period p is a Lyndon word. In particular, a pre-Lyndon word S with period $|S|$ is a Lyndon word.
(Fact 3) Given a pre-Lyndon word S with period p and a character $c \in \Sigma$, then
- Sc is a pre-Lyndon word of the same period if and only if $S[|S| - p + 1] = c$ and S is not the largest character in Σ.
- Sc is a Lyndon word if and only if $S[|S| - p + 1] < c$. In particular, if S is a Lyndon word, then Sc is a Lyndon word if and only if $S[1]$ is smaller than c.

Proof. **Fact 1** If S has a period less than $|S|$, then S is bordered.
Fact 2 If $S[1..|p|]$ would not be Lyndon, then there was a suffix X of S with $X \prec S[1..|X|]$, hence $XZ \prec SZ$ for every $Z \in \Sigma^*$, so S cannot be pre-Lyndon.
Fact 3 "⇒": If $T := Sc$ is a pre-Lyndon word with the same period as S, then T has a border $T[p+1..|T|] = T[1..|T|-p]$. "⇐": Follows from Fact 2 and [9, Corollary 1.4].

Checking pre-Lyndon Words. Now suppose that our stack S stores the text positions $(i_1, \ldots, i_\ell)$. To check whether $T[i_1]\cdots T[i_\ell]c$ for a character $c \in \Sigma$ is a pre-Lyndon word or whether it is a Lyndon word, we augment each position i_j stored in S with the period of $T[i_1]\cdots T[i_j]$, for $j \in [1..\ell]$, such that we can make use of Fact 3 to compute the period and check whether $T[i_1]\cdots T[i_j]c$ is a pre-Lyndon word, both in constant time, for $c \in \Sigma$.

Trie Navigation. To find the next text position $i_{\ell+1}$, we may need to scan $\mathcal{O}(n)$ characters in the text, and hence need $\mathcal{O}(n)$ time for walking down from a node to one of its children. If we restrict the alphabet to be integer, we can augment each text position i to store the smallest text position i_c with $i < i_c$ for each character $c \in \Sigma$ such that we can visit the trie nodes in constant time per node during our preorder traversal.

This gives already an algorithm that computes the longest Lyndon subsequence with $\mathcal{O}(n\sigma)$ space and time linear to the number of nodes in τ. However, since the number of nodes can be exponential in the text length, we present ways to omit nodes that do not lead to the solution. Our aim is to find a rule to judge whether a trie node contributes to the longest Lyndon subsequence to leave certain subtrees of the trie unexplored. For that, we use the following property:

Lemma 5. *Given a Lyndon word V and two strings U and W such that UW is a Lyndon word, $V \prec U$, and $|V| \geq |U|$, then VW is also a Lyndon word with $VW \prec UW$.*

Proof. Since $V \prec U$ and V is not a prefix of U, $U \succ VW$. In what follows, we show that $S \succ VW$ for every proper suffix S of VW.

- If S is a suffix of W, then $S \succeq UW \succeq U \succ VW$ because S is a suffix of the Lyndon word UW.
- Otherwise, ($|S| > |W|$), S is of the form $V'W$ for a proper suffix V' of V. Since V is a Lyndon word, $V' \succ V$, and V' is not a prefix of V (Lyndon words are border-free). Hence, $V'W \succeq V' \succ VW$.

Note that U in Lemma 5 is a pre-Lyndon word since it is the prefix of the Lyndon word UW.

Our algorithmic idea is as follows: We maintain an array $\mathsf{L}[1..n]$, where $\mathsf{L}[\ell]$ is the smallest text position i such that our traversal has already explored a length-ℓ Lyndon subsequence of $T[1..i]$. We initialize the entries of L with ∞ at the beginning. Now, whenever we visit a node u whose string label is a pre-Lyndon subsequence $U = T[i_1] \cdots T[i_\ell]$ with $\mathsf{L}[\ell] \leq i_\ell$, then we do not explore the children of u. In this case, we call u *irrelevant*. By skipping the subtree rooted at u, we do not omit the solution due to Lemma 5: When $\mathsf{L}[\ell] \leq i_\ell$, then there is a Lyndon subsequence V of $T[1..i_\ell]$ with $V \prec U$ (since we traverse the trie in lexicographically order) and $|V| = |U|$. Given there is a Lyndon subsequence UW of T, then we have already found VW earlier, which is also a Lyndon subsequence of T with $|VW| = |UW|$.

Next, we analyze the complexity of this algorithm, and propose an improved version. For that, we say that a string is *immature* if it is pre-Lyndon but not Lyndon. We also consider a subtree rooted at a node u as pruned if u is irrelevant, i.e., the algorithm does not explore this subtree. Consequently, irrelevant nodes are leaves in the pruned subtree, but not all leaves are irrelevant (consider a Lyndon subsequence using the last text position $T[n]$). Further, we call a node Lyndon or immature if its string label is Lyndon or immature, respectively. (All nodes in the trie are either Lyndon or immature.)

Time Complexity. Suppose that we have the text positions $(i_1, \ldots, i_\ell)$ on S such that $U := T[i_1] \cdots T[i_\ell]$ is a Lyndon word. If $\mathsf{L}[\ell] > i_\ell$, then we lower $\mathsf{L}[\ell] \leftarrow i_\ell$. We can lower an individual entry of L at most n times, or at most n^2 times in total for all entries. If a visited node is Lyndon, we only explore its subtree if we were able to lower an entry of L. Hence, we visit at most n^2

Lyndon nodes that trigger a decrease of the values in L. While each node can have at most σ children, at most one child can be immature due to Fact 3. Since the depth of the trie is at most n, we therefore visit $\mathcal{O}(n\sigma)$ nodes between two updates of L (we pop at most n nodes from the stack, and try to explore at most σ siblings of each node on the stack). These nodes are leaves (of the pruned trie) or immature nodes. Thus, we traverse $\mathcal{O}(n^3\sigma)$ nodes in total.

Theorem 6. *We can compute the longest Lyndon subsequence of a string of length n in $\mathcal{O}(n^3\sigma)$ time using $\mathcal{O}(n\sigma)$ words of space.*

4.2 Improving Time Bounds

We further improve the time bounds by avoiding visiting irrelevant nodes due to the following observation: First, we observe that the number of *relevant* (i.e., non-irrelevant) nodes that are Lyndon is $\mathcal{O}(n^2)$. Since all nodes have a depth of at most n, the total number of relevant nodes in $\mathcal{O}(n^3)$. Suppose we are at a node u, and S stores the positions $(i_1, \ldots, i_\ell)$ such that $T[i_1]\cdots T[i_\ell]$ is the string label of u. Let p denote the smallest period of $T[i_1]\cdots T[i_\ell]$. Then we do not want to consider all σ children of u, but only those whose edges to u have a label $c \geq T[i_{\ell-p+1}]$ such that c occurs in $T[i_\ell + 1..\mathsf{L}[\ell+1]-1]$ (otherwise, there is already a Lyndon subsequence of length $\ell+1$ lexicographically smaller than $T[i_1]\cdots T[i_\ell]c$). In the context of our preorder traversal, each such child can be found iteratively using range successor queries: starting from $b = T[i_{\ell-p+1}]-1$, we want to find the *lexicographically smallest* character $c > b$ such that c occurs in $T[i_\ell+1..\mathsf{L}[\ell+1]-1]$. In particular, we want to find the leftmost such occurrence. A data structure for finding c in this interval is the wavelet tree [14] returning the position of the *leftmost* such c (if it exists) in $\mathcal{O}(\lg\sigma)$ time. In particular, we can use the wavelet tree instead of the $\mathcal{O}(n\sigma)$ pointers to the subsequent characters to arrive at $\mathcal{O}(n)$ words of space. Finally, we do not want to query the wavelet tree each time, but only whenever we are sure that it will lead us to a relevant Lyndon node. For that, we build a range maximum query (RMQ) data structure on the characters of the text T in a preprocessing step. The RMQ data structure of [3] can be built in $\mathcal{O}(n)$ time; it answers queries in constant time. Now, in the context of the above traversal where we are at a node u with S storing $(i_1, \ldots, i_\ell)$, we query this RMQ data structure for the largest character c in $T[i_\ell+1..\mathsf{L}[\ell+1]-1]$ and check whether the sequence $S := T[i_1]\cdots T[i_\ell]c$ forms a (pre-)Lyndon word.

- If S is not pre-Lyndon, i.e., $T[i_{\ell-p+1}] > c$ for p being the smallest period of $T[i_1]\cdots T[i_\ell]$, we are sure that the children of u cannot lead to Lyndon subsequences [9, Prop. 1.5].
- If S is immature, i.e., $T[i_{\ell-p+1}] = c$, u has exactly one child, and this child's string label is S. Hence, we do not need to query for other Lyndon children.
- Finally, if S is Lyndon, i.e., $T[i_{\ell-p+1}] < c$, we know that there is at least one child of u that will trigger an update in L and thus is a relevant node.

This observation allows us to find all relevant children of u (including the single immature child, if any) by iteratively conducting $\mathcal{O}(k)$ range successor queries, where k is the number of children of u that are relevant Lyndon nodes. Thus, if we condition the execution of the aforementioned wavelet tree query with an RMQ query result on the same range, the total number of wavelet tree queries can be bounded by $\mathcal{O}(n^2)$. This gives us $\mathcal{O}(n^3 + n^2 \lg \sigma) = \mathcal{O}(n^3)$ time for $\sigma = \mathcal{O}(n)$ (which can be achieved by an $\mathcal{O}(n \log n)$ time re-enumeration of the alphabet in a preliminary step).

Theorem 7. *We can compute the longest Lyndon subsequence of a string of length n in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n)$ words of space.*

In particular, the algorithm computes the lexicographically smallest one among all longest Lyndon subsequences: Assume that this subsequence L is not computed, then we did not explore the subtree of the original trie τ (before pruning) containing the node with string label L. Further, assume that this subtree is rooted at an irrelevant node u whose string label is the pre-Lyndon subsequence U. Then U is a prefix of L, and because u is irrelevant (i.e., we have not explored u's children), there is a node v whose string label is a Lyndon word V with $V \prec U$ and $|V| = |U|$. In particular, the edge of v to v's parent is associated with a text position equal to or smaller than the associated text position of the edge between u and u's parent. Hence, we can extend V to the Lyndon subsequence $VL[|U| + 1]..]$ being lexicographically smaller than L, a contradiction.

4.3 Online Computation

If we allow increasing the space usage in order to maintain the trie data structure introduced in Section 3.2, we can modify our $\mathcal{O}(n^3\sigma)$-time algorithm of Section 4.1 to perform the computation online, i.e., with T given as a text stream. To this end, let us recall the trie τ of all pre-Lyndon subsequences introduced at the beginning of Section 4. In the online setting, when reading a new character c, for each subsequence S given by a path from τ's root (S may be empty), we add a new node for Sc if Sc is a pre-Lyndon subsequence that is not yet represented by such a path. Again, storing all nodes of τ explicitly would cost us too much space. Instead, we explicitly represent only the visited nodes of the trie τ with an explicit trie data structure τ' such that we can create pointers to the nodes. (In other words, τ' is a lazy representation of τ.) The problem is that we can no longer perform the traversal in lexicographic order, but instead keep multiple fingers in the trie τ' constructed up so far, and use these fingers to advance the trie traversal in text order.

With a different traversal order, we need an updated definition of $\mathsf{L}[1..n]$: Now, while the algorithm processes $T[i]$, the entry $\mathsf{L}[\ell]$ stores the lexicographically smallest length-ℓ Lyndon subsequence of $T[1..i]$ (represented by a pointer to the corresponding node of τ'). Further, we maintain σ lists storing pointers to nodes of τ'. Initially, τ' consists only of the root node, and each list stores only

the root node. Whenever we read a new character $T[i]$ from the text stream, for each node v of the $T[i]$-th list, we add a leaf λ connected to v by an edge with label $T[i]$. Our algorithm adheres to the invariant that λ's string label S is a pre-Lyndon word so that τ' is always a subtree of τ. If S is a Lyndon word satisfying $S \prec \mathsf{L}[|S|]$ (which can be tested using the data structure of Section 3.2), we further set $\mathsf{L}[|S|] := S$. This completes the process of updating $\mathsf{L}[1..n]$. Next, we clear the $T[i]$-th list and iterate again over the newly created leaves. For each such leaf λ with label S, we check whether λ is relevant, i.e., whether $S \preceq \mathsf{L}[|S|]$. If λ turns out irrelevant, we are done with processing it. Otherwise, we put λ into the c-th list for each character $c \in \Sigma$ such that Sc is a pre-Lyndon word. By doing so, we effectively create new events that trigger a call-back to the point where we stopped the trie traversal.

Overall, we generate exactly the nodes visited by the algorithm of Section 4.1. In particular, there are $\mathcal{O}(n^3)$ relevant nodes, and for each such node, we issue $\mathcal{O}(\sigma)$ events. The operations of Section 3.2 take constant amortized time, so the overall time and space complexity of the algorithm are $\mathcal{O}(n^3\sigma)$.

Theorem 8. *We can compute the longest Lyndon subsequence online in $\mathcal{O}(n^3\sigma)$ time using $\mathcal{O}(n^3\sigma)$ space.*

5 Longest Common Lyndon Subsequence

Given two strings X and Y, we want to compute the longest common subsequence of X and Y that is Lyndon. For that, we can extend our algorithm finding the longest Lyndon subsequence of a single string as follows. First, we explore in depth-first order the trie of all *common* pre-Lyndon subsequences of X and Y. A node is represented by a pair of positions (x, y) such that, given the path from the root to a node v of depth ℓ visits the nodes $(x_1, y_1), \ldots, (x_\ell, y_\ell)$ with $L = X[x_1] \cdots X[x_\ell] = Y[y_1] \cdots Y[y_\ell]$ being a pre-Lyndon word, L is neither a subsequence of $X[1..x_\ell - 1]$ nor of $Y[1..y_\ell - 1]$, i.e., x_ℓ and y_ℓ are the leftmost such positions. The depth-first search works like an exhaustive search in that it tries to extend L with each possible character in Σ having an occurrence in both remaining suffixes $X[x_\ell + 1..]$ and $Y[y_\ell + 1..]$, and then, after having explored the subtree rooted at v, visits its lexicographically succeeding sibling nodes (and descends into their subtrees) by checking whether $L[1..|L| - 1]$ can be extended with a character $c > L[|L|]$ appearing in both suffixes $X[x_{\ell-1} + 1..]$ and $Y[y_{\ell-1} + 1..]$.

The algorithm uses again the array L to check whether we have already found a lexicographically smaller Lyndon subsequence with equal or smaller ending positions in X and Y than the currently constructed pre-Lyndon subsequence. For that, $\mathsf{L}[\ell]$ stores not only one position, but a list of positions (x, y) such that $X[1..x]$ and $Y[1..y]$ have a *common* Lyndon subsequence of length ℓ. Although there can be n^2 such pairs of positions, we only store those that are pairwise non-dominated. A pair of positions (x_1, y_1) is called *dominated* by a pair $(x_2, y_2) \neq (x_1, y_1)$ if $x_2 \leq x_1$ and $y_2 \leq y_1$. A set storing pairs in $[1..n] \times [1..n]$ can have at most n elements that are pairwise non-dominated, and hence $|\mathsf{L}[\ell]| \leq n$.

At the beginning, all lists of L are empty. Suppose that we visit a node v with pair (x_ℓ, y_ℓ) representing a common Lyndon subsequence of length ℓ. Then we query whether $\mathsf{L}[\ell]$ has a pair dominating (x_ℓ, y_ℓ). In that case, we can skip v and its subtree. Otherwise, we insert (x_ℓ, y_ℓ) and remove pairs in $\mathsf{L}[\ell]$ that are dominated by (x_ℓ, y_ℓ). Such an insertion can happen at most n^2 times. Since $\mathsf{L}[1..n]$ maintains n lists, we can update L at most n^3 times in total. Checking for domination and insertion into L takes $\mathcal{O}(n)$ time. The former can be accelerated to constant time by representing $\mathsf{L}[\ell]$ as an array R_ℓ storing in $R_\ell[i]$ the value y of the tuple $(x, y) \in \mathsf{L}[\ell]$ with $x \leq i$ and the lowest possible y, for each $i \in [1..n]$. Then a pair $(x, y) \notin \mathsf{L}[\ell]$ is dominated if and only if $R_\ell[x] \leq y$.

Example 9. For $n = 10$, let $\mathsf{L}_\ell = [(3, 9), (5, 4), (8, 2)]$. Then all elements in L_ℓ are pairwise non-dominated, and $R_\ell = [\infty, \infty, 9, 9, 4, 4, 4, 2, 2, 2]$. Inserting $(3, 2)$ would remove all elements of L_ℓ, and update all entries of R_ℓ. Alternatively, inserting $(7, 3)$ would only involve updating $R_\ell[7] \leftarrow 3$; since the subsequent entry $R_\ell[8] = 2$ is less than $R_\ell[7]$, no subsequent entries need to be updated.

An update in $\mathsf{L}[\ell]$ involves changing $\mathcal{O}(n)$ entries of R_ℓ, but that cost is dwarfed by the cost for finding the next common Lyndon subsequence that updates L. Such a subsequence can be found while visiting $\mathcal{O}(n\sigma)$ irrelevant nodes during a naive depth-first search (cf. the solution of Section 3.1 computing the longest Lyndon sequence of a single string). Hence, the total time is $\mathcal{O}(n^4\sigma)$.

Theorem 10. *We can compute the longest common Lyndon subsequence of a string of length n in $\mathcal{O}(n^4\sigma)$ time using $\mathcal{O}(n^3)$ words of space.*

Open Problems Since we shed light on the computation of the longest (common) Lyndon subsequence for the very first time, we are unaware of the optimality of our solutions. It would be interesting to find non-trivial lower bounds that would justify our rather large time and space complexities.

Acknowledgments This work was supported by JSPS KAKENHI Grant Numbers JP20H04141 (HB), JP19K20213 (TI), JP21K17701 and JP21H05847 (DK). TK was supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.

References

1. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. SIAM J. Comput. **46**(5), 1501–1514 (2017)
2. de Beauregard Robinson, G.: On the representations of the symmetric group. American Journal of Mathematics **60**(3), 745–760 (1938)
3. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**(2), 75–94 (2005)

4. Biedl, T.C., Biniaz, A., Cummings, R., Lubiw, A., Manea, F., Nowotka, D., Shallit, J.O.: Rollercoasters: Long sequences without short runs. SIAM J. Discret. Math. **33**(2), 845–861 (2019)
5. Chen, K.T., Fox, R.H., Lyndon, R.C.: Free differential calculus, IV. The quotient groups of the lower central series. Annals of Mathematics pp. 81–95 (1958)
6. Chowdhury, S.R., Hasan, M.M., Iqbal, S., Rahman, M.S.: Computing a longest common palindromic subsequence. Fundam. Informaticae **129**(4), 329–340 (2014)
7. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005)
8. Dietz, P.F.: Finding level-ancestors in dynamic trees. In: Proc. WADS. LNCS, vol. 519, pp. 32–40 (1991)
9. Duval, J.: Factorizing words over an ordered alphabet. J. Algorithms **4**(4), 363–381 (1983)
10. Elmasry, A.: The longest almost-increasing subsequence. Inf. Process. Lett. **110**(16), 655–658 (2010)
11. Fujita, K., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Longest common rollercoasters. In: Proc. SPIRE. LNCS, vol. 12944, pp. 21–32 (2021)
12. Gawrychowski, P., Manea, F., Serafin, R.: Fast and longest rollercoasters. In: Proc. STACS. LIPIcs, vol. 126, pp. 30:1–30:17 (2019)
13. Glen, A., Simpson, J., Smyth, W.F.: Counting Lyndon factors. Electron. J. Comb. **24**(3), P3.28 (2017)
14. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA. pp. 841–850 (2003)
15. He, X., Xu, Y.: The longest commonly positioned increasing subsequences problem. J. Comb. Optim. **35**(2), 331–340 (2018)
16. Hirakawa, R., Nakashima, Y., Inenaga, S., Takeda, M.: Counting Lyndon subsequences. In: Proc. PSC. pp. 53–60 (2021)
17. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM **24**(4), 664–675 (1977)
18. Inenaga, S., Hyyrö, H.: A hardness result and new algorithm for the longest common palindromic subsequence problem. Inf. Process. Lett. **129**, 11–15 (2018)
19. Kiyomi, M., Horiyama, T., Otachi, Y.: Longest common subsequence in sublinear space. Inf. Process. Lett. **168**, 106084 (2021)
20. Knuth, D.: Permutations, matrices, and generalized Young tableaux. Pac. J. Math. **34**, 709–727 (1970)
21. Kosche, M., Koß, T., Manea, F., Siemer, S.: Absent subsequences in words. In: Proc. RP. LNCS, vol. 13035, pp. 115–131 (2021)
22. Kosowski, A.: An efficient algorithm for the longest tandem scattered subsequence problem. In: Proc. SPIRE. LNCS, vol. 3246, pp. 93–100 (2004)
23. Kutz, M., Brodal, G.S., Kaligosi, K., Katriel, I.: Faster algorithms for computing longest common increasing subsequences. J. Discrete Algorithms **9**(4), 314–325 (2011)
24. Lyndon, R.C.: On Burnside's problem. Transactions of the American Mathematical Society **77**(2), 202–215 (1954)
25. Schensted, C.: Longest increasing and decreasing subsequences. Canadian Journal of Mathematics **13**, 179–191 (1961)
26. Ta, T.T., Shieh, Y., Lu, C.L.: Computing a longest common almost-increasing subsequence of two sequences. Theor. Comput. Sci. **854**, 44–51 (2021)
27. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)