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 $\mathcal{O}(n)$ 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 $\mathcal{O}(n)$ 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 $\mathcal{O}(n)$ 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 $\Omega(n \lg n)$ 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 $\mathcal{O}(n)$ time, if the alphabet can be sorted in $\mathcal{O}(n)$ time (e.g., a polynomial integer alphabet $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$ on a word RAM of width $w \geq \log_2 n$). This is not a concern 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 $\mathcal{O}(n)$ time. The first one [5] computes the Lyndon array using only $\mathcal{O}(1)$ 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 $\mathcal{O}(n)$ 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 $\{i, i + 1, \dots, j\}$ (or $\emptyset$ if $i > j$). A *string* $x = x[1..n]$ over an *ordered alphabet* Σ is a sequence $x = x[1]x[2]\cdots x[n]$ of *symbols* drawn from some totally ordered set Σ. We write $|x| = n$ to denote the length of the string. The set Σ^* contains all strings over Σ, including the *empty string* ϵ of length 0. The concatenation of two strings $x[1..n]$ and $y[1..m]$ is the sequence $x[1]\dots x[n]y[1]\dots y[m]$, and simply written as xy (or $x \cdot y$). The total order of symbols from Σ induces the *lexicographical order* of strings from Σ^*. We say that x is *lexicographically smaller* than y and write $x \prec y$, if and only if either $y = xw$ for some non-empty string w, or $x = urv$ and $y = usw$ for (possibly empty) strings u, v, w and symbols $r < s$. We write $x \preceq y$ to denote $x = y \lor x \prec y$.

For $i, j \in [1, n]$ with $i \le j$, 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 ϵ. If $x[i..j] \neq x$ then $x[i..j]$ is a *proper* substring. A substring of the form $x[1..j]$ 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 $\textsc{lce}(i, j) = \max\{|u| \mid u, v, w \in \Sigma^* \land x_i = uv \land x_j = uw\}$.

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 $x_i = x_j x_{i+|x_j|}$, often complicates the notation of definitions and algorithms. This can be avoided by assuming that the text starts and ends with special *sentinel symbols* $x[1] = \texttt{\#}$ and $x[n] = \texttt{\$}$. The sentinels are smaller than all other symbols, i.e., $\forall k \in (1, n) : x[k] > \texttt{\$} > \texttt{\#}$. In definitions and lemmas we emphasize the presence of sentinels by writing $x = x[1..n] = \texttt{\#}x(1..n)\texttt{\$}$. (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[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 λ, which was also done in [5, 9]. Other notations are *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 $\mathsf{prev}[1..n]$*, and the* next smaller suffix array $\mathsf{next}[1..n]$ *of a string $x = \#x(1..n)\$$ are defined as:*

(i) $\forall i \in [1, n] : \lambda[i] = \max\{m \mid m \in [1, n - i + 1] \wedge x[i..i + m) \text{ is a Lyndon word}\}$,
i.e., $x[i..i + \lambda[i])$ is the longest Lyndon prefix of suffix x_i

(ii) $\forall i \in (1, n) : \mathsf{prev}[i] = \max\{j \mid j \in [1, i) \wedge x_j \prec x_i\}$, $\quad \mathsf{prev}[1] = 0$, $\quad \mathsf{prev}[n] = 1$,
i.e., $x_{\mathsf{prev}[i]}$ is the nearest suffix starting left of i that is lex. smaller than x_i

(iii) $\forall i \in (1, n) : \mathsf{next}[i] = \min\{j \mid j \in (i, n] \wedge x_j \prec x_i\}$, $\quad \mathsf{next}[1] = n + 1$, $\quad \mathsf{next}[n] = n + 1$,
i.e., $x_{\mathsf{next}[i]}$ 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 $\mathsf{prev}[i] = j$ (whenever the edge is directed from right to left) or $\mathsf{next}[i] = j$ (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 $\mathsf{next}[i] = i + \lambda[i]$ 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][2] proved the property in the form stated below.

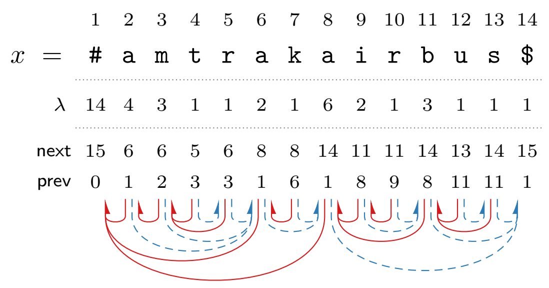
▶ **Lemma 3** ([12, 9, 11]). *Let $x = \#x(1..n)\$$ be a string with Lyndon array λ and next smaller suffix array next, then it holds $\forall i \in [1, n] : \mathsf{next}[i] = i + \lambda[i]$.*

Another property shown by Bille et al. [5] relates prev and Lyndon words:

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

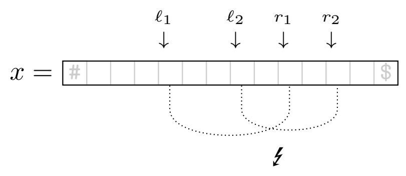
[1] see, e.g., lines 4–5 in file https://github.com/jonas-ellert/simple-lyndon/blob/main/lyndon.hpp
[2] Lemma 1 (b) in [11] should state "$x[i..j]$ is proto-Lyndon" rather than "$x[i..n]$ is proto-Lyndon"

3 Key Properties of Nearest Smaller Suffixes

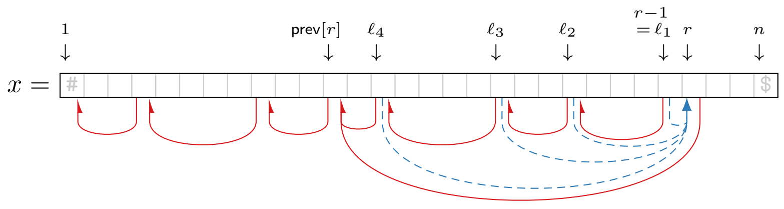


	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$x =$	#	a	m	t	r	a	k	a	i	r	b	u	s	$
λ	14	4	3	1	1	2	1	6	2	1	3	1	1	1
next	15	6	6	5	6	8	8	14	11	11	14	13	14	15
prev	0	1	2	3	3	1	6	1	8	9	8	11	11	1

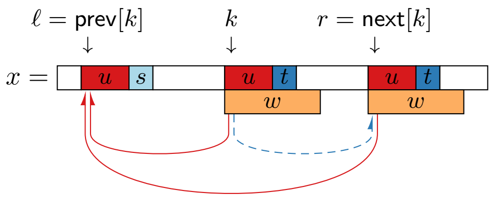
(a) Example of arrays λ, next and prev.



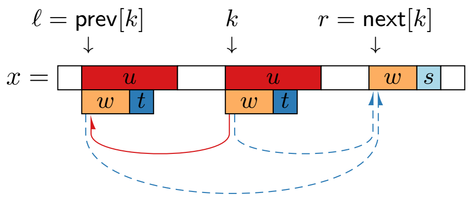
(b) Drawing for Lemma 5.



(c) Drawing for Lemma 6.



(d) Drawing for Lemma 7(ii).



(e) Drawing for Lemma 7(iii).

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 $\ell < r$.

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 $\ell_1, \ell_2, r_1, r_2 \in [1, n]$ 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 $\ell_1 < \ell_2 < r_1 < r_2$.*

Chains of previous smaller suffixes. For arbitrary index $r \in [1, n]$ and integer $e \geq 0$, we recursively define $\mathsf{prev}^0[r] = r$ and $\mathsf{prev}^{e+1}[r] = \mathsf{prev}^e[\mathsf{prev}[r]]$. We write $\ell = \mathsf{prev}^*[r]$ to denote that there is some integer $e \geq 0$ with $\ell = \mathsf{prev}^e[r]$. Note that generally $1 = \mathsf{prev}^*[r]$ (due to the sentinel $x[1] = \#$) and $r = \mathsf{prev}^*[r]$. In drawings, $\ell = \mathsf{prev}^e[r]$ means that there is a chain of PSS edges from r to ℓ. The following lemma states useful properties of PSS chains, which we will later use to compute the arrays prev and 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* prev *and* next*, and let $\ell, r \in [1, n]$ be arbitrary indices.*

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

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* prev *and* next*. Let $k \in (1, n)$ be an arbitrary index, and let $\ell = \mathsf{prev}[k]$ and $r = \mathsf{next}[k]$.*

(i) *If $\textsc{lce}(\ell, k) = \textsc{lce}(k, r)$, then $\textsc{lce}(\ell, r) \geq \textsc{lce}(k, r)$ and either $\mathsf{prev}[r] = \ell$ or $\mathsf{next}[\ell] = r$.*
(ii) *If $\textsc{lce}(\ell, k) < \textsc{lce}(k, r)$, then $\textsc{lce}(\ell, r) = \textsc{lce}(\ell, k)$ and $\mathsf{prev}[r] = \ell$.*
(iii) *If $\textsc{lce}(\ell, k) > \textsc{lce}(k, r)$, then $\textsc{lce}(\ell, r) = \textsc{lce}(k, r)$ and $\mathsf{next}[\ell] = r$.*

4 Algorithms to Compute the Lyndon Array

Due to Lemma 3, instead of designing algorithms that compute the Lyndon array λ, 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 $\mathcal{O}(n)$ time.

Algorithm 1: The general approach. Due to the sentinels, we can directly assign $\mathsf{prev}[1]$, $\mathsf{next}[1]$, and $\mathsf{next}[n]$ (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 $\mathsf{prev}[r]$, while also identifying all indices ℓ with $\mathsf{next}[\ell] = r$. 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 $\ell = \mathsf{prev}^*[r-1]$ one at a time, starting with $\ell = r-1$ (line 5). As long as $x_\ell \succ x_r$, we assign $\mathsf{next}[\ell] \leftarrow r$, and then continue with the next index $\ell \leftarrow \mathsf{prev}[\ell]$ on the chain of PSS edges (lines 6–8). As soon as $x_\ell \prec x_r$, we break out of the inner loop and finish the current iteration of the outer loop by assigning $\mathsf{prev}[r] \leftarrow \ell$ (line 9). (The sentinel $x[1] = \#$ ensures that $1 = \mathsf{prev}^*[r]$ and $x_1 \prec x_r$; thus we are guaranteed to reach some ℓ with $x_\ell \prec x_r$ 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 $\mathsf{lexrank}$, 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: $\mathsf{prev}[1..n] \leftarrow$ new array with $\mathsf{prev}[1] = 0$
2: $\mathsf{next}[1..n] \leftarrow$ new array with $\mathsf{next}[1] = \mathsf{next}[n] = n+1$
```

(a) Folklore

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

(b) ISA-NSV

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

(c) Naive LCE-NSS

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

(d) Improved LCE-NSS

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

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

```
function $\textsc{lce-extend}(\ell, r, m)$
    while $x[\ell + m] = x[r + m]$ do
        $m \leftarrow m + 1$
    return $m$
function $\textsc{lce-scan}(\ell, r)$
    return $\textsc{lce-extend}(\ell, r, 0)$
```

to achieve $\mathcal{O}(n)$ time, substitute:
line 7: $m \leftarrow \textsc{smart-lce}(\ell, r, 0)$
line 11: $m \leftarrow \textsc{smart-lce}(\mathsf{prev}[\ell], r, m)$
(see Algorithm 2)

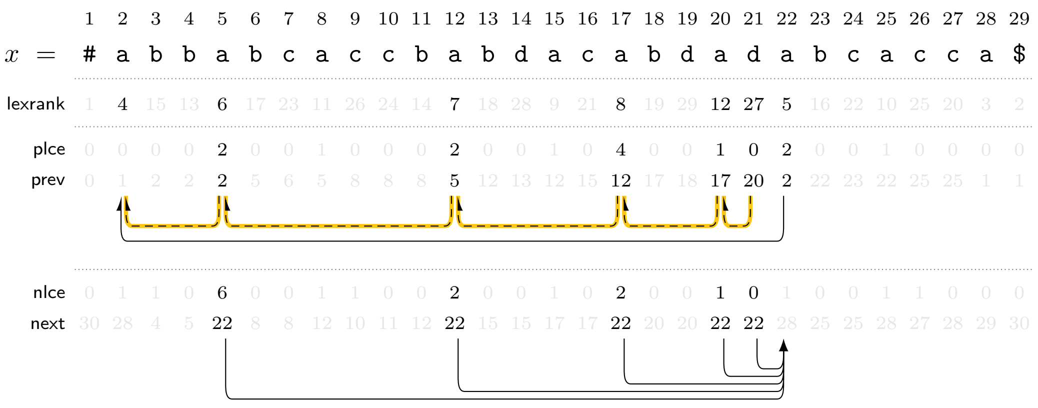

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
x = # a b b a b c a c c b a b d a c a b d a d a b c a c c a $
lexrank 1 4 15 13 6 17 23 11 26 24 14 7 18 28 9 21 8 19 29 12 27 5 16 22 10 25 20 3 2
plce 0 0 0 0 2 0 0 1 0 0 0 2 0 0 1 0 4 0 0 1 0 2 0 0 1 0 0 0 0
prev 0 1 2 2 2 5 6 5 8 8 8 5 12 13 12 15 12 17 18 17 20 2 22 23 22 25 25 1 1
nlce 0 1 1 0 6 0 0 1 1 0 0 2 0 0 1 0 2 0 0 1 0 1 0 0 1 1 0 0 0
next 30 28 4 5 22 8 8 12 10 11 12 22 15 15 17 17 22 20 20 22 22 28 25 25 28 27 28 29 30

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 $\mathsf{next}[\ell] \leftarrow 22$. We break out of the inner loop by discovering that $x_2 \prec x_{22}$, after which we assign $\mathsf{prev}[22] \leftarrow 2$. By design of the algorithm, and following Lemma 6, the values assumed by ℓ 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 $\mathsf{next}[\ell] \leftarrow r$ or $\mathsf{prev}[r] \leftarrow \ell$ immediately afterwards. Since each entry of 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 $\mathsf{next}[1] = \mathsf{next}[n] = n + 1$), and break out of the inner loop exactly $n - 1$ times (once per outer loop iteration, or equivalently once per entry of prev, but not for $\mathsf{prev}[1] = 0$). 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 $\mathsf{lexrank}[1..n]$ of $x[1..n]$ is the unique permutation of $[1, n]$ that satisfies $\forall \ell, r \in [1, n] : x_\ell \prec x_r \iff \mathsf{lexrank}[\ell] < \mathsf{lexrank}[r]$ (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 $\mathcal{O}(n \lg n)$ 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 lexrank.

Algorithm 1(c): Using LCEs with simple scanning. If $\ell \neq r$ and $x[n] = \$$, then it holds $x_\ell \prec x_r \iff x[\ell + \textsc{lce}(\ell, r)] < x[r + \textsc{lce}(\ell, r)]$ (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(e). 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 nlce and plce (lines 3–4, 9, and 15), where after termination it holds $\mathsf{plce}[i] = \textsc{lce}(\mathsf{prev}[i], i)$ and $\mathsf{nlce}[i] = \textsc{lce}(i, \mathsf{next}[i])$ for all $i \in (1, n)$. These arrays are of independent interest. For example, nlce is useful when computing maximal repetitions [8].

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

$\textsc{lce}(12, 22) = 2$, $\textsc{lce}(5, 22) = 6$ and $\textsc{lce}(5, 22) = 2$, and thus we perform 19 symbol comparisons. In the worst case, a single LCE scan takes $\mathcal{O}(n)$ time, and thus Algorithm 1(c) takes $\mathcal{O}(n^2)$ time (the bound is tight, e.g., for the string $x = \#\mathsf{a}^{n-2}\$$). If the string x is drawn uniformly at random from the set of length-n strings over Σ, where $|\Sigma| > 1$, then the expected running time of Algorithm 1(c) is $\mathcal{O}(n)$ (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 $\textsc{lce}(\ell, r)$ for multiple different values of ℓ. 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 in lines 10–13. At the point in time at which we reach line 10, let $k' = \ell$, $\ell' = \mathsf{prev}[\ell]$, and $r' = r$. Note that $\ell' = \mathsf{prev}[k']$ and $r' = \mathsf{next}[k']$, and thus we can use ℓ', k', and r' to invoke Lemma 7. We already computed $\textsc{lce}(\ell', k') = \mathsf{plce}[k']$ (due to the iteration order of the algorithm) and $\textsc{lce}(k', r') = \mathsf{nlce}[k'] = m$ (this is the most recently computed LCE). Now we compute $\textsc{lce}(\ell', r')$ according to the cases of Lemma 7:

- If $\textsc{lce}(\ell', k') = \textsc{lce}(k', r')$ then Lemma 7(i) implies $\textsc{lce}(\ell', r') \geq \textsc{lce}(k', r')$. We compute $\textsc{lce}(\ell', r')$ by scanning, but we skip $m = \textsc{lce}(k', r')$ symbol comparisons (lines 10–11).
- If $\textsc{lce}(\ell', k') < \textsc{lce}(k', r')$ then Lemma 7(ii) implies $\textsc{lce}(\ell', r') = \textsc{lce}(\ell', k')$. Since $\mathsf{plce}[k'] = \textsc{lce}(\ell', k')$, we can simply assign $m \leftarrow \mathsf{plce}[k']$ (lines 12–13). Note that Lemma 7(ii) also implies $\mathsf{prev}[r'] = \ell'$, which means that we will immediately break out of the inner loop and finish the current iteration of the outer loop.
- If $\textsc{lce}(\ell', k') > \textsc{lce}(k', r')$ then Lemma 7(iii) implies $\textsc{lce}(\ell', r') = \textsc{lce}(k', r')$. It already holds $m = \textsc{lce}(k', r')$, and thus there is no need to do anything.

For the example iteration in Figure 2, we entirely skip the computation of $\textsc{lce}(12, 22)$ due to Lemma 7(iii), as well as the computation of $\textsc{lce}(2, 22)$ due to Lemma 7(ii). Additionally, we skip two symbol comparisons when computing $\textsc{lce}(5, 22)$, and one symbol comparison when computing $\textsc{lce}(17, 22)$. The number of symbol comparisons for iteration 22 is 10 (significantly less than the 19 comparisons needed by Algorithm 1(c)). However, Algorithm 1(d) still takes $\mathcal{O}(n^2)$ time in the worst case. In the next section, we slightly modify the algorithm such that it achieves linear time.

A note on the space complexity. Algorithms 1(c) and 1(d) require $4n \lceil \log_2 n \rceil$ bits to store the arrays next, prev, nlce and plce. For a small practical improvement, it is possible to remove the array prev. This is because the only access to $\mathsf{prev}[\ell]$ occurs at the same time at which we assign $\mathsf{next}[\ell]$ (see lines 9 and 14). Thus, we only need to maintain access to the values $\mathsf{prev}[\ell]$ for positions with uninitialized $\mathsf{next}[\ell]$, 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 $3n \lceil \log_2 n \rceil + \mathcal{O}(\lg n)$ bits.

5 Achieving Linear Time

In order to achieve linear time, we use the function $\textsc{smart-lce}$ (Algorithm 2) to more efficiently compute LCEs. A call to $\textsc{smart-lce}(\ell, r, m)$ means that we want to compute $\textsc{lce}(\ell, r)$, and we have already established $\textsc{lce}(\ell, r) \geq m$. We modify Algorithm 1(d) by replacing line 7 with $m \leftarrow \textsc{smart-lce}(\ell, r, 0)$, and line 11 with $m \leftarrow \textsc{smart-lce}(\mathsf{prev}[\ell], r, m)$ (and leave everything else unchanged). In the remainder of the section, we show that $\textsc{smart-lce}$ works correctly, and that the total time spent for all invocations of $\textsc{smart-lce}$ is $\mathcal{O}(n)$. Then, it directly follows that the modified version of Algorithm 1(d) takes $\mathcal{O}(n)$ 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 $\textsc{lce}(\ell, r)$
1: global variable $c \leftarrow 0$
2: global variable $d \leftarrow 0$
3: function SMART-LCE($\ell, r, m$)
4:     if $r + m < c$ then
5:         if $\mathsf{next}[\ell - d] = r - d$ then $m \leftarrow \mathsf{nlce}[\ell - d]$
6:                                  else $m \leftarrow \mathsf{plce}[r - d]$
7:         if $r + m < c$ then return $m$
8:         $m \leftarrow c - r$
9:     while $x[\ell + m] = x[r + m]$ do
10:        $m \leftarrow m + 1$
11:    $c, d \leftarrow r + m, r - \ell$
12:    return $m$
```

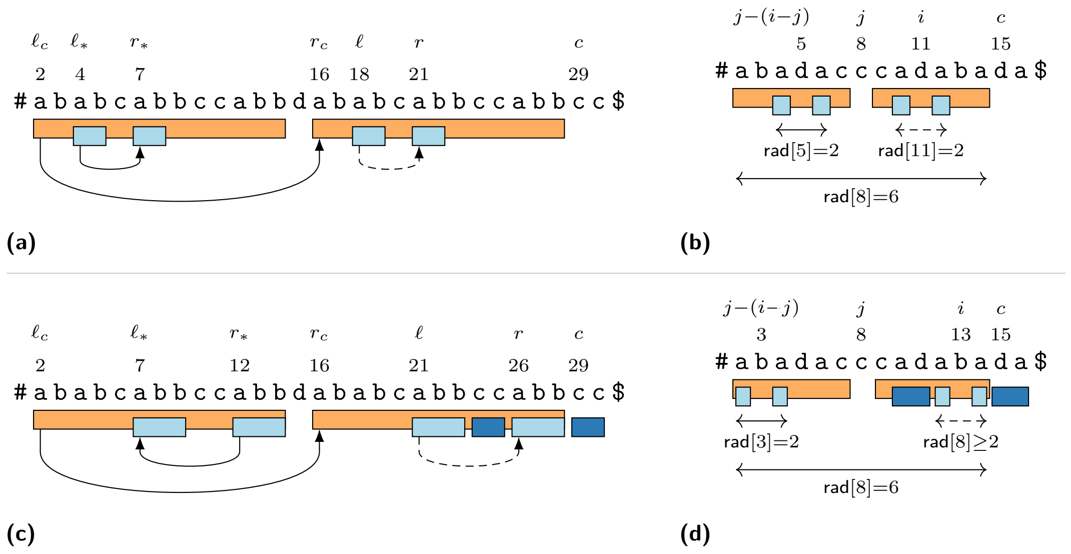
In the following description, whenever we use the variables ℓ, r, and m, we mean the arguments of the function SMART-LCE (rather than the identically named variables from Algorithm 1(d)). Now we explain how the new LCE function works. Generally speaking, it computes LCEs with two different methods: naive 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 LCE-EXTEND). Let m' be the initial value of m before the scan, and let $m'' = \textsc{lce}(\ell, r)$ 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 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 $\textsc{lce}(\ell, r)$ from previously computed LCEs (lines 4–8). Let r_c be the rightmost position for which we already computed some $\textsc{lce}(\ell_c, r_c)$ with $r_c + \textsc{lce}(\ell_c, r_c) = c$ (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)

(i) it holds $r_c \leq \ell < r < c$, and thus $\ell_c \leq \ell_* < r_* < \ell_c + \textsc{lce}(\ell_c, r_c)$, and

(ii) either $\mathsf{prev}[r_*] = \ell_*$ (and thus $\mathsf{plce}[r_*] = \textsc{lce}(\ell_*, r_*)$)
or $\mathsf{next}[\ell_*] = r_*$ (and thus $\mathsf{nlce}[\ell_*] = \textsc{lce}(\ell_*, r_*)$).



ℓ_c ℓ_* r_* r_c ℓ r c
2 4 7 16 18 21 29
# a b a b c a b b c c a b b d a b a b c a b b c c a b b c c $
(a)
$j-(i-j)$ j i c
5 8 11 15
# a b a d a c c c a d a b a d a $
rad[5]=2
rad[11]=2
rad[8]=6
(b)
ℓ_c ℓ_* r_* r_c ℓ r c
2 7 12 16 21 26 29
# a b a b c a b b c c a b b d a b a b c a b b c c a b b c c $
(c)
$j-(i-j)$ j i c
3 8 13 15
# a b a d a c c c a d a b a d a $
rad[3]=2
rad[8]≥2
rad[8]=6
(d)

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 $\textsc{lce}(\ell_*, r_*)$ (lines 5–6). Note that, by the definition of ℓ_* and r_*, the relative positions of ℓ_* and r_* within $x[\ell_c..\ell_c + \textsc{lce}(\ell_c, r_c))$ are the same as the relative positions of ℓ and r within $x[r_c..r_c + \textsc{lce}(\ell_c, r_c))$ (and the positions are indeed within these intervals due to (i)). If $r + \textsc{lce}(\ell_*, r_*) < c$ then

$$x[r..r + \textsc{lce}(\ell_*, r_*)] = x[r_*..r_* + \textsc{lce}(\ell_*, r_*)] \text{ and}$$
$$x[\ell..\ell + \textsc{lce}(\ell_*, r_*)] = x[\ell_*..\ell_* + \textsc{lce}(\ell_*, r_*)],$$

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

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

$$x[r..r + c - r) = x[r_*..r_* + c - r) = x[\ell_*..\ell_* + c - r) = x[\ell..\ell + c - r),$$

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

We compute the remaining part of $\textsc{lce}(\ell, r)$ 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 $\mathcal{O}(n)$. In Figure 3c, the scan extends the LCE by two more positions, and we obtain $\textsc{lce}(21, 26) = 5$. 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 $\textsc{smart-lce}(\ell, r, m)$ with $r+m < c$, then $r_c \leq \ell < r < c$. Since trivially $\ell < r \leq r+m < c$, we only have to show $r_c \leq \ell$. The property is readily proven for the call $\textsc{smart-lce}(\ell, r, 0)$ in line 7 of Algorithm 1(d). It holds $\ell = r-1$, and this is the first LCE that we compute between r and any smaller index. Since we already computed $\textsc{lce}(\ell_c, r_c)$, it holds $r > r_c$ and $\ell = r-1 \geq r_c$.

Now we consider the call $\textsc{smart-lce}(\ell, r, m)$ in line 11. As seen in the description of Algorithm 1(d), for this call it holds $m = \textsc{lce}(\ell, k) = \textsc{lce}(k, r)$, where $k \in (\ell, r)$ with $\mathsf{prev}[k] = \ell$ and $\mathsf{next}[k] = r$. For every $h \in (\ell, r)$, the definition of prev and next implies that $x_h \succeq x_k \succ x_r$. This also means that $m = \textsc{lce}(k, r) \geq \textsc{lce}(h, r)$. If $r = r_c$ then, because we already computed $\textsc{lce}(\ell_c, r)$, and due to the iteration order of the algorithm, it holds $\ell < \ell_c$. Then, however, $\ell_c \in (\ell, r)$ and thus $m = \textsc{lce}(k, r) \geq \textsc{lce}(\ell_c, r) = c - r$, which contradicts $r + m < c$. We have shown that $r > r_c$, which also implies $r_* > \ell_c$. If $\ell < r_*$ then $r_* \in (\ell, r)$ and thus $m \geq \textsc{lce}(r_*, r) = c - r$, which contradicts $r + m < c$. It follows that $\ell \geq r_* > \ell_c$. Finally, if $\ell \in (\ell_c, r_c)$ then $\ell_c < \ell < r_c < r$, which contradicts Lemma 5. The only remaining possibility is $\ell \geq r_c$, which is what we wanted to show.

Showing Property (ii). The property states that either $\mathsf{prev}[r_*] = \ell_*$, or $\mathsf{next}[\ell_*] = r_*$. By the definition of ℓ_* and r_*, and due to (i) and $x[\ell_c..\ell_c + \textsc{lce}(\ell_c, r_c)) = x[r_c..r_c + \textsc{lce}(\ell_c, r_c))$, it holds $x[\ell..r) = x[\ell_*..r_*)$. Since we want to compute $\textsc{lce}(\ell, r)$, it holds either $\mathsf{next}[\ell] = r$ or $\mathsf{prev}[r] = \ell$. Therefore, either Lemma 3 or Lemma 4 implies that $x[\ell..r) = x[\ell_*..r_*)$ is a Lyndon word. Due to Lemma 3, we know that $\mathsf{next}[\ell_*] \geq r_*$. If $\mathsf{next}[\ell_*] = r_*$ or $\mathsf{prev}[r_*] = \ell_*$, then there is nothing left to show. Thus, assume that $\mathsf{next}[\ell_*] > r_*$ and $\mathsf{prev}[r_*] > \ell_*$ (it cannot be that $\mathsf{prev}[r_*] < \ell_*$ because then $\mathsf{prev}[r_*] < \ell_* < r_* < \mathsf{next}[\ell_*]$ contradicts Lemma 5). Let $p_r = r - (r_* - \mathsf{prev}[r_*])$. Due to Lemma 4, the substring $x[\mathsf{prev}[r_*]..r_*) = x[p_r..r)$ is a Lyndon word, and Lemma 3 implies $\mathsf{next}[p_r] \geq r$. Since $p_r \in (\ell, r)$ it holds $\mathsf{next}[p_r] \leq r$ (otherwise we contradict Lemma 5), and the only possible option is $\mathsf{next}[p_r] = r$.

We have shown that $\mathsf{next}[p_r] = r$ and thus $x_{p_r} \succ x_r$. By the definition of prev, it also holds $x_{\mathsf{prev}[r_*]} \prec x_{r_*}$. Since we chose r_c to be the rightmost index with $r_c + \textsc{lce}(\ell_c, r_c) = c$, it holds $r + \textsc{lce}(p_r, r) < c$ (otherwise we would have updated r_c already). Therefore, we have

$$\begin{aligned} x[\mathsf{prev}[r_*]..\mathsf{prev}[r_*] + \textsc{lce}(p_r, r)] &= x[p_r..p_r + \textsc{lce}(p_r, r)], \text{ and} \\ x[r_*..r_* + \textsc{lce}(p_r, r)] &= x[r..r + \textsc{lce}(p_r, r)]. \end{aligned}$$

This, however, means that $x_{\mathsf{prev}[r_*]} \prec x_{r_*} \iff x_{p_r} \prec x_r$, which contradicts our previous observation that $x_{p_r} \succ x_r$ and $x_{\mathsf{prev}[r_*]} \prec x_{r_*}$. It follows that the assumption $\mathsf{next}[\ell_*] > r_*$ and $\mathsf{prev}[r_*] > \ell_*$ was wrong, and it holds $\mathsf{next}[\ell_*] = r_*$ or $\mathsf{prev}[r_*] = \ell_*$.

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 $\mathsf{rad}[1..n]$, where $\forall i \in [1, n]$:

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

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: rad[1..n] ← new array initialized
              with rad[1] = rad[n] = 1

2: global variable $c \leftarrow 0$
3: global variable $j \leftarrow 0$

4: for $i = 2$ to $n - 1$ do
5:     rad[$i$] ← SMART-PALINDROME($i, 1$)
```

```
1: function SMART-PALINDROME($i, m$)
2:     if $i + m < c$ then
3:         $m \leftarrow$ rad[$j - (i - j)$]
4:         if $i + m < c$ then return $m$
5:         $m \leftarrow c - i$

6:     while $x[i - m] = x[i + m]$ do
7:         $m \leftarrow m + 1$

8:     $c, j \leftarrow i + m, i$
9:     return $m$
```

deduce an entry, see Figures 3b and 3d. This is highly similar to our observations for LCEs in Figures 3a and 3c. A possible implementation of Manacher's algorithm is provided in Algorithm 3. It computes rad from left to right, while keeping track of the rightmost inspected position of the string. Whenever possible, the function SMART-PALINDROME partially or fully deduces rad$[i]$. Note that the functions SMART-LCE and SMART-PALINDROME are structurally identical and use the same algorithmic ideas. Due to space constraints, we omit further details on how and why Algorithm 3 functions as intended, and why it takes $\mathcal{O}(n)$ time. (However, we invite the reader to produce a proof of correctness on their own. This is much easier for Algorithm 3 than for Algorithm 1(d) and Algorithm 2. Particularly, it requires no complicated technicalities like properties (i) and (ii) in Section 5.)

7 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 $x_{\ell_2} \succ x_{r_1}$. 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 $x_{\ell_2} \prec x_{r_1}$, 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$[r] =$ prev$^*[r - 1]$.
(ii) *It holds* next$[\ell] = r$ *if and only if $\ell =$* prev$^*[r - 1]$ *and $\ell >$* prev$[r]$.

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

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

For the counter direction, assume that prev$[r] < \ell$ and $\ell =$ prev$^*[r - 1]$. By the definition of prev, and due to $\ell \in ($prev$[r], r)$, it holds $x_\ell \succ x_r$. It is easy to see that $\ell =$ prev$^*[r - 1]$ implies $\forall k \in (\ell, r) : x_k \succ x_\ell$ (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 $x_\ell \succ x_r$ and $\forall k \in (\ell, r) : x_k \succ x_\ell$ follows next$[\ell] = r$. ◀

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

(i) *If $\textsc{lce}(\ell, k) = \textsc{lce}(k, r)$, then $\textsc{lce}(\ell, r) \geq \textsc{lce}(k, r)$ and either $\mathsf{prev}[r] = \ell$ or $\mathsf{next}[\ell] = r$.*
(ii) *If $\textsc{lce}(\ell, k) < \textsc{lce}(k, r)$, then $\textsc{lce}(\ell, r) = \textsc{lce}(\ell, k)$ and $\mathsf{prev}[r] = \ell$.*
(iii) *If $\textsc{lce}(\ell, k) > \textsc{lce}(k, r)$, then $\textsc{lce}(\ell, r) = \textsc{lce}(k, r)$ and $\mathsf{next}[\ell] = r$.*

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

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

References

1 Golnaz Badkobeh, Maxime Crochemore, Jonas Ellert, and Cyril Nicaud. Back-to-front online Lyndon forest construction. In *Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, pages 13:1–13:23, Prague, Czech Republic, June 2022. doi:10.4230/LIPIcs.CPM.2022.13.

2 Uwe Baier. Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 23:1–23:12, Tel Aviv, Israel, June 2016. doi:10.4230/LIPIcs.CPM.2016.23.

3 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

4 Nico Bertram, Jonas Ellert, and Johannes Fischer. Lyndon words accelerate suffix sorting. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021)*, pages 15:1–15:13, Lisbon, Portugal (Virtual Conference), September 2021. doi:10.4230/LIPIcs.ESA.2021.15.

5 Philip Bille, Jonas Ellert, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. Space efficient construction of Lyndon arrays in linear time. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, pages 14:1–14:18, Saarbrücken, Germany (Virtual Conference), July 2020. doi:10.4230/LIPIcs.ICALP.2020.14.

6 Maxime Crochemore, Thierry Lecroq, and Wojciech Rytter. *125 Problems in Text Algorithms*. Cambridge University Press, 2021. 334 pages.

7 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

8 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, pages 63:1–63:16, Glasgow, Scotland (Virtual Conference), July 2021. doi:10.4230/LIPIcs.ICALP.2021.63.

9 Frantisek Franek, A. S. M. Sohidull Islam, Mohammad Sohel Rahman, and William F. Smyth. Algorithms to compute the Lyndon array. In *Proceedings of the Prague Stringology Conference 2016*, pages 172–184, Prague, Czech Republic, August 2016. URL: http://www.stringology.org/event/2016/p15.html.

10 Frantisek Franek and Michael Liut. Algorithms to compute the Lyndon array revisited. In *Proceedings of the Prague Stringology Conference 2019*, pages 16–28, Prague, Czech Republic, 2019. URL: http://www.stringology.org/event/2019/p03.html.

11 Frantisek Franek and Michael Liut. Computing maximal Lyndon substrings of a string. *Algorithms*, 13(11), 2020. doi:10.3390/a13110294.

12 Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003. doi:10.1016/S0304-3975(03)00099-9.

13 Felipe A. Louza, Sabrina Mantaci, Giovanni Manzini, Marinella Sciortino, and Guilherme P. Telles. Inducing the Lyndon array. In *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019)*, pages 138–151, Segovia, Spain, October 2019. doi:10.1007/978-3-030-32686-9_10.

14 Glenn Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.

15 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the First Annual Symposium on Discrete Algorithms (SODA 1990)*, pages 319–327, San Francisco, California, USA, January 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

16 Kazuya Tsuruta, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar-compressed self-index with Lyndon words. *CoRR*, abs/2004.05309, 2020. arXiv:2004.05309.