Lyndon Arrays in Sublinear Time

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

Jonas Ellert
Technical University of Dortmund, Germany

Abstract

A Lyndon word is a string that is lexicographically smaller than all of its non-trivial suffixes. For example, `airbus` is a Lyndon word, but `amtrak` is not a Lyndon word due to its suffix `ak`. The Lyndon array stores the length of the longest Lyndon prefix of each suffix of a string. For a length-n string over a general ordered alphabet, the array can be computed in $\mathcal{O}(n)$ time (Bille et al., ICALP 2020). However, on a word-RAM of word-width $w \geq \log_2 n$, linear time is not optimal if the string is over integer alphabet $\{0, \dots, \sigma\}$ with $\sigma \ll n$. In this case, the string can be stored in $\mathcal{O}(n \log \sigma)$ bits (or $\mathcal{O}(n / \log_\sigma n)$ words) of memory, and reading it takes only $\mathcal{O}(n / \log_\sigma n)$ time. We show that $\mathcal{O}(n / \log_\sigma n)$ time and words of space suffice to compute the succinct $2n$-bit version of the Lyndon array. The time is optimal for $w = \mathcal{O}(\log n)$. The algorithm uses precomputed lookup tables to perform significant parts of the computation in constant time. This is possible due to properties of periodic substrings, which we carefully analyze to achieve the desired result. We envision that the algorithm has applications in the computation of runs (maximal periodic substrings), where the Lyndon array plays a central role in both theoretically and practically fast algorithms.

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

Keywords and phrases Lyndon forest, Lyndon table, Lyndon array, sublinear time algorithms, word RAM algorithms, word packing, tabulation, lookup tables, periodicity

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.14

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP20H04141

1 Introduction

A Lyndon word is a string that is lexicographically smaller than all of its non-trivial suffixes. For example, `airbus` is a Lyndon word, but `amtrak` is not a Lyndon word due to its suffix `ak`. The Lyndon array stores the length of the longest Lyndon prefix of each suffix of a string (a precise definition follows later). In this article, we propose a new algorithm that computes the (succinct version of) the Lyndon array of a length-n string over alphabet $\{0, \dots, \sigma\}$ in $\mathcal{O}(n / \log_\sigma n)$ time and words of space on a word-RAM of word-width $w \geq \log_2 n$.

Background and Applications. Since their introduction in the field of combinatorics on words almost 70 years ago [32], Lyndon words have proven to be useful for designing efficient algorithms. The Lyndon factorization of a string uniquely decomposes it into lexicographically non-increasing Lyndon words [9, 15]. It can easily be obtained from the Lyndon array, and it has recently been used to capture overlaps between reads for next generation DNA sequencing [7, 8]. The closely related standard factorization $T = RS$ of a Lyndon word T is uniquely defined by S, which is the longest proper Lyndon suffix, or equivalently the lexicographically smallest proper suffix of T. It is guaranteed that R is also a Lyndon word [9]. By recursively factorizing R and S in the same manner until all segments are single symbols, we obtain the binary Lyndon tree. This tree can also be defined for a non-Lyndon word T, since prepending an infinitely small symbol makes any word Lyndon. The Lyndon tree encodes the same information as the Lyndon array (see, e.g., [13, 2]).

Hohlweg and Reutenauer [24] showed that the Lyndon factorization encodes the list of left-to-right suffix (lexicographical) minima of a string. Crochemore and Russo [13] showed that the Lyndon array (respectively the Lyndon tree) of a string encodes the left-to-right minima tree (respectively the Cartesian tree) of its inverse suffix array (the array that stores for each suffix its lexicographical rank among all the suffixes). This shows the close relation between the Lyndon array and the suffix array [33], one of the major data structures in string algorithmics, with countless applications in compression and indexing. Lyndon words and the Lyndon array have been used to efficiently compute the suffix array (e.g., [3, 5, 38]) and vice versa (e.g., [21, 31]). The Lyndon array can be computed from the suffix array in $\mathcal{O}(n)$ time; however, this requires a linearly-sortable alphabet (e.g., $\Sigma = \{0, \dots, n^{\mathcal{O}(1)}\}$) due to the information-theoretic lower bound on comparison sorting. For general ordered alphabets, there is an $\mathcal{O}(n)$ time algorithm that computes the Lyndon array without the suffix array by exploiting combinatorial properties of Lyndon words [6] (see [16] for a simplified description). This algorithm can also be used to output a succinct $2n$-bit encoding of the Lyndon array, which is based on a balanced parentheses sequence of the tree structure of the Lyndon array.

The perhaps most important application of the Lyndon array is the computation of runs (maximal periodic substrings). Kolpakov and Kucherov showed that there are $\mathcal{O}(n)$ runs in a length-n string, and conjectured that this upper bound can be improved to n [29]. A series of results gradually improved the best known bound [40, 10, 11, 39]. Ultimately, the Lyndon array and its rich combinatorial properties were used to prove the conjecture [4], which also resulted in a remarkably simple proof. The Lyndon array is also one of the two main ingredients of a simple $\mathcal{O}(n)$ time algorithm for computing all the runs (see, e.g., [4, 13]). The second ingredient is a data structure for longest common extensions (LCEs), which answers queries of the type "Given i, j, what is the longest shared prefix between $T[i..n]$ and $T[j..n]$?". For linearly-sortable alphabets, an LCE data structure with constant query time can be constructed in $\mathcal{O}(n)$ time (e.g., [20]), which results in an $\mathcal{O}(n)$ time runs algorithm. It was conjectured that the same time can be achieved for general ordered alphabets [30], which resulted in a series of new LCE data structures aimed at these alphabets [30, 22, 12]. The first algorithm that achieves linear time [17] and hence proves the conjecture does not use an LCE data structure at all. Instead, it relies on the combinatorial structure of the Lyndon array to explicitly compute all the LCEs in overall linear time.

The existing $\mathcal{O}(n)$ time algorithms for the Lyndon array are optimal if the string is over a general ordered alphabet. However, they are not optimal on a word-RAM of word-width $w \geq \log_2 n$ if the alphabet is $\{0, \dots, \sigma\}$ with $\sigma \ll n$. In this case, $\lfloor \log_\sigma n \rfloor$ symbols can be packed in a single word of memory, and hence they can be processed simultaneously. Word packing has lead to faster algorithms for many problems in string algorithmics. For example, $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time suffices to construct any of the following data structures for a packed string: compressed suffix arrays and trees with $\mathcal{O}(\log^\epsilon n)$ time operations [27]; an index that counts occurrences of a length-m query pattern in $\mathcal{O}(m/\log_\sigma n + \log^\epsilon n)$ time [27]; the Burrows–Wheeler transform [26]; the wavelet tree [1, 35] with a fast practical implementation [25]. A novel LCE data structure by Kempa and Kociumaka can be constructed in even faster $\mathcal{O}(n/\log_\sigma n)$ time and answers queries in $\mathcal{O}(1)$ time [26].

Our Contributions. We show that the succinct $2n$ bit representation of the Lyndon array can be computed in $\mathcal{O}(n/\log_\sigma n)$ time and words of space on a word RAM of word-width $w \geq \log_2 n$. The time bound is optimal under the common assumption that the input size scales with the word-width, i.e., $w = \mathcal{O}(\log n)$. The algorithm uses the same ideas as previous $\mathcal{O}(n)$ time algorithms, but processes the string one word (rather than one position) at a time.

We accelerate the computation with lookup tables and the LCE data structure by Kempa and Kociumaka [26]. By carefully analyzing properties of periodic substrings, we are able to design the lookup tables in a way that minimizes the number of required LCE queries. The new algorithm will hopefully lead to a sublinear time algorithm for the computation of runs.

The remainder of the paper is structured as follows. In Section 2, we introduce basic definitions and notation, as well as a simple linear time algorithm for the succinct Lyndon array. This algorithm is the starting point of our new sublinear time algorithm, which we present in Section 3. It uses some auxiliary data structures, which we first use as a black box. In Sections 4 and 5, we show how to implement these data structures.

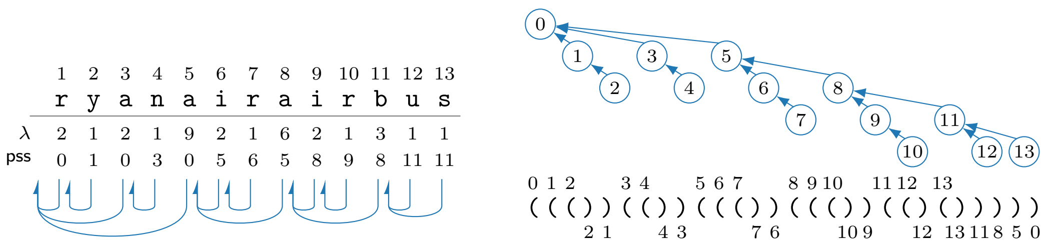
2 Preliminaries

Strings and Computational Model. For $i, j \in \mathbb{N}$, we write $[i, j] = (i-1, j] = [i, j+1) = (i-1, j+1)$ to denote the integer interval $\{i, i+1, \ldots, j\}$ (or the empty set if $i > j$). A string $T \in \Sigma^n$ is a sequence of $|T| = n$ symbols from some alphabet Σ. The empty string of length 0 is denoted by ε. For $i, j \in [1, n]$, we denote the i^{th} symbol of the sequence by $T[i]$. The substring $T[i..j] = T(i-1..j] = T[i..j+1) = T(i-1..j+1)$ is the sequence of length $j - i + 1$ that starts with the i^{th} and ends with the j^{th} symbol. For $j < i$ we define $T[i..j] = \varepsilon$. If $i \leq j$, then the longest common extension (LCE) at positions i and j is defined as $\textsc{lce}(i, j) = \textsc{lce}(j, i) = \max(\{\ell \in [0, n-j+1] \mid T[i..i+\ell) = T[j..j+\ell)\})$. Substrings $T[1..i]$ and $T[i..n]$ are respectively called prefix and suffix of T. A substring $T[i..j]$ is proper if $T[i..j] \neq T$, and non-trivial if it is proper and $T[i..j] \neq \varepsilon$. If the alphabet Σ is totally ordered, then it induces a lexicographical order as follows. For strings $S \in \Sigma^m$ and $T \in \Sigma^n$, it holds $S \prec T$ (and we say that S is lexicographically smaller than T) if and only if either S is a prefix of T, or there is some $\ell \in [1, \min(m, n)]$ such that $S[1..\ell) = T[1..\ell)$ and $S[\ell] < T[\ell]$. We write $S \preceq T$ to denote that T is not lexicographically smaller than S. A string $T[1..n]$ has period $p \in [1, n]$ if $T[1..n-p] = T[1+p..n]$. We then call $T[1..n-p]$ a border of T. The concatenation of two string S and T is denoted by ST. For $k \in \mathbb{N}^0$, the k-times concatenation (or k-power) of T is denoted by T^k (with $T^0 = \varepsilon$).

We work on a word RAM (see, e.g., [23]) with words of width $w \geq \log_2 n$ bits (where n is the length of the input string). A string $T \in [0, \sigma)^n$ can be stored in packed representation, i.e., the binary representation of each symbol is stored in $\lceil \log_2 \sigma \rceil$ bits, and the entire string occupies $n \lceil \log_2 \sigma \rceil$ consecutive bits or $\mathcal{O}(n / \log_\sigma n)$ words of memory. (The number of bits can be improved to $\lceil n \log_2 \sigma \rceil + \mathcal{O}(\log^2 n)$ while retaining fast access [14, Theorem 1].) Primitive bitwise operations suffice to extract any substring of length $\mathcal{O}(\log_\sigma n)$ in constant time because such a substring fits in a constant number of words. Since a (sub-)string S is a bit string of length $|S| \cdot \lceil \log_2 \sigma \rceil$, it can be interpreted as an integer in range $[1, 2^{|S| \cdot \lceil \log_2 \sigma \rceil}]$ (by reading the bit string as a binary number and adding one). We write $\mathtt{int}(S)$ to denote the integer value associated with S (we also use this notations for bitvectors, as they are merely strings with $\sigma = 2$). In this model of computation, a data structure by Kempa and Kociumaka can be constructed in sublinear time and answers LCE queries (i.e., outputs the LCE of any two positions) in constant time.

▶ **Lemma 1** ([26, Theorem 5.4]). *Given a string $T \in [0, \sigma)^n$ in packed representation, LCE queries can be answered in $\mathcal{O}(1)$ time after an $\mathcal{O}(n / \log_\sigma n)$ time preprocessing.*

We do not use uninitialized memory or similar techniques, and hence the words of space used by the algorithm is upper bounded by the time spent (this includes Lemma 1). Therefore, we will not discuss any space complexities in the remainder of the paper, and instead only show the $\mathcal{O}(n / \log_\sigma n)$ time bound, which implies that $\mathcal{O}(n / \log_\sigma n)$ words of space suffice.



(a) Lyndon array λ and PSS array pss. **(b)** The previous smaller suffix tree and its BPS.

Figure 1 Data structures for the string `ryanairairbus`. Edges point to previous smaller suffixes.

Lyndon Arrays. A Lyndon word is a non-empty string $T[1..n]$ over a totally ordered alphabet that is lexicographically smaller than its non-trivial suffixes, i.e., $\forall i \in [2, n] : T \prec T[i..n]$.

▶ **Definition 2.** *Let $T[1..n]$ be a string over a totally ordered alphabet. The Lyndon array $\lambda[1..n]$ and the previous smaller suffix (PSS) array* $\mathsf{pss}[1..n]$ *are defined $\forall i \in [1, n]$ by*

- $\lambda[i] = \max(\{\ell \in [1, n - i + 1] \mid T[i..i + \ell)$ *is a Lyndon word*$\})$*, and*
- $\mathsf{pss}[i] = \max(\{j \in [1, i) \mid T[j..n] \prec T[i..n]\} \cup \{0\})$.

An example is provided in Figure 1a. We may interpret the PSS array as a rooted tree. The root is an artificial node with label 0. Every text position is a node, and for any position i, there is an edge from i to its parent $\mathsf{pss}[i]$. Each position is a child of a smaller position or the artificial root node, and hence it is easy to see that this indeed yields a tree with root 0. An example of this so-called *previous smaller suffix tree (PSS tree)* is provided in Figure 1b.

A balanced parentheses sequence (BPS) encodes the tree in $2n + 2$ bits (see, e.g., [36]), which can be described in the following constructive way. We write the sequence in an append-only manner. We perform a depth-first traversal of the tree, during which we visit the children of each node in increasing order. Whenever we walk down an edge from $\mathsf{pss}[i]$ to i, we append the opening parenthesis of node i. Whenever we walk up an edge from i to $\mathsf{pss}[i]$, we write the closing parenthesis of node i. An example is provided in Figure 1b. If we assign preorder-numbers during this traversal, then node i has preorder-number i (see [19, Lemma 1] and [6]). Hence the i^{th} opening parenthesis belongs to node i. In practice, the parentheses sequence is a bitvector, where 1-bits are opening parentheses, and 0-bits are closing ones. Bille et al. [6] showed that the PSS tree inherently encodes the Lyndon array because $\lambda[i]$ is exactly the size of the subtree rooted in node i. Hence we need to augment the parentheses sequence such that $\mathsf{subtree\text{-}size}(i)$ can be answered in constant time. There are multiple support data structures that achieve this with $o(n)$ bits of additional space (see, e.g., [37]), but their current construction algorithms do not achieve sublinear time. However, we only require a small and simple subset of operations supported by these data structures (namely rank, select, and find-close). We plan to cover the efficient construction of support data structures for these operations in a future full version of the paper.

Simple Construction Algorithm for the PSS Tree. A simple algorithm computes the PSS tree in $\mathcal{O}(n)$ time, which will be the starting point for the $\mathcal{O}(n/\log_\sigma n)$ time algorithm in Section 3. Suppose that we have already computed the subtree induced by nodes $[0, i)$. Attaching node i requires finding $\mathsf{pss}[i]$. A strategy for this follows from the property that

(a) $\mathsf{pss}[i]$ is one of the nodes on the already computed path from $i - 1$ to the root 0, and
(b) on this path, $\mathsf{pss}[i]$ is the deepest node j such that either $j = 0$ or $T[j..n] \prec T[i..n]$.

This observation has been used by previous results, and it has been proven, e.g., in [16, Lemma 6]. It implies an algorithm in which the positions are inserted into the tree one by one in left-to-right order. This is also the approach of Algorithm 1, which directly computes the BPS of the tree (see [6, Algorithm 1] for a more abstract version of this algorithm).

Algorithm 1 Linear time construction of the PSS tree.

```
Require: Packed string $T \in [0, \sigma)^n$.
Ensure: BPS $\mathcal{B}$ of the PSS tree of $T$.
1:  $\mathcal{B} \leftarrow$ (                                  ▷ opening parenthesis of node 0
2:  $\mathcal{Q} \leftarrow$ stack that contains only 0
3:  for $i = 1$ to $n$ do
4:      $j \leftarrow \mathcal{Q}.top()$
5:      while $j > 0$ and $T[i..n] \prec T[j..n]$ do     ▷ evaluate with LCE data structure
6:          append ) to $\mathcal{B}$                        ▷ closing parenthesis of node $j$
7:          $\mathcal{Q}.pop()$
8:          $j \leftarrow \mathcal{Q}.top()$
9:      append ( to $\mathcal{B}$                            ▷ opening parenthesis of node $i$
10:     $\mathcal{Q}.push(i)$
11: append $|\mathcal{Q}|$ times ) to $\mathcal{B}$          ▷ closing parentheses of nodes on path from $n$ to 0
```

At the time at which the algorithm starts processing position i, the sequence $\mathcal{B}$ contains the prefix of the BPS that ends with the opening parenthesis of node $i-1$, and the stack $\mathcal{Q}$ contains exactly the nodes on the path from $i-1$ (topmost stack element) to the root 0 (bottommost stack element). A loop is used to find the topmost element j on the stack that satisfies $j = 0$ or $T[j..n] \prec T[i..n]$ (lines 4–8). By properties (a) and (b), the final value of j is the previous smaller suffix of i, which means that node i will be attached as a child of j. Hence we pop the nodes on the path from $i-1$ to j (but excluding j) from the stack, and then push i on the stack (lines 7 and 10). As explained earlier, the BPS encodes a depth-first traversal of this tree. In terms of this traversal, we just moved from node $i-1$ up to node j, and then down to node i. Thus, we write one closing parenthesis for each step up (line 6), and then one opening parenthesis for moving down to node i (line 9). After processing position n, we write the closing parentheses of the nodes on the path from n to 0 (line 11).

The correctness follows from properties (a) and (b). Each line takes constant time, except for the lexicographical suffix comparison in line 5. It holds $T[i..n] \prec T[j..n]$ if and only if either $\textsc{lce}(i,j) = n-i+1$ or $T[i+\textsc{lce}(i,j)] < T[j+\textsc{lce}(i,j)]$. Thus, the LCE data structure from Lemma 1 suffices to lexicographically compare suffixes in constant time (we use this technique repeatedly throughout the paper). The number of inner loop iterations is less than the number of closing parentheses, and hence the total time needed by the algorithm is $\mathcal{O}(n)$.

3 A Blockwise Algorithm for the PSS Tree

In this section, we modify Algorithm 1 such that instead of processing a single index at a time, it processes blocks of indices in each step. The block size $k = \left\lfloor \frac{\log_2 n}{8\lceil \log_2 \sigma \rceil} \right\rfloor$ is approximately one eighth of the number of symbols that fit into one word of memory, and hence there are $N = \lceil \frac{n}{k} \rceil = \Theta(n/\log_\sigma n)$ blocks. Let $B_1, \dots, B_N$ with $\forall x \in [1, N] : B_x = (xk-k, xk]$ be the sequence of blocks (where without loss of generality we assume that k divides n).

In the PSS tree, each block B_x induces a forest that contains exactly the nodes that are members of the block. For any node $j \in B_x$, if $\mathsf{pss}[j] \in B_x$, then $\mathsf{pss}[j]$ is the parent of j in the forest induced by B_x. Otherwise, j is the root of a tree in the forest. We call these trees *small trees*, and their roots *small roots*. The small roots are exactly the left-to-right lexicographical minima of suffixes starting in B_x, i.e., $i \in B_x$ is a root if and only if $\forall i' \in B_x : i' < i \implies T[i'..n] \succ T[i..n]$. Just like in the PSS tree, we arrange the children of each node in increasing order. The BPS of the forest is the concatenation of the BPSs of its small trees in left-to-right order.

High-Level Description of the Blockwise Algorithm. We process the blocks one at a time in left-to-right order. At the time at which we process block B_x, we have already computed the partial PSS tree induced by all previous blocks, i.e., by $[0, xk - k]$. For B_x, we first obtain its induced PSS forest. Our goal is to attach the small roots (including their small trees) to the respective previous smaller suffixes, which lie on the path from $xk - k$ to 0 in the partial PSS tree. This is schematically shown in Figure 2a. Note that small roots further to the right will be attached further up in the path. This is because suffixes on the path are lexicographically decreasing towards the root, while the suffixes corresponding to small roots are lexicographically decreasing from left to right. Hence our task is to lexicographically *interleave* the path with the small roots. For an efficient implementation of this interleaving process, it is crucial that we maintain the path from $xk - k$ to 0 in a blockwise manner. Just like in Algorithm 1, we maintain a stack of the nodes on the path. However, each stack element is a pair $(y, \mathcal{L})$, where y indicates that we consider block B_y, and $\mathcal{L}[1..k]$ is a bitvector indicating which of the positions in the block are relevant. For $j' \in [1, k]$, it holds $\mathcal{L}[j'] = 1$ if and only if $yk - k + j'$ lies on the path from $xk - k$ to 0. The stack then contains exactly the blocks with at least one 1-bit in the bitvector. This is visualized in Figure 2a.

3.1 Detailed Description of the Blockwise Algorithm

So far, we described the algorithm in terms of the PSS tree. However, we want to directly compute its BPS. After $\mathcal{O}(N)$ preprocessing time, we can obtain the BPS of the forest induced by any block in $\mathcal{O}(1)$ time. This follows directly from the lemma below.

▶ **Lemma 3.** *Let $T \in [0, \sigma)^n$ be a string in packed representation and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, the following type of query can be answered in $\mathcal{O}(1)$ time. Given a range $[i, i + \ell) \subseteq [1, n]$ of length $\ell \leq \frac{\log_2 n}{(2+\epsilon)\lceil \log_2 \sigma \rceil}$, output the BPS of the PSS forest induced by $[i, i + \ell)$, as well as a bitvector $\mathcal{R}[1..\ell]$ such that for $j \in [1, \ell]$ it holds $\mathcal{R}[j] = 1$ if and only if $i + j - 1$ is the root of a tree in the forest.*

The proof of the lemma is provided in Section 4. When we lexicographically interleave the suffixes, we will repeatedly encounter another type of query. Given a small root, we have to find its previous smaller suffix within a block on the stack. A solution for this is provided by the lemma below, which we prove in Section 5.

▶ **Lemma 4.** *Let $T \in [0, \sigma)^n$ be a string in packed representation and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, we can answer the following type of query in $\mathcal{O}(1)$ time. Given a position $i \in [1, n]$ and a non-empty interval $[j, j+\ell) \subseteq [1, n]$ of length $\ell \leq \frac{\log_2 n}{(5+\epsilon)\cdot\lceil \log_2 \sigma \rceil}$, find the position $j_{\max} = \max(\{j' \in [j, j + \ell) \mid T[j'..n] \prec T[i..n]\} \cup \{j - 1\})$.*

Now we have all the tools needed to describe the algorithm. We start with an empty stack $\mathcal{Q}$ and $\mathcal{B} =$ (, i.e., with the opening parenthesis of the artificial root node 0. Now we process the blocks $B_1, \ldots, B_N$ in left-to-right order. At the time at which we start processing

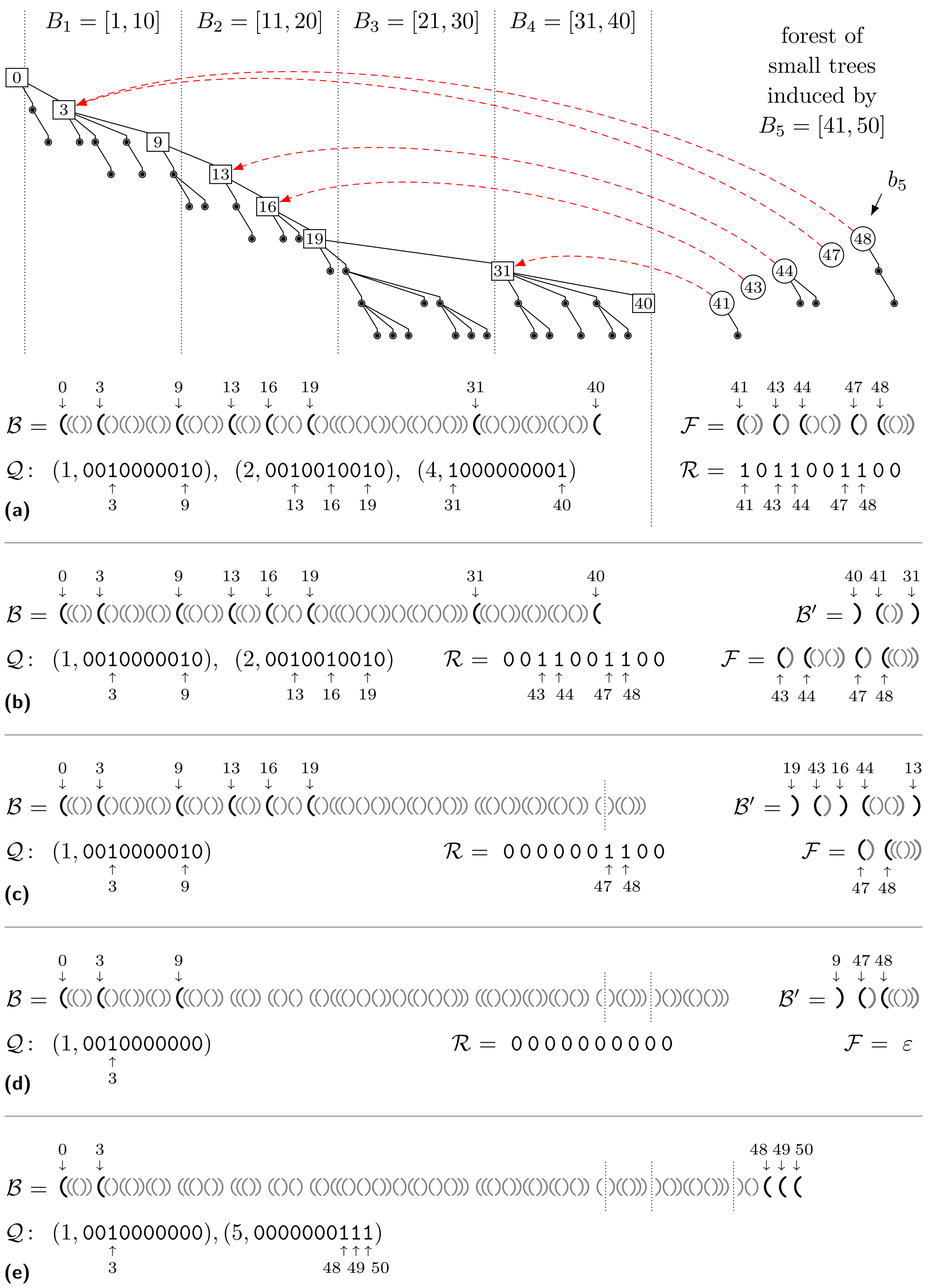


Figure 2 Data structures during the execution of the blockwise algorithm with block size $k = 10$. While processing $B_5 = [41..50]$, the dashed edges will be inserted into the partial PSS tree induced by $[0..40]$. The drawings show the state of the relevant data structures before calling the subroutine during the first iteration of the main routine (a), and after calling the subroutine in the first (b), second (c), and third (d) iteration of the main routine. The state after finalizing B_5 is shown in (e).

B_x, the stack $\mathcal{Q}$ contains the nodes on the path from $xk-k$ to 0 in the blockwise manner described above, and $\mathcal{B}$ contains the prefix of the BPS of the PSS tree that ends with the opening parenthesis of node $xk-k$. We start by querying Lemma 3 with B_x and obtain the BPS $\mathcal{F}$ of the forest induced by B_x, as well as the bitvector $\mathcal{R}$ indicating the small roots. We then find the rightmost 1-bit in $\mathcal{R}$ in constant time (there are only $2^k = \mathcal{O}(\sqrt[8]{n})$ possible values of $\mathcal{R}$, hence a lookup table for rightmost or leftmost 1-bits can be precomputed in $o(n/\log n)$ time). If this bit is at position $\mathcal{R}[b'_x]$, then $b_x = b'_x + xk - k$ is the rightmost small root in the forest induced by B_x. Note that $T[b_x..n]$ is the lexicographically smallest suffix starting in B_x. The state after this step is visualized in Figure 2a. Now we repeatedly run the interleaving main routine described below, during which we will alter $\mathcal{F}$, $\mathcal{R}$, $\mathcal{Q}$, and $\mathcal{B}$.

Main Routine. The goal of this routine is to interleave (the remaining small trees of) B_x with the topmost block on the stack. If $\mathcal{F}$ is empty (which happens if and only if $\mathcal{R}$ contains only zeroes), then we have attached all small trees and the main routine terminates. Otherwise, if $\mathcal{Q}$ is empty, the remaining small trees need to be attached to the root of the PSS tree, and we append $\mathcal{F}$ to $\mathcal{B}$. This takes $\mathcal{O}(1)$ time and also terminates the main routine.

If neither $\mathcal{F}$ nor $\mathcal{Q}$ are empty, then we retrieve and pop the topmost pair $(y, \mathcal{L})$ from $\mathcal{Q}$. We use Lemma 4 to obtain $b_y = \max(\{j' \in B_y \mid T[j'..n] \prec T[b_x..n]\} \cup \{yk-k\})$, and the corresponding within-block offset $b'_y = b_y - yk + k$. If $\mathsf{pss}[b_x] \in B_y$ then $b_y = \mathsf{pss}[b_x]$, and all remaining small trees have to be attached to nodes from $B_y \cap [b_y, n]$. Since b_x will be attached to b_y, none of the nodes from $B_y \cap (b_y, n]$ will remain on the stack. Hence we compute a bitvector $\mathcal{L}'[1..k]$ where for $j' \in [1,k]$ it holds $\mathcal{L}'[j'] = 1$ if and only if $\mathcal{L}[j'] = 1$ and $j' \leq b'_y$ (this takes constant time using bitwise operations). We then push $(y, \mathcal{L}')$ back onto the stack. If however $\mathsf{pss}[b_x] \notin B_y$, then $b_y = yk - k$ (the first position to the left of B_y) and b_x will be attached to a node in a block left of B_y. This means that block B_y will no longer be on the stack. Note that either way $\forall j' \in (b_y, b_x) : T[j'..n] \succ T[b_x..n]$.

Our next task is as follows. We have to attach some (possibly none, possibly all) of the remaining small trees to nodes in B_y. We reflect this change in $\mathcal{F}$ and $\mathcal{R}$ by removing the corresponding prefix of $\mathcal{F}$, and setting the corresponding bits in $\mathcal{R}$ to 0. Simultaneously, we extend $\mathcal{B}$ such that it contains the newly attached small trees, possibly interleaved with additional closing parentheses of nodes from B_y. This is realized by the following interleaving subroutine, which we run in a loop (and which will later be replaced by a single constant time table lookup). A sequence $\mathcal{B}'$ is used to buffer the parentheses that we will append to $\mathcal{B}$.

Subroutine. If either $\mathcal{L}$ or $\mathcal{R}$ consists only of 0-bits, we terminate the subroutine. Otherwise, we obtain the rightmost 1-bit of $\mathcal{L}$ (with a lookup table). If this bit is at position $\mathcal{L}[j']$, then the corresponding absolute position is $j = j' + yk - k$. If $j' = b'_y$ (which is equivalent to $j = b_y = \mathsf{pss}[b_x]$), then all remaining small trees need to be attached to j, and we append $\mathcal{F}$ to $\mathcal{B}'$. We replace $\mathcal{F}$ with ε and $\mathcal{R}$ with an all-zero bitvector. This terminates the subroutine.

Otherwise (i.e., if $j' > b'_y$ or equivalently $j > b_y$), we obtain the leftmost 1-bit of $\mathcal{R}$ (with a lookup table). If this bit is at position $\mathcal{R}[i']$, then $i = i' + xk - k$ is the leftmost small root that we still have to attach. Now we have to determine if $T[i..n] \prec T[j..n]$. (This state is equivalent to reaching the head of the inner loop of Algorithm 1 with the current values of i and j.) It holds $T[i..n] \prec T[j..n]$ if and only if $T[i..b_x) \preceq T[j..j+b_x-i)$. This is because $j + b_x - i \in (b_y, b_x)$, and hence we have already established that $T[b_x..n] \prec T[j+b_x-i..n]$. Thus, if $T[i..b_x) = T[j..j+b_x-i)$, it immediately follows that $T[i..n] \prec T[j..n]$. Note that $T[i..b_x)$ and $T[j..j+b_x-i)$ are substrings of $T(xk-k..xk]$ and $T(yk-k..yk+k]$ respectively, which will later be relevant for an efficient implementation. If $T[i..n] \prec T[j..n]$, then we append) to $\mathcal{B}'$ (this is the closing parenthesis of node j), and we assign $\mathcal{L}[j'] = 0$. If, however, $T[i..n] \succ T[j..n]$, then $\mathsf{pss}[i] = j$. In this case, we take the prefix of $\mathcal{F}$ that corresponds to the small tree rooted in i (which is the shortest balanced prefix of $\mathcal{F}$), and append it to $\mathcal{B}'$. We remove this prefix from $\mathcal{F}$ and assign $\mathcal{R}[i'] = 0$. We then continue with the next iteration of the subroutine. After the subroutine terminates, we append $\mathcal{B}'$ to $\mathcal{B}$ and continue with the next iteration of the main routine. Figures 2b–2d shows the result of the subroutine in three consecutive iterations of the main routine.

Finalizing the Block. Once the main routine terminates for block B_x, we have attached all the small trees of B_x to $\mathcal{B}$. The stack $\mathcal{Q}$ contains the blockwise representation of all the nodes on the path from xk to 0, except for the ones in block B_x. Before we can continue with the next iteration of the main routine, we have to push $(x, \mathcal{L}'')$ on the stack, where the 1-bits in $\mathcal{L}''$ correspond to the nodes on the path from xk to b_x. Note that this information can be obtained from the state of $\mathcal{F}$ at the beginning of the main routine iteration. Since $\mathcal{F}$ is a bitvector of length $2k$, a lookup table $W[1..2^{2k}]$ suffices to store the bitvector $\mathcal{L}''$ for each possible $\mathcal{F}$. The table has $\mathcal{O}(\sqrt[4]{n})$ entries and can be filled naively in $\mathcal{O}(\sqrt[4]{n} \cdot \text{polylog}(n)) \subset \mathcal{O}(n/\log n)$ time. Once we need $\mathcal{L}''$, we simply lookup $W[\mathsf{int}(\mathcal{F})]$ in constant time. Finally, in order to continue, the last written parenthesis needs to be the opening parenthesis of xk. Hence we remove the at most k trailing closing parentheses of $\mathcal{B}$ (in constant time, using another lookup table), and then continue by processing block B_{x+1}. Figure 2e shows the running example after finalizing the processed block.

After block B_N has been processed, we finish the algorithm execution by appending the $2n + 2 - |\mathcal{B}|$ closing parentheses of the nodes on the path from n to 0. This can be done in $\mathcal{O}(n/\log n)$ time by appending them one word (rather than one parenthesis) at a time.

3.2 Analyzing the Time Complexity

The initial and final processing of each block (i.e., computing $\mathcal{F}$, $\mathcal{R}$, b_x, and the pair $(x, \mathcal{L}'')$ to push on the stack) takes constant time. There are exactly N terminal iterations of the main routine, i.e., iterations where either $\mathcal{F}$ or $\mathcal{Q}$ is empty. Each terminal iteration takes constant time. In each of the non-terminal iterations, we pop a pair $(y, \mathcal{L})$ from the stack. If we do not push an updated pair $(y, \mathcal{L}')$ back onto the stack, then block B_y will never participate in the stack again, and hence this case occurs at most N times. If, however, we do push an updated pair $(y, \mathcal{L}')$ back onto the stack, then during the same main routine iteration we will also attach all remaining small trees of B_x to the partial PSS tree, which can also occur only N times. Hence the total number of iterations of the main routine is $\mathcal{O}(N)$. In each non-terminal iteration of the main routine, we call the subroutine exactly once (even though a single call may lead to multiple iterations of the subroutine). Apart from this call, each iteration of the main routine takes constant time.

It remains to be shown how to implement the subroutine such that the $\mathcal{O}(N)$ calls take $\mathcal{O}(n/\log_\sigma n)$ time in total. A straightforward naive implementation takes $\mathcal{O}(\text{poly}(k)) \subseteq \mathcal{O}(\text{polylog}(n))$ time per call. Note that the subroutine only accesses the following information: $\mathcal{L}$, b'_y, $\mathcal{R}$ (which allows access to b'_x), $\mathcal{F}$, and substrings $T_y = T(yk - k..yk + k]$ and $T_x = T(xk - k..xk]$. Bitvectors $\mathcal{L}$ and $\mathcal{R}$ are of length k bits each; b'_y is an integer from $[0, k]$ and hence can be encoded in $\lceil \log_2(k + 1)\rceil \leq \lfloor 0.99k \rfloor$ bits (for sufficiently large k); sequence $\mathcal{F}$ is of length at most $2k$ bits; strings T_y and T_x in packed representation require $2k \lceil \log_2 \sigma \rceil$ and $k \lceil \log_2 \sigma \rceil$ bits respectively. This motivates a lookup table

$$M[1..2^k][1..2^{\lfloor 0.99k \rfloor}][1..2^k][1..2^{2k}][1..2^{2k\lceil \log_2 \sigma \rceil}][1..2^{k\lceil \log_2 \sigma \rceil}].$$

In entry $M[\mathsf{int}(\mathcal{L})][b'_y][\mathsf{int}(\mathcal{R})][\mathsf{int}(\mathcal{F})][\mathsf{int}(T_y)][\mathsf{int}(T_x)]$, we store $\mathcal{B}'$ as well as the new values of $\mathcal{R}$ and $\mathcal{F}$ after running the subroutine. Note that $\mathcal{B}'$ is of length at most $3k$ because it contains at most all the parentheses from $\mathcal{F}$ and one closing parenthesis per 1-bit in $\mathcal{L}$. Hence the information stored in each table entry fits in a constant number of words, and can be retrieved in constant time. We fill the table in a lazy manner. Initially, we mark each entry as uninitialized. When accessing $M[\mathsf{int}(\mathcal{L})][b'_y][\mathsf{int}(\mathcal{R})][\mathsf{int}(\mathcal{F})][\mathsf{int}(T_y)][\mathsf{int}(T_x)]$, we check if this entry is marked. If it is, then we run the naive $\mathcal{O}(\text{polylog}(n))$ time algorithm for the subroutine, store the result in the entry, and remove its marking. Otherwise, the entry already contains the values of $\mathcal{B}'$, $\mathcal{R}$, and $\mathcal{F}$ after running the subroutine, and we return them in constant time. The lookup table has at most $2^{7.99k\lceil \log_2 \sigma \rceil} \leq 2^{\log_2 n \cdot 7.99/8} = n^{7.99/8}$ entries. Computing one entry takes $\mathcal{O}(\text{polylog}(n))$ time. Thus, the entire time spent on filling the table (i.e., on running the subroutine naively) is $\mathcal{O}(n^{7.99/8} \cdot \text{polylog}(n)) \subset \mathcal{O}(n/\log n)$. Additional $\mathcal{O}(n/\log_\sigma n)$ preprocessing time is needed for Lemmas 1, 3, and 4.

4 Proving Lemma 3

A key insight for the proof of Lemma 3 is that the lexicographical order of suffixes starting in a small range almost entirely depends on a short substring. This is formally expressed by the auxiliary lemma below.

▶ **Lemma 5.** *Let $T[1..n]$ be a string over a totally ordered alphabet, and let $[i, i+2\ell) \subseteq [1, n]$ be a non-empty interval of even length. Then at least one of the following properties holds:*

- $\forall x, y \in [i, i+\ell) : T[x..n] \prec T[y..n] \iff T[x..i+2\ell) \prec T[y..i+2\ell)$, *or*
- $\forall x, y \in [i, i+\ell) : T[x..n] \prec T[y..n] \iff T[x..i+2\ell)\# \prec T[y..i+2\ell)\#$,

  where $\#$ is an infinitely large symbol, i.e., $\forall i' \in [1, n] : T[i'] < \#$.

Proof. Let $\tilde{n} = i + 2\ell$. Assume that the lemma does not hold, then there are indices $x_1, x_2, y_1, y_2 \in [i, i+\ell)$ such that $T[x_1..n] \prec T[y_1..n]$ but $T[x_1..\tilde{n}) \succ T[y_1..\tilde{n})$, and $T[x_2..n] \prec T[y_2..n]$ but $T[x_2..\tilde{n})\# \succ T[y_2..\tilde{n})\#$. It is easy to see that this implies

$$\begin{aligned} T[y_1..\tilde{n}) &\prec T[x_1..\tilde{n}) \prec T[x_1..n] \prec T[y_1..n] = T[y_1..\tilde{n})T[\tilde{n}..n], \text{ and} \\ T[x_2..\tilde{n})\# &\succ T[y_2..\tilde{n})\# \succ T[y_2..n] \succ T[x_2..n] = T[x_2..\tilde{n})T[\tilde{n}..n]. \end{aligned}$$

Due to first condition, $T[y_1..\tilde{n})$ is a proper prefix of $T[x_1..\tilde{n})$ and it holds $x_1 < y_1$. Note that $T[y_1..\tilde{n})$ is therefore also a proper suffix (and hence a border) of $T[x_1..\tilde{n})$, and thus $T[x_1..\tilde{n})$ has period $p_1 = (y_1 - x_1)$. By the same reasoning, the second condition implies that $T[x_2..\tilde{n})$ is a border of $T[y_2..\tilde{n})$. Hence $y_2 < x_2$, and $T[y_2..\tilde{n})$ has period $p_2 = (x_2 - y_2)$. By combining these observations with the initial assumption, we obtain

$$\begin{aligned} T[y_1..\tilde{n})T[\tilde{n} - p_1..n] &= T[x_1..n] \prec T[y_1..n] = T[y_1..\tilde{n})T[\tilde{n}..n], \qquad \text{and} \\ T[x_2..\tilde{n})T[\tilde{n}..n] &= T[x_2..n] \prec T[y_2..n] = T[x_2..\tilde{n})T[\tilde{n} - p_2..n]. \end{aligned}$$

The former implies $T[\tilde{n} - p_1..n] \prec T[\tilde{n}..n]$, the latter implies $T[\tilde{n}..n] \prec T[\tilde{n} - p_2..n]$. Hence

$$T[\tilde{n} - p_1..\tilde{n})T[\tilde{n}..n] \prec T[\tilde{n}..n] \prec T[\tilde{n} - p_2..\tilde{n})T[\tilde{n}..n].$$

Since $T[\max(x_1, y_2)..\tilde{n})$ is a suffix of both $T[x_1..\tilde{n})$ and $T[y_2..\tilde{n})$, it has periods p_1 and p_2. Note that $x_1 < i + \ell - p_1$ and $y_2 < i + \ell - p_2$, and hence $T[\max(x_1, y_2)..\tilde{n})$ is of length $\tilde{n} - \max(x_1, y_2) > \tilde{n} - i - \ell + \min(p_1, p_2) = \ell + \min(p_1, p_2) > p_1 + p_2$. Therefore, it follows from the periodicity lemma [18] that $T[\max(x_1, y_2)..\tilde{n})$ has period $p_0 = \gcd(p_1, p_2)$. Since both $T[\tilde{n}-p_1..\tilde{n})$ and $T[\tilde{n}-p_2..\tilde{n})$ are suffixes of $T[\max(x_1,y_2)..\tilde{n})$, they also have period p_0. Let $\alpha = T[\tilde{n}-p_0..\tilde{n})$, $k_1 = p_1/p_0$ and $k_2 = p_2/p_0$. Then both k_1 and k_2 are positive integers, and it holds $T[\tilde{n}-p_1..\tilde{n}) = \alpha^{k_1}$ and $T[\tilde{n}-p_2..\tilde{n}) = \alpha^{k_2}$. Let k_0 be the largest integer (possibly 0) such that $T[\tilde{n}..n] = \alpha^{k_0}T[\tilde{n}+k_0p_0..n]$, and let $\beta = T[\tilde{n}+k_0p_0..n]$. Then α is not a prefix of β. The inequality above can be written as $\alpha^{k_1+k_0}\beta \prec \alpha^{k_0}\beta \prec \alpha^{k_2+k_0}\beta$. However, this is equivalent to $\alpha^{k_1}\beta \prec \beta \prec \alpha^{k_2}\beta$, which implies that α is a prefix of β. Due to this contradiction, the initial assumption must be false, and the lemma holds. ◀

Now we are ready to show Lemma 3, which is restated below.

▶ **Lemma 3.** *Let $T \in [0,\sigma)^n$ be a string in packed representation and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, the following type of query can be answered in $\mathcal{O}(1)$ time. Given a range $[i, i+\ell) \subseteq [1,n]$ of length $\ell \leq \frac{\log_2 n}{(2+\epsilon)\lceil\log_2\sigma\rceil}$, output the BPS of the PSS forest induced by $[i,i+\ell)$, as well as a bitvector $\mathcal{R}[1..\ell]$ such that for $j \in [1,\ell]$ it holds $\mathcal{R}[j] = 1$ if and only if $i+j-1$ is the root of a tree in the forest.*

Proof. The answer to any query $[i,i+\ell)$ is a parentheses sequence of length exactly 2ℓ and a bitvector of length ℓ. Hence it fits in a constant number of words. Let $\ell_{\max} = \left\lfloor \frac{\log_2 n}{(2+\epsilon)\cdot\lceil\log_2\sigma\rceil} \right\rfloor$. We precompute a 2D lookup table $E[1..2\ell_{\max}][1..\ell_{\max}]$ with the purpose of answering the subset of queries that satisfy $i + 2\ell_{\max} > n$. For any such query $[i,i+\ell)$, it holds $n-i+1 \in [1, 2\ell_{\max}]$, and we explicitly store its answer in $E[n-i+1][\ell]$. Since these queries only consider suffixes of length $\mathcal{O}(\log n)$, each of the $\mathcal{O}(\log^2 n)$ table entries can be computed naively in $\mathcal{O}(\text{polylog}(n))$ time.

We answer the remaining queries using the LCE data structure from Lemma 1 and additional lookup tables. For each possible value of ℓ, we construct tables $A_\ell[1..2^{2\ell\lceil\log_2\sigma\rceil}]$ and $B_\ell[1..2^{2\ell\lceil\log_2\sigma\rceil}]$. For every string $S \in [0,\sigma)^{2\ell}$, we store at position $A_\ell[\mathsf{int}(S)]$ the BPS of the PSS forest of S that is induced by $[1,\ell]$, as well as the bitvector that indicates the roots. At position $B_\ell[\mathsf{int}(S)]$, we store the BPS of the PSS forest of $S\#$ that is induced by $[1,\ell]$, as well as the bitvector that indicates the roots.

When answering query $[i,i+\ell)$, we first extract $T' = T[i,i+2\ell)$. Due to Lemma 5, the answer to the query is either $A_\ell[\mathsf{int}(T')]$ or $B_\ell[\mathsf{int}(T')]$. A table $C_\ell[1..2^{2\ell\lceil\log_2\sigma\rceil}]$ is used to decide which answer is correct. For every string $S \in [0,\sigma)^{2\ell}$, we store at position $C_\ell[\mathsf{int}(S)]$ an integer pair $(x,y) \in [1,\ell]^2$ such that $S[x..2\ell] \prec S[y..2\ell]$ and $S[x..2\ell]\# \succ S[y..2\ell]\#$ (or $x = y = 1$ if such a pair does not exist). This is as a witness pair of suffixes for which S and $S\#$ disagree on the lexicographical order. At query time, we lookup $(\hat{x},\hat{y}) = C_\ell[T']$. If $T[i+\hat{x}-1..n] \prec T[i+\hat{y}-1..n]$, then we return $A_\ell[\mathsf{int}(T')]$, and otherwise we return $B_\ell[\mathsf{int}(T')]$. The correctness follows from Lemma 5. Testing $T[i+\hat{x}-1..n] \prec T[i+\hat{y}-1..n]$ takes constant time with the LCE data structure from Lemma 1. Extracting T' and performing table lookups also takes constant time because T' fits in a single word of memory.

A single lookup table entry can be computed naively in $\mathcal{O}(\text{polylog}(n))$ time. There are $\mathcal{O}(\log n)$ tables, each storing at most $2^{2\ell_{\max}\lceil\log_2\sigma\rceil} \leq 2^{\log_2 n/(1+\epsilon/2)} = \sqrt[1+\epsilon/2]{n}$ entries. Thus, the precomputation of lookup tables takes $\mathcal{O}(\sqrt[1+\epsilon/2]{n}\cdot\text{polylog}(n))$ time, which is dominated by the $\mathcal{O}(n/\log_\sigma n)$ time needed to construct the LCE data structure. ◀

5 Proving Lemma 4

The proof of Lemma 4 relies on the properties of periodic substrings that are stated below.

▶ **Proposition 6.** *Let α, β, and γ be arbitrary strings. The following properties hold.*

1. *If $\alpha\beta \succ \beta$ and $\alpha\gamma \prec \gamma$ then $\beta \prec \gamma$.*
2. *If $\alpha\gamma \prec \gamma$ and α is not a prefix of γ, then $\forall x,y \in \mathbb{N}^0 : x > y \implies \alpha^x\beta \prec \alpha^y\gamma$.*

Proof. We start with (1). Let $k \in \mathbb{N}^0$ be the maximal value such that both $\beta = \alpha^k\beta'$ and $\gamma = \alpha^k\gamma'$ for some (possibly empty) strings β' and γ'. If $\alpha^{k+1}\beta' \succ \alpha^k\beta'$ and $\alpha^{k+1}\gamma' \prec \alpha^k\gamma'$, then $\beta' \prec \alpha\beta'$ and $\alpha\gamma' \prec \gamma'$. Now assume that $\gamma' \preceq \beta'$, then $\alpha\gamma' \prec \gamma' \preceq \beta' \prec \alpha\beta'$. However, this implies that α is a prefix of both β and γ, which contradicts the definition of k. Thus $\beta' \prec \gamma'$, which also implies $\beta = \alpha^k\beta' \prec \alpha^k\gamma' = \gamma$. For (2), consider any $x, y \in \mathbb{N}^0$ with $x > y$, and assume that $\alpha\gamma \prec \gamma$. Since α is not a prefix of γ, it follows from $\alpha\gamma \prec \gamma$ that $\alpha\delta \prec \gamma$ for every string δ. Hence also $\alpha^{x-y}\beta \prec \gamma$, which implies $\alpha^x\beta = \alpha^y\alpha^{x-y}\beta \prec \alpha^y\gamma$. ◀

Now we are ready to show Lemma 4, which is restated below.

▶ **Lemma 4.** *Let $T \in [0,\sigma)^n$ be a string in packed representation and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, we can answer the following type of query in $\mathcal{O}(1)$ time. Given a position $i \in [1,n]$ and a non-empty interval $[j, j+\ell) \subseteq [1,n]$ of length $\ell \leq \frac{\log_2 n}{(5+\epsilon)\cdot\lceil\log_2\sigma\rceil}$, find the position $j_{\max} = \max(\{j' \in [j, j+\ell) \mid T[j'..n] \prec T[i..n]\} \cup \{j-1\})$.*

Proof. Similarly to what was done in Lemma 3, we spend $\mathcal{O}(\text{polylog}(n))$ time to precompute the answers to all queries that satisfy $j + 3\ell \geq n$ or $i + 2\ell \geq n$. For any of the remaining queries, we consider the set

$$C = \{j' \in [j, j+\ell) \mid T[j'..j'+2\ell) = T[i..i+2\ell)\} = \{c_1, c_2, \ldots, c_h\}$$

with $c_1 < c_2 < \cdots < c_h$. This set contains exactly the positions $j' \in [j, j+\ell)$ for which we cannot easily determine whether $T[j'..n] \prec T[i..n]$ by inspecting only a small number of symbols. Hence it captures the difficult part of answering a query, and we treat it separately from the rest. We answer the query using the following subsets of $[j, j+\ell)$:

- $D' = \{j' \in C \mid T[j'..n] \prec T[i..n]\}$ (the hard subset), and
- $D'' = \{j' \in [j, j+\ell) \setminus C \mid T[j'..n] \prec T[i..n]\}$ (the easy subset).

The result of the query is $j_{\max} = \max(j'_{\max}, j''_{\max})$, where $j'_{\max} = \max(D' \cup \{j-1\})$ and $j''_{\max} = \max(D'' \cup \{j-1\})$. We start with the significantly harder task of computing $j'_{\max}$. First, we outline the algorithmic approach and the combinatorial properties of the present substrings (without giving details of an efficient implementation). Later, we describe lookup tables that achieve the claimed preprocessing and query times.

Periodicity of $T[i..i+2\ell)$ and $T[c_1..c_h+2\ell)$. We show that, if $|C| \geq 2$, then there is some p such that $T[c_1..c_h+2\ell)$ has period p, and $\forall x \in [1,h) : c_{x+1} - c_x = p$. This is similar to [34, Lemma 1] and [28, Lemma 2]. Assume that $|C| \geq 2$. For $x \in [1,h)$, let $p_x = c_{x+1} - c_x < \ell$. By design of C, it holds $T[c_x..c_x+2\ell-p_x) = T[c_{x+1}..c_{x+1}+2\ell-p_x) = T[c_x+p_x..c_x+2\ell)$. This means that $T[i..i+2\ell) = T[c_x..c_x+2\ell)$ has a border of length $2\ell - p_x$, and therefore it has period p_x. Let p be the smallest period of $T[i..i+2\ell)$. If there was some $x \in [1,h)$ such that $p_x < p$, then p would not be the smallest period of $T[i..i+2\ell)$. Hence $p_x \geq p$. Now we show that $\forall x \in [1,h) : p_x = p$. For the sake of contradiction, assume $p_x > p$. By definition of C, it holds $p_x < \ell$, which means that $T[c_x..c_x+2\ell)$ and $T[c_{x+1}..c_{x+1}+2\ell)$ overlap by $c_x + 2\ell - c_{x+1} = 2\ell - p_x > \ell > p$ symbols. Due to this overlap, and because the identical substrings $T[c_x..c_x+2\ell)$ and $T[c_{x+1}..c_{x+1}+2\ell)$ have period p, it is clear that also their union $T[c_x..c_{x+1}+2\ell)$ has period p. However, this implies $T[c_x..c_x+2\ell) = T[c_x+p..c_x+p+2\ell)$, which means that $c_x + p$ should be in C. Due to this contradiction, it holds $p_x = p$. It also follows that $T[c_1..c_h+2\ell)$ has period p.

Computing $j'_{\max}$ from c_1, c_h, and p. We will later introduce lookup tables that output c_1, c_h, and p for any query in constant time. The tables might return that c_1 and c_h do not exist (i.e., $|C| = 0$), in which case we report $j'_{\max} = j - 1$. Otherwise, it might be that $c_1 = c_h$ (i.e., $|C| = 1$). In this case, we report that $j'_{\max} = c_1$ if $T[i..n] \succ T[c_1..n]$ (using an LCE query for the comparison). Otherwise, we report $j'_{\max} = j - 1$. It remains to be shown how to compute $j'_{\max}$ if $c_1 \neq c_h$ (i.e., if $|C| \geq 2$, and the previously described periodicity exists).

We evaluate $T[i..n] \prec T[i + p..n]$ and $T[c_h..n] \prec [c_h + p..n]$ (using LCE queries). Due to the periodicity of $T[c_1..c_h + 2\ell)$, for $x \in [1, h)$ it holds $T[c_h..n] \prec T[c_h + p..n]$ if and only if

$$T[c_x..n] \;=\; T[i..i + p)^{k-x}T[c_h..n] \;\prec\; T[i..i + p)^{k-x}T[c_h + p..n] \;=\; T[c_{x+1}..n].$$

Hence either $T[c_1..n] \prec T[c_2..n] \prec \cdots \prec T[c_h..n]$ or $T[c_1..n] \succ T[c_2..n] \succ \cdots \succ T[c_h..n]$, and we already know which of the two applies. Depending on the outcome of the lexicographical comparisons, we report $j'_{\max}$ according to one of the following three cases.

Case 1: $T[c_1..n] \succ T[c_2..n] \succ \cdots \succ T[c_h..n]$.

For the computation of $j'_{\max}$, we are only interested in the rightmost $x \in [1, h]$ such that $T[i..n] \succ T[c_x..n]$. Since $T[c_h..n]$ is both rightmost and lexicographically minimal among all the possible $T[c_x..n]$, we simply use another LCE query to check if $T[i..n] \succ T[c_h..n]$. If yes, we report $j'_{\max} = c_h$. Otherwise, we report $j'_{\max} = j - 1$.

Case 2: $T[i..n] \succ T[i + p..n]$ and $T[c_1..n] \prec T[c_2..n] \prec \cdots \prec T[c_h..n]$.

Let $\alpha = T[i..i + p)$, $\beta = T[i + p..n]$, and $\gamma = T[c_2..n]$. The precondition of this case means that $\alpha\beta \succ \beta$ and $\alpha\gamma \prec \gamma$. Proposition 6.1 implies $\beta \prec \gamma$, and thus also $T[i..n] = \alpha\beta \prec \alpha\gamma = T[c_1..n] \prec T[c_2..n] \prec \cdots \prec T[c_h..n]$. Hence we report $j'_{\max} = j - 1$.

Case 3: $T[i..n] \prec T[i + p..n]$ and $T[c_1..n] \prec T[c_2..n] \prec \cdots \prec T[c_h..n]$.

Let $\alpha = T[i..i+p)$. We compute $r = \lfloor \text{LCE}(i, i + p)/p \rfloor + 1$ and $s = \lfloor \text{LCE}(c_1, c_1 + p)/p \rfloor + 1$, i.e., the respectively maximal integer powers with $T[i..n] = \alpha^r\beta$ and $T[c_1..n] = \alpha^s\gamma$, where $\beta = T[i + rp..n]$ and $\gamma = T[c_1 + sp..n]$. The precondition of this case means that $\alpha^s\gamma \prec \alpha^{s-1}\gamma$, and thus also $\alpha\gamma \prec \alpha$. Note that α is not a prefix of γ. Hence Proposition 6.2 implies that $\alpha^r\beta \prec \alpha^{s'}\gamma$ for any $s' < r$. Thus, for $x \in [1, h]$, if $s-x+1 < r$ then $T[i..n] = \alpha^r\beta \prec \alpha^{s-x+1}\gamma = T[c_x..n]$. Hence we only have to consider $x \leq s - r + 1$. On the other hand, the precondition of the case also implies $\alpha^r\beta \prec \alpha^{r-1}\beta$, and thus $\alpha\beta \prec \alpha$. Also, α is not a prefix of β. Hence Proposition 6.2 (with swapped roles of β and γ) implies that $\alpha^r\beta \succ \alpha^{s'}\gamma$ for any $s' > r$. For $x \in [1, h]$, if $x \leq s - r$ then $T[i..n] = \alpha^r\beta \succ \alpha^{s-x+1}\gamma = T[c_x..n]$.
This motivates the following strategy. If $s - r + 1 < 1$, then there is no suitable choice of x and we report $j'_{\max} = j - 1$. If $k \leq s - r$, then $T[i..n] \succ T[c_h..n]$ and we report $j'_{\max} = c_h$. We are left with the case where $s - r + 1 \in [1, h]$. If $T[i..n] \succ T[c_{s-r+1}..n]$, then we report $j'_{\max} = c_{s-r+1}$ (we use another LCE query to achieve constant time). If we still have not reported anything, then we report $j'_{\max} = c_{s-r}$ if and only if $s-r \in [1, h]$ (we have already established that $T[i..n] \succ T[c_{s-r}..n]$). If, however, $s - r \notin [1, h]$, then we report $j'_{\max} = j - 1$.

The three cases are exhaustive, and it takes constant time to determine which case applies. Regardless of the case, we report $j'_{\max}$ in constant time. We require the LCE data structure from Lemma 1, and hence the preprocessing time is $\mathcal{O}(n/\log_\sigma n)$.

Lookup Tables for c_1, c_h, p, and $j''_{\max}$. As described above, we can compute $j'_{\max}$ in constant time if we can determine c_1, c_h, and p in constant time. Note that these values depend solely on the substrings $T[i..i+2\ell)$ and $T[j..j+3\ell)$. This also holds for $j''_{\max}$, which can be written as $j''_{\max} = \max(\{j' \in [j, j+\ell) \mid T[j'..j'+2\ell) \prec T[i..i+2\ell)\} \cup \{j-1\})$. For each possible value of ℓ, we compute a lookup table $L_\ell[1..2^{2\ell\lceil\log_2\sigma\rceil}][1..2^{3\ell\lceil\log_2\sigma\rceil}]$. Let $S_1 \in [0,\sigma)^{2\ell}$ and $S_2 \in [0,\sigma)^{3\ell}$ be packed strings. In entry $L_\ell[\mathsf{int}(S_1)][\mathsf{int}(S_2)]$, we store the quadruple $\langle \hat{p}, \hat{c}_{\min}, \hat{c}_{\max}, \hat{y}_{\max}\rangle$, where

- $\hat{p}$ is the shortest period of S_1,
- $\hat{c}_{\min} = \min(\{x' \in [1,\ell] \mid S_2[x'..x'+2\ell) = S_1\} \cup \{\infty\})$,
- $\hat{c}_{\max} = \max(\{x' \in [1,\ell] \mid S_2[x'..x'+2\ell) = S_1\} \cup \{-\infty\})$,
- $\hat{y}_{\max} = \max(\{x' \in [1,\ell] \mid S_2[x'..x'+2\ell) \prec S_1\} \cup \{-\infty\})$.

A single entry $L_\ell[\mathsf{int}(S_1)][\mathsf{int}(S_2)]$ can be computed naively in $\mathcal{O}(\text{poly}(\ell)) \subseteq \mathcal{O}(\text{polylog}(n))$ time. Table L_ℓ has $2^{5\ell\lceil\log_2\sigma\rceil} \le 2^{\log_2 n/(1+\epsilon/5)} = \sqrt[1+\epsilon/5]{n}$ entries, and there are $\mathcal{O}(\log n)$ tables. Thus, the entire preprocessing time is $\mathcal{O}(\sqrt[1+\epsilon/5]{n} \cdot \text{polylog}(n)) \subset \mathcal{O}(n/\log_\sigma n)$. Whenever we have to answer a query i, $[j, j+\ell)$, we extract $T' = T[i..i+2\ell)$ and $T'' = T[j..j+3\ell)$ and lookup $\langle p, c_{\min}, c_{\max}, y_{\max}\rangle = L_\ell[\mathsf{int}(T')][\mathsf{int}(T'')]$. This takes constant time because T' and T'' fit in a single word of memory. From the construction of L_ℓ, it is clear that

- p is the shortest period of $T[i..i+2\ell)$.
- If $c_{\min} \ne \infty$ then $c_1 = j + c_{\min} - 1$. Otherwise, c_0 does not exist.
- If $c_{\max} \ne -\infty$ then $c_h = j + c_{\max} - 1$. Otherwise, c_h does not exist.
- If $y_{\max} \ne -\infty$ then $j''_{\max} = j + y_{\max} - 1$. Otherwise, $j''_{\max} = j - 1$.

Hence we can compute $j'_{\max}$ in constant time as described above, and output the query result $j_{\max} = \min(j'_{\max}, j''_{\max})$ in constant time. ◀

References

1 Maxim Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the 26th Annual Symposium on Discrete Algorithms (SODA 2015)*, pages 572–591, San Diego, CA, USA, January 2015. doi:10.1137/1.9781611973730.39.

2 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.

3 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.

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

5 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.

6 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.

7 P. Bonizzoni, M. Costantini, C. De Felice, A. Petescia, Y. Pirola, M. Previtali, R. Rizzi, J. Stoye, R. Zaccagnino, and R. Zizza. Numeric lyndon-based feature embedding of sequencing reads for machine learning approaches. *Information Sciences*, 607:458–476, 2022. doi:10.1016/j.ins.2022.06.005.

8 Paola Bonizzoni, Alessia Petescia, Yuri Pirola, Raffaella Rizzi, Rocco Zaccagnino, and Rosalba Zizza. Kfinger: Capturing overlaps between long reads by using lyndon fingerprints. In *Proceedings of the International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO 2022)*, pages 436–449, Gran Canaria, Spain, June 2022. `doi:10.1007/978-3-031-07802-6_37`.

9 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958. `doi:10.2307/1970044`.

10 Maxime Crochemore and Lucian Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, 74(5):796–807, 2008. `doi:10.1016/j.jcss.2007.09.003`.

11 Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011. `doi:10.1016/j.tcs.2010.06.019`.

12 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, pages 22–34, Beppu, Japan, October 2016. `doi:10.1007/978-3-319-46049-9_3`.

13 Maxime Crochemore and Luís M. S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806:1–9, 2020. `doi:10.1016/j.tcs.2018.08.011`.

14 Yevgeniy Dodis, Mihai Pătraşcu, and Mikkel Thorup. Changing base without losing space. In *Proceedings of the 42nd Symposium on Theory of Computing (STOC 2010)*, pages 593–602, Cambridge, MA, USA, June 2010. `doi:10.1145/1806689.1806771`.

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

16 Jonas Ellert. Lyndon Arrays Simplified. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*, pages 48:1–48:14, Potsdam, Germany, September 2022. `doi:10.4230/LIPIcs.ESA.2022.48`.

17 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`.

18 N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. `doi:10.2307/2034009`.

19 Johannes Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the 9th Latin American Symposium on Theoretical Informatics (LATIN 2010)*, pages 158–169, Oaxaca, Mexico, April 2010. `doi:10.1007/978-3-642-12200-2_16`.

20 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, pages 36–48, Barcelona, Spain, July 2006. `doi:10.1007/11780441_5`.

21 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 (PSC 2016)*, pages 172–184, Prague, Czech Republic, August 2016. URL: `http://www.stringology.org/event/2016/p15.html`.

22 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 5:1–5:13, Tel Aviv, Israel, June 2016. `doi:10.4230/LIPIcs.CPM.2016.5`.

23 Torben Hagerup. Sorting and searching on the word ram. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, pages 366–398, Paris, France, February 1998. `doi:10.1007/BFb0028575`.

24 Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theoretical Computer Science*, 307(1):173–178, 2003. `doi:10.1016/S0304-3975(03)00099-9`.

25 Yusaku Kaneta. Fast wavelet tree construction in practice. In *Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE 2018)*, pages 218–232, Lima, Peru, October 2018. doi:10.1007/978-3-030-00479-8_18.

26 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual Symposium on Theory of Computing (STOC 2019)*, pages 756–767, Phoenix, AZ, USA, June 2019. doi:10.1145/3313276.3316368.

27 Dominik Kempa and Tomasz Kociumaka. Breaking the o(n)-barrier in the construction of compressed suffix arrays and suffix trees. In *Proceedings of the 34th Annual Symposium on Discrete Algorithms (SODA)*, pages 5122–5202, Florence, Italy, January 2023. doi:10.1137/1.9781611977554.ch187.

28 Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003. doi:10.1016/S0304-3975(02)00426-7.

29 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*, pages 596–604, New York, NY, USA, October 1999. doi:10.1109/SFFCS.1999.814634.

30 Dmitry Kosolobov. Computing runs on a general alphabet. *Information Processing Letters*, 116(3):241–244, 2016. doi:10.1016/j.ipl.2015.11.016.

31 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.

32 R. C. Lyndon. On Burnside problem I. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954. doi:10.2307/1990868.

33 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual Symposium on Discrete Algorithms (SODA 1990)*, pages 319–327, San Francisco, CA, USA, January 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

34 Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997)*, pages 1–11, Aarhus, Denmark, June 1997. doi:10.1007/3-540-63220-4_45.

35 J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. In *Proceedings of the 21st International Symposium on String Processing and Information Retrieval (SPIRE 2014)*, Ouro Preto, Brazil, October 2014. doi:10.1007/978-3-319-11918-2_10.

36 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

37 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3), May 2014. doi:10.1145/2601073.

38 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. On the optimisation of the GSACA suffix array construction algorithm. In *Proceedings of the 29th International Symposium on String Processing and Information Retrieval (SPIRE 2022)*, pages 99–113, Concepción, Chile, November 2022. doi:10.1007/978-3-031-20643-6_8.

39 Simon J. Puglisi, Jamie Simpson, and W.F. Smyth. How many runs can a string contain? *Theoretical Computer Science*, 401(1):165–171, 2008. doi:10.1016/j.tcs.2008.04.020.

40 Wojciech Rytter. The number of runs in a string: Improved analysis of the linear upper bound. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006)*, pages 184–195, Marseille, France, February 2006. doi:10.1007/11672142_14.