Algorithmica (2024) 86:735–756
https://doi.org/10.1007/s00453-023-01125-z

Computing Longest Lyndon Subsequences and Longest Common Lyndon Subsequences

Hideo Bannai[1] · **Tomohiro I.**[2] · **Tomasz Kociumaka**[3] · **Dominik Köppl**[1,4] · **Simon J. Puglisi**[5]

Received: 29 September 2022 / Accepted: 6 April 2023 / Published online: 6 May 2023

Abstract

Given a string T of length n whose characters are drawn from an ordered alphabet of size σ, its longest Lyndon subsequence is a maximum-length 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 $\mathcal{O}(n^3)$ space and time. Our first result can be extended to find the longest common Lyndon subsequence of two strings of length at most n in $\mathcal{O}(n^4\sigma)$ time using $\mathcal{O}(n^2)$ space.

Keywords Lyndon word · Subsequence · Lexicographic order · Dynamic programming

Parts of this work have already been presented at the 33rd International Workshop on Combinatorial Algorithms [4].

✉ Dominik Köppl
dominik.koeppl@uni-muenster.de

Hideo Bannai
hdbn.dsc@tmd.ac.jp

Tomohiro I.
tomohiro@ai.kyutech.ac.jp

Tomasz Kociumaka
tomasz.kociumaka@mpi-inf.mpg.de

Simon J. Puglisi
simon.puglisi@helsinki.fi

1 M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

2 Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Japan

3 Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

4 Department of Computer Science, University of Münster, Münster, Germany

5 Department of Computer Science, Helsinki University, Helsinki, Finland

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 [11, 33]. Several variants of this problem have been proposed [14, 28]. These problems generalize to the task of finding such a subsequence that is not only present in one string, but common to two given strings [21, 31, 34], which can also be viewed as a specialization of the longest common subsequence problem [23, 27, 35].

More recently, the problem of computing the longest square word that is a subsequence [30], the longest palindrome that is a subsequence [9, 25], the lexicographically smallest absent subsequence [29], and longest rollercoasters [6, 16, 18] have been considered.

Here, we focus on subsequences that are Lyndon words, i.e., strings that are lexicographically smaller than all of their non-empty proper suffixes [32]. Lyndon words are objects of longstanding combinatorial interest (see, e.g., [19]), and they have also proved to be useful algorithmic tools in various contexts (see, e.g., [3]). The longest Lyndon *substring* of a string is the longest factor of the Lyndon factorization of the string [8], and it can be computed in linear time [13]. 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, e.g., for 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 [22], and thus deriving the longest Lyndon subsequence from all distinct Lyndon subsequences can be infeasible. In this article, we focus on the algorithmic aspects of computing this longest Lyndon subsequence in polynomial time without the need to consider all Lyndon subsequences. Specifically, we study the problems of computing:

1. the lexicographically smallest (common) subsequence of each length (in Sect. 3), and
2. the longest Lyndon subsequence (in Sect. 4), with two variations considering online computation (in Sect. 4.3) and the restriction that this subsequence has to be common to two given strings (in Sect. 5).

The first problem serves as an appetizer. Although the notions of *Lyndon* and *lexicographically smallest* subsequences share common traits, our solutions to the two problems are mostly independent (except for some tools shared by the online algorithms for both problems).

Compared to an earlier conference version of this paper [4], we describe here an algorithm with significantly improved time complexity for the online setting. Additionally, we added more illustrations, examples, and the analysis of special cases with simpler algorithmic ideas to ease the understanding of the article. Last but not least (in Sect. 6), we evaluate the implementation of one of our proposed algorithms on commonly studied datasets.

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^*. We denote the empty string with ε. Given a string $S \in \Sigma^*$, we denote its length with $|S|$ and its i-th symbol with $S[i]$ for $i \in [1..|S|]$.[1] Further, for integers $1 \le i \le j \le |S|$, we write $S[i..j] = S[i] \cdots S[j]$ to denote the substring of $|S|$ starting at position i and ending at position j and $S[i..] = S[i..|S|]$ to denote the suffix of S starting at position i. The empty string is a substring of every string S and can be referred to as $S[j+1..j]$ for any $j \in [0..|S|]$. For $\ell \in [0..|S|]$, a length-ℓ subsequence of a string S is a string $S[i_1] \cdots S[i_\ell]$ with $i_1 < \cdots < i_\ell$. For a string V, we denote $\mathsf{pos}_S(V) = \min\{i \in [0..|S|] : V \text{ is a subsequence of } S[1..i]\}$; in particular, $\mathsf{pos}_S(V) = 0$ if $V = \varepsilon$ and, following the convention that $\min \emptyset = \infty$, we assume $\mathsf{pos}_S(V) = \infty$ if V is not a subsequence of S.

A non-empty string is a Lyndon word [32] if it is lexicographically smaller than all its non-empty proper suffixes. Equivalently, a string is a Lyndon word if and only if it is smaller than all its proper cyclic rotations.

The algorithms we present in the following assume that the input consists of strings of length at most n whose characters are drawn from an integer alphabet $\Sigma := [1..\sigma]$ of size $\sigma = \mathcal{O}(n)$.[2]

3 Lexicographically Smallest Subsequence

As a starter, we propose a solution for the following related problem: Maintain, for each length ℓ, the lexicographically smallest length-ℓ subsequence of T as the characters of T arrive online one at a time (in the left-to-right order).

3.1 Dynamic Programming Approach

The idea is to apply dynamic programming that computes, for all lengths $0 \le \ell \le i \le n$, the lexicographically smallest length-ℓ subsequence of $T[1..i]$, denoted by $D[i, \ell]$. We observe that $D[i, 0] = \varepsilon$ is the empty word and $D[i, i] = T[1..i]$. In the remaining cases, our algorithm considers $D[i-1, \ell]$ or $D[i-1, \ell-1] \cdot T[i]$ as candidates for $D[i, \ell]$; see Algorithm 1 for a pseudocode and Fig. 1 for an example.

Lemma 1 *For all $0 \le \ell \le i \le n$, Algorithm 1 correctly computes $D[i, \ell]$, the lexicographically smallest subsequence of $T[1..i]$ of length ℓ.*

[1] For arbitrary integers p, q, we write $[p..q] = \{i \in \mathbb{Z} : p \le i \le q\}$.

[2] One can reduce the alphabet to an integer alphabet by sorting the characters of the input string 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 $[1.. \min(\sigma, n)]$. However, such a reduction does not work for online algorithms, and it would constitute a bottleneck for algorithms running in $o(n \lg n)$ time.

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

```
1 D[0, 0] ← ε
2 for i ← 1 to n do                          ▷ Deduce D[i, ·] from D[i − 1, ·]
3     D[i, 0] ← ε
4     for ℓ ← 1 to i − 1 do
5         D[i, ℓ] ← min(D[i − 1, ℓ], D[i − 1, ℓ − 1] · T[i])
6     D[i, i] ← D[i − 1, i − 1] · T[i]
```

$T =$	b	c	c	a	d	b	a	c	c	b	c	d
ℓ \ i	1	2	3	4	5	6	7	8	9	10	11	12
1	b	b	b	a	a	a	a	a	a	a	a	a
2	⊥	bc	bc	ba	ad	ab	aa	aa	aa	aa	aa	aa
3	⊥	⊥	bcc	bca	bad	adb	aba	aac	aac	aab	aab	aab
4	⊥	⊥	⊥	bcca	bcad	badb	adba	abac	aacc	aacb	aabc	aabc
5	⊥	⊥	⊥	⊥	bccad	bcadb	babda	adbac	abacc	aaccb	aacbc	aabcd
⋮												

Fig. 1 The lexicographically smallest subsequences of the prefixes of the example string $T =$ bccadbaccbcd. We only show the output for $\ell \in [1..5]$ and denote the undefined values $D[i, \ell]$ (for $i < \ell$) with $\perp$

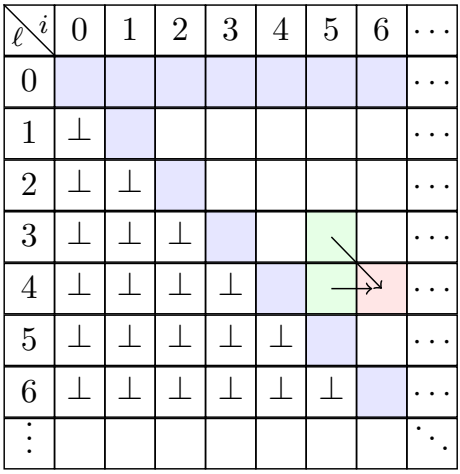


Fig. 2 Sketch of the proof of Lemma 1. We can easily fill the fields shaded in blue (the 0-th row and the main diagonal). Further, the entries to the left of the diagonal are all undefined (denoted $\perp$). A cell to the right of it (red) is based on its left-preceding and diagonal-preceding cell (green) (Color figure online)

Proof The proof is done by induction over the prefix length i. We first observe that $D[i, 0] = \varepsilon$ (the only length-0 subsequence of any string) and $D[i, i] = T[1..i]$ (the only length-i subsequence of $T[1..i]$).

In what follows, we show that the claim also holds for $D[i, \ell]$ with $0 \leq \ell < i$ assuming that all entries $D[i-1, \cdot]$ have been computed correctly. Note that $D[i-1, \ell]$ and $D[i-1, \ell-1] \cdot T[i]$ are both length-ℓ subsequences of $T[1..i]$. Hence, it suffices to prove that one of these two subsequences is the lexicographically smallest one. For a proof by contradiction, suppose that $T[1..i]$ has a length-ℓ subsequence 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. However, $D[i, \ell] \preceq D[i-1, \ell]$; 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. However, $D[i, \ell] \preceq D[i-1, \ell-1] \cdot T[i] \preceq L[1..\ell-1] \cdot T[i] = L$; a contradiction. Hence, $D[i, \ell]$ is indeed the lexicographically smallest subsequence of $T[1..i]$ of length ℓ. □

Let us analyze the complexity of Algorithm 1. If we stored the subsequences explicitly, the entries of our two-dimensional table $D[0..n, 0..n]$ would occupy $\mathcal{O}(n^3)$ space in total. However, in order to reduce the space consumption to $\mathcal{O}(n^2)$, we just store a flag that determines whether we built $D[i, j]$ from $D[i-1, \ell]$ or $D[i-1, \ell-1] \cdot T[i]$. To restore the string represented by $D[i, j]$, we backtrack with the help of the stored flags while reading $\mathcal{O}(n)$ cells and characters. In this setting, the initialization of entries $D[i, 0]$ and $D[i, i]$ costs $\mathcal{O}(n^2)$ time. Line 5, where we compute the lexicographical minimum of two subsequences, is executed $\mathcal{O}(n^2)$ times. If we perform this computation with naive character comparisons, for which we need to check $\mathcal{O}(n)$ characters (which we first need to restore by reading $\mathcal{O}(n)$ previous cells), we pay $\mathcal{O}(n^3)$ time in total, which is the bottleneck of this algorithm.

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

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 of 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, $T = \texttt{aba}$ has the longest Lyndon subsequence `ab`, while the lexicographically smallest length-2 subsequence is `aa`.

3.2 Speeding Up String Comparisons

Below, we improve the time bound of Lemma 2 by maintaining the entries of the $D[0..n, 0..n]$ table in a *trie* [15]. Mathematically, the trie of a string family is defined as a rooted tree whose nodes represent all the prefixes of the strings in the family. (Multiple strings may share the same prefix.) The root represents the empty prefix and, for every non-empty prefix P, the parent of the node representing P is the node representing $P[1..|P|-1]$ and the edge to the parent is labeled by the character $P[|P|]$; see Fig. 3 for an example. We develop a custom trie implementation which supports the following methods in constant time:

- **insert**(v, c): inserts a new leaf attached to a node v using an edge labeled with character c, and returns a handle to the created leaf; the node v cannot already have an outgoing edge labeled with c.
- **parent**(v): returns the handle to the parent of a node v (or $\perp$ if v is the root).
- **edge-label**(v): returns the label of the incoming edge of a node v (or $\perp$ if v is the root).
- **precedes**(u, v): decides whether the string represented by a node u is lexicographically smaller than the string represented by a node v.

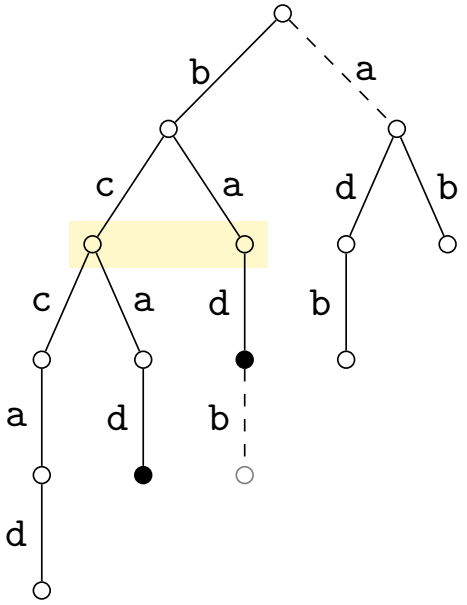


Fig. 3 A trie described in Sect. 3.2 speeding up comparisons. The trie is a snapshot of the example computation shown in Fig. 1, where we are at $i = 6$ and $\ell = 4$. There, we want to decide whether $D[5, 4] = \texttt{bcad}$ or $D[5, 3] \cdot T[6] = \texttt{badb}$ is lexicographically smaller. For that, we take the LCA v of the nodes (highlighted by black circles) representing $D[5, 4]$ and $D[5, 3]$; v is the child of the root with label b. Next, we compare the labels of v's two children leading to $D[5, 4]$ and $D[5, 3]$, respectively (these two children are marked in yellow). It suffices to compare the labels of these children to determine that $D[5, 3] \cdot T[6]$ is lexicographically smaller than $D[5, 4]$

Implementation of the trie For each node v, we explicitly store its parent **parent**(v), label **edge-label**(v), and depth **depth**(v). We do not keep pointers from v to its children, and thus each node occupies constant space. Moreover, we maintain the underlying (unlabeled) tree using the dynamic data structures of [2, 10], answering lowest common ancestor (LCA) and level ancestor queries (**level-anc**(u, d) returns the ancestor of a node u on depth d), respectively, in constant time. Both data structures support the insertion of *leaves* in constant time and, consequently, their space consumption is proportional to the tree size. In order to implement **precedes**(u, v), we first compute the lowest common ancestor w of u and v. For the special case that u is an ancestor of v, or vice versa, we return **false** if $v = w$ and **true** if $u = w \neq v$. Otherwise, we use level ancestor queries **level-anc**$(u,$ **depth**$(w) + 1)$ and **level-anc**$(v,$ **depth**$(w) + 1)$, to select the children u' and v' of w on the paths towards u and v, respectively. In that case, we return **true** if **edge-label**$(u') \prec$ **edge-label**(v') and **false** otherwise; see Fig. 3.

Application of the trie Instead of a flag, each cell of D now stores a handle to its respective trie node. The root node of the trie represents the empty string ε, so we associate $D[i, 0] = \varepsilon$ with the root node for all i.

To implement Line 5 of Alg 1, we first retrieve the handles to nodes u and v representing $D[i - 1, \ell - 1]$ and $D[i - 1, \ell]$, respectively. We proceed as follows for

deciding whether $D[i-1,\ell-1]\cdot T[i] \prec D[i-1,\ell]$: If $u = \textbf{parent}(v)$, we only have to compare $T[i]$ with **edge-label**(v), which is the last character of $D[i-1,\ell]$. Otherwise, we know that $D[i-1,\ell-1]$ is not a prefix of $D[i-1,\ell]$, and hence $D[i-1,\ell-1]\cdot T[i] \prec D[i-1,\ell]$ holds if and only if $D[i-1,\ell-1] \prec D[i-1,\ell]$, which we determine using **precedes**(u,v); see Fig. 3 for an example. If $D[i,\ell] = D[i-1,\ell]$, we store the handle to v at $D[i,\ell]$. Otherwise, we call **insert**$(u,T[i])$ and store the resulting handle at $D[i,\ell]$. This insertion is valid (meaning that u has no outgoing edge with label $T[i]$ yet) because all trie nodes at depth ℓ correspond to $D[j,\ell]$ for $j \in [\ell..i-1]$, and all these subsequences are at least as large as $D[i-1,\ell]$ in the lexicographic order.

As for Line 6, we retrieve the handle to node u representing $D[i-1,i-1]$, call **insert**$(u,T[i])$, and store the resulting handle at $D[i,i]$. This insertion is valid because the trie does not yet have any node at depth i.

Complexity Analysis The number of trie operations is $\mathcal{O}(n^2)$ (constantly many for each entry $D[i,\ell]$), and each of them is implemented in constant time. Hence, the overall time and space complexities become $\mathcal{O}(n^2)$.

Theorem 3 *We can compute the table $D[0..n,0..n]$ online in $\mathcal{O}(n^2)$ time using $\mathcal{O}(n^2)$ space.*

3.3 Most Competitive Subsequence

If we want to find only the lexicographically smallest subsequence of the whole string T for a fixed length ℓ, this problem is also called *Find the Most Competitive Subsequence*.[3] It admits a folklore linear-time solution that scans T from left to right and maintains, in a stack S, a subsequence of $T[1..i]$ of length between $\ell+i-n$ and ℓ chosen to minimize $\mathsf{S}\cdot\$$ in the lexicographic order, where $\$ \succ \max\Sigma$ is a sentinel character. Here, the lower bound $\ell+i-n$ guarantees that, when we are near the end of the text, we have enough characters to extend S to a length-ℓ subsequence of $T[1..n]$. Let top denote the top element of S. When processing text position i, we recursively pop top as long as (a) S is not empty, (b) $T[\mathsf{top}] \succ T[i]$, and (c) $|\mathsf{S}| \geq \ell+i-n$. Finally, we push $T[i]$ 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.

Observe that we can repeatedly use this solution to compute the lexicographically smallest subsequences of T of multiple lengths. The overall running time for all lengths $\ell \in [1..n]$ is $\mathcal{O}(n^2)$ and the algorithm uses $\mathcal{O}(n)$ working space, but it does not produce intermediate answers for the prefixes of T (as online algorithms do).

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 $T[3] \prec T[2]$, we do not discard $T[2] = \mathtt{b}$ stored on S since we need to produce a subsequence of length $\ell = 2$. A more elaborate execution on our running example is given in Fig. 4.

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

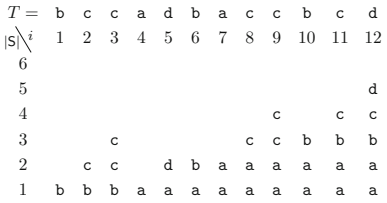


Fig. 4 Computing the most competitive subsequence of length $\ell = 5$ of the example string $T =$ `bccadbaccbcd`. The stack is shown vertically below of T for each step of the algorithm of Sect. 3.3. For $\ell = 6$, our stack would first differ at text position $i = 10$, where we would discard only the topmost `c` (instead of both of them). Then, the stack would store the subsequences `aacb` for $i = 10$, `aacbc` for $i = 11$, and `aacbcd` for $i = 12$

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 Sects. 3.1 and 3.2 can be straightforwardly translated. For that, our matrix D becomes a cube $D_3[0..L, 0..|X|, 0..|Y|]$, where $L := \mathrm{LCS}[|X|, |Y|]$ and $\mathrm{LCS}[x, y]$ denotes the length of a longest common subsequence of $X[1..x]$ and $Y[1..y]$. The entries $D_3[\ell, x, y]$ are well-defined for $\ell \leq \mathrm{LCS}[x, y]$ and computed by taking the lexicographically smallest string among at most three candidates for $\ell, x, y \geq 1$:

$$D_3[\ell, x, y] = \min \begin{cases} D_3[\ell - 1, x - 1, y - 1] \cdot X[x] & \text{if } X[x] = Y[y], \\ D_3[\ell, x - 1, y] & \text{if } \ell \leq \mathrm{LCS}[x - 1, y], \\ D_3[\ell, x, y - 1] & \text{if } \ell \leq \mathrm{LCS}[x, y - 1]. \end{cases}$$

Moreover, $D_3[0, x, y] = \varepsilon$ for all $x \in [0..|X|]$ and $y \in [0..|Y|]$, which gives us an induction basis similar to the one used in the proof of Lemma 1, so 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 handle to a node in the trie data structure proposed in Sect. 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). By monotonicity of D_3 (that is, due to $D_3[\ell, x, y] = \min_{x' \in [0..x], y' \in [0..y]: \mathrm{LCS}[x', y'] \geq \ell} D_3[\ell, x', y']$) and since L is already a common subsequence of $X[1..\mathsf{pos}_X(L)]$ and $Y[1..\mathsf{pos}_Y(L)]$, we must have $x = \mathsf{pos}_X(L)$ and $y = \mathsf{pos}_Y(L)$. Moreover, the monotonicity of D_3 further implies that all other entries $D_3[\ell, x', y'] = L$ satisfy $D_3[\ell, x' - 1, y'] = L$ (if $x' > x$) or $D_3[\ell, x', y' - 1] = L$ (if $y' > y$), so we copy the handle to the trie node representing L instead of calling **insert** when filling out $D_3[\ell, x', y'] = L$.

Theorem 4 *Given two strings X, Y of length at most 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.*

Fig. 5 Longest Lyndon subsequences of selected prefixes of a text T. The i-th row of bars below T depicts the selection of characters forming a Lyndon subsequence. In particular, the i-th row corresponds to the longest Lyndon 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) also corresponds to a longest Lyndon subsequence of $T[1..10]$ and $T[1..11]$ (when extended with $T[11]$). Extending the second Lyndon subsequence (blue) with $T[12]$ also gives a Lyndon subsequence, but 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 $T[1..i+1]$ (Color figure online)

4 Computing the Longest Lyndon Subsequence

In the following, we want to compute the longest Lyndon subsequence of T. See Fig. 5 for examples of longest Lyndon subsequences. As a starter, let us consider the following special case.

Theorem 5 *Given a string of length n, in which each character only appears once, we can compute its longest Lyndon subsequence in (a) $\mathcal{O}(n^2)$ time using $\mathcal{O}(1)$ space, or (b) $\mathcal{O}(n\sqrt{\lg n})$ time using $\mathcal{O}(n)$ space.*

Proof For each text position $i \in [1..n]$, we consider all characters in $T[i..n]$ that are at least as large as $T[i]$. These characters form the longest Lyndon subsequence starting at $T[i]$. Our answer is the longest among these n candidates. We can compute the length of each candidate in $\mathcal{O}(n)$ time, and thus obtain our first solution. For the second solution, we use the offline orthogonal range counting procedure of Chan and Pătraşcu [7, Corollary 2.3]. Specifically, we apply it for points $(j, T[j])$ for $j \in [1..n]$ and rectangles $[i+1..n] \times [T[i]+1..\sigma]$ for $i \in [1..n]$. This call takes $\mathcal{O}(n\sqrt{\lg n})$ time and outputs the number of input points located in each rectangle, which is $|\{j \in [i+1..n] : T[j] > T[i]\}|$ for the i-th rectangle. □

For the general case, compared to the dynamic programming approach for the lexicographically smallest subsequences introduced above, 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 of T, where a word is pre-Lyndon if it is a prefix of a Lyndon word. The trie τ is a subgraph of the trie storing all subsequences of T, sharing the same root. This subgraph is connected since, by definition, if S is a pre-Lyndon word, then all prefixes of S are also pre-Lyndon (if S is a prefix of a Lyndon word V, then all prefixes of S are also prefixes of V). We say that the string label of a node v is the string read from the edges on the path from the root to v. For every node v of T, we store $\mathsf{pos}_T(V)$, where V is the string label of v. Observe that, unless v is the root, the label of the incoming edge, which is the last character of V, equals $T[\mathsf{pos}_T(V)]$.

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 $(i_1, \ldots, i_\ell)$ associated with the path from the root to the node v we currently visit, i.e., if V is the string label of v, then $i_j = \mathsf{pos}_T(V[1..j])$ and thus $V[j] = T[i_j]$. At each node v, we first check whether V is a Lyndon word (if so, it is considered as an answer). Then, we recursively traverse the subtree of v. For this, we need to iterate, in the lexicographic order, over all characters c such that Vc is a pre-Lyndon word. For each such character, we determine $\mathsf{pos}_T(Vc)$, which is the smallest text position $i_{\ell+1} > i_\ell$ with $T[i_{\ell+1}] = c$. If there is such position $i_{\ell+1}$, we push it onto S, recurse, and then pop $i_{\ell+1}$. We apply the following facts to check whether a given subsequence is a Lyndon or a pre-Lyndon word.

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 [13, Prop. 1.1]. 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 $i \in [1..n - p]$. We use the following facts:

(Fact 1) The shortest period of a Lyndon word S is the length $|S|$.
(Fact 2) (Fact The prefix $S[1..p]$ of a pre-Lyndon word S with shortest period p is a Lyndon word. In particular, a pre-Lyndon word S is a Lyndon word if and only if its shortest period is $|S|$.
(Fact 3) Consider a pre-Lyndon word S with shortest period p and a character $c \in \Sigma$. Then:

- If $c \succ S[|S| - p + 1]$, then Sc is a Lyndon word.
- If $c = S[|S| - p + 1]$ and S is not the largest character of Σ,[4] then Sc is a pre-Lyndon word with shortest period p.
- Otherwise, Sc is not a pre-Lyndon word.

Proof

Fact 1. If S has a period smaller than $|S|$, then S is bordered.
Fact 2. If $S[1..p]$ was not Lyndon, then there would be 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. Follows from Fact 2 and [13, Lemma 1.6].

□

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

[4] There is no Lyndon word longer than 1 that starts with the largest character of Σ.

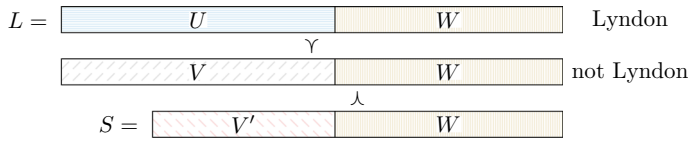


Fig. 6 Sketch of the second case in proof of Lemma 7, where the suffix S is assumed to be longer than W

whether $T[i_1] \cdots T[i_j] \cdot c$ is a pre-Lyndon word and, if so, retrieve its shortest period, both in constant time for any $c \in \Sigma$.

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 any of its children. However, for each text position i and each character $c \in \Sigma$, we can store the leftmost occurrence $i' \geq i$ of the smallest character $c' \succeq c$ that occurs in $T[i..n]$. As a result, we can traverse the trie in constant time per node during our preorder traversal.

This already gives an algorithm that computes the longest Lyndon subsequence with $\mathcal{O}(n\sigma)$ space and time linear in the number of nodes in τ. However, since the number of nodes can be exponential in the text length, we develop ways to omit nodes that do not lead to the solution. Our aim is to find a rule to prune trie nodes that surely do not contribute to the longest Lyndon subsequence of T. For that, we use the following notion of irrelevance:

Definition 6 Consider a pre-Lyndon subsequence U of T. We say that U is irrelevant if T has a Lyndon subsequence V of length $|V| = |U|$ such that $V \prec U$ and $\mathsf{pos}_T(V) \leq \mathsf{pos}_T(U)$. Otherwise, U is relevant.

Lemma 7 *If L is the lexicographically smallest length-ℓ Lyndon subsequence of T (for some $\ell \in [1..n]$), then all prefixes of L are relevant.*

Proof For a proof by contradiction, suppose that $L = UW$ for an irrelevant prefix U. Consider an integer i such that U is a subsequence of $T[1..i]$ and W is a subsequence of $T[i+1..n]$. By definition of irrelevance, $T[1..i]$ contains a Lyndon subsequence $V \prec U$ of length $|V| = |U|$. These conditions imply that VW is a subsequence of T that satisfies $VW \prec UW$. By definition of $L = UW$, this means that VW is not a Lyndon word, i.e., it contains a proper suffix $S \prec VW$. We consider two cases:

- If S is a suffix of W, then S is also a suffix of the Lyndon word UW, and hence $S \succ UW \succ VW$, a contradiction.
- Otherwise ($|S| > |W|$, see Fig. 6 for a visualization), S is of the form $V'W$ for a proper suffix V' of V. Since V is a Lyndon word, we have $V' \succ V$. Moreover, V is not a prefix of V', so this implies $S = V'W \succeq V' \succ VW$, a contradiction.

□

Due to Lemma 7, we do not omit the solution if we skip the subtrees rooted at irrelevant nodes, i.e., nodes whose string labels are irrelevant. Algorithmically, we exploit this observation as follows: We maintain an array $\mathsf{L}[1..n]$, where $\mathsf{L}[\ell]$ is the

smallest position $\mathsf{pos}_T(V)$ among the length-ℓ Lyndon subsequences V explored so far. We initialize all entries of L with ∞. Now, whenever we visit a node u whose string label is a length-ℓ pre-Lyndon subsequence U, then U is irrelevant if and only if $\mathsf{L}[\ell] \leq \mathsf{pos}_T(U)$: indeed, since we traverse the trie in the lexicographic order, the condition $\mathsf{L}[\ell] \leq \mathsf{pos}_T(U)$ is equivalent to the existence of a Lyndon subsequence $V \prec U$ of length ℓ with $\mathsf{pos}_T(V) \leq \mathsf{pos}_T(U)$.

Time Complexity Next, we analyze the complexity of this algorithm. For that, we say that a string is immature if it is pre-Lyndon but not Lyndon. Let us first bound the number of relevant Lyndon nodes visited. Whenever the algorithm processes a relevant Lyndon subsequence U of length ℓ, it decreases $\mathsf{L}[\ell]$ from a value strictly larger than $\mathsf{pos}_T(U)$ (if $\mathsf{L}[\ell] \leq \mathsf{pos}_T(U)$, then U would be irrelevant) to $\mathsf{pos}_T(U)$. We can decrease an individual entry of L at most n times, so there are at most n^2 relevant Lyndon subsequences in total. While each node can have at most σ children, due to Fact 3, at most one child can be immature. Since the depth of the trie is at most n, we therefore visit at most n^3 immature nodes, and at most $\mathcal{O}(n^3)$ relevant nodes in total. All irrelevant nodes are leaves in the pruned tree, so the overall number of visited nodes is $\mathcal{O}(n^3\sigma)$. As noted above, our trie navigation infrastructure allows for traversing the pruned trie in constant time per node.

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

4.2 Improving Time Bounds

We further improve the time bounds by avoiding visiting irrelevant nodes. For that, we make use of the following queries:

- Range maximum query: Given an interval $[i..j] \subseteq [1..n]$, retrieve the position of the largest character of the substring $T[i..j]$, i.e., return $\arg\max_{k\in[i..j]} T[k]$;
- Range successor query: Given an interval $[i..j] \subseteq [1..n]$ and a character c, retrieve the position $k \in [i..j]$ of the lexicographically smallest character $T[k]$ in $T[i..j]$ with $T[k] \succeq c$, i.e., return $\arg\min_{k\in[i..j]:\, T[k]\succeq c} T[k]$.

Each query returns a text position. In case of ties, they return the *leftmost* among candidate positions.

Now, suppose we are at a relevant node u with string label U of length ℓ and period p. Then, we want to consider all characters c such that Uc is a relevant pre-Lyndon subsequence of T. By Fact 3, all these characters satisfy $c \succeq U[\ell-p+1]$ (so that Uc is pre-Lyndon) and occur in $T[\mathsf{pos}_T(U)+1..\mathsf{L}[\ell+1]-1]$ (so that Uc is relevant). In the context of our preorder traversal, each such child can be found iteratively using range successor queries: starting from $b = U[\ell-p+1]$, we want to find the *lexicographically smallest* character $c \succeq b$ such that c occurs in $T[\mathsf{pos}_T(U)+1..\mathsf{L}[\ell+1]-1]$ and locate the leftmost such occurrence. This task can be accomplished using the wavelet tree [20] of T, which can be constructed in $\mathcal{O}(n \log \sigma)$ time and answers range successor queries in $\mathcal{O}(\lg \sigma)$ time [17, Theorem 7]. In particular, we can use the wavelet tree

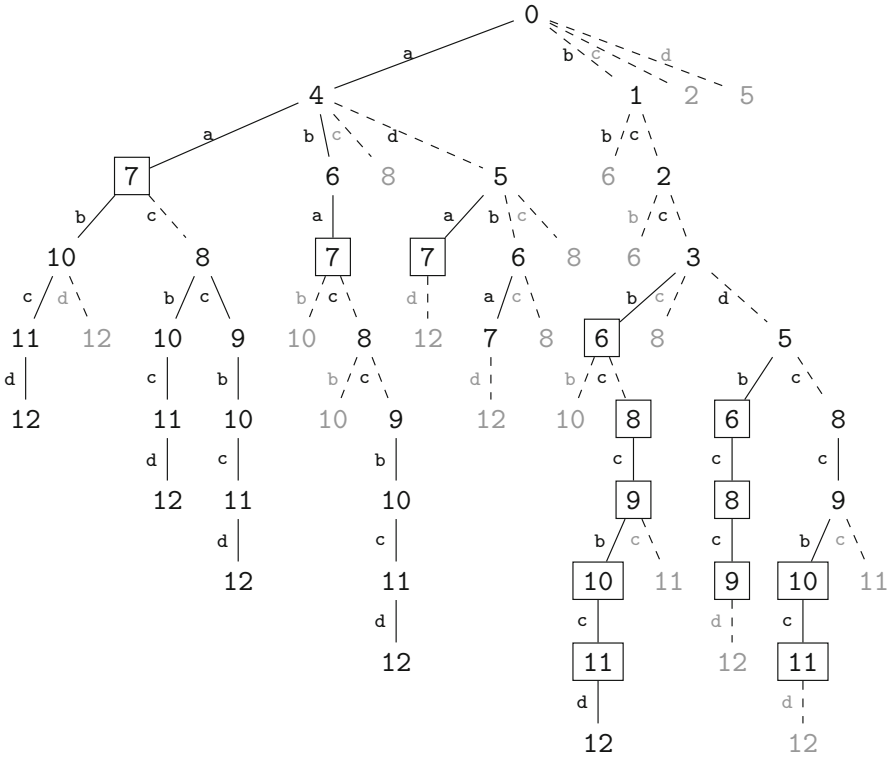


Fig. 7 The trie τ traversed by the algorithm of Theorem 8, with each node labeled by the value $\mathsf{pos}_T(V)$ computed for its string label V. Irrelevant nodes (whose subtrees are pruned) are drawn in gray and have a dashed incoming edge. For simplification, we sometimes omit irrelevant nodes representing subsequences ending with the last character of the text (every node which does not yet have an outgoing edge with label d should have such an irrelevant child). Each immature node is surrounded by a rectangle box. Remembering that immature nodes do not contribute to our pruning technique, we cannot prune `bccdc` with its leftmost occurrence ending at text position 8, since all formerly found subsequences with the same length ending at or before 8 are immature. The relevant Lyndon nodes have the property that, when fixing a depth, reading the Lyndon nodes from left to right gives a decreasing sequence of text positions. When pruning the node with string label `abab` and label 10, we have $\mathsf{L} = [4, 6, 8, 9, 10, 11, 12, \infty, \ldots]$ and can prune this node because $\mathsf{L}[|\texttt{abab}|] = 9$ does not exceed the label 10

instead of the $\mathcal{O}(n\sigma)$ pointers to the subsequent characters to arrive at $\mathcal{O}(n)$ space. The time complexity reduces to $\mathcal{O}(n^3 \log \sigma)$.

In order to bring the time down to $\mathcal{O}(n^3)$, we do not want to query the wavelet tree each time, but only whenever we are sure that u has at least one relevant Lyndon child. For that, we build a data structure of [5], which can be constructed in $\mathcal{O}(n)$ time and answers range maximum queries (RMQ) on T in $\mathcal{O}(1)$ time.[5] When we are at the relevant node u, we issue an RMQ to locate the leftmost occurrence of the largest character c in $T[\mathsf{pos}_T(U)+1..\mathsf{L}[\ell+1]-1]$. Then, we analyze the sequence Uc using Fact 3:

[5] Recall that if a character has multiple occurrences in T, then we rank these occurrences by their positions so that an RMQ on interval $[i..j]$ retrieves the *leftmost* position of the largest character in $T[i..j]$.

- If Uc is not pre-Lyndon, then u has no relevant children.
- If Uc is immature, then u has no relevant Lyndon children. Moreover, $\mathsf{pos}_T(Uc)$ is the position reported by the range maximum query. Hence, we do not need to use the wavelet tree.
- Finally, if Uc is Lyndon, we know that u has at least one relevant Lyndon child: while Uc might still be irrelevant if $\mathsf{L}[\ell+1]$ is decreased before we visit Uc, the only nodes that may decrease $\mathsf{L}[\ell+1]$ before we visit Uc are relevant Lyndon children of u.

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 relevant Lyndon children of u. Thus, the total number of wavelet tree queries asked is $\mathcal{O}(n^2)$ and the overall runtime is $\mathcal{O}(n^3 + n^2 \lg \sigma) = \mathcal{O}(n^3)$.

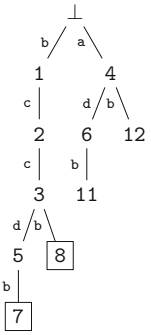
Theorem 9 *We can compute the longest Lyndon subsequence of a string of length n in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n)$ space.*

We remark that, by Lemma 7, our algorithm can be easily modified to compute, for each length ℓ, the lexicographically smallest length-ℓ Lyndon subsequence of T (if one exists). For this, it suffices to output, for each ℓ, the first visited Lyndon subsequence of length ℓ.

4.3 Online Computation

If we allow for more space to maintain the trie data structure introduced in Sect. 3.2, we can modify our $\mathcal{O}(n^3\sigma)$-time algorithm of Sect. 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 Sect. 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, so we prune irrelevant nodes obtaining a trie τ' of size $\mathcal{O}(n^3\sigma)$. The problem is that we can no longer perform the traversal in lexicographic order, so we 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, once the algorithm starts processing $T[i]$, the entry $\mathsf{L}[\ell]$ stores the lexicographically smallest length-ℓ Lyndon subsequence of $T[1..i-1]$ (represented by a pointer to the corresponding node of τ') or is empty if no such subsequence exists. Further, we maintain σ lists P_c ($c \in [1..\sigma]$) storing pointers to nodes of τ'. Once the algorithm starts processing $T[i]$, the list P_c contains pointers to all relevant nodes with string label U such that Uc is a pre-Lyndon word that is not a subsequence of $T[1..i-1]$ (i.e., $\mathsf{pos}_T(Uc) \geq i$). Initially, τ' consists only of the root node, and each list P_c stores only the root node. Whenever we read a new character $T[i]$ from the text stream, for each node v with string label V in $\mathsf{P}_{T[i]}$, we insert a leaf with string label $S := V \cdot T[i]$ (as a child of v). The characterization of $\mathsf{P}_{T[i]}$ guarantees that $\mathsf{pos}_T(S) = i$, so such a node does not exist yet. In order to keep the table $\mathsf{L}[1..n]$ up-to-date, we also check



$\mathsf{P}_\mathtt{a}$	4, 6, 11, 12
$\mathsf{P}_\mathtt{b}$	
$\mathsf{P}_\mathtt{c}$	7, 8
$\mathsf{P}_\mathtt{d}$	

ℓ	$\lvert\mathsf{L}[\ell]\rvert$	$\lvert\mathsf{W}[\ell,\mathtt{a}]\rvert$	$\lvert\mathsf{W}[\ell,\mathtt{b}]\rvert$	$\lvert\mathsf{W}[\ell,\mathtt{c}]\rvert$	$\lvert\mathsf{W}[\ell,\mathtt{d}]\rvert$
0		$\bot$	$\bot$	$\bot$	$\bot$
1	4	4	1		
2	12	12	2		
3	11	11	3		
4	5		5	8	
5				7	

Fig. 8 Online computation on the prefix bccadb of our running example. The trie on the left shows τ, where nodes are labeled by a rank reflecting the order at which a node has been created. This rank is used in the lists P and the table W as pointers to the trie nodes. Like before, nodes with rectangular boxes have immature string labels. On the right, we show the non-empty entries of L and W, where each row corresponds to one length ℓ. On reading the first d from the text, we do not create a node for bd since we already have bc, which also needs a character larger or equal to c to be extended to a Lyndon subsequence of length three

whether S is a Lyndon word satisfying $S \prec \mathsf{L}[|S|]$ (which can be tested using the data structure of Sect. 3.2) and, if so, we further set $\mathsf{L}[|S|] := S$. Next, we clear $\mathsf{P}_{T[i]}$ and iterate again over the newly created leaves. For each such leaf λ with label S, we check whether λ is relevant by performing a comparison $S \preceq \mathsf{L}[|S|]$. If λ is relevant, we put λ into P_c 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 Sect. 4.1 (although in a different order). In particular, there are $\mathcal{O}(n^3)$ relevant nodes, and we issue $\mathcal{O}(\sigma)$ events for each such node. The operations of Sect. 3.2 take constant time, so the total time and space complexity of the algorithm are $\mathcal{O}(n^3\sigma)$.

Theorem 10 *We can compute the longest Lyndon subsequence online in $\mathcal{O}(n^3\sigma)$ time using $\mathcal{O}(n^3\sigma)$ space.*

We can improve space and time bounds by treating immature subsequences and Lyndon subsequences separately. First, we only add a leaf λ with string label S into P_c if Sc is immature (i.e., we no longer store λ in P_c if Sc is Lyndon). Second, we treat Sc being Lyndon now differently with a table $\mathsf{W}[0..n, 1..\sigma]$ of size $\mathcal{O}(n\sigma)$. Throughout the execution of the algorithm, $\mathsf{W}[0, c]$ stores (a pointer to) the root node for each $c \in [1..\sigma]$. For each length $\ell \geq 1$ and character $c \in [1..\sigma]$, the entry $\mathsf{W}[\ell, c]$ stores a pointer to a relevant node with string label S of length ℓ such that Sc is immature. If there is no such node, the entry $\mathsf{W}[\ell, c]$ remains empty. If there are multiple candidates, we pick the one with the lexicographically smallest string label S. This choice is dictated by the following corollary:

Corollary 11 *(of Lemma 7) Consider two nodes u and v of τ' with string labels U and V, respectively, such that $|U| = |V|$ (u and v are on the same depth), $V \prec U$, and Uc and Vc are immature. Assume that we construct, later on, a child of u whose string label is Lyndon. Then this child is actually irrelevant.*

Proof Suppose that we read a character $T[i] \succ c$ such that we can create a child node u' of u with string label $U \cdot T[i]$. Then $V \cdot T[i]$ is also a Lyndon word, and we can apply Lemma 7. □

When reading character $T[i]$, for each length $\ell \in [1..i]$, we might create at most one relevant Lyndon node of length ℓ (the string label S of this node satisfies $\mathsf{pos}_T(S) = i$). By Lemma Corollary 11, the parent of this Lyndon word is among the nodes $\mathsf{W}[\ell-1, 1..T[i]-1]$. For each node v with string label V in $\mathsf{W}[\ell-1, 1..T[i]-1]$, the string $V \cdot T[i]$ is a Lyndon subsequence of $T[1..i]$. If there are multiple candidates, it suffices to consider one with the lexicographically smallest label V. If $V \cdot T[i] \prec L[\ell]$ holds for this label, then $V \cdot T[i]$ is relevant and satisfies $\mathsf{pos}_T(V \cdot T[i]) = i$. Hence, we create a new leaf linked with v using an edge with label $T[i]$. Moreover, we set $\mathsf{L}[\ell] := V \cdot T[i]$. This way, we add all the new relevant Lyndon nodes. As for the immature nodes, we scan $\mathsf{P}_{T[i]}$: given a node v in $\mathsf{P}_{T[i]}$ with string label V, if $V \cdot T[i] \prec L[\ell]$, we create a new leaf linked with v using an edge with label $T[i]$. Before we complete processing $T[i]$, we need to update the lists P and the table W. Thus, we clear $\mathsf{P}_{T[i]}$ and, for each newly created leaf λ with string label S, we use Fact 3 to compute the character c such that Sc is immature. We then append λ to P_c and, if $\mathsf{W}[|S|, c]$ is empty or Sc is lexicographically smaller than the string label of $\mathsf{W}[|S|, c]$, we also put λ at $\mathsf{W}[|S|, c]$.

Per read character $T[i]$, we scan W in $\mathcal{O}(n\sigma)$ time, which results in insertion of $\mathcal{O}(n)$ relevant Lyndon nodes. Moreover, we process $\mathsf{P}_{T[i]}$ (in time proportional to its length), which results in insertion of some relevant immature nodes. The total number of relevant nodes is $\mathcal{O}(n^3)$ and, for each such node, we issue one event into the lists P. Hence, the total running time is $\mathcal{O}(n^3)$.

Theorem 12 *We can compute the longest Lyndon subsequence online in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n^3)$ space.*

5 Longest Common Lyndon Subsequence

Given two strings X and Y, we want to compute the longest common subsequence (LCS) of X and Y that is Lyndon. In the special case that all characters in X and Y are unique, we can make use of an algorithm computing the LCS of two strings of length $\mathcal{O}(n)$ with unique letters in $\mathcal{O}(n \log\log n)$ time and $\mathcal{O}(n)$ space [24, Theorem 2(b)]: Similar to Theorem 5, we scan X from left to right, and compute, for each visited character $X[i]$, the LCS of $\pi_{X[i]}(X[i+1..])$ and $\pi_{X[i]}(Y[\mathsf{pos}_Y(X[i])+1..])$, where $\pi_{X[i]}$ discards all characters that are smaller than $X[i]$; if $X[i]$ does not occur in Y, we omit text position i and directly continue with $i+1$. If we take the maximum of all these at most n LCS lengths and increment this maximum by one for the matched character, we obtain the longest common Lyndon subsequence of X and Y in $\mathcal{O}(n^2 \log\log n)$ total time using $\mathcal{O}(n)$ space.

For the general case, we can extend our algorithm finding the longest Lyndon subsequence of a single string as follows. First, we explore, in lexicographic order, the trie of all *common* pre-Lyndon subsequences of X and Y. A node with string label L of length ℓ is represented by a stack $(x_1, y_1), \ldots, (x_\ell, y_\ell)$ with $x_j = \mathsf{pos}_X(L[1..j])$

and $y_j = \mathsf{pos}_Y(L[1..j])$. The depth-first search works like an exhaustive search in that it tries to extend L with subsequent characters $c \in \Sigma$ such that Lc is pre-Lyndon and c occurs in both $X[x_\ell + 1..]$ and $Y[y_\ell + 1..]$. For each such character c, the pair $(x_{\ell+1}, y_{\ell+1})$ consists of the positions of the leftmost occurrences of Lc in X and Y, respectively, which can be precomputed in $\mathcal{O}(n\sigma)$ time and space.

The algorithm uses again the array L to check, while processing a pre-Lyndon subsequence U, whether we have already found a Lyndon subsequence V of the same length satisfying $V \prec U$, $\mathsf{pos}_X(V) \leq \mathsf{pos}_X(U)$, and $\mathsf{pos}_Y(V) \leq \mathsf{pos}_Y(U)$. 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. We then 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 13 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 decrease all finite entries of R_ℓ to 2. 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 further 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 Sect. 3.1 computing the longest Lyndon sequence of a single string). Hence, the total time is $\mathcal{O}(n^4\sigma)$. The space complexity is dominated by the representation of the array L with the arrays R_ℓ. Since each R_ℓ uses $\mathcal{O}(n)$ space for $\ell \in [1..n]$, the total space is bounded by $\mathcal{O}(n^2)$.

Theorem 14 *We can compute the longest common Lyndon subsequence of a string of length n in $\mathcal{O}(n^4\sigma)$ time using $\mathcal{O}(n^2)$ space.*

6 Experiments

We implemented our algorithm computing the longest Lyndon subsequence of Theorem 8 and benchmarked this implementation on various texts. Our implementation,

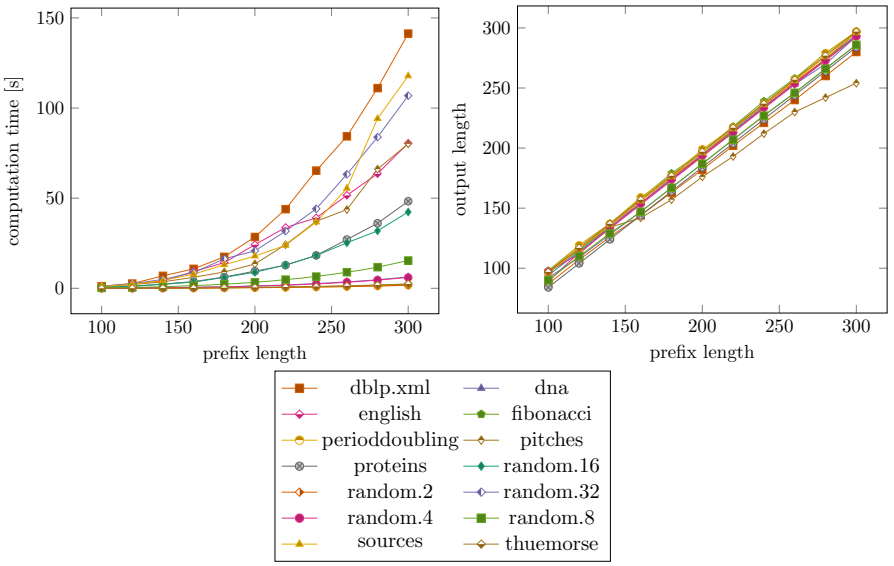


Fig. 9 Computing the longest Lyndon subsequence on prefixes of various texts. Left: Computation time. Right: Lengths of the longest Lyndon subsequence

written in Rust, is publicly available.[6] We evaluated our implementation on a server with an Intel i3-9100 CPU, running Debian 11. Since the time complexity of our algorithm is far from linear, we only benchmarked the computation on a few hundred numbers of characters. For such short strings, the task of the RMQ data structure and the wavelet tree can be performed by a linear scan on the text without degrading the performance too much (in fact, linear scan on such short strings is particularly fast due to data locality and its cache-friendly nature). We tested our algorithm on prefixes of the datasets of the Pizza&Chili corpus,[7] and on artificial random datasets random.x, where $x \in \{2, 4, 16, 32\}$ denotes the alphabet size. Additionally, we took the Thue–Morse, Fibonacci, and the period-doubling sequence. Figure 9 depicts the evaluation results. We observe that the running time is super-linear on all instances. The time also depends on the alphabet size since we need more time for random.x than for random.y with $x > y$. Another observation is that the lengths of the longest Lyndon subsequences we output grow linearly with the input size. Here, the dataset pitches has a slightly shorter output. Despite the fact that the length-300 prefixes of pitches and english have 126 and 44 distinct characters, respectively, the running time on both datasets for these two prefixes is roughly the same. We therefore conclude that the distribution of the characters has also an impact on the running time.

[6] https://github.com/koeppl/longestlyndonsubsequence.

[7] http://pizzachili.dcc.uchile.cl/.

Table 1 Algorithmic complexities for computing subsequences of various kinds studied in this article

Subsequence type	Time	Space	Reference	Speciality
Lexicographically smallest (all lengths)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Theorem 3	Online
Lexicographically smallest (fixed length)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Folklore	
Lexicographically smallest common (all lengths)	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	Theorem 4	
Longest Lyndon	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Theorem 5	All characters distinct
	$\mathcal{O}(n\sqrt{\lg n})$	$\mathcal{O}(n)$	Theorem 5	All characters distinct
	$\mathcal{O}(n^3)$	$\mathcal{O}(n)$	Theorem 9	
	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	Theorem 12	Online
Longest common Lyndon	$\mathcal{O}(n^2 \log\log n)$	$\mathcal{O}(n)$	Sect. 5	All characters distinct
	$\mathcal{O}(n^4\sigma)$	$\mathcal{O}(n^2)$	Theorem 14	

7 Conclusion

This article has shed light, for the very first time, on the computation of the longest Lyndon subsequence. We began by studying the lexicographically smallest subsequence and the most competitive subsequence. Both problems are related to Lyndon subsequences in that they are all based on the lexicographic order. In the main part of this article, we focused on the computation of the longest Lyndon subsequence, for which we proposed algorithms for the offline and the online case. Finally, we extended our offline algorithm to compute the longest common Lyndon subsequence of two strings. Different but much easier solutions can be obtained in the special case when all characters are unique. Table 1 summarizes the algorithmic complexities we obtained or observed during the analysis of our algorithms computing the subsequences we studied.

Open Problems

It is known that the longest common subsequence of two strings of length n cannot be computed in $\mathcal{O}(n^{2-\epsilon})$ time for any $\epsilon > 0$ unless the strong exponential time hypothesis (SETH) is false [1]. This conditional lower bound has been translated to other variations like finding the longest square subsequence [26, Section 4]. Unfortunately, we do not see whether we can find similar (conditional) lower bounds for the problems studied in this article. Lower bounds would either justify our time and space complexities, or give hope in finding better algorithms.

For the online computation studied in Sect. 4.3, the current bottleneck is the trie representation used, which represents $\mathcal{O}(n^3)$ nodes explicitly, and therefore needs $\mathcal{O}(n^3)$ time and space. We wonder whether we can find an implicit representation for the immature and irrelevant nodes that improves both complexities.

On the practical side, it is possible to enhance our implementation of Sect. 6 to cover also the algorithmic improvements described in Sect. 4.2. To be competitive with the current implementation, efficient implementations of range minimum queries and range successor queries need to be used. However, we are not aware of any optimized implementation of range successor queries.

Finally, we remark that we can extend our techniques for a special case of so-called *Galois words* [12, Section 6]. Galois words are defined in the setting of the alternating order $\prec_{alt}$, which is given by ranking odd positions with the classic lexicographic order, but even positions in the opposite order, when comparing two strings character by character. For instance, $\texttt{ab} \prec_{alt} \texttt{aa} \prec_{alt} \texttt{bb} \prec_{alt} \texttt{ba}$. A Galois word is then a word that is strictly smaller than all its cyclic rotations. A major difference to the lexicographic order is that a prefix of a string S is only smaller than S if its length is even, e.g., $\texttt{ab} \prec_{alt} \texttt{a}$. Now, if we stipulate that a prefix P of a string S always exhibits $P \prec_{alt} S$ (so we slightly modify the standard definition), then we can directly translate our techniques to compute the longest non-bordered Galois subsequence. This is because a non-bordered string is Galois if all its proper suffixes are $\prec_{alt}$-larger than itself. However, it is not clear to us how to find the longest bordered one, because our modified definition of $\prec_{alt}$ for the prefixes does not make sense when regarding bordered Galois subsequences. For instance, $\texttt{aba}$ is a bordered Galois word in the standard definition of the $\prec_{alt}$-order.

Acknowledgements This work was supported by JSPS KAKENHI Grants Number JP20H04141 (HB), JP19K20213 (TI), JP21K17701, JP22H03551, and JP23H04378 (DK). While preparing the preliminary version of this work, TK was at University of California, Berkeley, supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship. SJP was supported by the Academy of Finland via grant 339070. We are grateful to Gabriele Fici for suggesting the problem addressed in 5, and to Marinella Sciortino for introducing us to Galois words.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. In: Guruswami, V. (ed.) Proceedings of FOCS. pp. 59–78. IEEE Computer Society (2015). https://doi.org/10.1109/FOCS.2015.14
2. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Proceedings of ICALP. LNCS, vol. 1853, pp. 73–84 (2000). https://doi.org/10.1007/3-540-45022-x_8
3. Bannai, H., Tomohiro, I., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. SIAM J. Comput. **46**(5), 1501–1514 (2017). https://doi.org/10.1137/15m1011032

4. Bannai, H., Tomohiro, I., Kociumaka, T., Köppl, D., Puglisi, S.J.: Computing longest (common) Lyndon subsequences. In: Proceedings of IWOCA. LNCS, vol. 13270, pp. 128–142. Springer (2022). https://doi.org/10.1007/978-3-031-06678-8_10
5. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algor. **57**(2), 75–94 (2005). https://doi.org/10.1016/j.jalgor.2005.08.001
6. 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). https://doi.org/10.1137/18m1192226
7. Chan, T.M., Patrascu, M.: Counting inversions, offline orthogonal range counting, and related problems. In: SODA. pp. 161–173. SIAM (2010). https://doi.org/10.1137/1.9781611973075.15
8. 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). https://doi.org/10.1007/978-1-4612-2096-1_10
9. Chowdhury, S.R., Hasan, M.M., Iqbal, S., Rahman, M.S.: Computing a longest common palindromic subsequence. Fundam. Informaticae **129**(4), 329–340 (2014). https://doi.org/10.3233/fi-2014-974
10. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005). https://doi.org/10.1137/s0097539700370539
11. de Beauregard Robinson, G.: On the representations of the symmetric group. Am. J. Math. **60**(3), 745–760 (1938). https://doi.org/10.2307/2371609
12. Dolce, F., Restivo, A., Reutenauer, C.: On generalized Lyndon words. Theor. Comput. Sci. **777**, 232–242 (2019). https://doi.org/10.1016/j.tcs.2018.12.015
13. Duval, J.: Factorizing words over an ordered alphabet. J. Algor. **4**(4), 363–381 (1983). https://doi.org/10.1016/0196-6774(83)90017-2
14. Elmasry, A.: The longest almost-increasing subsequence. Inf. Process. Lett. **110**(16), 655–658 (2010). https://doi.org/10.1016/j.ipl.2010.05.022
15. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960). https://doi.org/10.1145/367390.367400
16. Fujita, K., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Longest common rollercoasters. In: Proceeding of SPIRE. LNCS, vol. 12944, pp. 21–32 (2021). https://doi.org/10.1007/978-3-030-86692-1_3
17. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. **426**, 25–41 (2012). https://doi.org/10.1016/j.tcs.2011.12.002
18. Gawrychowski, P., Manea, F., Serafin, R.: Fast and longest rollercoasters. In: Proceeding of STACS. LIPIcs, vol. 126, pp. 30:1–30:17 (2019). https://doi.org/10.1007/s00453-021-00908-6
19. Glen, A., Simpson, J., Smyth, W.F.: Counting Lyndon factors. Electron. J. Comb. **24**(3), P3.28 (2017). https://doi.org/10.37236/6915
20. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceeding of SODA. pp. 841–850 (2003), http://dl.acm.org/citation.cfm?id=644108.644250
21. He, X., Xu, Y.: The longest commonly positioned increasing subsequences problem. J. Comb. Optim. **35**(2), 331–340 (2018). https://doi.org/10.1007/s10878-017-0170-9
22. Hirakawa, R., Nakashima, Y., Inenaga, S., Takeda, M.: Counting Lyndon subsequences. In: Proceeding of PSC, pp. 53–60 (2021), http://www.stringology.org/event/2021/p05.html
23. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM **24**(4), 664–675 (1977). https://doi.org/10.1145/322033.322044
24. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest subsequences. Commun. ACM **20**(5), 350–353 (1977). https://doi.org/10.1145/359581.359603
25. Inenaga, S., Hyyrö, H.: A hardness result and new algorithm for the longest common palindromic subsequence problem. Inf. Process. Lett. **129**, 11–15 (2018). https://doi.org/10.1016/j.ipl.2017.08.006
26. Inoue, T., Inenaga, S., Hyyrö, H., Bannai, H., Takeda, M.: Computing longest common square subsequences. In: Proceeding of CPM. LIPIcs, vol. 105, pp. 15:1–15:13 (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.15
27. Kiyomi, M., Horiyama, T., Otachi, Y.: Longest common subsequence in sublinear space. Inf. Process. Lett. **168**, 106084 (2021). https://doi.org/10.1016/j.ipl.2020.106084
28. Knuth, D.: Permutations, matrices, and generalized Young tableaux. Pac. J. Math. **34**, 709–727 (1970). https://doi.org/10.2140/pjm.1970.34.709

29. Kosche, M., Koß, T., Manea, F., Siemer, S.: Absent subsequences in words. In: Proceeding of RP. LNCS, vol. 13035, pp. 115–131 (2021). https://doi.org/10.1007/978-3-030-89716-1_8
30. Kosowski, A.: An efficient algorithm for the longest tandem scattered subsequence problem. In: Proceeding of SPIRE. LNCS, vol. 3246, pp. 93–100 (2004). https://doi.org/10.1007/978-3-540-30213-1_13
31. Kutz, M., Brodal, G.S., Kaligosi, K., Katriel, I.: Faster algorithms for computing longest common increasing subsequences. J. Discrete Algor. **9**(4), 314–325 (2011). https://doi.org/10.1016/j.jda.2011.03.013
32. Lyndon, R.C.: On Burnside's problem. Trans. Am. Math. Soc. **77**(2), 202–215 (1954). https://doi.org/10.2307/1990868
33. Schensted, C.: Longest increasing and decreasing subsequences. Can. J. Math. **13**, 179–191 (1961). https://doi.org/10.1007/978-0-8176-4842-8_21
34. 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). https://doi.org/10.1016/j.tcs.2020.11.035
35. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974). https://doi.org/10.1145/321796.321811