Lyndon Arrays Simplified
Jonas Ellert
Department of Computer Science, Technische Universität Dortmund, Germany

Abstract

A Lyndon word is a string that is lexicographically smaller than all of its proper suffixes (e.g., *airbus* is a Lyndon word; *amtrak* is not a Lyndon word because its suffix *ak* is lexicographically smaller than *amtrak*). The Lyndon array (sometimes called Lyndon table) identifies the longest Lyndon prefix of each suffix of a string. It is well known that the Lyndon array of a length-$n$ string can be computed in $O(n)$ time. However, most of the existing algorithms require the suffix array, which has theoretical and practical disadvantages. The only known algorithms that compute the Lyndon array in $O(n)$ time without the suffix array (or similar data structures) do so in a particularly space efficient way (Bille et al., ICALP 2020), or in an online manner (Badkobeh et al., CPM 2022). Due to the additional goals of space efficiency and online computation, these algorithms are complicated in technical detail. Using the main ideas of the aforementioned algorithms, we provide a simpler and easier to understand algorithm that computes the Lyndon array in $O(n)$ time.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Lyndon table, Lyndon array, Lyndon word, nearest smaller suffixes, lexicographical ordering, general ordered alphabets, combinatorial algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.48

Supplementary Material Software (Source Code): https://github.com/jonas-ellert/simple-lyndon; archived at swh:1:dir:dbbf2b4ac2fa652eb0865cdc6719924ce8a81952

1 Related Work

A Lyndon word is a string that is lexicographically smaller than all of its proper suffixes (e.g., *airbus* is a Lyndon word; *amtrak* is not a Lyndon word because its suffix *ak* is lexicographically smaller than *amtrak*). The Lyndon array (sometimes called Lyndon table) identifies the longest Lyndon prefix of each suffix of a string (a precise definition follows later). It has both theoretical and practical applications related to repetitiveness in strings. Most notably, it is a crucial component for showing that a length-$n$ string contains less than $n$ maximal repetitions (the “Runs” theorem by Bannai et al. [3]), and it is useful for computing all of these maximal repetitions in optimal time [8]. Other applications of the Lyndon array include text compression and indexing [16].

There is a close relation [12] between the Lyndon array and the suffix array (one of the most fundamental data structures in string algorithmics [15]). This inspired research focused on computing the Lyndon array from the suffix array [9], computing both arrays simultaneously [2, 13], and also using properties of the Lyndon array to compute the suffix array [2, 4]. One of the conceptually simplest methods for computing the Lyndon array combines the (inverse) suffix array with a folklore algorithm for the computation of nearest smaller values [9, Algorithm ISA-NSV] (we will discuss this algorithm in Section 4).

However, using the suffix array is both a theoretical and practical drawback. From a theoretical point of view, computing the suffix array of a length-$n$ string over a general ordered alphabet takes $O(n \log n)$ time. This is due to the well-known information-theoretic lower bound on the number of comparisons for sorting. We can only compute the suffix array in optimal $O(n)$ time, if the alphabet can be sorted in $O(n)$ time (e.g., a polynomial integer alphabet $\Sigma = \{1, \ldots, n^{O(1)}\}$ on a word RAM of width $w \geq \log_2 n$). This is not a concern...
Lyndon Arrays Simplified

in practice, where this requirement is virtually always met. Nevertheless, even the fastest algorithms for the suffix array are relatively slow in practice, and the computation appears to be somewhat excessive for the seemingly simpler task of computing the Lyndon array.

There are two known algorithms that compute the Lyndon array by mere symbol comparisons, without using the suffix array, and in optimal $O(n)$ time. The first one [5] computes the Lyndon array using only $O(1)$ words of additional working space, or the succinct $2n$-bit version of the Lyndon array using only $o(n)$ bits of additional working space. The second one [1] is an online algorithm that computes the Lyndon array from right to left. The technical details of these algorithms are rather intricate, and many of the used techniques are only needed due to the goals of space efficiency and online computation.

**Contributions.** We present a simple $O(n)$ time and space algorithm that computes the Lyndon array of a length-$n$ string in the comparison model (i.e., the only elementary operations allowed on symbols of the string are comparisons of the form less-greater-equal). It requires no precomputed data structures (like the suffix array), and works by exploiting the powerful combinatorial properties of Lyndon words. Many of the used techniques are simpler versions of what was done in [5, 1]. The final trick used to achieve linear time is similar to Manacher’s classic algorithm for longest palindromic substrings [14].

The proof of correctness is still rather technical, but the resulting algorithm is incredibly easy to implement. The provided C++ implementation consists of not even 50 lines of code, and uses neither external nor standard libraries.

## 2 Preliminaries

We use the interval notations $[i, j] = [i, j + 1) = (i - 1, j] = (i - 1, j + 1]$ to denote the integer set \{i, i + 1, \ldots , j\} (or $\emptyset$ if $i > j$). A string $x = x[1..n]$ over an ordered alphabet $\Sigma$ is a sequence $x = x[1]x[2] \cdots x[n]$ of symbols drawn from some totally ordered set $\Sigma$. We write $|x| = n$ to denote the length of the string. The set $\Sigma^*$ contains all strings over $\Sigma$, including the empty string $\epsilon$ of length 0. The concatenation of two strings $x[1..n]$ and $y[1..m]$ is the sequence $x[1] \cdots x[n]y[1] \cdots y[m]$, and simply written as $xy$ (or $x \cdot y$). The total order of symbols from $\Sigma$ induces the lexicographical order of strings from $\Sigma^*$. We say that $x$ is lexicographically smaller than $y$ and write $x < y$, if and only if either $y = xw$ for some non-empty string $w$, or $x = uv$ and $y = usw$ for (possibly empty) strings $u, v, w$ and symbols $r < s$. We write $x \leq y$ to denote $x = y$ or $x < y$.

For $i, j \in [1, n]$ with $i \leq j$, the substring $x[i..j] = x[i..j + 1) = x(i-1..j] = x(i-1..j + 1]$ of $x[1..n]$ is the sequence $x[i]x[i+1] \cdots x[j]$. If $i > j$, then $x[i..j]$ equals the empty string denoted by $\epsilon$. If $x[i..j] \neq x$ then $x[i..j]$ is a proper substring. A substring of the form $x[i..j]$ is called prefix of $x$, while $x[i..n]$ is called suffix of $x$. We use the simplified notation $x_i = x[i..n]$ for the suffix starting at position $i$. The longest common extension (LCE) of two suffixes $x_i$ and $x_j$ is defined as the length of the longest common prefix of the suffixes, formally $\text{LCE}(i, j) = \max\{|u| \mid u, v, w \in \Sigma^* \wedge x_i = uv \land x_j = uw\}$.

**Sentinel Symbols.** Throughout this work, we often compare two suffixes $x_i, x_j$ of the same string $x[1..n]$. The special case where one suffix is a prefix of another, e.g., $i < j$ and $x_i = x_jx_{i+1} \cdots x_{j}$, often complicates the notation of definitions and algorithms. This can be avoided by assuming that the text starts and ends with special sentinel symbols $x[1] = \#$ and $x[n] = \$. The sentinels are smaller than all other symbols, i.e., $\forall k \in (1, n) : x[k] > \$ > \#$. In definitions and lemmas we emphasize the presence of sentinels by writing $x = x[1..n] = \#x(1..n)\$. (Note...
that $n$ is the length of the string including the sentinels.) The usage of sentinels is of purely cosmetic nature. Particularly, they do not affect the lexicographical order of suffixes, i.e., it holds $x[i..n]$ $\prec x[j..n]$ if and only if $x[i..n]$ $\prec x[j..n]$. In practice, we can add sentinels to any string by either physically prepending and appending them, or by using an appropriate wrapper function\(^\text{1}\) when accessing the string.

**Lyndon Words and Arrays**

There are multiple equivalent definitions of Lyndon words. We use Duval’s characterization based on the lexicographical order of suffixes:

> **Definition 1** ([7, Proposition 1.2]). A string $x[1..n]$ is a Lyndon word, if and only if it is lexicographically smaller than all of its proper non-empty suffixes, i.e., $\forall i \in [2, n]: x \prec x_i$.

The Lyndon array of a string and its close relatives, the nearest smaller suffix arrays, capture the combinatorial structure of Lyndon substrings. We denote the Lyndon array by $\lambda$, which was also done in [5, 9]. Other notations are $\text{Lyn}$ in [6, 1], $l$ in [3], and $\mathcal{L}$ in [10, 11].

> **Definition 2.** The Lyndon array $\lambda[1..n]$, the previous smaller suffix array $\text{prev}[1..n]$, and the next smaller suffix array $\text{next}[1..n]$ of a string $x = \#x(1..n)$ are defined as:

(i) $\forall i \in [1, n]: \lambda[i] = \max \{m \mid m \in [1, n-i+1] \land x[i..i+m] \text{ is a Lyndon word} \}$,

i.e., $x[i..i+\lambda[i]]$ is the longest Lyndon prefix of suffix $x_i$

(ii) $\forall i \in (1, n): \text{prev}[i] = \max \{j \mid j \in [1, i) \land x_j \prec x_i\}$, \text{prev}[1] = 0, \text{prev}[n] = 1,

i.e., $x_{\text{prev}[i]}$ is the nearest suffix starting left of $i$ that is lex. smaller than $x_i$

(iii) $\forall i \in (1, n): \text{next}[i] = \min \{j \mid j \in (i, n] \land x_j \prec x_i\}$, \text{next}[1] = n + 1, \text{next}[n] = n + 1,

i.e., $x_{\text{next}[i]}$ is the nearest suffix starting right of $i$ that is lex. smaller than $x_i$

Figure 1a shows an example of these arrays. In drawings, we use a directed edge from position $i$ to position $j$ of a string to indicate that either $\text{prev}[i] = j$ (whenever the edge is directed from right to left) or $\text{next}[i] = j$ (whenever the edge is directed from left to right). We refer to these edges as PSS and NSS edges. It is no coincidence that in the example it holds $\text{next}[i] = i + \lambda[i]$ for all $i$. In fact, this is a fundamental combinatorial property of the Lyndon array, which was first (indirectly in a different form) shown by Hohlweg and Reutenauer [12]. Subsequently, Franek et al. [9, Lemma 15] and Franek and Liut [11, Lemma 1]\(^\text{2}\) proved the property in the form stated below.

> **Lemma 3** ([12, 9, 11]). Let $x = \#x(1..n)$ be a string with Lyndon array $\lambda$ and next smaller suffix array $\text{next}$, then it holds $\forall i \in [1, n]: \text{next}[i] = i + \lambda[i]$.

Another property shown by Bille et al. [5] relates $\text{prev}$ and Lyndon words:

> **Lemma 4** ([5, Lemma 4]). Let $x = \#x(1..n)$ be a string with previous smaller suffix array $\text{prev}$. For every $i \in [2, n]$, the string $x[\text{prev}[i].i]$ is a Lyndon word.

\(^1\)see, e.g., lines 4–5 in file https://github.com/jonas-ellert/simple-lyndon/blob/main/lyndon.hpp

\(^2\)Lemma 1 (b) in [11] should state “$x[i..j]$ is proto-Lyndon” rather than “$x[i..n]$ is proto-Lyndon”
3 Key Properties of Nearest Smaller Suffixes

![Example of arrays λ, next and prev.](image)

![Drawing for Lemma 5.](image)

![Drawing for Lemma 6.](image)

![Drawing for Lemma 7(ii).](image)

![Drawing for Lemma 7(iii).](image)

**Figure 1** Examples and technical drawings for Sections 2 and 3. PSS edges are solid and red. NSS edges are dashed and blue. Dotted black edges are either NSS or PSS edges.

In this section, we show combinatorial properties of *prev* and *next* that are essential for all algorithms presented in this paper. Some of the properties have been shown (in a similar form) in [5, 1]. Proving Lemmas 5–7 is helpful for truly understanding the mechanisms at play. The reader is encouraged to do so on their own, with the help of the provided supplementary drawings. The full proofs are given in Section 7.

**Naming of variables.** Throughout the remainder of the paper, we use the variables *ℓ* and *r* to denote positions of the string. The intended meaning of these variables is *left* and *right*, i.e., whenever we use *ℓ* and *r* it holds *ℓ < r*.

**PSS and NSS edges are intersection-free.** The first property that we show is that, when drawn underneath the text as in the previous example, none of the PSS and NSS edges intersect. This is formally expressed by the following lemma (see also Figure 1b).

**Lemma 5.** Let \( x = #x(1..n) $ \) be a string with previous and next smaller suffix array *prev* and *next*. Let \( ℓ_1, ℓ_2, r_1, r_2 \in [1, n] \) be indices with either \( \text{next}[ℓ_1] = r_1 \) or \( \text{prev}[r_1] = ℓ_1 \), and also either \( \text{next}[ℓ_2] = r_2 \) or \( \text{prev}[r_2] = ℓ_2 \). Then it does not hold \( ℓ_1 < ℓ_2 < r_1 < r_2 \).
Chains of previous smaller suffixes. For arbitrary index \( r \in [1, n] \) and integer \( e \geq 0 \), we recursively define \( \text{prev}^e[r] = r \) and \( \text{prev}^{e+1}[r] = \text{prev}^{e}[\text{prev}[r]] \). We write \( \ell = \text{prev}^*[r] \) to denote that there is some integer \( e \geq 0 \) with \( \ell = \text{prev}^e[r] \). Note that generally \( 1 = \text{prev}^*[r] \) (due to the sentinel \( x[1] = \# \) ) and \( r = \text{prev}^*[r] \). In drawings, \( \ell = \text{prev}[r] \) means that there is a chain of PSS edges from \( r \) to \( \ell \). The following lemma states useful properties of PSS chains, which we will later use to compute the arrays \( \text{prev} \) and \( \text{next} \). The lemma is visualized in Figure 1c.

Lemma 6. Let \( x = #x[1..n]\# \) be a string with previous and next smaller suffix arrays \( \text{prev} \) and \( \text{next} \), and let \( \ell, r \in [1, n] \) be arbitrary indices.

(i) It holds \( \text{prev}[r] = \text{prev}^*[r-1] \).
(ii) It holds \( \text{next}[\ell] = r \) if and only if \( \ell = \text{prev}^*[r-1] \) and \( \ell > \text{prev}[r] \).

Finally, for some indices that are related via next and prev, we can deduce LCEs. The following lemma is visualized in Figures 1d and 1e.

Lemma 7. Let \( x = #x[1..n]\# \) be a string with previous and next smaller suffix arrays \( \text{prev} \) and \( \text{next} \). Let \( k \in (1, n) \) be an arbitrary index, and let \( \ell = \text{prev}[k] \) and \( r = \text{next}[k] \).

(i) If \( \text{LCE}(\ell,k) = \text{LCE}(k,r) \), then \( \text{LCE}(\ell,r) \geq \text{LCE}(k,r) \) and either \( \text{prev}[r] = \ell \) or \( \text{next}[\ell] = r \).
(ii) If \( \text{LCE}(\ell,k) < \text{LCE}(k,r) \), then \( \text{LCE}(\ell,r) = \text{LCE}(k,k) \) and \( \text{prev}[r] = \ell \).
(iii) If \( \text{LCE}(\ell,k) > \text{LCE}(k,r) \), then \( \text{LCE}(\ell,r) = \text{LCE}(k,k) \) and \( \text{next}[\ell] = r \).

4 Algoritms to Compute the Lyndon Array

Due to Lemma 3, instead of designing algorithms that compute the Lyndon array \( \lambda \), we can design algorithms that compute the next smaller suffix array next. This has been done, e.g., by Bille et al. [5] and Crochemore et al. [1], and is also the general approach used in this paper. All of the presented algorithms are based on a simple folklore algorithm for nearest smaller values, where instead of comparing values we lexicographically compare suffixes (Algorithm 1(a)). We obtain three different versions of this algorithm depending on how the lexicographical comparisons are implemented: Algorithm 1(b) uses the inverse suffix array and is only shown because it is a standard solution for computing the Lyndon array. Algorithm 1(c) implements lexicographical comparisons with naively computed LCEs. Algorithm 1(d) refines the LCE computation such that it is more time efficient. In the remainder of this section, we explain each version of the algorithm in detail.

Algorithms 1(c) and 1(d) require super-linear time, but they can be seen as incremental stepping stones towards the final solution. In Section 5, we modify Algorithm 1(d) such that it runs in \( O(n) \) time.

Algorithm 1: The general approach. Due to the sentinel, we can directly assign \( \text{prev}[1] \), \( \text{next}[1] \), and \( \text{next}[n] \) (lines 1–2). We compute the arrays next and prev in \( n-1 \) iterations of a simple for-loop (line 4). The goal of iteration \( r \) is to compute \( \text{prev}[r] \), while also identifying all indices \( \ell \) with \( \text{next}[\ell] = r \). A simple strategy for this is dictated by Lemma 6, which states that all the relevant indices lie on the chain of PSS edges that starts at position \( r-1 \). We thus inspect the positions \( \ell = \text{prev}^*[r-1] \) one at a time, starting with \( \ell = r-1 \) (line 5). As long as \( x_\ell > x_r \), we assign \( \text{next}[\ell] \leftarrow r \), and then continue with the next index \( \ell \leftarrow \text{prev}[\ell] \) on the chain of PSS edges (lines 6–8). As soon as \( x_\ell < x_r \), we break out of the inner loop and finish the current iteration of the outer loop by assigning \( \text{prev}[r] \leftarrow \ell \) (line 9). (The sentinel \( x[1] = \# \) ensures that \( 1 = \text{prev}^*[r] \) and \( x_1 < x_r \); thus we are guaranteed to reach some \( \ell \) with \( x_\ell < x_r \) eventually.) The correctness of the algorithm follows directly from Lemma 6. An example of an outer loop iteration is provided in Figure 2 (the arrays lexrank, plce, and nlce will be relevant later and can be ignored for now).
Algorithm 1 Various algorithms for computing nearest smaller suffixes.

Require: string \( x = x[1..n] = \#x(i..n) \)
Ensure: previous and next smaller suffix arrays prev and next

1: prev[1..n] ← new array with prev[1] = 0
2: next[1..n] ← new array with next[1] = next[n] = n + 1

(a) Folklore

3: —
4: for \( r = 2 \) to \( n \) do
5: \( \ell \leftarrow r - 1 \)
6: while \( x_{\ell} \succ x_r \) do
7: \( \text{next}[\ell] \leftarrow r \)
8: \( \ell \leftarrow \text{prev}[\ell] \)
9: prev[r] ← \( \ell \)

(b) ISA-NSV

3: lexrank ← inverse suffix array of \( x \)
4: for \( r = 2 \) to \( n \) do
5: \( \ell \leftarrow r - 1 \)
6: while lexrank[\( \ell \)] > lexrank[\( r \)] do
7: \( \text{next}[\ell] \leftarrow r \)
8: \( \ell \leftarrow \text{prev}[\ell] \)
9: prev[r] ← \( \ell \)

(c) Naive LCE-NSS

3: plce[1..n] ← array filled with 0
4: nlce[1..n] ← array filled with 0
5: for \( r = 2 \) to \( n \) do
6: \( \ell \leftarrow r - 1 \)
7: \( m \leftarrow \text{LCE-SCAN}(\ell, r) \)
8: while \( x[\ell + m] > x[r + m] \) do
9: \( \text{next}[\ell], \text{nlce}[\ell] \leftarrow r, m \)
10: \( m \leftarrow \text{LCE-SCAN}(\text{prev}[\ell], r) \)
11: —
12: —
13: —
14: \( \ell \leftarrow \text{prev}[\ell] \)
15: prev[r], plce[r] ← \( \ell, m \)

(d) Improved LCE-NSS

3: plce[1..n] ← array filled with 0
4: nlce[1..n] ← array filled with 0
5: for \( r = 2 \) to \( n \) do
6: \( \ell \leftarrow r - 1 \)
7: \( m \leftarrow \text{LCE-SCAN}(\ell, r) \)
8: while \( x[\ell + m] > x[r + m] \) do
9: \( \text{next}[\ell], \text{nlce}[\ell] \leftarrow r, m \)
10: if \( m = \text{plce}[\ell] \) then
11: \( m \leftarrow \text{LCE-EXTEND}(\text{prev}[\ell], r, m) \)
12: else if \( m > \text{plce}[\ell] \) then
13: \( m \leftarrow \text{plce}[\ell] \)
14: \( \ell \leftarrow \text{prev}[\ell] \)
15: prev[r], plce[r] ← \( \ell, m \)

e) LCE Functions for (c) and (d)

\[
\text{function LCE-EXTEND}(\ell, r, m) \\
\quad \text{while } x[\ell + m] = x[r + m] \text{ do} \\
\quad \quad m \leftarrow m + 1 \\
\quad \text{return } m \\
\]

\[
\text{function LCE-SCAN}(\ell, r) \\
\quad \text{return } \text{LCE-EXTEND}(\ell, r, 0) \\
\]
to achieve \( O(n) \) time, substitute:
line 7: \( m \leftarrow \text{SMART-LCE}(\ell, r, 0) \)
line 11: \( m \leftarrow \text{SMART-LCE}(\text{prev}[\ell], r, m) \)
(see Algorithm 2)
Figure 2 For any of the Algorithms 1(a)–(d), we perform six suffix comparisons in outer loop iteration \( r = 22 \). We evaluate \( x_\ell \succ x_r \) for each of the values \( \ell = 21, 20, 17, 12, 5, 2 \) (precisely in this order). For the first five values \( \ell = 21, 20, 17, 12, 5 \), we enter the body of the inner loop and thus assign \( \text{next}[\ell] \to 22 \). We break out of the inner loop by discovering that \( x_2 \prec x_{22} \), after which we assign \( \text{prev}[22] \to 2 \). By design of the algorithm, and following Lemma 6, the values assumed by \( \ell \) form a consecutive chain of PSS edges starting at position \( r = 1 = 21 \) (dashed edges).

Whenever we perform the lexicographical comparison of suffixes in line 6, we correctly assign either \( \text{next}[\ell] \to r \) or \( \text{prev}[r] \to \ell \) immediately afterwards. Since each entry of \( \text{prev} \) and next gets assigned exactly once, we perform exactly \( 2n - 3 \) suffix comparisons; we enter the body of the inner loop exactly \( n - 2 \) times (once per entry of next, but not for \( \text{next}[1] = \text{next}[n] = n + 1 \)), and break out of the inner loop exactly \( n - 1 \) times (once per outer loop iteration, or equivalently once per entry of \( \text{prev} \), but not for \( \text{prev}[1] = 0 \)). It follows that the algorithm takes \( \mathcal{O}(n) \) time, plus the time needed to perform the suffix comparisons.

Next, we discuss three possible implementations of these comparisons.

Algorithm 1(b): Using the inverse suffix array. The inverse suffix array \( \text{lexrank}[1..n] \) of \( x[1..n] \) is the unique permutation of \( [1, n] \) that satisfies \( \forall \ell, r \in [1, n] : x_\ell \prec x_r \iff \text{lexrank}[\ell] < \text{lexrank}[r] \) (an example is provided in Figure 2). Algorithm 1(b) precomputes the inverse suffix array, and then uses it to perform the lexicographical suffix comparisons. This idea was first proposed by Franek et al. [9, Algorithm NSVISA]. As discussed in Section 1, the precomputation takes \( \mathcal{O}(n \log n) \) time for general ordered alphabets, or \( \mathcal{O}(n) \) time for linearly-sortable alphabets. The remainder of the algorithm takes \( \mathcal{O}(n) \) time because we perform \( \mathcal{O}(n) \) suffix comparisons, each of which takes constant time when using \( \text{lexrank} \).

Algorithm 1(c): Using LCEs with simple scanning. If \( \ell \neq r \) and \( x[n] = \$ \), then it holds \( x_\ell \prec x_r \iff x[\ell + \text{LCE}(\ell, r)] < x[r + \text{LCE}(\ell, r)] \) (both of the conditions are satisfied for the comparisons performed by our algorithms). The LCE can be computed by simple scanning, as shown in Algorithm 1(c). Due to the sentinel \( x[n] = \$ \), no suffix is prefix of another suffix, and we always find a mismatching symbol eventually. Algorithm 1(c) implements the lexicographical suffix comparisons with LCEs (lines 7, 8, and 10). Additionally, it stores the computed LCEs in two arrays \( \text{nice} \) and \( \text{pice} \) (lines 3–4, 9, and 15), where after termination it holds \( \text{pice}[i] = \text{LCE} (\text{prev}[i], i) \) and \( \text{nice}[i] = \text{LCE} (i, \text{next}[i]) \) for all \( i \in (1, n) \). These arrays are of independent interest. For example, \( \text{nice} \) is useful when computing maximal repetitions [8].

Computing some \( \text{LCE}(\ell, r) \) by scanning takes \( \mathcal{O}(\text{LCE}(\ell, r)) + 1 \) symbol comparisons: \( \text{LCE}(\ell, r) \) comparisons with outcome “equal”, and one comparison with outcome “not equal”. For the example iteration in Figure 2, we compute \( \text{LCE}(21, 22) = 0 \), \( \text{LCE}(20, 22) = 1 \), \( \text{LCE}(17, 22) = 2 \),
Lyndon Arrays Simplified

LCE\((12, 22) = 2\), LCE\((5, 22) = 6\) and LCE\((5, 22) = 2\), and thus we perform 19 symbol comparisons. In the worst case, a single LCE scan takes \(O(n)\) time, and thus Algorithm 1(c) takes \(O(n^2)\) time (the bound is tight, e.g., for the string \(x = \#a^{n-2}\$\)). If the string \(x\) is drawn uniformly at random from the set of length-\(n\) strings over \(\Sigma\), where \(|\Sigma| > 1\), then the expected running time of Algorithm 1(c) is \(O(n)\) (see [1, Theorem 7]).

Algorithm 1(d): Using LCEs with improved scanning. In a single outer loop iteration \(r\) of Algorithm 1(c), we may compute LCE\((\ell, r)\) for multiple different values of \(\ell\). So far, we always scanned each new LCE entirely from scratch (line 10). For many of the LCEs, we can avoid (a part of) the scan by utilizing Lemma 7. This is done in Algorithm 1(d), which is identical to Algorithm 1(c), except for the highlighted computation of the LCE.

We always scanned each new LCE entirely from scratch (line 10). For many of the LCEs, we can eliminate (a part of) the scan by utilizing Lemma 7. This is done in Algorithm 1(d), which is identical to Algorithm 1(c), except for the highlighted computation of the LCE.

A note on the space complexity. Algorithms 1(c) and 1(d) require \(4n \lceil \log_2 n \rceil\) bits to store the arrays \(\text{next}, \text{prev}, \text{nLCE}\) and \(\text{pLCE}\). For a small practical improvement, it is possible to remove the array \(\text{prev}\). This is because the only access to \(\text{prev}[\ell]\) occurs at the same time at which we assign \(\text{next}[\ell]\) (see lines 9 and 14). Thus, we only need to maintain access to the values \(\text{prev}[\ell]\) for positions with uninitialized \(\text{next}[\ell]\), which means that we can use a single array for storing both PSS and NSS information. The total working space (without the input string) then becomes \(3n \lceil \log_2 n \rceil + O(\lg n)\) bits.

5 Achieving Linear Time

In order to achieve linear time, we use the function SMART-LCE (Algorithm 2) to more efficiently compute LCEs. A call to SMART-LCE\((\ell, r, m)\) means that we want to compute LCE\((\ell, r)\), and we have already established LCE\((\ell, r) \geq m\). We modify Algorithm 1(d) by replacing line 7 with \(m \leftarrow \text{SMART-LCE}(\ell, r, 0)\), and line 11 with \(m \leftarrow \text{SMART-LCE}(\text{prev}[\ell], r, m)\) (and leave everything else unchanged). In the remainder of the section, we show that SMART-LCE works correctly, and that the total time spent for all invocations of SMART-LCE is \(O(n)\). Then, it directly follows that the modified version of Algorithm 1(d) takes \(O(n)\) time. Note that Algorithm 2 is tailored to (and thus only works as a part of) Algorithm 1(d).
Algorithm 2 Efficient LCE computation (only works in conjunction with Algorithm 1(d)).

Require: string \( x = x[1..n] = \#x(i..n)\$ with \( x[\ell..\ell + m] = x[r..r + m] \).
Ensure: longest common extension \( LCE(\ell, r) \)

1: global variable \( c \leftarrow 0 \)
2: global variable \( d \leftarrow 0 \)
3: function \( \text{SMART-LCE}(\ell, r, m) \)
4: \hspace{1em} if \( r + m < c \) then
5: \hspace{2em} if \( \text{next}[\ell - d] = r - d \) then \( m \leftarrow \text{nlce}[\ell - d] \)
6: \hspace{2em} else \( m \leftarrow \text{plce}[r - d] \)
7: \hspace{1em} if \( r + m < c \) then return \( m \)
8: \hspace{1em} \( m \leftarrow c - r \)
9: \hspace{1em} while \( x[\ell + m] = x[r + m] \) do
10: \hspace{2em} \( m \leftarrow m + 1 \)
11: \hspace{1em} \( c, d \leftarrow r + m, r - \ell \)
12: \hspace{1em} return \( m \)

In the following description, whenever we use the variables \( \ell, r, \) and \( m \), we mean the arguments of the function \( \text{SMART-LCE} \) (rather than the identically named variables from Algorithm 1(d)). Now we explain how the new LCE function works. Generally speaking, it computes LCEs with two different methods: naive scanning (as done before), and deduction from previously computed LCEs. Sometimes, a combination of both is necessary. Both methods rely on a global variable \( c \) (persistent between the function calls) that stores at all times the rightmost position of the string that we have already inspected (line 2).

Scanning LCEs. We start by explaining the simpler method of naive scanning. If at the beginning of the function call it holds \( r + m \geq c \) (line 4), then we simply scan the remainder of the LCE (lines 9–10; identical to what we did in \( \text{LCE-EXTEND} \)). Let \( m' \) be the initial value of \( m \) before the scan, and let \( m'' = \text{LCE}(\ell, r) \) be the final value of \( m \) after performing the scan. After the scan, the rightmost inspected position is \( r + m'' \), and we update \( c \) accordingly (line 11; the variable \( d \) is not relevant for now). Since we only perform the scan if \( r + m' \geq c \), the assignment \( c \leftarrow r + m'' \) increases \( c \) by at least \( m'' - m' \). Note that \( m'' - m' \) is also exactly the number of times we execute line 10. Since \( c \) never exceeds \( n \), we execute line 10 no more than \( n \) times during all the calls to \( \text{SMART-LCE} \) that initially satisfy \( r + m \geq c \). It follows that, for all of these calls together, we spend at most \( \mathcal{O}(n) \) time.

Deducing LCEs. If at the beginning of the function call it holds \( r + m < c \), then we try to deduce \( LCE(\ell, r) \) from previously computed LCEs (lines 4–8). Let \( r_c \) be the rightmost position for which we already computed some \( \text{LCE}(\ell_c, r_c) \) with \( r_c + \text{LCE}(\ell_c, r_c) = c \) (such a position must exist because otherwise we would not have inspected \( x[c] \) yet). The global variable \( d \) contains at all times the distance \( r_c - \ell_c \) (line 2; we update \( d \) together with \( c \), see line 11). Let \( \ell_* = \ell - d \) and \( r_* = r - d \). The example in Figure 3 helps with understanding the notation. Later, we will show that (as suggested by the examples)

1. it holds \( r_c \leq \ell < r < c \), and thus \( \ell_c \leq \ell_* < r_* < \ell_c + \text{LCE}(\ell_c, r_c) \), and
2. either \( \text{prev}[r_*] = \ell_* \) (and thus \( \text{plce}[r_*] = \text{LCE}(\ell_c, r_c) \))
   or \( \text{next}[\ell_*] = r_* \) (and thus \( \text{nlce}[\ell_*] = \text{LCE}(\ell_c, r_c) \)).
Figure 3 Deducing LCEs with Algorithm 2 in (a) and (c), and deducing longest palindromes with Manacher’s algorithm in (b) and (d). Boxes of equal color indicate equal substrings. In (b) and (d), boxes of equal color sometimes indicate substrings that are the reverse of each other.

When deducing LCEs, we first use (ii) to obtain $LCE(\ell, r)$ (lines 5–6). Note that, by the definition of $\ell$ and $r$, the relative positions of $\ell$ and $r$ within $x[\ell..\ell + LCE(\ell, r)]$ are the same as the relative positions of $\ell$ and $r$ within $x[r..r + LCE(\ell, r)]$ (and the positions are indeed within these intervals due to (i)). If $r + LCE(\ell, r) < c$ then

$$x[r..r + LCE(\ell, r)] = x[\ell..\ell + LCE(\ell, r)]$$

and

$$x[\ell..\ell + LCE(\ell, r)] = x[\ell..\ell + LCE(\ell, r)],$$

where both equalities follow from $x[\ell..\ell + LCE(\ell, r)] = x[r..r + LCE(\ell, r)]$. This implies $LCE(\ell, r) = LCE(\ell, r)$, and we return $LCE(\ell, r)$ in constant time (line 7). Since it holds $r + LCE(\ell, r) < c$, there is no need to update $c$ and $d$. In Figure 3a, we have $21 + LCE(4, 7) = 23 < 29 = c$, and thus $LCE(18, 21) = LCE(4, 7) = 2$.

If, however, $r + LCE(\ell, r) \geq c$ then we cannot immediately deduce the exact value of $LCE(\ell, r)$ (as is the case in Figure 3c). We can still obtain some useful information because of

$$x[r..r + c - r] = x[\ell..\ell + c - r] = x[\ell..\ell + c - r],$$

where the first and the third equality follow from $x[\ell..\ell + LCE(\ell, r)] = x[r..r + LCE(\ell, r)]$, and the second equality follows from $r + LCE(\ell, r) \geq c$, which is equal to $LCE(\ell, r) \geq c - r$. The equation implies $LCE(\ell, r) \geq c - r$, and we update $m$ accordingly (line 8). In Figure 3c, we have $26 + LCE(7, 12) = 29 = c$, and thus $LCE(21, 26) \geq 29 - 26 = 3$.

We compute the remaining part of $LCE(\ell, r)$ by scanning (lines 9–10), and then update $c$ and $d$ (line 11). Since we assign $m \leftarrow (c - r)$ immediately before starting the scan, we can use the same argument as in the previous paragraph about scanning LCEs: For every symbol comparison of the scan (except for the last one), we will increase $c$ by one. Therefore, the total number of symbol comparisons for all calls of SMART-LCE is $O(n)$. In Figure 3c, the scan extends the LCE by two more positions, and we obtain $LCE(21, 26) = 5$. We then have to update $c \leftarrow 26 + 5 = 31$ and $d \leftarrow 26 - 21 = 5$.

The correctness of the algorithm follows from its description and the properties (i) and (ii), which we will show in the next paragraphs.
Showing Property (i). The property states that, if we call smart-lce(ℓ, r, m) with r + m < c, then r_c ≤ ℓ < r < c. Since trivially ℓ < r ≤ r + m < c, we only have to show r_c ≤ ℓ. The property is readily proven for the call smart-lce(ℓ, r, 0) in line 7 of Algorithm 1(d). It holds ℓ = r − 1, and this is the first LCE that we compute between r and any smaller index. Since we already computed LCE(ℓ_c, r_c), it holds r > r_c and ℓ = r − 1 ≥ r_c.

Now we consider the call smart-lce(ℓ, r, m) in line 11. As seen in the description of Algorithm 1(d), for this call it holds m = LCE(ℓ, k) = LCE(k, r), where k ∈ (ℓ, r) with prev[k] = ℓ and next[k] = r. For every h ∈ (ℓ, r), the definition of prev and next implies that x_h ≥ x_k > x_r. This also means that m = LCE(k, r) ≥ LCE(h, r). If r = r_c then, because we already computed LCE(ℓ_c, r), and due to the iteration order of the algorithm, it holds ℓ < ℓ_c.

Then, however, ℓ_c ∈ (ℓ, r) and thus m = LCE(k, r) ≥ LCE(ℓ_c, r) = c − r, which contradicts r + m < c. We have shown that r > r_c, which also implies r_c > ℓ_c. If ℓ < r_c then r_c ∈ (ℓ, r) and thus m ≥ LCE(r_c, r) = c − r, which contradicts r + m < c. It follows that ℓ ≥ r_c > ℓ_c. Finally, if ℓ ∈ (ℓ_c, r_c) then ℓ_c < ℓ < r_c < r, which contradicts Lemma 5. The only remaining possibility is ℓ ≥ c, which is what we wanted to show.

Showing Property (ii). The property states that either prev[r_c] = ℓ_c, or next[ℓ_c] = r_c. By the definition of ℓ_c and r_c, and due to (i) and x[ℓ_c..ℓ_c + LCE(ℓ_c, r_c)] = x[r_c..r_c + LCE(ℓ_c, r_c)], it holds x[ℓ..r] = x[ℓ_c..r_c]. Since we want to compute LCE(ℓ, r), it holds either next[ℓ] = r or prev[r] = ℓ. Therefore, either Lemma 3 or Lemma 4 implies that x[ℓ..r] = x[ℓ_c..r_c] is a Lyndon word. Due to Lemma 3, we know that next[ℓ_c] ≥ r_c. If next[ℓ_c] = r_c or prev[r_c] = ℓ_c, then there is nothing left to show. Thus, assume that next[ℓ_c] > r_c and prev[r_c] > ℓ_c (it cannot be that prev[r_c] < ℓ_c, because then prev[r_c] < ℓ_c < r_c < next[ℓ_c] contradicts Lemma 5).

Let p_c = r − (r_c − prev[r_c]). Due to Lemma 4, the substring x[prev[r_c]..r_c] = x[p_c..r] is a Lyndon word, and Lemma 3 implies next[p_c] ≥ r. Since p_c ∈ (ℓ, r) it holds next[p_c] ≤ r (otherwise we contradict Lemma 5), and the only possible option is next[p_c] = r.

We have shown that next[p_c] = r and thus x_p < x_r. By the definition of prev, it also holds x[prev[r_c]] < x_r. Since we chose r_c to be the rightmost index with r_c + LCE(ℓ_c, r_c) = c, it holds r + LCE(p_c, r) < c (otherwise we would have updated r_c already). Therefore, we have

\[ x[prev[r_c]..prev[r_c] + LCE(p_c, r)] = x[p_c..p_r + LCE(p_c, r)], \]
\[ x[r_c..r_c + LCE(p_c, r)] = x[r..r + LCE(p_c, r)]. \]

This, however, means that x[prev[r_c]] < x_r \iff x_p < x_r, which contradicts our previous observation that x_p < x_r and x[prev[r_c]] < x_r. It follows that the assumption next[ℓ_c] > r_c and prev[r_c] > ℓ_c was wrong, and it holds next[ℓ_c] = r_c or prev[r_c] = ℓ_c.

6 Similarity to Manacher’s Algorithm

In this section, we want to briefly highlight the similarity between the technique of Section 5 and Manacher’s algorithm for computing maximal palindromes [14]. For simplicity, we only consider odd palindromes. An odd palindrome of radius |w| + 1 is a string of the form ws\overline{w}, where s is a symbol, w is some possibly empty string, and \overline{w} is the reverse string of w defined by |\overline{w}| = |w| and \forall i \in [1, |w|] : \overline{w}[i] = w[|w| - i + 1]. For a string x[1..n], the presented version of Manacher’s algorithm computes an array rad[1..n], where \forall i \in [1, n] :

\[ rad[i] = \max\{m \mid m \in [1, \min(i, n - i + 1)] \text{ and } x(i - m..i + m) \text{ is an odd palindrome}\}. \]

If we compute the entries of rad in left-to-right order, then we can sometimes fully or partially
Algorithm 3 Manacher’s algorithm for odd palindromes.

Require: string $x = x[1..n] = \#x(1..n)\$. 
Ensure: array rad containing the radii of the longest odd palindromes 

1: $\text{rad}[1..n] \leftarrow \text{new array initialized with } \text{rad}[1] = \text{rad}[n] = 1$
2: global variable $c \leftarrow 0$
3: global variable $j \leftarrow 0$
4: for $i = 2$ to $n - 1$ do 
5: \[ \text{rad}[i] \leftarrow \text{SMART-PALINDROME}(i, 1) \]

function SMART-PALINDROME$(i, m)$ 
1: if $i + m < c$ then 
2: \[ m \leftarrow \text{rad}[j - (i - j)] \]
3: if $i + m < c$ then return $m$
4: \[ m \leftarrow c - i \]
5: \[ \text{while } x[i - m] = x[i + m] \text{ do} \]
6: \[ m \leftarrow m + 1 \]
7: $c, j \leftarrow i + m, i$
8: return $m$

Proofs for Section 3

Lemma 5. Let $x = \#x(1..n)$ be a string with previous and next smaller suffix array prev and next. Let $\ell_1, \ell_2, r_1, r_2 \in [1, n]$ be indices with either next$[\ell_1] = r_1$ or prev$[r_1] = \ell_1$, and also either next$[\ell_2] = r_2$ or prev$[r_2] = \ell_2$. Then it does not hold $\ell_1 < \ell_2 < r_1 < r_2$.

Proof. Assume $\ell_1 < \ell_2 < r_1 < r_2$. Since $\ell_2 \in (\ell_1, r_1)$ and either next$[\ell_1] = r_1$ or prev$[r_1] = \ell_1$, Definition 2 implies $x_{\ell_2} > x_{\ell_1}$. However, it also holds $r_1 \in (\ell_2, r_2)$ and either next$[\ell_2] = r_2$ or prev$[r_2] = \ell_2$. Thus, Definition 2 also implies $x_{\ell_2} < x_{r_1}$, which is a contradiction.

Lemma 6. Let $x = \#x(1..n)$ be a string with previous and next smaller suffix arrays prev and next, and let $\ell, r \in [1, n]$ be arbitrary indices.

(i) It holds prev$[r] = \text{prev}^*[r - 1]$.

(ii) It holds next$[\ell] = r$ if and only if $\ell = \text{prev}^*[r - 1]$ and $\ell > \text{prev}[r]$.

Proof. For (i), assume that prev$[r] \neq \text{prev}^*[r - 1]$. Then there must be some $r' = \text{prev}^*[r - 1]$ with prev$[r'] \in (\text{prev}[r'], r')$. However, prev$[r'] < \text{prev}[r] < r' < r$ contradicts Lemma 5.

For (ii), we show both directions separately. Assume that next$[\ell] = r$ then $\forall k \in [\ell, r)$ : $x_k > x_r$, which means prev$[r] \notin [\ell, r)$ and thus prev$[r] < \ell$. Assume that $\ell \neq \text{prev}^*[r - 1]$, then there must be some $r' = \text{prev}^*[r - 1]$ with $\ell \in (\text{prev}[r'], r')$. However, then prev$[r'] < \ell < r' < r$ and Lemma 5 contradict the assumption that next$[\ell] = r$. Thus $\ell \neq \text{prev}^*[r - 1]$.

For the counter direction, assume that prev$[r] < \ell$ and $\ell = \text{prev}^*[r - 1]$. By the definition of prev, and due to $\ell \in (\text{prev}[r'], r)$, it holds $x_k > x_r$. It is easy to see that $\ell = \text{prev}^*[r - 1]$ implies $\forall k \in (\ell, r) : x_k > x_\ell$ (when following a chain of PSS edges, by definition of prev, the visited suffixes are lexicographically decreasing, and all suffixes skipped by a PSS edge are lexicographically larger). From $x_\ell > x_r$ and $\forall k \in (\ell, r) : x_k > x_\ell$ follows next$[\ell] = r$. 

Proof. Assume $\ell_1 < \ell_2 < r_1 < r_2$. Since $\ell_2 \in (\ell_1, r_1)$ and either next$[\ell_1] = r_1$ or prev$[r_1] = \ell_1$, Definition 2 implies $x_{\ell_2} > x_{\ell_1}$. However, it also holds $r_1 \in (\ell_2, r_2)$ and either next$[\ell_2] = r_2$ or prev$[r_2] = \ell_2$. Thus, Definition 2 also implies $x_{\ell_2} < x_{r_1}$, which is a contradiction. 

Proof. For (i), assume that prev$[r] \neq \text{prev}^*[r - 1]$. Then there must be some $r' = \text{prev}^*[r - 1]$ with prev$[r'] \in (\text{prev}[r'], r')$. However, prev$[r'] < \text{prev}[r] < r' < r$ contradicts Lemma 5.

For (ii), we show both directions separately. Assume that next$[\ell] = r$ then $\forall k \in [\ell, r)$ : $x_k > x_r$, which means prev$[r] \notin [\ell, r)$ and thus prev$[r] < \ell$. Assume that $\ell \neq \text{prev}^*[r - 1]$, then there must be some $r' = \text{prev}^*[r - 1]$ with $\ell \in (\text{prev}[r'], r')$. However, then prev$[r'] < \ell < r' < r$ and Lemma 5 contradict the assumption that next$[\ell] = r$. Thus $\ell \neq \text{prev}^*[r - 1]$.

For the counter direction, assume that prev$[r] < \ell$ and $\ell = \text{prev}^*[r - 1]$. By the definition of prev, and due to $\ell \in (\text{prev}[r'], r)$, it holds $x_k > x_r$. It is easy to see that $\ell = \text{prev}^*[r - 1]$ implies $\forall k \in (\ell, r) : x_k > x_\ell$ (when following a chain of PSS edges, by definition of prev, the visited suffixes are lexicographically decreasing, and all suffixes skipped by a PSS edge are lexicographically larger). From $x_\ell > x_r$ and $\forall k \in (\ell, r) : x_k > x_\ell$ follows next$[\ell] = r$. 

Proof. Assume $\ell_1 < \ell_2 < r_1 < r_2$. Since $\ell_2 \in (\ell_1, r_1)$ and either next$[\ell_1] = r_1$ or prev$[r_1] = \ell_1$, Definition 2 implies $x_{\ell_2} > x_{\ell_1}$. However, it also holds $r_1 \in (\ell_2, r_2)$ and either next$[\ell_2] = r_2$ or prev$[r_2] = \ell_2$. Thus, Definition 2 also implies $x_{\ell_2} < x_{r_1}$, which is a contradiction.
Lemma 7. Let $x = \#x(1..n)$ be a string with previous and next smaller suffix arrays $\text{prev}$ and $\text{next}$, Let $k \in (1, n)$ be an arbitrary index, and let $\ell = \text{prev}[k]$ and $r = \text{next}[k]$.

(i) If $\text{lce}([\ell, k]) = \text{lce}([k, r])$, then $\text{lce}([\ell, r]) \geq \text{lce}([k, r])$ and either $\text{prev}[r] = \ell$ or $\text{next}[\ell] = r$.

(ii) If $\text{lce}([\ell, k]) < \text{lce}([k, r])$, then $\text{lce}([\ell, r]) = \text{lce}([k, r])$ and $\text{prev}[r] = \ell$.

(iii) If $\text{lce}([\ell, k]) > \text{lce}([k, r])$, then $\text{lce}([\ell, r]) = \text{lce}([k, r])$ and $\text{next}[\ell] = r$.

Proof. We start with (i). Definition 2 implies $\forall i \in (\ell, k) \cup (k, r) : x_k < x_i$. Since $\ell = \text{prev}[k]$ we have $x_\ell < x_k$ and thus $\forall i \in (\ell, r) : x_\ell < x_r$. Analogously, due to $r = \text{next}[k]$ we have $\forall i \in (\ell, r) : x_r < x_i$. Thus, if $x_\ell < x_r$ then $\text{prev}[r] = \ell$, and if $x_\ell > x_r$ then $\text{next}[\ell] = r$.

For showing (ii), let $u = x[\ell..\ell + \text{lce}([\ell, k])]$, $s = x[\ell + \text{lce}([\ell, k])]$, and $t = x[k + \text{lce}([\ell, k])]$. By the definition of LCEs, it holds $s \neq t$. Due to $\ell = \text{prev}[k]$ we have $us \cdot x_{t+1} | us | = x_\ell \prec x_k = ut \cdot x_{k+1} | ut |$, and therefore $s < t$. Because of $\text{lce}([\ell, k]) < \text{lce}([k, r])$, suffix $x_r$ has prefix $ut$. Thus, it holds $x_\ell = us \cdot x_{t+1} | us | \prec ut \cdot x_{r+1} | ut | = x_r$. Due to (i), this means $\text{next}[\ell] = r$.

The proof of (iii) works analogously to the one for (ii).