Article

Computing Maximal Lyndon Substrings of a String

Frantisek Franek ^{1,†} and Michael Liut ^{2,*,†}

¹ Department of Computing and Software, McMaster University, Hamilton, ON L8S 4K1, Canada; franek@mcmaster.ca

² Department of Mathematical and Computational Sciences, University of Toronto Mississauga, Mississauga, ON L5L 1C6, Canada

* Correspondence: michael.liut@utoronto.ca

† These authors contributed equally to this work.

Received: 23 September 2020; Accepted: 10 November 2020; Published: 12 November 2020

Abstract: There are two reasons to have an efficient algorithm for identifying all right-maximal Lyndon substrings of a string: firstly, Bannai et al. introduced in 2015 a linear algorithm to compute all runs of a string that relies on knowing all right-maximal Lyndon substrings of the input string, and secondly, Franek et al. showed in 2017 a linear equivalence of sorting suffixes and sorting right-maximal Lyndon substrings of a string, inspired by a novel suffix sorting algorithm of Baier. In 2016, Franek et al. presented a brief overview of algorithms for computing the Lyndon array that encodes the knowledge of right-maximal Lyndon substrings of the input string. Among those presented were two well-known algorithms for computing the Lyndon array: a quadratic in-place algorithm based on the iterated Duval algorithm for Lyndon factorization and a linear algorithmic scheme based on linear suffix sorting, computing the inverse suffix array, and applying to it the *next smaller value* algorithm. Duval's algorithm works for strings over any ordered alphabet, while for linear suffix sorting, a constant or an integer alphabet is required. The authors at that time were not aware of Baier's algorithm. In 2017, our research group proposed a novel algorithm for the Lyndon array. Though the proposed algorithm is linear in the average case and has $O(n \log(n))$ worst-case complexity, it is interesting as it emulates the fast Fourier algorithm's recursive approach and introduces τ -reduction, which might be of independent interest. In 2018, we presented a linear algorithm to compute the Lyndon array of a string inspired by Phase I of Baier's algorithm for suffix sorting. This paper presents the theoretical analysis of these two algorithms and provides empirical comparisons of both of their C++ implementations with respect to the iterated Duval algorithm.

Keywords: combinatorics on words; string algorithms; regularities in strings; suffix sorting; Lyndon substrings; Lyndon arrays; right-maximal Lyndon substrings; tau-reduction algorithm; Baier's sort algorithm; iterative Duval algorithm

1. Introduction

In *combinatorics on words*, Lyndon words play a very important role. *Lyndon words*, a special case of Hall words, were named after Roger Lyndon, who was looking for a suitable description of the generators of free Lie algebras [1]. Despite their humble beginnings, to date, Lyndon words have facilitated many applications in mathematics and computer science, some of which are: constructing de Bruijn sequences, constructing bases in free Lie algebras, finding the lexicographically smallest or largest substring in a string, and succinct prefix matching of highly periodic strings; see Marcus et al. [2], and the informative paper by Berstel and Perrin [3] and the references therein.

The pioneering work on Lyndon decomposition was already introduced by Chen, Fox, and Lyndon in [4]. The *Lyndon decomposition theorem* is not explicitly stated there; nevertheless, it follows from the work presented there.

Theorem 1 (Lyndon decomposition theorem, Chen+Fox+Lyndon, 1958). *For any word x , there are unique Lyndon words $u_1, \dots, u_k$ so that $u_{i+1} \prec u_i$ for any $1 \leq i < k$, and $x = u_1 u_2 \dots u_k$, where $\prec$ denotes the lexicographic ordering.*

As there exists a bijection between Lyndon words over an alphabet of cardinality k and irreducible polynomials over F_k [5], many results are known about this factorization: the average number of factors, the average length of the longest factor [6] and of the shortest [7]. Several algorithms deal with Lyndon factorization. Duval gave in [8] an elegant algorithm that computes, in linear time and in-place, the factorization of a word into Lyndon words. More about its implementation can be found in [9]. In [10], Fredricksen and Maiorana presented an algorithm generating all Lyndon words up to a given length in lexicographical order. This algorithm runs in a constant average time.

Two of the latest applications of Lyndon words are due to Bannai et al. In [11], they employed Lyndon roots of runs to prove the runs conjecture that *the number of runs in a string is bounded by the length of the string*. In the same paper, they presented an algorithm to compute all the runs in a string in linear time that requires the knowledge of all right-maximal Lyndon substrings of the input string with respect to an order of the alphabet and its inverse. The latter result was the major reason for our interest in computing the right-maximal Lyndon substrings of a given string. Though the terms *word* and *string* are interchangeable, and so are the terms *subword* and *substring*, in the following, we prefer to use exclusively the terms *string* and *substring* to avoid confusing the reader.

There are at least two reasons for having an efficient algorithm for identifying all right-maximal Lyndon substrings of a string: firstly, Bannai et al. published in 2017 [11] a linear algorithm to compute all runs in a string that depends on knowing all right-maximal Lyndon substrings of the input string, and secondly, in 2017, Franek et al. in [12] showed a linear equivalence of sorting suffixes and sorting right-maximal Lyndon substrings, inspired by Phase II of a suffix sorting algorithm introduced by Baier in 2015 (Master's thesis [13]) and published in 2016 [14].

The most significant feature of the runs algorithm presented in [11] is that it relies on knowing the right-maximal Lyndon substrings of the input string for some order of the alphabet and for the inverse of that order, while all other linear algorithms for runs rely on Lempel–Ziv factorization of the input string. It also raised the issue about which approach may be more efficient: to compute the Lempel–Ziv factorization or to compute all right-maximal Lyndon substrings. There are several efficient linear algorithms for Lempel–Ziv factorization (e.g., see [15,16] and the references therein).

Interestingly, Kosolobov [17] showed that for a general alphabet, in the decision tree model, the runs problem is easier than the Lempel–Ziv decomposition. His result supports the conjecture that there must be a linear random access memory model algorithm finding all runs.

Baier introduced in [13], and published in [14], a new algorithm for suffix sorting. Though Lyndon strings were never mentioned in [13,14], it was noticed by Cristoph Diegelmann in a personal communication [18] that Phase I of Baier's suffix sort identifies and sorts all right-maximal Lyndon substrings.

The right-maximal Lyndon substrings of a string $x = x[1..n]$ can be best encoded in the so-called **Lyndon array**, introduced in [19] and closely related to the *Lyndon tree* of [20]: an integer array $\mathcal{L}[1..n]$ so that for any $i \in 1..n$, $\mathcal{L}[i] = \text{the length of the right-maximal Lyndon substring starting at the position } i$.

In an overview [19], Franek et al. discussed an algorithm based on an iterative application of Duval's Lyndon factorization algorithm [8], which we refer to here as *IDLA*, and an algorithmic scheme based on Hohlweg and Reutenauer's work [20], which we refer to as *SSLA*. The authors were not aware of Baier's algorithm at that time. Two additional algorithms were presented there, a quadratic recursive application of Duval's algorithm and an algorithm *NSV** with possibly $\mathcal{O}(n \log(n))$ worst-case complexity based on ranges that can be compared in constant time for constant alphabets. The correctness of *NSV** and its complexity were discussed there just informally.

The algorithm *IDLA* (see Figure 1) is simple and in-place, so no additional space is required except for the storage for the string and the Lyndon array. It is completely independent of the alphabet of the string and does not require the alphabet to be sorted; all it requires is that the alphabet be ordered,

i.e., only pairwise comparisons of the alphabet symbols are needed. Its weakness is its quadratic worst-case complexity, which becomes a problem for longer strings with long right-maximal Lyndon substrings, as one of our experiments showed (see Figure 11 in Section 7).

In our empirical work, we used *IDLA* as a control for comparison and as a verifier of the results. Note that the reason the procedure *MaxLyn* of Figure 1 really computes the longest Lyndon prefix is not obvious and is based on the properties of periods of prefixes; see [8] or Observation 6 and Lemma 11 in [19].

Lemma 1, below, characterizes right-maximal Lyndon substrings in terms of the relationships of the suffixes and follows from the work of Hohlweg and Reutenauer [20]. Though the definition of the proto-Lyndon substring is formally given in Section 2 below, it suffices to say that it means that it is a prefix of a Lyndon substring of the string. The definition of the lexicographic ordering $\prec$ is also given Section 2. The proof of this lemma is delayed to the end of Section 2, where all the technical terms needed are defined.

```

procedure MaxLyn( $x[1..n], j, \Sigma, \prec$ ) | integer
   $i \leftarrow j + 1$ ;  $\text{max} \leftarrow 1$ 
  while  $i \leq n$  do
     $k \leftarrow 0$ 
    while  $x[j + k] = x[i + k]$  do
       $k \leftarrow k + 1$ 
    if  $x[j + k] \prec x[i + k]$  then
       $i \leftarrow i + k + 1$ ;  $\text{max} \leftarrow i - 1$ 
    else
      return  $\text{max}$ 
procedure IDLA( $x[1..n], j, \Sigma, \prec$ ) | integer array
   $i \leftarrow 1$ 
  while  $i < n$  do
     $L[i] = \text{MaxLyn}(x[i..n], j, \Sigma, \prec)$ 
     $i \leftarrow i + 1$ 
   $L[n] \leftarrow 1$ 
  return  $L$ 

```

Figure 1. Algorithm IDLA.

Lemma 1. Consider a string $x[1..n]$ over an alphabet ordered by $\prec$. A substring $x[i..j]$ is proto-Lyndon if and only if:

(a) $x[i..n] \prec x[k..n]$ for any $i < k \leq j$.

A substring $x[i..j]$ is right-maximal Lyndon if and only if:

(b) $x[i..n]$ is proto-Lyndon and

(c) either $j = n$ or $x[j+1..n] \prec x[i..n]$.

Thus, the Lyndon array is an *NSV* (Next Smaller Value) array of the inverse suffix array. Consequently, the Lyndon array can be computed by sorting the suffixes, i.e., computing the suffix array, then computing the inverse suffix array, and then applying *NSV* to it; see [19]. Computing the inverse suffix array and applying *NSV* are “naturally” linear, and computing the suffix array can be implemented to be linear; see [19,21] and the references therein. The execution and space characteristics are dominated by those of the first step, i.e., computation of the suffix array. We refer to this scheme as *SSLA*.

In 2018, a linear algorithm to compute the Lyndon array from a given Burrows–Wheeler transform was presented [22]. Since the Burrows–Wheeler transform is computed in linear time from the suffix array, it is yet another scheme of how to obtain the Lyndon array via suffix sorting: compute the suffix array; from the suffix array, compute the Burrows–Wheeler transform, then compute the Lyndon array during the inversion of the Burrows–Wheeler transform. We refer to this scheme as *BWLA*.

The introduction of Baier’s suffix sort in 2015 and the consequent realization of the connection to right-maximal Lyndon substrings brought up the realization that there was an elementary (not relying on a pre-processed global data structure such as a suffix array or a Burrows–Wheeler transform) algorithm to compute the Lyndon array, and that, despite its original clumsiness, could be eventually refined to outperform any *SSLA* or *BWLA* implementation: any implementation of a suffix sorting-based scheme requires a full suffix sort and then some additional processing, while Baier’s approach is “just” a partial suffix sort; see [23].

In this work, we present two additional algorithms for the Lyndon array not discussed in [19]. The C++ source code of the three implementations *IDLA*, *TRLA*, and *BSLA* is available; see [24]. Note that the procedure *IDLA* is in the `lynarr.hpp` file.

The first algorithm presented here is *TRLA*. *TRLA* is a τ -reduction based Lyndon array algorithm that follows Farach’s approach used in his remarkable linear algorithm for suffix tree construction [25] and reproduced very successfully in all linear algorithms for suffix sorting (e.g., see [21,26] and the references therein). Farach’s approach follows the Cooley–Tukey algorithm for the fast Fourier transform relying on recursion to lower the quadratic complexity to $O(n \log(n))$ complexity; see [27]. *TRLA* was first introduced by the authors in 2019 (see [28,29]) and presented as a part of Liut’s Ph.D. thesis [30].

The second algorithm, *BSLA*, is a Baier’s sort-based Lyndon array algorithm. *BSLA* is based on the idea of Phase I of Baier’s suffix sort, though our implementation necessarily differs from Baier’s. *BSLA* was first introduced at the Prague Stringology Conference 2018 [23] and also presented as a part of Liut’s Ph.D. thesis [30] in 2019; here, we present a complete and refined theoretical analysis of the algorithm and a more efficient implementation than that initially introduced.

The paper is structured as follows: In Section 2, the basic notions and terminology are presented. In Section 3, the *TRLA* algorithm is presented and analysed. In Section 4, the *BSLA* algorithm is presented and analysed. In Section 5, the datasets with random strings of various lengths and over various alphabets and other datasets used in the empirical tests are described. In Section 6, the conclusion of the research and the future work are presented. The results of the empirical measurements of the performance of *IDLA*, *TRLA*, and *BSLA* on those datasets are presented in Section 7 in both tabular and graphical forms.

2. Basic Notation and Terminology

Most of the fundamental notions, definitions, facts, and string algorithms can be found in [31–34]. For the ease of access, this section includes those that are directly related to the work herein.

The set of integers is denoted by $\mathbb{Z}$. For two integers $i \leq j$, the *range* $i..j = \{k \in \mathbb{Z} \mid i \leq k \leq j\}$. An *alphabet* is a finite or infinite set of *symbols*, or equivalently called *letters*. We always assume that the sentinel symbol $\$$ is not in the alphabet and is always assumed to be lexicographically the smallest. A *string* over an alphabet $\mathcal{A}$ is a finite sequence of symbols from $\mathcal{A}$. A *$\$$ -terminated string* over $\mathcal{A}$ is a string over $\mathcal{A}$ terminated by $\$$. We use the array notation indexing from 1 for strings; thus, $x[1..n]$ indicates a string of length n ; the first symbol is the symbol with index 1, i.e., $x[1]$; the second symbol is the symbol with index 2, i.e., $x[2]$, etc. Thus, $x[1..n] = x[1]x[2]...x[n]$. For a $\$$ -terminated string x of length n , $x[n+1] = \$$. The *alphabet of string* x , denoted as $\mathcal{A}_x$, is the set of all distinct alphabet symbols occurring in x .

We use the term *strings over a constant alphabet* if the alphabet is a fixed finite alphabet. The *integer alphabet* is the infinite alphabet $\mathcal{A} = \{0, 1, 2, \dots\}$. We use the term *strings over integer alphabet* for the strings over the alphabet $\{0, 1, 2, \dots\}$ with an additional constraint that all letters occurring in the string are all smaller than the length of the string, i.e., in this paper, $x[1..n]$ is a string over the integer alphabet if it is a string over the alphabet $\{0, 1, \dots, n-1\}$. Many authors use a more general definition; for instance, Burkhardt and Kärkkäinen [35] defined it as any set of integers of size $n^{o(1)}$; however, our results can easily be adapted to such more general definitions without changing their essence.

We use a **bold font** to denote strings; thus, x denotes a string, while x denotes some other mathematical entity such as an integer. The **empty string** is denoted by ε and has length zero. The **length** or **size** of string $x = x[1..n]$ is n . The length of a string x is denoted by $|x|$. For two strings $x = x[1..n]$ and $y = y[1..m]$, the **concatenation** xy is a string u

$$\text{where } u[i] = \begin{cases} x[i] & \text{for } i \leq n, \\ y[i-n] & \text{for } n < i \leq n+m. \end{cases}$$

If $x = uvw$, then u is a **prefix**, v a **substring**, and w a **suffix** of x . If u (respectively, v , w) is empty, then it is called an **empty prefix** (respectively, **empty substring**, **empty suffix**); if $|u| < |x|$ (respectively, $|v| < |x|$, $|w| < |x|$), then it is called a **proper prefix** (respectively, **proper substring**, **proper suffix**). If $x = uv$, then vu is called a **rotation** or a **conjugate** of x ; if either $u = \varepsilon$ or $v = \varepsilon$, then the rotation is called **trivial**. A non-empty string x is **primitive** if there is no string y and no integer $k \geq 2$ so that $x = y^k = \underbrace{yy \cdots y}_{k \text{ times}}$.

A non-empty string x has a non-trivial border u if u is both a non-empty proper prefix and a non-empty proper suffix of x . Thus, both ε and x are trivial borders of x . A string without a non-trivial border is called **unbordered**.

Let $\prec$ be a total order of an alphabet $\mathcal{A}$. The order is extended to all finite strings over the alphabet $\mathcal{A}$: for $x = x[1..n]$ and $y = y[1..m]$, $x \prec y$ if either x is a proper prefix of y or there is a $j \leq \min\{n, m\}$ so that $x[1] = y[1], \dots, x[j-1] = y[j-1]$ and $x[j] \prec y[j]$. This total order induced by the order of the alphabet is called a **lexicographic** order of all non-empty strings over $\mathcal{A}$. We denote by $x \preceq y$ if either $x \prec y$ or $x = y$. A string x over $\mathcal{A}$ is **Lyndon** for a given order $\prec$ of $\mathcal{A}$ if x is strictly lexicographically smaller than any non-trivial rotation of x . In particular:

$$x \text{ is Lyndon} \Rightarrow x \text{ is unbordered} \Rightarrow x \text{ is primitive}$$

Note that the reverse implications do not hold: aba is primitive but neither unbordered, nor Lyndon, while $acaab$ is unbordered, but not Lyndon. A substring $x[i..j]$ of $x[1..n]$, $1 \leq i < j \leq n$ is a **right-maximal Lyndon substring** of x if it is Lyndon and either $j = n$ or for any $k > j$, $x[i..k]$ is not Lyndon.

A substring $x[i..j]$ of a string $x[1..n]$ is **proto-Lyndon** if there is a $j \leq k \leq n$ so that $x[i..k]$ is Lyndon. The **Lyndon array** of a string $x = x[1..n]$ is an integer array $\mathcal{L}[1..n]$ so that $\mathcal{L}[i] = j$ where $j \leq n-i$ is a maximal integer such that $x[i..i+j-1]$ is Lyndon. Alternatively, we can define it as an integer array $\mathcal{L}'[1..n]$ so that $\mathcal{L}'[i] = j$ where j is the last position of the right-maximal Lyndon substring starting at the position i . The relationship between those two definitions is straightforward: $\mathcal{L}'[i] = \mathcal{L}[i] + i - 1$ or $\mathcal{L}[i] = \mathcal{L}'[i] - i + 1$.

Proof of Lemma 1. We first prove the following claim:

Claim. Substring $x[i..j]$ is right-maximal Lyndon if and only if:

- (b') $x[i..n] \prec x[k..n]$ for any $i < k \leq j$ and
- (c) either $j = n$ or $x[j+1..n] \prec x[i..n]$.

Let $x[i..j]$ be right-maximal Lyndon. We need to show that (b') and (c) hold. Let $i < k \leq j$. Since it is Lyndon, $x[i..j] \prec x[k..j]$. Thus, there is $0 \leq r$ so that $i+r \leq j$ and $j+r \leq j$ and $x[i+r] = x[j+r]$ for any $0 \leq \ell < r$ and $x[i+r] \prec x[j+r]$. It follows that $x[i..n] \prec x[j..n]$, and (b') is satisfied.

If $j = n$, then (c) holds. Therefore, let us assume that $j < n$.

- (1) If $x[i] \prec x[j+1]$, then $x[i..j+1] \prec x[j+1..j+1]$; together with $x[i..j] \prec x[k..j]$ for any $i < k \leq j$, it shows that $x[i..j+1]$ is Lyndon, contradicting the right-maximality of $x[i..j]$.
- (2) If $x[i] \succ x[j+1]$, then $x[i..n] \succ x[j+1..n]$, and (c) holds.
- (3) If $x[i] = x[j+1]$, then there are a prefix u , strings v and w , and an integer $r \geq 1$ so that $uv = x[i..j]$, $x[i..n] = uvu^r w$, and $x[j+1..n] = u^r w$. Let us take a maximal such u and a maximal such r .

- (3a) Let $r \geq 2$. Then, since $uv = x[i..j]$ is Lyndon, $u < v$, and so, $uu < uv$; hence, $u^r w < uvu^r w$, and (c) holds.
- (3b) Let $r = 1$, i.e., $x[i..n] = uvuw$ and $x[j+1..n] = uw$. There is no common prefix of v and w as it would contradict the maximality of u . There are thus two mutually exclusive cases, either $v[1] < w[1]$ or $v[1] > w[1]$. Let $v[1] = c_1$ and $w[1] = c_2$. If $c_1 < c_2$, then $uc_1 < uc_2$, and so, $uv < uc_2$. For any $u = u_1u_2$, $uv < u_2v$ as uv is Lyndon, so $uv < u_2c_2$, giving uvc_2 to be Lyndon, a contradiction with the right-maximality of uv . Therefore, $c_1 > c_2$; thus, $uw < uvuw$, and (c) holds.

Now, we go in the opposite direction. Assuming (b') and (c), we need to show that $x[i..j]$ is right-maximal Lyndon.

Consider the right-maximal substring of x starting at the position i ; it is $x[i..\ell]$ for some $i \leq \ell \leq n$.

- (1) If $\ell < j$, then by the first part of this proof, $x[i..n] > x[\ell+1..n]$, contradicting the assumption (b') as $\ell+1 \leq j$.
- (2) If $j < \ell$, then $j+1 \leq \ell$, and by the first part of this proof, $x[i..n] < x[j+1..n]$, contradicting the assumption (c).
- (3) If $\ell = j$, then $x[i..j]$ is right-maximal Lyndon.

Now, we can prove (a). Let $x[i..j]$ be a proto-Lyndon substring of x . By definition, it is a prefix of a Lyndon substring of x and, hence, a prefix of a right-maximal Lyndon substring of x , say $x[i..\ell]$ for some $j \leq \ell \leq n$. It follows from the claim that $x[i..n] < x[k..n]$ for any $i < k \leq \ell$. Since $j \leq \ell$, (a) holds. For the opposite direction, if (a) holds, then there are two possibilities: either there is $j < \ell \leq n$ so that $x[i..n] > x[\ell..n]$ or $x[i..n] < x[k..n]$ for any $i < k \leq n$. By the claim, in the former case, $x[i..\ell-1]$ is a right-maximal Lyndon substring of x , while in the latter case, $x[i..n]$ is a right-maximal Lyndon substring of x . Thus, in both cases, $x[i..j]$ is a prefix of a Lyndon substring of x .

With (a) proven, we can now replace (b') in the claim with (b), completing the proof. $\square$

3. τ -Reduction Algorithm (TRLA)

The purpose of this section is to introduce a recursive algorithm *TRLA* for computing the Lyndon array of a string. As will be shown below, the most significant aspect is the so-called τ -reduction of a string and how the Lyndon array of the τ -reduced string can be expanded to a partially filled Lyndon array for the whole string, as well as how to compute the missing values. This section thus provides the mathematical justification for the algorithm and, in so doing, proves the correctness of the algorithm. The mathematical understanding of the algorithm provides the bases for the bounding of its worst-case complexity by $O(n \log(n))$ and determining the linearity of the average-case complexity.

The first idea of the algorithm was proposed in Paracha's 2017 Ph.D. thesis [36]. It follows Farach's approach [25]:

- (1) reduce the input string x to y ;
- (2) by recursion, compute the Lyndon array of y ; and
- (3) from the Lyndon array of y , compute the Lyndon array of x .

The input strings for the algorithm are $\$$ -terminated strings over an integer alphabet. The reduction computed in (1) is important. All linear algorithms for suffix array computations use the proximity property of suffixes: comparing $x[i..n]$ and $x[j..n]$ can be done by comparing $x[i]$ and $x[j]$ and, if they are the same, comparing $x[i+1..n]$ with $x[j+1..n]$. For instance, in the first linear algorithm for the suffix array by Kärkkäinen and Sanders [37], obtaining the sorted suffixes for positions $i \equiv 0 \pmod{3}$ and $i \equiv 1 \pmod{3}$ via the recursive call is sufficient to determine the order of suffixes for the $i \equiv 2 \pmod{3}$ positions, then merging both lists together. However, there is no such proximity property for right-maximal Lyndon substrings, so the reduction itself must have a property that helps determine some of the values of the Lyndon array of x from the Lyndon array of y and computing the rest.

In our algorithm, we use a special reduction, which we call τ -reduction, defined in Section 3.2, that reduces the original string to at most $\frac{1}{2}$ and at least $\frac{2}{3}$ of its length. The algorithm computes y as a τ -reduction of the input string x in Step (1) in linear time. In Step (3), it expands the Lyndon array of the reduced string computed by Step (2) to an incomplete Lyndon array of the original string also in linear time. The incomplete Lyndon array computed in (3) is about $\frac{1}{2}$ to $\frac{2}{3}$ full, and for every position i with an unknown value, the values at positions $i-1$ and $i+1$ are known. In particular, the values at Position 1 and position n are both known. Therefore, much information is provided by the recursive Step (2). For instance, for 00011001, via the recursive call, we would identify the right-maximal Lyndon substrings that are underlined in 00011001 and would need to compute the missing right-maximal Lyndon substrings that are underlined in 00011001.

However, computing the missing values of the incomplete Lyndon array takes at most $O(n \log(n))$ steps, as we will show, resulting in the overall worst-case complexity of $O(n \log(n))$. When the input string is such that the missing values of the incomplete Lyndon array of the input string can be computed in linear time, the overall execution of the algorithm is linear as well, and thus, the average case complexity will be shown to be linear in the length of the input string.

In the following subsections, we describe the τ -reduction in several steps: first, the τ -pairing, then choosing the τ -alphabet, and finally, the computation of the $\tau(x)$. The τ -reduction may be of some general interest as it preserves (see Lemma 6) some right-maximal Lyndon substrings of the original string.

3.1. τ -Pairing

Consider a $\$$ -terminated string $x = x[1..n]$ whose alphabet $\mathcal{A}_x$ is ordered by $\prec$ where $x[n+1] = \$$ and $\$ \prec a$ for any $a \in \mathcal{A}_x$. A τ -pair consists of a pair of adjacent positions from the range $1..n+1$. The τ -pairs are computed by induction:

1. the initial τ -pair is $(1, 2)$;
2. if $(i-1, i)$ is the last τ -pair computed, then:
 - if $i = n-1$ then
 - the next τ -pair is set to $(n, n+1)$
 - stop
 - elseif $i \geq n$ then
 - stop
 - elseif $x[i-1] \succ x[i]$ and $x[i] \preceq x[i+1]$ then
 - the next τ -pair is set to $(i, i+1)$; repeat 2.
 - else
 - the next τ -pair is set to $(i+1, i+2)$; repeat 2.

Every position of the input string that occurs in some τ -pair as the first element is labelled **black**; all others are labelled **white**. Note that Position 1 is always black, while the last position n can be either black or white; however, the positions $n-1$ and n cannot be simultaneously both black. Note also that most of the τ -pairs do not overlap; if two τ -pairs overlap, they overlap in a position i such that $1 < i < n$ and $x[i-1] \succ x[i]$ and $x[i] \preceq x[i+1]$. The first position and the last position never figure in an overlap of τ -pairs. Moreover, a τ -pair can be involved in at most one overlap; for an illustration, see Figure 2; for the formal proof see Lemma 2.

Lemma 2. Let $(i_1, i_1+1) \dots (i_k, i_k+1)$ be the τ -pairs of a string $x = x[1..n]$. Then, for any $j, \ell \in 1..k$:

- (1) if $|(i_j, i_j+1) \cap (i_\ell, i_\ell+1)| = 1$, then for any $m \neq j, \ell$, $|(i_j, i_j+1) \cap (i_m, i_m+1)| = 0$,
- (2) $|(i_j, i_j+1) \cap (i_\ell, i_\ell+1)| \leq 1$.

Figure 2. Illustration of the τ -reduction of a string **011023122**. The rounded rectangles indicate symbol τ -pairs; the ovals indicate the τ -pairs. below are the colour labels of the positions; at the bottom is the τ -reduction.

Proof. This is by induction; trivially true for $|x| = 1$ as $(1, 2)$ is the only τ -pair. Assume it is true for $|x| \leq n-1$.

1. Case $(i_k, i_k+1) = (n, n+1)$:
Then, $(i_{k-1}, i_{k-1}+1) = (n-2, n-1)$, and so, $(i_1, i_1+1) \dots (i_{k-1}, i_{k-1}+1)$ are τ -pairs of $x[1..n-1]$; thus, they satisfy (1) and (2) by the induction hypothesis. However, $(n, n+1) \cap (i_\ell, i_{\ell+1}) = \emptyset$ for $1 \leq \ell < k$, so (1) and (2) hold for $(i_1, i_1+1) \dots (i_k, i_k+1)$.
2. Cases $(i_k, i_k+1) = (n-1, n)$ and $(i_{k-1}, i_{k-1}+1) = (n-2, n-1)$:
Therefore, $(i_1, i_1+1) \dots (i_{k-1}, i_{k-1}+1)$ are τ -pairs of $x[1..n-1]$, and thus, they satisfy (1) and (2) by the induction hypothesis. However, $(i_k, i_k+1) \cap (i_\ell, i_{\ell+1}) = \emptyset$ for $1 \leq \ell < k-1$, and $(i_k, i_k+1) \cap (i_{k-1}, i_{k-1}+1) = \{i_{k-1}\} = n-1$; so, $|(i_k, i_k+1) \cap (i_{k-1}, i_{k-1}+1)| \leq 1$, and so, (1) and (2) hold for $(i_1, i_1+1) \dots (i_k, i_k+1)$.
3. Cases $(i_k, i_k+1) = (n-1, n)$ and $(i_{k-1}, i_{k-1}+1) = (n-3, n-2)$:
Then, $(i_1, i_1+1) \dots (i_{k-1}, i_{k-1}+1)$ are τ -pairs of $x[1..n-2]$, so they satisfy (1) and (2) by the induction hypothesis. However, $(i_k, i_k+1) \cap (i_\ell, i_{\ell+1}) = \emptyset$ for $1 \leq \ell < k$, so (1) and (2) hold for $(i_1, i_1+1) \dots (i_k, i_k+1)$.

□

3.2. τ -Reduction

For each τ -pair $(i, i+1)$, we consider the pair of alphabet symbols $(x[i], x[i+1])$. We call them **symbol τ -pairs**. They are in a total order $\triangleleft$ induced by $\prec$: $(x[i_j], x[i_j+1]) \triangleleft (x[i_\ell], x[i_\ell+1])$ if either $x[i_j] \prec x[i_\ell]$, or $x[i_j] = x[i_\ell]$ and $x[i_j+1] \prec x[i_\ell+1]$. They are sorted using the radix sort with a key of size two and assigned letters from a chosen τ -alphabet that is a subset of $\{0, 1, \dots, |\tau(x)|\}$ so that the assignment preserves the order. Since the input string is over an integer alphabet, the radix sort is linear.

In the example (Figure 2), the τ -pairs are $(1, 2)(3, 4)(4, 5)(6, 7)(7, 8)(9, 10)$, and so, the symbol τ -pairs are $(0, 1)(1, 0)(0, 2)(3, 1)(1, 2)(2, \$)$. The sorted symbol τ -pairs are $(0, 1)(0, 2)(1, 0)(1, 2)(2, \$)(3, 1)$. Thus, we chose as our τ -alphabet $\{0, 1, 2, 3, 4, 5\}$, and so, the symbol τ -pairs are assigned these letters: $(0, 1) \rightarrow 0$, $(0, 2) \rightarrow 1$, $(1, 0) \rightarrow 2$, $(1, 2) \rightarrow 3$, $(2, \$) \rightarrow 4$, and $(3, 1) \rightarrow 5$. Note that the assignments respect the order $\triangleleft$ of the symbols τ -pairs and the natural order $<$ of $\{0, 1, 2, 3, 4, 5\}$.

The τ -letters are substituted for the symbol τ -pairs, and the resulting string is terminated with $\$$. This string is called the **τ -reduction** of x and denoted $\tau(x)$, and it is a $\$$ -terminated string over an integer alphabet. For our running example from Figure 2, $\tau(x) = 021534$. The next lemma justifies calling the above transformation a reduction.

Lemma 3. For any string x , $\frac{1}{2}|x| \leq |\tau(x)| \leq \frac{2}{3}|x|$.

Proof. There are two extreme cases; the first is when all the τ -pairs do not overlap at all, then $|\tau(x)| = \frac{1}{2}|x|$; and the second is when all the τ -pairs overlap, then $|\tau(x)| = \frac{2}{3}|x|$. Any other case must be in between. $\square$

Let $\mathcal{B}(x)$ denote the set of all black positions of x . For any $i \in 1..|\tau(x)|$, $b(i) = j$ where j is a black position in x of the τ -pair corresponding to the new symbol in $\tau(x)$ at position i , while $t(j)$ assigns each black position of x the position in $\tau(x)$ where the corresponding new symbol is, i.e., $b(t(j)) = j$ and $t(b(i)) = i$. Thus,

$$1..|\tau(x)| \xrightleftharpoons[t]{b} \mathcal{B}(x)$$

In addition, we define p as the mapping of the τ -pairs to the τ -alphabet.

In our running example from Figure 2, $t(1) = 1$, $t(3) = 2$, $t(4) = 3$, $t(6) = 4$, $t(7) = 5$, and $t(9) = 6$, while $b(1) = 1$, $b(2) = 3$, $b(3) = 4$, $b(4) = 6$, $b(5) = 7$, and $b(6) = 9$. For the letter mapping, we get $p(1,2) = 0$, $p(3,4) = 2$, $p(4,5) = 1$, $p(6,7) = 5$, $p(7,8) = 3$, and $p(9,10) = 4$.

3.3. Properties Preserved by τ -Reduction

The most important property of τ -reduction is a preservation of right-maximal Lyndon substrings of x that start at black positions. This means there is a closed formula that gives, for every right-maximal Lyndon substring of $\tau(x)$, a corresponding right-maximal Lyndon substring of x . Moreover, the formula for any black position can be computed in constant time. It is simpler to present the following results using $\mathcal{L}'$, the alternative form of the Lyndon array, the one where the end positions of right-maximal Lyndon substrings are stored rather than their lengths. More formally:

Theorem 2. Let $x = x[1..n]$; let $\mathcal{L}'_{\tau(x)}[1..m]$ be the Lyndon array of $\tau(x)$; and let $\mathcal{L}'_x[1..n]$ be the Lyndon array of x .

$$\text{Then, for any black } i \in 1..n, \mathcal{L}'_x[i] = \begin{cases} b(r) & \text{if } b(r) = n \text{ or } x[b(r)+1] \preceq x[i] \\ b(r)+1 & \text{otherwise,} \end{cases}$$

where $r = \mathcal{L}'_{\tau(x)}[t(i)]$.

The proof of the theorem requires a series of lemmas that are presented below. First, we show that τ -reduction preserves the relationships of certain suffixes of x .

Lemma 4. Let $x = x[1..n]$, and let $\tau(x) = \tau(x)[1..m]$. Let $i \neq j$ and $1 \leq i, j \leq n$. If i and j are both black positions, then $x[i..n] \prec x[j..n]$ implies $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

Proof. Since i and j are both black positions, both $t(i)$ and $t(j)$ are defined, and $t(i) \neq t(j)$. Let us assume that $x[i..n] \prec x[j..n]$. The proof is argued in several cases determined by the nature of the relationship $x[i..n] \prec x[j..n]$.

- (1) Case: $x[i..n]$ is a proper prefix of $x[j..n]$.
Then, $|x[i..n]| = n - i + 1 < |x[j..n]| = n - j + 1$, and so, $j < i$. It follows that $x[j..j+n-i] = x[i..n]$, and thus, $x[i..n]$ is a border of $x[j..n]$.
 - (1a) Case: $j+n-i$ is black.
Since n may be either black or white, we need to discuss two cases.
 - (1aα) Case: n is white.
Since n is white, the last τ -pair of x must be $(n-1, n)$. The τ -pairs of $x[j..j+n-i]$ must be the same as the τ -pairs of $x[i..n]$; the last τ -pair of $x[j..j+n-i]$ must

be $(j+n-i-1, j+n-i)$. Since $j+n-i$ is black by our assumption (1a), the next τ -pair of x must be $(j+n-i, j+n-i+1)$, as indicated in the following diagram:

Thus, $\tau(x)[t(j)..t(j+n-i-1)] = \tau(x)[t(i)..t(n-1)]$. Since $t(n-1) = m$, we have $\tau(x)[t(j)..t(j+n-i-1)] = \tau(x)[t(i)..m]$, and so, $\tau(x)[t(i)..m]$ is a proper prefix of $\tau(x)[t(j)..m]$ giving $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(1a β) Case: n is black.

Then, the last τ -pair of x must be $(n, n+1)$, and hence, the last τ -pair of $x[j..j+n-i]$, so the next τ -pair is $(j+n-i, j+n-i+1)$; since $n-1$ cannot be black when n is, the situation is as indicated in the following diagram:

Thus, $\tau(x)[t(i)..t(n-2)] = \tau(x)[t(j)..t(j+n-i-2)]$. Since $x[j+n-i] = x[n]$ and $(x[n], x[n+1]) = (x[n], \$)$, we have $(x[j+n-i], x[j+n-i+1]) \triangleleft (x[n], x[n+1])$, and so, $\tau(x)[t(j+n-i)] \prec \tau(x)[t(n)]$, giving $\tau(x)[t(j)..t(n)] \prec \tau(x)[t(i)..t(n)]$. Since $t(n) = m$, we have $\tau(x)[t(j)..m] \prec \tau(x)[t(i)..m]$.

(1b) Case: $j+n-i$ is white.

Then, $j+n-i-1$ is black; hence, $n-1$ is black; so, n must also be white, and thus, $\tau(x)[t(j)..t(j+n-i-1)] = \tau(x)[t(i)..t(n-1)]$, as indicated by the following diagram:

Since $t(n-1) = m$, we have $\tau(x)[t(j)..t(j+n-i-1)] = \tau(x)[t(i)..m]$, and so, $\tau(x)[t(i)..m]$ is a proper prefix of $\tau(x)[t(j)..m]$, giving $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(2) Case: $x[i] \prec x[j]$ or $(x[i] = x[j] \text{ and } x[i+1] \prec x[j+1])$.

Then, $(x[i], x[i+1]) \triangleleft (x[j], x[j+1])$, and so, $\tau(x)[t(i)] \prec \tau(x)[t(j)]$, and thus, $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(3) Case: for some $\ell \geq 3$, $x[i..i+\ell-1] = x[j..j+\ell-1]$, while $x[i+\ell] \prec x[j+\ell]$.

First note that $i+\ell-2$ and $j+\ell-2$ are either both black, or both are white:

- If $i+\ell-2$ is white, then the τ -pairs $((i, i+1), \dots, (i+\ell-3, i+\ell-2))$ of $x[i..n]$ correspond one-to-one to the τ -pairs $((j, j+1), \dots, (j+\ell-3, j+\ell-2))$ of $x[j..n]$. To determine what follows $(i+\ell-3, i+\ell-2)$, we need to know the relationship between the values $x[i+\ell-3]$, $x[i+\ell-2]$, and $x[i+\ell-1]$. Since $x[i+\ell-3] = x[j+\ell-3]$, $x[i+\ell-2] = x[j+\ell-2]$, and $x[i+\ell-1] = x[j+\ell-1]$, the values $x[j+\ell-3]$, $x[j+\ell-2]$, and $x[j+\ell-1]$ have the same relationship, and thus, the τ -pair following $(j+\ell-3, j+\ell-2)$ will be the “same” as the τ -pair following $(i+\ell-3, i+\ell-2)$. Since $i+\ell-2$ is white, the τ -pair following $(i+\ell-3, i+\ell-2)$ is $(i+\ell-1, i+\ell)$, and so, the τ -pair following $(j+\ell-3, j+\ell-2)$ is $(j+\ell-1, j+\ell)$, making $j+\ell-2$ white as well.
- If $i+\ell-2$ is black, then the τ -pairs $((i, i+1), \dots, (i+\ell-2, i+\ell-1))$ of $x[i..n]$ correspond one-to-one to the τ -pairs $((j, j+1), \dots, (j+\ell-2, j+\ell-1))$ of $x[j..n]$. It follows that $j+\ell-2$ is black as well.

We proceed by discussing these two cases for the colours of $i+\ell-2$ and $j+\ell-2$.

(3a) Case when $i+\ell-2$ and $j+\ell-2$ are both white.

Therefore, we have the τ -pairs $((i, i+1), \dots, (i+\ell-3, i+\ell-2), (i+\ell-1, i+\ell))$ for $x[i..n]$

that correspond one-to-one to the τ -pairs $((j, j+1), \dots, (j+\ell-3, j+\ell-2), (j+\ell-1, j+\ell))$ for $x[j..n]$. It follows that $\tau(x)[t(i)..t(i+\ell-3)] = \tau(x)[t(j)..t(j+\ell-3)]$. $\tau(x)[t(i+\ell-1)] = p(i+\ell-1, i+\ell)$ and $\tau(x)[t(j+\ell-1)] = p(j+\ell-1, j+\ell)$. Since $x[i+\ell-1] = x[j+\ell-1]$ and, by our assumption (3), $x[i+\ell] \prec x[j+\ell]$, it follows that $(i+\ell-1, i+\ell) \triangleleft (j+\ell-1, j+\ell)$, giving $p(i+\ell-1, i+\ell) \prec p(j+\ell-1, j+\ell)$, and so, $\tau(x)[t(i+\ell-1)] \prec \tau(x)[t(j+\ell-1)]$. Since $t(i+\ell-3)+1 = t(i+\ell-1)$ and $t(j+\ell-3)+1 = t(j+\ell-1)$, we have $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(3b) Case when $i+\ell-2$ and $j+\ell-2$ are both black.

Therefore, we have the τ -pairs $((i, i+1), \dots, (i+\ell-2, i+\ell-1))$ for $x[i..n]$ that correspond one-to-one to the τ -pairs $((j, j+1), \dots, (j+\ell-2, j+\ell-1))$ for $x[j..n]$. It follows that $\tau(x)[t(i)..t(i+\ell-2)] = \tau(x)[t(j)..t(j+\ell-2)]$.

We need to discuss the four cases based on the colours of $i+\ell-1$ and $j+\ell-1$.

(3b α) Both $i+\ell-1$ and $j+\ell-1$ are black.

It follows that the next τ -pair for $x[i..n]$ is $(i+\ell-1, i+\ell)$, and the next τ -pair for $x[j..n]$ is $(j+\ell-1, j+\ell)$. It follows that $t(i+\ell-2)+1 = t(i+\ell-1)$ and $t(j+\ell-2)+1 = t(j+\ell-1)$. Hence, $\tau(x)[t(i+\ell-2)+1] = p(i+\ell-1, i+\ell)$ and $\tau(x)[t(j+\ell-2)+1] = p(j+\ell-1, j+\ell)$. Since $x[i+\ell-1] = x[j+\ell-1]$ and, by Assumption (3), $x[i+\ell] \prec x[j+\ell]$, we have $(x[i+\ell-1], x[i+\ell]) \triangleleft (x[j+\ell-1], x[j+\ell])$, and so, $p(x[i+\ell-1], x[i+\ell]) \prec p(x[j+\ell-1], x[j+\ell])$, giving us $\tau(x)[t(i+\ell-2)+1] \prec \tau(x)[t(j+\ell-2)+1]$. It follows that $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(3b β) $i+\ell-1$ is white, and $j+\ell-1$ is black.

It follows that the next τ -pair for $x[i..n]$ is $(i+\ell, i+\ell+1)$, and the next τ -pair for $x[j..n]$ is $(j+\ell-1, j+\ell)$. It follows that $t(i+\ell-2)+1 = t(i+\ell)$, while $t(j+\ell-2)+1 = t(j+\ell-1)$. Thus, $\tau(x)[t(i+\ell-2)+1] = p(i+\ell, i+\ell+1)$ and $\tau(x)[t(j+\ell-2)+1] = p(j+\ell-1, j+\ell)$. Since $j+\ell-1$ is black, we know that $x[j+\ell-2] \succ x[j+\ell-1] \preceq x[j+\ell]$. Since $x[i+\ell-2] = x[j+\ell-2]$ and $x[i+\ell-1] = x[j+\ell-1]$, we have $x[i+\ell-2] \succ x[i+\ell-1]$, and so, $x[i+\ell-1] \succ x[i+\ell]$, as otherwise, $i+\ell-1$ would be black. This gives us $x[j+\ell-1] \succ x[i+\ell]$. Thus, $(x[i+\ell], x[i+\ell+1]) \triangleleft (x[j+\ell-1], x[j+\ell])$, giving $p(i+\ell, i+\ell+1) \prec p(j+\ell-1, j+\ell)$ and, ultimately, $\tau(x)[t(i+\ell)] \prec \tau(x)[t(j+\ell-1)]$. The last step is to realize that $t(i+\ell-2)+1 = t(i+\ell)$ and $t(j+\ell-2)+1 = t(j+\ell-2)$, which gives us $\tau(x)[t(i+\ell-2)+1] \prec \tau(x)[t(j+\ell-2)+1]$. It follows that $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(3b γ) $i+\ell-1$ is black, and $j+\ell-1$ is white.

It follows that the next τ -pair for $x[i..n]$ is $(i+\ell-1, i+\ell)$, and the next τ -pair for $x[j..n]$ is $(j+\ell, j+\ell+1)$. It follows that $t(i+\ell-2)+1 = t(i+\ell-1)$, while $t(j+\ell-2)+1 = t(j+\ell)$. Thus, $\tau(x)[t(i+\ell-2)+1] = p(i+\ell-1, i+\ell)$ and $\tau(x)[t(j+\ell-2)+1] = p(j+\ell, j+\ell+1)$. Since $i+\ell-1$ is black, we know that $x[i+\ell-2] \succ x[i+\ell-1] \preceq x[i+\ell] \prec x[j+\ell]$, where the last inequality is our Assumption (3). Therefore, $x[j+\ell-1] = x[i+\ell-1] \prec x[j+\ell]$. Thus, $(x[i+\ell-1], x[i+\ell]) \triangleleft (x[j+\ell], x[j+\ell+1])$, giving $p(i+\ell-1, i+\ell) \prec p(j+\ell, j+\ell+1)$, $\tau(x)[t(i+\ell-1)] \prec \tau(x)[t(j+\ell)]$, and ultimately, $\tau(x)[t(i+\ell-2)+1] = \tau(x)[t(i+\ell-1)] \prec \tau(x)[t(j+\ell)] = \tau(x)[t(j+\ell-2)+1]$. It follows that $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

(3b δ) Both $i+\ell-1$ and $j+\ell-1$ are white.

Then, the next τ -pair for $x[i..n]$ is $(i+\ell, i+\ell+1)$, and the next τ -pair for $x[j..n]$ is $(j+\ell, j+\ell+1)$. It follows that $t(i+\ell-2)+1 = t(i+\ell)$, while $t(j+\ell-2)+1 = t(j+\ell)$. Thus, $\tau(x)[t(i+\ell-2)+1] = p(i+\ell, i+\ell+1)$ and

$\tau(x)[t(j+\ell-2)+1] = p(j+\ell, j+\ell+1)$. Since $x[i+\ell-1] = x[j+\ell-1]$ and, by our Assumption (3), $x[i+\ell] \prec x[j+\ell]$, $(x[i+\ell], x[i+\ell-1]) \prec (x[j+\ell], x[j+\ell-1])$, giving $p(i+\ell, i+\ell+1) \prec p(j+\ell, j+\ell+1)$, $\tau(x)[t(i+\ell)] \prec \tau(x)[t(j+\ell)]$, and ultimately, $\tau(x)[t(i+\ell-2)+1] = \tau(x)[t(i+\ell)] \prec \tau(x)[t(j+\ell)] = \tau(x)[t(j+\ell-2)+1]$. It follows that $\tau(x)[t(i)..m] \prec \tau(x)[t(j)..m]$.

□

Lemma 5 shows that τ -reduction preserves the proto-Lyndon property of certain proto-Lyndon substrings of x .

Lemma 5. Let $x = x[1..n]$, and let $\tau(x) = \tau(x)[1..m]$. Let $1 \leq i < j \leq n$. Let $x[i..j]$ be a proto-Lyndon substring of x , and let i be a black position.

Then, $\begin{cases} \tau(x)[t(i)..t(j)] \text{ is proto-Lyndon} & \text{if } j \text{ is black} \\ \tau(x)[t(i)..t(j-1)] \text{ is proto-Lyndon} & \text{if } j \text{ is white.} \end{cases}$

Proof. Let us first assume that j is black.

Since both i and j are black, $t(i)$ and $t(j)$ are defined. Let $i_1 = t(i)$, $j_1 = t(j)$, and consider k_1 , so that $i_1 < k_1 \leq j_1$. Let $k = b(k_1)$. Then, $t(k) = k_1$ and $i < k \leq j$, and so, $x[i..n] \prec x[k..n]$ by Lemma 1 as $x[i..j]$ is proto-Lyndon. It follows that $\tau(x)[t(i)..m] \prec \tau(x)[t(k)..m]$ by Lemma 4. Thus, $\tau(x)[i_1..m] \prec \tau(x)[k_1..m]$ for any $i_1 < k_1 \leq j_1$, and so, $\tau(x)[i_1..j_1]$ is proto-Lyndon by Lemma 1.

Now, let us assume that j is white.

Then, $j-1$ is black, and $x[i..j-1]$ is proto-Lyndon, so as in the previous case, $\tau(x)[t(i)..t(j-1)]$ is proto-Lyndon. □

Now, we can show that τ -reduction preserves some right-maximal Lyndon substrings.

Lemma 6. Let $x = x[1..n]$, and let $\tau(x) = \tau(x)[1..m]$. Let $1 \leq i < j \leq n$. Let $x[i..j]$ be a right-maximal Lyndon substring, and let i be a black position.

Then, $\begin{cases} \tau(x)[t(i)..t(j)] \text{ is a right-maximal Lyndon substring} & \text{if } j \text{ is black} \\ \tau(x)[t(i)..t(j-1)] \text{ is a right-maximal Lyndon substring} & \text{if } j \text{ is white.} \end{cases}$

Proof. Since $x[i..j]$ is Lyndon and hence proto-Lyndon, by Lemma 5, we know that $\tau(x)[t(i)..t(j)]$ is proto-Lyndon for j black, while for white j , $\tau(x)[t(i)..t(j-1)]$ is proto-Lyndon. Thus, in order to conclude that the respective strings are right-maximal Lyndon substrings, we only need to prove that the property (c) of Lemma 1 holds in both cases.

Since $x[i..j]$ is right-maximal Lyndon, either $j = n$ or $x[j+1..n] \prec x[i..n]$ by Lemma 1, giving $j = n$ or $x[j+1] \preceq x[i]$. Since $x[i..j]$ is Lyndon and hence unbordered, $x[i] \prec x[j]$. Thus, either $j = n$ or $x[j+1] \preceq x[i] \prec x[j]$.

If $j = n$, then there are two simple cases. If n is white, $n-1$ is black and $m = t(n-1)$, so $t(j-1) = m$, giving us (c) of Lemma 1 for $\tau(x)[t(i)..t(j-1)]$. On the other hand, if n is black, then $m = t(n)$, and so, $m = t(j)$ giving us (c) of Lemma 1 for $\tau(x)[t(i)..t(j)]$.

Thus, in the following, we can assume that $j < n$ and that $x[j+1] \preceq x[i] \prec x[j]$. We will proceed by discussing two possible cases, one where j is black and the other where j is white.

(1) Case: j is black.

We need to show that either $t(j) = m$ or $\tau(x)[t(i)..m] \succ \tau(x)[t(j)+1..m]$.

If $j = n$, then $t(j) = m$, and we are done. Thus, we can assume that $j < n$. We must show that $\tau(x)[t(j)+1..m] \prec \tau(x)[t(i)..m]$.

- (1a) Case: $x[j+1] \preceq x[j+2]$.
Then, $x[j] \succ x[j+1]$ and $x[j+1] \preceq x[j+2]$, and so, $j+1$ is black. It follows that $t(j)+1 = t(j+1)$. By Lemma 4, $\tau(x)[t(j+1)..m] \prec \tau(x)[t(i)..m]$ because $x[j+1..n] \prec x[i..n]$, thus $\tau(x)[t(j)+1..m] \prec \tau(x)[t(i)..m]$.
- (1b) Case: $x[j+1] \succ x[j+2]$.
Then, $x[j] \succ x[i] \succeq x[j+1] \succ x[j+2]$. It follows that the τ -pair $(j, j+1)$ is followed by a τ -pair $(j+2, j+3)$, and thus, $t(j)+1 = t(j+2)$. Thus, $(x[j+2], x[j+3]) \triangleleft (x[i], x[i+1]) \triangleleft (x[j], x[j+1])$; hence, $p(j+2, j+3) \prec p(i, i+1) \prec p(j, j+1)$. Since $\tau(x)[t(j)+1] = p(j+2, j+3)$, $\tau(x)[t(i)] = p(j, i+1)$, and $\tau(x)[t(j)] = p(j, j+1)$, it follows that $\tau(x)[t(j)+1] \prec \tau(x)[t(i)]$, and so, $\tau(x)[t(j)+1..m] \prec \tau(x)[t(i)..m]$.
- (2) Case: j is white.
We need to prove that $\tau(x)[t(j-1)+1..m] \prec \tau(x)[t(i)..m]$. Since j is white, necessarily both $j-1$ and $j+1$ are black and $t(j-1)+1 = t(j+1)$. By Lemma 5, $\tau(x)[t(i)..t(j-1)]$ is proto-Lyndon as both i and $j-1$ are black and $x[i..j-1]$ is proto-Lyndon. Since $x[i..n] \succ x[j+1..n]$ and both i and $j+1$ are black, by Lemma 4, we get $\tau(x)[t(i)..m] \succ \tau(x)[t(j+1)..m] = \tau(x)[t(j-1)+1..m]$.
□

Now, we are ready to tackle the proof of Theorem 2.

Proof of Theorem 2. Let $\mathcal{L}'_x[i] = j$ where i is black. Then, $t(i)$ is defined, and $x[i..j]$ is a right-maximal Lyndon substring of x . We proceed by analysis of the two possible cases of the label for the position j . Let $(*)$ denote the condition from the theorem, i.e.,

$$(*) \quad b(\mathcal{L}'_{\tau(x)}[t(i)]) = n \quad \text{or} \quad x[b(\mathcal{L}'_{\tau(x)}[t(i)]) + 1] \preceq x[i]$$

- (1) Case: j is black.

Then, by Lemma 6, $\tau(x)[t(i)..t(j)]$ is a right-maximal Lyndon substring of $\tau(x)$; hence, $\mathcal{L}'_{\tau(x)}[t(i)] = t(j)$. Therefore, $b(\mathcal{L}'_{\tau(x)}[t(i)]) = b(t(j)) = j = \mathcal{L}'_x[i]$. We have to also prove that the condition $(*)$ holds.

If $j = n$, then the condition $(*)$ holds. Therefore, assume that $j < n$. Since $x[i..j]$ is right-maximal, by Lemma 1, $x[j+1..n] \prec x[i..n]$, and so, $x[j+1] \preceq x[i]$. Then, $x[b(\mathcal{L}'_{\tau(x)}[t(i)]) + 1] = x[b(t(j)) + 1] = x[j+1] \preceq x[i]$.

- (2) Case: j is white.

Then, $j-1$ is black, and $\tau(x)[t(j-1)] = p(j-1, j)$. By Lemma 6, $\tau(x)[t(i)..t(j-1)]$ is a right-maximal Lyndon substring of $\tau(x)$; hence, $\mathcal{L}'_{\tau(x)}[t(i)] = t(j-1)$, so $b(\mathcal{L}'_{\tau(x)}[t(i)]) = b(t(j-1)) = j-1$, giving $b(\mathcal{L}'_{\tau(x)}[t(i)] + 1) = j$.

We want to show that the condition $(*)$ does not hold.

If $b(\mathcal{L}'_{\tau(x)}[t(i)]) = n$, then $j-1 = n$, which is impossible as $j \leq n$. Since $x[i..j]$ is Lyndon, $x[i] \prec x[j]$, and so, $x[i] \prec x[b(\mathcal{L}'_{\tau(x)}[t(i)] + 1)]$. Thus, Condition $(*)$ does not hold.

□

3.4. Computing $\mathcal{L}'_x$ from $\mathcal{L}'_{\tau(x)}$

Theorem 2 indicates how to compute the partial $\mathcal{L}'_x$ from $\mathcal{L}'_{\tau(x)}$. The procedure is given in Figure 3.

```

for  $i \leftarrow 1$  to  $n$ 
  if  $i = 1$  or  $(x[i-1] \succ x[i] \text{ and } x[i] \preceq x[i+1])$  then
    if  $x[b(\mathcal{L}'_{\tau(x)}[t(i)]) + 1] \preceq x[i]$  then
       $\mathcal{L}'_x[i] \leftarrow b(\mathcal{L}'_{\tau(x)}[t(i)])$ 
    else
       $\mathcal{L}'_x[i] \leftarrow b(\mathcal{L}'_{\tau(x)}[t(i)]) + 1$ 
  else
     $\mathcal{L}'_x[i] \leftarrow nil$ 

```

Figure 3. Computing the partial Lyndon array of the input string.

To compute the missing values, the partial array is processed from right to left. When a missing value at position i is encountered (note that it is recognized by $\mathcal{L}'_x[i] = nil$), the Lyndon array $\mathcal{L}'_x[i+1..n]$ is completely filled, and also, $\mathcal{L}'_x[i-1]$ is known. Recall that $\mathcal{L}'_x[i+1]$ is the ending position of the right-maximal Lyndon substring starting at the position $i+1$. In several cases, we can determine the value of $\mathcal{L}'_x[i]$ in constant time:

1. if $i = n$, then $\mathcal{L}'_x[i] = i$.
2. if $x[i] \succ x[i+1]$, then $\mathcal{L}'_x[i] = i$.
3. if $x[i] = x[i+1]$ and $\mathcal{L}'_x[i+1] = i+1$ and either $i+1 = n$ or $i+1 = \mathcal{L}'_x[i-1]$, then $\mathcal{L}'_x[i] = i$.
4. if $x[i] \prec x[i+1]$ and $\mathcal{L}'_x[i+1] = i+1$ and either $i+1 = n$ or $i+1 = \mathcal{L}'_x[i-1]$, then $\mathcal{L}'_x[i] = i+1$.
5. if $x[i] \preceq x[i+1]$ and $\mathcal{L}'_x[i+1] > i+1$ and either $\mathcal{L}'_x[i+1] = n$ or $\mathcal{L}'_x[i+1] = \mathcal{L}'_x[i-1]$, then $\mathcal{L}'_x[i] = \mathcal{L}'_x[i+1]$.

We call such points *easy*. All others will be referred to as *hard*. For a *hard* point i , it means that $x[i]$ is followed by at least two consecutive right-maximal Lyndon substrings before reaching either $\mathcal{L}'_x[i-1]$ or n , and we might need to traverse them all.

The **while** loop, seen in Figure 4's procedure, is the likely cause of the $\mathcal{O}(n \log(n))$ complexity. At first glance, it may seem that the complexity might be $\mathcal{O}(n^2)$; however, the doubling of the length of the string when a hard point is introduced actually trims it down to an $\mathcal{O}(n \log(n))$ worst-case complexity. See Section 3.5 for more details and Section 7 for the measurements and graphs.

```

 $\mathcal{L}'_x[n] \leftarrow n$ 
for  $i \leftarrow n-1$  downto 2
  if  $\mathcal{L}'[i] = nil$  then
    if  $x[i] \succ x[i+1]$  then
       $\mathcal{L}'[i] \leftarrow i$ 
    else
      if  $\mathcal{L}'[i-1] = i-1$  then
         $stop \leftarrow n$ 
      else
         $stop \leftarrow \mathcal{L}'[i-1]$ 
         $\mathcal{L}'[i] \leftarrow \mathcal{L}'[i+1]$ 
        while  $\mathcal{L}'[i] < stop$  do
          if  $x[i..\mathcal{L}'[i]] \prec x[\mathcal{L}'[i]+1..\mathcal{L}'[\mathcal{L}'[i]+1]]$  then
             $\mathcal{L}'[i] \leftarrow \mathcal{L}'[\mathcal{L}'[i]+1]$ 
          else
            break

```

Figure 4. Computing missing values of the Lyndon array of the input string.

Consider our running example from Figure 2. Since $\tau(x) = 021534$, we have $\mathcal{L}'_{\tau(x)}[1..6] = 6, 2, 6, 4, 6, 6$ giving $\mathcal{L}'_x[1..9] = 9, \bullet, 3, 9, \bullet, 6, 9, \bullet, 9$. Computing $\mathcal{L}'_x[8]$ is easy as $x[8] = x[9]$,

and so, $\mathcal{L}'_x[8] = 8$. $\mathcal{L}'_x[5]$ is more complicated and an example of a hard point: we can extend the right-maximal Lyndon substring from $\mathcal{L}'_x[6]$ to the left to 23, but no more, so $\mathcal{L}'_x[5] = 6$. Computing $\mathcal{L}'_x[2]$ is again easy as $x[2] = x[3]$, and so, $\mathcal{L}'_x[2] = 2$. Thus, $\mathcal{L}'_x[1..9] = 9, 2, 3, 9, 6, 6, 9, 8, 9$.

3.5. The Complexity of TRLA

To determine the complexity of the algorithm, we attach to each position i a counter $red[i]$ initialized to zero. Imagine a *hard* point j indicated by the following diagram:

A_1 represents the right-maximal Lyndon substring starting at the position $j+1$; A_2 represents the right-maximal Lyndon substring following immediately A_1 and so forth. To make j a *hard* point, $r \geq 2$ and $x[j] \preceq x[j+1]$. The value of $stop$ is determined by:

$$stop = \begin{cases} \mathcal{L}'_x[j-1] & \text{if } \mathcal{L}'_x[j-1] > j-1 \\ n & \text{otherwise} \end{cases}.$$

To determine the right-maximal Lyndon substring starting at the *hard* position j , we need first to check if A_1 can be left-extended by $x[j]$ to make jA_1 Lyndon; we are using abbreviated notation jA_1 for the substring $x[j..k]$ where $A_1 = x[j+1..k]$; in simple words, jA_1 represents the left-extension of A_1 by one position. If jA_1 is proto-Lyndon, we have to check whether A_2 can be left-extended by jA_1 to a Lyndon substring. If jA_1A_2 is Lyndon, we must continue until we check whether $jA_1A_2...A_{r-1}$ is Lyndon. If so, we must check whether $jA_1...A_r$ is Lyndon. We need not go beyond $stop$.

How do we check if $jA_1...A_k$ can left-extend A_{k+1} to a Lyndon substring? If $jA_1...A_k \succeq A_{k+1}$, we can stop, and $jA_1...A_k$ is the right-maximal Lyndon substring starting at position j . If $jA_1...A_k \prec A_{k+1}$, we need to continue. Since $stop$ is the last position of the right-maximal Lyndon substring at the position $j-1$ or n , we are assured to stop there. When comparing the substring $jA_1...A_k$ with A_{k+1} , we increment the counter $red[i]$ at every position of A_{k+1} used in the comparison. When done with the whole array, the value of $red[i]$ represents how many times i was used in various comparisons, for any position i .

Consider a position i that was used k times for $k \geq 4$, i.e., $red[i] = k$. In the next four diagrams and related text, the upper indices of A and C do not represent powers; they are just indices. The next diagram indicates the configuration when the counter $red[i]$ was incremented for the first time in the comparison of $j_1A_1^1...A_{r_1-1}^1$ and $A_{r_1}^1$ during the computation of the missing value $\mathcal{L}'_x[j_1]$ where:

$$stop_1 = \begin{cases} \mathcal{L}'_x[j_1-1] & \text{if } \mathcal{L}'_x[j_1-1] > j_1-1 \\ n & \text{otherwise} \end{cases}$$

The next diagram indicates the configuration when the counter $red[i]$ was incremented for the second time in the comparison of $j_2 A_1^2 \dots A_{r_2-1}^2$ and $A_{r_2}^2$ during the computation of the missing value $\mathcal{L}'_{\mathbf{x}}[j_2]$ where:

$$stop_2 = \begin{cases} \mathcal{L}'_{\mathbf{x}}[j_2-1] & \text{if } \mathcal{L}'_{\mathbf{x}}[j_2-1] > j_2-1 \\ n & \text{otherwise} \end{cases}$$

The next diagram indicates the configuration when the counter $red[i]$ was incremented for the third time in the comparison of $j_3 A_1^3 \dots A_{r_3-1}^3$ and $A_{r_3}^3$ during the computation of the missing value $\mathcal{L}'_{\mathbf{x}}[j_3]$ where:

$$stop_3 = \begin{cases} \mathcal{L}'_{\mathbf{x}}[j_3-1] & \text{if } \mathcal{L}'_{\mathbf{x}}[j_3-1] > j_3-1 \\ n & \text{otherwise} \end{cases}$$

The next diagram indicates the configuration when the counter $red[i]$ was incremented for the fourth time in the comparison of $j_4 A_1^4 \dots A_{r_4-1}^4$ and $A_{r_4}^4$ during the computation of the missing value $\mathcal{L}'_{\mathbf{x}}[j_4]$ where:

$$stop_4 = \begin{cases} \mathcal{L}'_{\mathbf{x}}[j_4-1] & \text{if } \mathcal{L}'_{\mathbf{x}}[j_4-1] > j_4-1 \\ n & \text{otherwise} \end{cases}$$

and so forth until the k -th increment of $red[i]$. Thus, if $red[i] = k$, then $n \geq 2^{k-1}(n_1+1) \geq 2^k$ as $n_1+1 \geq 2$. Thus, $n \geq 2^k$, and so, $k \leq \log(n)$. Thus, either $k < 4$ or $k \leq \log(n)$. Therefore, the overall complexity is $\mathcal{O}(n \log(n))$.

To show that the average case complexity is linear, we first recall that the overall complexity of TRLA is determined by the procedure filling the missing values. We showed above that there are at most $\log(n)$ missing values (*hard* positions) that cannot be determined in constant time. We overestimate the number of strings of length n over an alphabet of size Σ , $2 \leq \Sigma \leq n$, which will force a non-linear

computation, by assuming that every possible $\log(n)$ subset of indices with any possible letter assignment forces the worst performance. Thus, there are $\Sigma^n - \binom{n}{\log(n)} \Sigma^{\log(n)}$ strings that are processed in linear time, say with a constant K_1 , and there are $\binom{n}{\log(n)} \Sigma^{\log(n)}$ strings that are processed in the worst time, with a constant K_2 . Let $K = \max(K_1, K_2)$. Then, the average time is bounded by:

$$\begin{aligned} & \frac{(\Sigma^n - \binom{n}{\log(n)} \Sigma^{\log(n)}) Kn + \binom{n}{\log(n)} \Sigma^{\log(n)} Kn \log(n)}{\Sigma^n} = \\ & Kn + Kn \frac{\binom{n}{\log(n)} \Sigma^{\log(n)}}{\Sigma^n} (\log(n) - 1) \leq \\ & Kn + Kn \frac{\binom{n}{\log(n)} \Sigma^{\log(n)}}{\Sigma^n} \log(n) \leq \\ & Kn + Kn \frac{n^{\log(n)} \Sigma^{\log(n)}}{\log(n)! \Sigma^n} \log(n) \leq \\ & Kn + Kn \frac{n^{2\log(n)}}{2^n} \leq \\ & Kn + Kn = 2Kn \end{aligned}$$

for $n \geq 2^7$. The last step follows from the fact that $n^{2\log(n)} \leq 2^n$ for any $n \geq 2^7$.

The combinatorics of the processing is too complicated to ascertain whether the worst-case complexity is linear or not. We tried to generate strings that might give the worst performance. We used three different formulas to generate the strings, nesting the white indices that might require non-constant computation: the dataset `extreme_trla` of binary strings is created using the recursive formula $u_{k+1} = 00u_k0u_k$, using the first 100 shortest binary Lyndon strings as the start u_0 . The moment the size u_k exceeds the required length of the string, the recursion stops, and the string is trimmed to the required length. For the `extreme_trla1` dataset, we used the same approach with the formula $u_{k+1} = 000u_k00u_k$, and for the `extreme_trla2` dataset, we used the formula $u_{k+1} = 0000u_k00u_k$.

The space complexity of our C++ implementation is bounded by $9n$ integers. This upper bound is derived from the fact that a `Tau` object (see `Tau.hpp` [24]) requires $3n$ integers of space for a string of length n . Therefore, the first call to `TRLA` requires $3n$, the next recursive call at most $3\frac{2}{3}n$, the next recursive call at most $3(\frac{2}{3})^2n$, ...; thus, $3n + 3\frac{2}{3}n + 3(\frac{2}{3})^2n + 3(\frac{2}{3})^3n + \dots = 3n(1 + \frac{2}{3} + (\frac{2}{3})^2 + (\frac{2}{3})^3 + (\frac{2}{3})^4 + \dots) = 3n \frac{1}{1-\frac{2}{3}} = 9n$. However, it should be possible to bring it down to $6n$ integers.

4. The Algorithm BSLA

The purpose of this section is to present a linear algorithm *BSLA* for computing the Lyndon array of a string over an integer alphabet. The algorithm is based on a series of refinements of a list of groups of indices of the input string. The refinement is driven by a group that is already complete, and the refinement process makes the immediately preceding group also complete. In turn, this newly completed group is used as the driver of the next round of the refinement. In this fashion, the refinement proceeds from right to left until all the groups in the list are complete. The initial list of groups consists of the groups of indices with the same alphabet symbol. The section contains proper definitions of all these terms—group, complete group, and refinement. In the process of refinement, each newly created group is assigned a specific substring of the input string referred to as the context of the group. Throughout the process, the list of the groups is maintained in an increasing lexicographic order by their contexts. Moreover, at every stage, the contexts of all the groups are Lyndon substrings of x with an additional property that the contexts of the complete groups are right-maximal Lyndon substrings. Hence, when the refinement is completed, the contexts of all the groups in the list represent all the right-maximal Lyndon substrings of x . The mathematics of the process of refinement is necessary in

order to ascertain its correctness and completeness and to determine the worst-case complexity of the algorithm.

4.1. Notation and Basic Notions of BSLA

For the sake of simplicity, we fix a string $x = x[1..n]$ for the whole Section 4.1; all the definitions and the observations apply and refer to this x .

A **group** G is a non-empty set of indices of x . The group G is assigned a **context**, i.e., a substring $con(G)$ of x with the property that for any $i \in G$, $x[i..i+|con(G)|-1] = con(G)$. If $i \in G$, then $\mathcal{C}(i)$ denotes the occurrence of the context of G at the position i , i.e., the substring $\mathcal{C}(i) = x[i..i+|con(G)|-1]$. We say that a group G' is **smaller than** or **precedes** a group G'' if $con(G') \prec con(G'')$.

Definition 1. An ordered list of groups $\langle G_k, G_{k-1}, \dots, G_2, G_1 \rangle$ is a **group configuration** if:

- (C₁) $G_k \cup G_{k-1} \cup \dots \cup G_2 \cup G_1 = 1..n$;
- (C₂) $G_j \cap G_\ell = \emptyset$ for any $1 \leq \ell < j \leq k$;
- (C₃) $con(G_k) \prec con(G_{k-1}) \prec \dots \prec con(G_2) \prec con(G_1)$;
- (C₄) For any $j \in 1..k$, $con(G_j)$ is a Lyndon substring of x .

Note that (C₁) and (C₂) guarantee that $\langle G_k, G_{k-1}, \dots, G_2, G_1 \rangle$ is a disjoint partitioning of $1..n$. For $i \in \{1, \dots, n\}$, $gr(i)$ denotes the unique group to which i belongs, i.e., if $i \in G_t$, then $gr(i) = G_t$. Note that using this notation, $\mathcal{C}(i) = x[i..i+|con(gr(i))|-1]$.

The mapping **prev** is defined by $prev(i) = \max\{j < i \mid con(gr(j)) \prec con(gr(i))\}$ if such j exists, otherwise $prev(i) = nil$.

For a group G from a group configuration, we define an equivalence $\sim$ on G as follows: $i \sim j$ iff $gr(prev(i)) = gr(prev(j))$ or $prev(i) = prev(j) = nil$. The symbol $[i]_\sim$ denotes the class of equivalence $\sim$ that contains i , i.e., $[i]_\sim = \{j \in G \mid j \sim i\}$. If $prev(i) = nil$, then the class $[i]_\sim$ is called trivial. An interesting observation states that if G is viewed as an ordered set of indices, then a non-trivial $[i]_\sim$ is an interval:

Observation 7. Let G be a group from a group configuration for x . Consider an $i \in G$ such that $prev(i) \neq nil$. Let $j_1 = \min[i]_\sim$ and $j_2 = \max[i]_\sim$. Then, $[i]_\sim = \{j \in G \mid j_1 \leq j \leq j_2\}$.

Proof. Since $prev(j_1)$ is a candidate to be $prev(j)$, $prev(j) \neq nil$ and $prev(j_1) \leq prev(j) \leq prev(j_2) = prev(j_1)$, so $prev(j) = prev(j_1) = prev(j_2)$. $\square$

On each non-trivial class of $\sim$, we define a relation $\approx$ as follows: $i \approx j$ iff $|j-i| = |con(G)|$; in simple terms, it means that the occurrence $\mathcal{C}(i)$ of $con(G)$ is immediately followed by the occurrence $\mathcal{C}(j)$ of $con(G)$. The transitive closure of $\approx$ is a relation of equivalence, which we also denote by $\approx$. The symbol $[i]_\approx$ denotes the class of equivalence $\approx$ containing i , i.e., $[i]_\approx = \{j \in [i]_\sim \mid j \approx i\}$.

For each j from a non-trivial $[i]_\sim$, we define the **valence** by $val(j) = |[i]_\approx|$. In simple terms, $val(i)$ is the number of elements from $[i]_\sim$ that are $\approx i$. Thus, $1 \leq val(i) \leq |G|$.

Interestingly, if G is viewed as an ordered set of indices, then $[i]_\approx$ is a subinterval of the interval $[i]_\sim$:

Observation 8. Let G be a group from a group configuration for x . Consider an $i \in G$ such that $prev(i) \neq nil$. Let $j_1 = \min[i]_\approx$ and $j_2 = \max[i]_\approx$. Then, $[i]_\approx = \{j \in [i]_\sim \mid j_1 \leq j \leq j_2\}$.

Proof. We argue by contradiction. Assume that there is an $j \in [i]_\sim$ so that $j_1 < j < j_2$ and $j \notin [i]_\approx$. Take the minimal such j . Consider $j' = j - |con(G)|$. Then, $j' \in [i]_\sim$, and since, $j' < j$, $j' \in [i]_\approx$ due to the minimality of j . Therefore, $i \approx j' \approx j$, and so, $j \approx i$, a contradiction. $\square$

Definition 2. A group G is **complete** if for any $i \in G$, the occurrence $\mathcal{C}(i)$ of $\text{con}(G)$ is a right-maximal Lyndon substring of x .

A group configuration $\langle G_k, G_{k-1}, \dots, G_2, G_1 \rangle$ is **t -complete**, $1 \leq t \leq k$, if

- (C₅) the groups $G_t, \dots, G_1$ are complete;
- (C₆) the mapping prev is **proper** on G_t :
for any $i \in G_t$, if $\text{prev}(i) \neq \text{nil}$ and $v = \text{val}(i)$, then there are $i_1, \dots, i_v \in G_t$, $i \in \{i_1, \dots, i_v\}$, $\text{prev}(i) = \text{prev}(i_1) = \dots = \text{prev}(i_v)$, and so that $\mathcal{C}(\text{prev}(i))\mathcal{C}(i_1)\dots\mathcal{C}(i_v)$ is a prefix of $x[j..n]$;
- (C₇) the family $\{\mathcal{C}(i) \mid i \in 1..n\}$ is **proper**:
(a) if $\mathcal{C}(j)$ is a proper substring of $\mathcal{C}(i)$, i.e., $\mathcal{C}(j) \subsetneq \mathcal{C}(i)$, then $\text{con}(G_t) \prec \text{con}(\text{gr}(j))$,
(b) if $\mathcal{C}(i)$ is followed immediately by $\mathcal{C}(j)$, i.e., when $i + |\text{con}(\text{gr}(i))| = j$, and $\mathcal{C}(i) \prec \mathcal{C}(j)$, then $\text{con}(\text{gr}(j)) \preceq \text{con}(G_t)$;
- (C₈) the family $\{\mathcal{C}(i) \mid i \in 1..n\}$ has the **Monge** property, i.e., if $\mathcal{C}(i) \cap \mathcal{C}(j) \neq \emptyset$, then $\mathcal{C}(i) \subseteq \mathcal{C}(j)$ or $\mathcal{C}(j) \subseteq \mathcal{C}(i)$.

The condition (C₆) is all-important for carrying out the refinement process (see (R₃) below). The conditions (C₇) and (C₈) are necessary for asserting that the condition (C₆) is preserved during the refinement process.

4.2. The Refinement

For the sake of simplicity, we fix a string $x = x[1..n]$ for the whole Section 4.2; all the definitions, lemmas, and theorems apply and refer to this x .

Lemma 9. Let $\mathcal{A}_x = \{a_1, \dots, a_k\}$ and $a_1 \prec a_2 \prec \dots \prec a_k$. For $1 \leq \ell \leq k$, define $G_\ell = \{i \in 1..n \mid x[i] = a_{k+1-\ell}\}$ with context $a_{k+1-\ell}$. Then, $\langle G_k, \dots, G_1 \rangle$ is a one-complete group configuration.

Proof. (C₁), (C₂), (C₃), and (C₄) are straightforward to verify. To verify (C₅), we need to show that G_1 is complete. Any occurrence of a_k in x is a right-maximal Lyndon substring, so G_1 is complete. To verify (C₆), consider $j = \text{prev}(i)$ and $\text{val}(i) = v$ for $i \in G_1$. Consider any r such that $j < r < i$. If $x[r] \neq a_k$, then $\text{prev}(i) < r$, which contradicts the definition of prev . Hence, $x[r] = a_k$, and so, $x[j+1] = \dots = x[i] = \dots x[j+v+1] = a_k$, while $x[j] = a_q$ for some $q < k$. It follows that $x[j..n]$ has $a_q(a_k)^v$ as a prefix.

The condition (C₇(a)) is trivially satisfied as no $\mathcal{C}(i)$ can have a proper substring. If $\mathcal{C}(i)$ is immediately followed by $\mathcal{C}(j)$ and $\mathcal{C}(i) \prec \mathcal{C}(j)$, then $\mathcal{C}(i) = x[i]$, $j = i+1$, $\mathcal{C}(j) = x[i+1]$, and $x[i] \prec x[i+1]$. Then, $\text{con}(\mathcal{C}(j)) = x[i+1] \preceq a_k = \text{con}(G_1)$, so (C₇(b)) is also satisfied.

To verify (C₈), consider $\mathcal{C}(i) \cap \mathcal{C}(j) \neq \emptyset$. Then, $\mathcal{C}(i) = x[i] = x[j] = \mathcal{C}(j)$. $\square$

Let $\langle G_k, \dots, G_t, \dots, G_1 \rangle$ by a t -complete group configuration. The refinement is driven by the group G_t , and it might only partition the groups that precede it, i.e., the groups $G_k, \dots, G_{t+1}$, while the groups $G_t, \dots, G_1$ remain unchanged.

- (R₁) Partition G_t into classes of the equivalence $\sim$.
 $G_t = [i_1]_{\sim} \cup [i_2]_{\sim} \cup \dots \cup [i_p]_{\sim} \cup X$ where $X = \{i \in G_t \mid \text{prev}(i) = \text{nil}\}$ may be possibly empty and $i_1 < i_2 < \dots < i_p$.
- (R₂) Partition every class $[i_\ell]_{\sim}$, $1 \leq \ell \leq p$, into classes of the equivalence $\approx$.
 $[i_\ell]_{\sim} = [j_{\ell,1}]_{\approx} \cup [j_{\ell,2}]_{\approx} \cup \dots \cup [j_{\ell,m_\ell}]_{\approx}$ where $\text{val}(j_{\ell,1}) < \text{val}(j_{\ell,2}) < \dots < \text{val}(j_{\ell,m_\ell})$.
- (R₃) Therefore, we have a list of classes in this order: $[j_{1,1}]_{\approx}, [j_{1,2}]_{\approx}, \dots, [j_{1,m_1}]_{\approx}, [j_{2,1}]_{\approx}, [j_{2,2}]_{\approx}, \dots, [j_{2,m_2}]_{\approx}, \dots, [j_{p,1}]_{\approx}, [j_{p,2}]_{\approx}, \dots, [j_{p,m_p}]_{\approx}$. This list is processed from left to right. Note that for each $i \in [j_{\ell,k}]_{\approx}$, $\text{prev}(i) \in \text{gr}(j_{\ell,k})$, and $\text{val}(i) = \text{val}(j_{\ell,k})$.
For each $j_{\ell,k}$, move all elements $\{\text{prev}(i) \mid i \in [j_{\ell,k}]_{\approx}\}$ from the group $\text{gr}(\text{prev}(j_{\ell,k}))$ into a new

group H , place H in the list of groups right after the group $gr(prev(j_{\ell,k}))$, and set its context to $con(gr(prev(j_{\ell,k})))con(gr(j_{\ell,k}))^{val(j_{\ell,k})}$. (Note, that this “doubling of the contexts” is possible due to (C_6)). Then, update $prev$:

All values of $prev$ are correct except possibly the values of $prev$ for indices from H . It may be the case that for $i \in H$, there is $i' \in gr(j_{\ell,k})$, so that $prev(i) < i'$, so $prev(i)$ must be reset to the maximal such i' . (Note that before the removal of H from $gr(j_{\ell,k})$, the index i' was not eligible to be considered for $prev(i)$ as i and i' were both from the same group.)

Theorem 3 shows that having a t -complete group configuration $\langle G_k, \dots, G_{t+1}, G_t, \dots, G_1 \rangle$ and refining it by G_t , then the resulting system of groups is a $(t+1)$ -complete group configuration. This allows carrying out the refinement in an iterative fashion.

Theorem 3. Let $Conf = \langle G_k, \dots, G_{t+1}, G_t, \dots, G_1 \rangle$ be a t -complete group configuration, $1 \leq t$. After performing the refinement of $Conf$ by group G_t , the resulting system of groups denoted as $Conf'$ is a $(t+1)$ -complete group configuration.

Proof. We carry the proof in a series of claims. The symbols $gr()$, $con()$, $\mathcal{C}()$, $prev()$, and $val()$ denote the functions for $Conf$, while $gr'()$, $con'()$, $\mathcal{C}'()$, $prev'()$, and $val'()$ denote the functions for $Conf'$. When a group G_{t+1} is partitioned, a part of it is moved as the next group in the list, and we call it H_{t+1} ; thus, $G_{t+1} \prec H_{t+1} \prec G_t$. For details, please see (R_3) above.

Claim 1. $Conf'$ is a group configuration, i.e., (C_1) , (C_2) , (C_3) , and (C_4) for $Conf'$ hold.

Proof of Claim 1. (C_1) and (C_2) follow from the fact that the process is a refinement, i.e., a group is either preserved as is or is partitioned into two or more groups. The doubling of the contexts in Step (R_3) guarantees that the increasing order of the contexts is preserved, i.e., (C_3) holds. For any $j \in G_t$ so that $j = prev(i) \neq nil$, $con(gr(prev(j)))$ is Lyndon, and $con(gr(j))$ is also Lyndon, while $con(gr(prev(j))) \prec con(gr(j))$, so $con(gr(prev(j)))con(gr(j))^{val(j)}$ is Lyndon as well; thus, (C_4) holds.

To illustrate the concatenation: let us call $con(gr(prev(j)))$ as A and $con(gr(j))$ as B , and let $val(j) = m$, then we know that A is Lyndon and B is Lyndon and $A \prec B$; so, AB^m is clearly Lyndon as if A and B were letters. $\square$

This concludes the proof of Claim 1.

Claim 2. $\{\mathcal{C}'(i) \mid i \in 1..n\}$ is proper and has the Monge property, i.e., (C_7) and (C_8) for $Conf'$ hold.

Proof of Claim 2. Consider $\mathcal{C}'(i)$ for some $i \in 1..n$. There are two possibilities:

1. $\mathcal{C}'(i) = \mathcal{C}(i)$ or
2. $\mathcal{C}'(i) = \mathcal{C}(i)\mathcal{C}(i_1)\dots\mathcal{C}(i_v)$, for some $i_1, i_2, \dots, i_v \in G_t$, so that for any $1 \leq \ell \leq v$, $i = prev(i_\ell)$, $\mathcal{C}(i_\ell) = con(G_t)$, $v = val(i_\ell)$ and for any $1 \leq \ell < k$ and $i_{\ell+1} = i_\ell + |con(G_t)|$. Note that $con(gr(i)) \prec con(G_t)$.

Consider $\mathcal{C}'(i)$ and $\mathcal{C}'(j)$ for some $1 \leq i < j \leq n$.

1. Case $\mathcal{C}'(i) = \mathcal{C}(i)$ and $\mathcal{C}'(j) = \mathcal{C}(j)$.
 - (a) Show that $(C_7(a))$ holds.
If $\mathcal{C}'(j) \subsetneq \mathcal{C}'(i)$, then $\mathcal{C}(j) \subsetneq \mathcal{C}(i)$, and so, by $(C_7(a))$ for $Conf$, $con(G_t) \prec con(gr(j))$, and thus, $con'(H_{t+1}) \prec con(G_t) \prec con(gr(j)) = con'(gr'(j))$. Therefore, $(C_7(a))$ for $Conf'$ holds.

- (b) Show that (C_8) holds. If $\mathcal{C}'(i) \cap \mathcal{C}'(j) \neq \emptyset$, then $\mathcal{C}(i) \cap \mathcal{C}(j) \neq \emptyset$, so $\mathcal{C}(j) \subseteq \mathcal{C}(i)$, and so, $\mathcal{C}'(j) \subseteq \mathcal{C}'(i)$; so, (C_8) for Conf' holds.
2. Case $\mathcal{C}'(i) = \mathcal{C}(i)$ and $\mathcal{C}'(j) = \mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w)$, where $w = \text{val}(j_1)$, $\mathcal{C}(j_1) = \dots = \mathcal{C}(j_w) = \text{con}(G_t)$, and $j_1 \approx \dots \approx j_w$.
- (a) Show that $(C_7(a))$ holds.
If $\mathcal{C}'(j) \subsetneq \mathcal{C}'(i)$, then $\mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w) \subsetneq \mathcal{C}(i)$; hence, $\mathcal{C}(j) \subsetneq \mathcal{C}(i)$, and so, by $(C_7(a))$ for Conf , $\text{con}(G_t) \prec \text{con}(gr(j))$. By the t -completeness of Conf , $\mathcal{C}(j)$ is a right-maximal Lyndon substring, a contradiction with $\mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w)$ being Lyndon. This is an impossible case.
- (b) Show that (C_8) holds.
If $\mathcal{C}'(i) \cap \mathcal{C}'(j) \neq \emptyset$, then $\mathcal{C}(j) \subseteq \mathcal{C}(i)$ by (C_8) for Conf . By $(C_7(a))$ for Conf , $\mathcal{C}(j)$ cannot be a suffix of $\mathcal{C}(i)$ as $\text{con}(gr(j)) \prec \text{con}(G_t)$. Hence, $\mathcal{C}(i) \cap \mathcal{C}(j_1) \neq \emptyset$, and so, $\mathcal{C}(j)\mathcal{C}(j_1) \subseteq \mathcal{C}(i)$; and since $\mathcal{C}(j_1)$ cannot be a suffix of $\mathcal{C}(i)$ as $gr(j_1) = G_t$, it follows that $\mathcal{C}(i) \cap \mathcal{C}(j_2) \neq \emptyset$, ..., ultimately giving $\mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w) \subseteq \mathcal{C}(i)$. Therefore, (C_8) for Conf' holds.
3. Case $\mathcal{C}'(i) = \mathcal{C}(i)\mathcal{C}(i_1)..\mathcal{C}(i_v)$ and $\mathcal{C}'(j) = \mathcal{C}(j)$, where $v = \text{val}(i_1)$, $\mathcal{C}(i_1) = \dots = \mathcal{C}(i_v) = \text{con}(G_t)$, and $i_1 \approx \dots \approx i_v$.
- (a) Show that $(C_7(a))$ holds.
If $\mathcal{C}'(j) \subsetneq \mathcal{C}'(i)$, then either $\mathcal{C}(j) \subsetneq \mathcal{C}(i)$, which implies by $(C_7(a))$ for Conf that $\text{con}(G_t) \prec \text{con}(gr(j))$, giving $\text{con}'(H_{t+1}) \prec \text{con}'(G_t) = \text{con}(G_t) \prec \text{con}(gr(j)) = \text{con}'(gr'(j))$, or $\mathcal{C}(j) \subseteq \mathcal{C}(i_\ell)$ for some $1 \leq \ell \leq v$. If $\mathcal{C}(j) = \mathcal{C}(i_\ell)$, then $gr(j) = gr(i_\ell) = G_t$, giving $\text{con}'(H_{t+1}) \prec \text{con}(G_t) = \text{con}(gr(j))$. Therefore $(C_7(a))$ for Conf' holds.
- (b) Show that (C_8) holds.
Let $\mathcal{C}'(i) \cap \mathcal{C}'(j) \neq \emptyset$. Consider $\mathcal{D} = \{i_\ell \mid 1 \leq \ell \leq v \text{ and } \mathcal{C}(j) \cap \mathcal{C}(i_\ell) \neq \emptyset\}$. Assume that $\mathcal{D} \neq \emptyset$:
By (C_8) for Conf , either $\mathcal{C}(j) \subseteq \bigcup_{i_\ell \in \mathcal{D}} \mathcal{C}(i_\ell) \subseteq \mathcal{C}'(i)$, and we are done, or $\bigcup_{i_\ell \in \mathcal{D}} \mathcal{C}(i_\ell) \subseteq \mathcal{C}(j)$. Let i_k be the smallest element of $\mathcal{D}$. Since $\mathcal{C}(i_k)$ cannot be the prefix of $\mathcal{C}(j)$, it means that $i_k = i_1$. Since $\mathcal{C}(i_1)$ cannot be a prefix of $\mathcal{C}(j)$, it means that $\mathcal{C}(i) \cap \mathcal{C}(j) \neq \emptyset$, and so, $\mathcal{C}(j) \subseteq \mathcal{C}(i)$, which contradicts the fact that $\mathcal{C}(j) \subseteq \bigcup_{i_\ell \in \mathcal{D}} \mathcal{C}(i_\ell) \subseteq \mathcal{C}'(i)$.
Assume that $\mathcal{D} = \emptyset$:
Then, $\mathcal{C}(i) \cap \mathcal{C}(j) \neq \emptyset$, and so, by (C_8) for Conf , $\mathcal{C}(j) \subseteq \mathcal{C}(i) \subseteq \mathcal{C}'(i)$ as $i < j$.
4. Case $\mathcal{C}'(i) = \mathcal{C}(i)\mathcal{C}(i_1)..\mathcal{C}(i_v)$ and $\mathcal{C}'(j) = \mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w)$, where $v = \text{val}(i_1)$, $\mathcal{C}(i_1) = \dots = \mathcal{C}(i_v) = \text{con}(G_t)$, and $i_1 \approx \dots \approx i_v$ and where $v = \text{val}(j_1)$, $\mathcal{C}(j_1) = \dots = \mathcal{C}(j_w) = \text{con}(G_t)$, and $j_1 \approx \dots \approx j_w$.
- (a) Show that $(C_7(a))$ holds.
Let $\mathcal{C}'(j) \subsetneq \mathcal{C}'(i)$. Then, either $\mathcal{C}(j) \subseteq \mathcal{C}(i)$, and so, $\text{con}(G_t) \prec \text{con}(gr(j))$, implying that $\mathcal{C}(j)$ is maximal contradicting $\mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w)$ being Lyndon. Thus, $\mathcal{C}(j) \subsetneq \mathcal{C}(i_\ell)$ for some $1 \leq \ell \leq v$. However, then, $\text{con}(G_t) \prec \text{con}(gr(j))$, implying that $\mathcal{C}(j)$ is maximal, again a contradiction. This is an impossible case.
- (b) Show that (C_8) holds.
Let $\mathcal{C}'(i) \cap \mathcal{C}'(j) \neq \emptyset$. Let us first assume that $\mathcal{C}(i) \cap \mathcal{C}(j) \neq \emptyset$. Then, $\mathcal{C}(j) \subseteq \mathcal{C}(i)$. Since $\mathcal{C}(j)$ cannot be a suffix of $\mathcal{C}(i)$, it follows that $\mathcal{C}(i) \cap \mathcal{C}(j_1) \neq \emptyset$. Therefore, $\mathcal{C}(j)\mathcal{C}(j_1) \subseteq \mathcal{C}(i)$. Repeating this argument leads to $\mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w) \subseteq \mathcal{C}(i)$, and we are done.
Therefore, assume that $\mathcal{C}(i) \cap \mathcal{C}(j) = \emptyset$. Let $1 \leq \ell \leq v$ be the smallest such that $\mathcal{C}(i_\ell) \cap \mathcal{C}(j) \neq \emptyset$. Such an ℓ must exist. Then, $i_\ell \leq j$. If $i_\ell = j$, then either $\mathcal{C}(i_\ell)$ is a prefix of $\mathcal{C}(j)$ or vice versa, both impossibilities; hence, $i_\ell < j$. Repeating the same arguments as for i , we get that $\mathcal{C}(j)\mathcal{C}(j_1)..\mathcal{C}(j_w) \subseteq \mathcal{C}(i_\ell)$, and so, we are done.

It remains to show that $(C_7(b))$ for $Conf'$ holds.

Consider $C'(i)$ immediately followed by $C'(j)$ with $C'(i) \prec C'(j)$.

1. Assume that $gr'(j) \in \{G_{t-1}, \dots, G_1\}$.
Then, $con(G_t) = con'(G_t)$, $gr(j) = gr'(j)$, and $con(gr(j)) = con'(gr'(j))$. If $C'(i) = C(i)$, then $C(i) \prec C(j)$, and $C(i)$ is immediately followed by $C(j)$, so by $(C_7(b))$ for $Conf$, we have a contradiction. Thus, $C'(i) = C(i)C(i_1) \dots C(i_v)$ for $v = val(i)$ and $con(gr(i_v)) = con(G_t) \prec con(gr(j))$, and $C(i_v)$ is immediately followed by $C(j)$, a contradiction by $(C_7(b))$ for $Conf$.
2. Assume that $gr'(j) = G_t$.
Then, the group $gr(i)$ is partitioned when refining by G_t , and so, $C'(i) = con'(gr'(i)) = con(gr(i))C(j)^v$ for $v = val(j)$. Since $C'(i)$ is immediately followed by $C'(j) = con(G_t)$, we have again a contradiction, as it implies that $val(j) = v+1$.

□

This concludes the proof of Claim 2.

Claim 3. *The function $prev'$ is proper on H_{t+1} , i.e., (C_6) for $Conf'$ holds.*

Proof of Claim 3. Let $j = prev'(i)$ and $i \in H_{t+1}$ with $val'(i) = v$. Then, $|[i]_{\approx}| = v$, and so, $[i]_{\approx} = \{i_1, \dots, i_v\}$, where $i_1 < i_2 < \dots < i_v$. Hence, $i_1, \dots, i_v \in H_{t+1}$, $C'(i_1) = \dots = C'(i_v) = con'(H_{t+1})$, and $j = prev'(i) = prev'(i_1) = \dots = prev'(i_v)$, and so, $j < i_1$. It remains to show that $C'(j)C'(i_1) \dots C'(i_v)$ is a prefix of $x[j..n]$. It suffices to show that $C'(j)$ is immediately followed by $C'(i_1)$.

If $C'(j) \cap C'(i_1) \neq \emptyset$, then by the Monge property (C_8) , $C'(i_1) \subseteq C'(j)$ as $j < i_1$, and so, by $(C_7(a))$, $con'(H_{t+1}) \prec con'(gr'(i_1)) = con'(H_{t+1})$, a contradiction.

Thus, $C'(j) \cap C'(i_1) = \emptyset$. Set $j_1 = j + |con'(gr'(j))|$. It follows that $j_1 \leq i_1$. Assume that $j_1 < i_1$. Since $j = prev'(i_1)$ and $j < i_1$, $con'(gr'(j_1)) \succeq con'(gr'(i_1)) = con'(H_{t+1})$. Since $j_1 \notin H_{t+1}$, $con'(gr'(j_1)) \succ con'(H_{t+1})$. Consider $C'(j_1)$. If $C'(j_1) \cap C'(i_1) \neq \emptyset$, then by (C_8) , $C'(i_1) \subseteq C'(j_1)$, and so, by $(C_7(a))$, $con'(H_{t+1}) \prec con'(gr'(i_1)) = con'(H_{t+1})$, a contradiction. Thus, $C'(j_1) \cap C'(i_1) = \emptyset$. Since $C'(j_1)$ immediately follows $C'(j)$, by $(C_7(b))$, $con'(gr'(j_1)) \preceq con'(H_{t+1})$, a contradiction. Therefore, $j_1 = i_1$, and so, $prev'$ is proper on H_{t+1} . □

This concludes the proof of Claim 3.

Claim 4. *H_{t+1} is a complete group, i.e., (C_5) for $Conf'$ holds.*

Proof of Claim 4. Assume that there is $i \in H_{t+1}$ so that $C'(i)$ is not maximal, i.e., for some $k \geq i + |con'(H_{t+1})|$, $x[i..k]$ is a right-maximal Lyndon substring of x .

Either $k = n$ and so $con'(gr'(k)) = x[k]$, and so, $C'(k)$ is a suffix of $x[i..k]$, or $k < n$, and then, $x[k+1] \prec x[k]$, since $x[k+1] \preceq x[k]$ implies that $x[i..k+1]$ is Lyndon, a contradiction with the right-maximality of $x[i..k]$. Consider $C'(k)$, then $C'(k) \subseteq x[i..k]$, and so, $C'(k) = x[k]$.

Therefore, there is j_1 so that $i + |con'(H_{t+1})| \leq j_1 \leq k$, and $C'(j_1)$ is a suffix of $x[i..k]$. Take the smallest such j_1 . If $j_1 = i + |con'(H_{t+1})|$, then $C'(i) \prec C'(j_1)$ as $x[i..k] = C'(i)C'(j_1)$ is Lyndon. By $(C_7(b))$, $C'(j_1) \preceq con'(H_{t+1})$, so we have $con'(H_{t+1}) = C'(i) \prec C'(j_1) \preceq con'(H_{t+1})$, a contradiction.

Therefore, $j_1 > i + |con'(H_{t+1})|$. Consider $x[j_1-1]$. If $x[j_1-1] \preceq x[j_1]$, $x[j_1-1..k]$ is Lyndon, and since $x[j_1..k] = C'(j_1)$, $x[j_1-1..k]$ would be a context of $gr'(j_1-1)$, this contradicts the fact j_1 was chosen to be the smallest such one. Therefore, $x[j_1-1] \succ x[j_1]$, and so, $con'(gr'(j_1-1)) = x[j_1-1]$. Thus, there is j_2 , $i + |con'(H_{t+1})| \leq j_2 < j_1 \leq k$, and $C'(j_2)$ is a suffix of $x[i..j_1-1]$. Take the smallest such j_2 . If $C'(j_2) \prec C'(j_1)$, then by $(C_7(b))$, $C'(j_1) \preceq con'(H_{t+1})$, a contradiction. Hence, $C'(j_2) \succeq C'(j_1)$. If $j_2 = i + i + |con'(H_{t+1})|$, then $x[i..k] = C'(i)C'(j_2)C'(j_1)$, and so, by $(C_7(b))$, $C'(j_2) \preceq con'(H_{t+1})$, a contradiction. Hence, $i + |con'(H_{t+1})| < j_2$.

The same argument done for j_2 can now be done for j_3 . We end up with $i + |con'(H_{t+1})| \leq j_3 < j_2 < j_1 \leq k$ and with $C'(j_3) \succeq C'(j_2) \succeq C'(j_1) \succ con'(H_{t+1})$. If $i + |con'(H_{t+1})| = j_3$,

then we have a contradiction, so $i + |\text{con}'(H_{t+1})| < j_3$. These arguments can be repeated only finitely many times, and we obtain $i + |\text{con}'(H_{t+1})| = j_\ell < j_{\ell-1} < \dots < j_2 < j_1 \leq k$ so that $x[i..k] = C'(i)C'(j_\ell)C'(j_{\ell-1})\dots C'(j_2)C'(j_1)$, which is a contradiction.

Therefore, our initial assumption that $C'(i)$ is not maximal always leads to a contradiction. $\square$

This concludes the proof of Claim 4.

The four claims show that all the conditions $(C_1) \dots (C_8)$ are satisfied for Conf' , and that proves Theorem 3. $\square$

As the last step, we show that when the process of refinement is completed, all right-maximal Lyndon substrings of x are identified and sorted via the contexts of the groups of the final configuration.

Theorem 4.

Let $\text{Conf}_1 = \langle G_{k_1}^1, G_{k_1-1}^1, \dots, G_2^1, G_1^1 \rangle$ with $gr_1()$, $con_1()$, $C_1()$, $prev_1()$, and $val_1()$ be the initial 1-complete group configuration from Lemma 9.

Let $\text{Conf}_2 = \langle G_{k_2}^2, G_{k_2-1}^2, \dots, G_2^2, G_1^2 \rangle$ with $gr_2()$, $con_2()$, $C_2()$, $prev_2()$, and $val_2()$ be the 2-complete group configuration obtained from Conf_1 through the refinement by the group G_1^1 .

Let $\text{Conf}_3 = \langle G_{k_3}^3, G_{k_3-1}^3, \dots, G_2^3, G_1^3 \rangle$ with $gr_3()$, $con_3()$, $C_3()$, $prev_3()$, and $val_3()$ be the 3-complete group configuration obtained from Conf_2 through the refinement by the group G_2^2 .

...

Let $\text{Conf}_r = \langle G_{k_r}^r, G_{k_r-1}^r, \dots, G_2^r, G_1^r \rangle$ with $gr_r()$, $con_r()$, $C_r()$, $prev_r()$, and $val_r()$ be the r -complete group configuration obtained from Conf_{r-1} through the refinement by the group G_{r-1}^{r-1} . Let Conf_r be the final configuration after the refinement runs out.

Then, $x[i..k]$ is a right-maximal Lyndon substring of x iff $x[i..k] = C_r(i) = con_r(gr_r(i))$.

Proof. That all the groups of Conf_r are complete follows from Theorem 3, and hence, every $C_r(i)$ is a right-maximal Lyndon string. Let $x[i..k]$ be a right-maximal Lyndon substring of x . Consider $C_r(i)$; since it is maximal, it must be equal to $x[i..k]$. $\square$

4.3. Motivation for the Refinement

The process of refinement is in fact a process of the gradual revealing of the Lyndon substrings, which we call the *water draining method*:

- lower the water level by one;
- extend the existing Lyndon substrings; the revealed letters are used to extend the existing Lyndon substrings where possible, or became Lyndon substrings of length one otherwise;
- consolidate the new Lyndon substrings; processed from the right, if several Lyndon substrings are adjacent and can be joined to a longer Lyndon substring, they are joined.

The diagram in Figure 5 and the description that follows it illustrate the method for a string 011023122. The input string is visualized as a curve, and the height at each point is the value of the letter at that position.

In Figure 5, we illustrate the process:

- We start with the string 011023122 and a full tank of water.
- We drain one level; only 3 is revealed; there is nothing to extend, nothing to consolidate.
- We drain one more level, and three 2's are revealed; the first 2 extends 3 to 23, and the remaining two 2's form Lyndon substrings 2 of length one; there is nothing to consolidate.
- We drain one more level, and three 1's are revealed; the first two 1's form Lyndon substrings 1 of length one; the third 1 extends 22 to 122; there is nothing to consolidate.

- (5) We drain one more level, and two 0's are revealed; the first 0 extends 11 to 011; the second 0 extends 23 to 023; in the consolidation phase, 023 is joined with 122 to form a Lyndon substring 023122, and then, 011 is joined with 023122 to form a Lyndon substring 011023122.

Figure 5. The water draining method for 011023122. Stages (1)–(6) explained in the text.

Therefore, during the process, the following right-maximal Lyndon substrings were identified: 3 at Position 6, 23 at Position 5, 2 at Positions 8 and 9, 1 at Positions 2 and 3, 122 at Position 7, 023 at Position 4, and finally, 011023122 at Position 1. Note that all positions are accounted for; we really have all right-maximal Lyndon substrings of the string 011023122.

In Figure 6, we present an illustrative example for the string 011023122, where the arrows represent the *prev* mapping shown only on the group used for the refinement. The groups used for the refinement are indicated by the bold font.

0	1	1	0	2	3	1	2	2
1	2	3	4	5	6	7	8	9

$$G_0 = \{1, 4\} \quad G_1 = \{2, 3, 7\} \quad G_2 = \{5, 8, 9\} \quad G_3 = \{6\}$$

$$G_0 = \{1, 4\} \quad G_1 = \{2, 3, 7\} \quad G_2 = \{8, 9\} \quad G_{23} = \{5\} \quad G_3 = \{6\}$$

$$G_0 = \{1\} \quad G_{023} = \{4\} \quad G_1 = \{2, 3, 7\} \quad G_2 = \{8, 9\} \quad G_{23} = \{5\} \quad G_3 = \{6\}$$

$$G_0 = \{1\} \quad G_{023} = \{4\} \quad G_1 = \{2, 3\} \quad G_{122} = \{7\} \quad G_2 = \{8, 9\} \quad G_{23} = \{5\} \quad G_3 = \{6\}$$

$$G_0 = \{1\} \quad G_{023122} = \{4\} \quad G_1 = \{2, 3\} \quad G_{122} = \{7\} \quad G_2 = \{8, 9\} \quad G_{23} = \{5\} \quad G_3 = \{6\}$$

$$G_{011} = \{1\} \quad G_{023122} = \{4\} \quad G_1 = \{2, 3\} \quad G_{122} = \{7\} \quad G_2 = \{8, 9\} \quad G_{23} = \{5\} \quad G_3 = \{6\}$$

$$G_{011023122} = \{1\} \quad G_{023122} = \{4\} \quad G_1 = \{2, 3\} \quad G_{122} = \{7\} \quad G_2 = \{8, 9\} \quad G_{23} = \{5\} \quad G_3 = \{6\}$$

Figure 6. Group refinement for 011023122.

4.4. The Complexity of BSLA

The computation of the initial configuration can be done in linear time. To compute the initial value of *prev* in linear time, a stack-based approach similar to the NSV algorithm is used. Since all groups are non-empty, there can never be more groups than n . Theorem 3 is at the heart of the algorithm. The refinement by the last completed group is linear in the size of the group, including the

update of *prev*. Therefore, the overall worst-case complexity of *BSLA* is linear in the length of the input string.

5. Data and Measurements

Initially, computations were performed on the Department of Computing and Software's *moore* server; memory: 32 GB (DDR4 @ 2400 MHz), CPU: $8 \times$ Intel Xeon E5-2687W v4 @ 3.00 GHz, OS: Linux Version 2.6.18-419.el5 (gcc Version 4.1.2 and Red Hat Version 4.1.2-55). To verify correctness, new randomized data were produced and computed independently on the University of Toronto Mississauga's *octolab* cluster; memory: $8 \times$ 32 GB (DDR4 @ 3200 MHz), CPU: $8 \times$ AMD Ryzen Threadripper 1920X (12-Core) @ 4.00 GHz, OS: Ubuntu 16.04.6 LTS (gcc Version 5.4.0). The results of both were extremely similar, and those reported herein are those generated using the *moore* server. All the programs were compiled without any additional level of optimization (i.e., neither `-O1`, nor `-O2`, nor `-O3` flags were specified for the compilation). The CPU time was measured in clock ticks with 1,000,000 clock ticks per second. Since the execution time was negligible for short strings, the processing of the same string was repeated several times (the repeat factor varied from 10^6 , for strings of length 10, to one, for strings of length 5×10^6), resulting in a higher precision. Thus, for graphing, the logarithmic scale was used for both, x -axis representing the length of the strings and y -axis representing the time. We used four categories of randomly generated datasets:

- (1) **bin**
random strings over an integer alphabet with exactly two distinct letters (kind of binary strings).
- (2) **dna**
random strings over an integer alphabet with exactly four distinct letters (kind of random DNA strings).
- (3) **eng**
random strings over an integer alphabet with exactly 26 distinct letters (kind of random English).
- (4) **int**
random strings over an integer alphabet (i.e., over the alphabet $\{0, \dots, n-1\}$).

Each dataset contains 100 randomly generated strings of the same length. For each category, there were datasets for lengths 10, 50, 10^2 , 5×10^2 , ..., 10^5 , 5×10^5 , 10^6 , and 5×10^6 . The minimum, average, and maximum times for each dataset were computed. Since the variance for each dataset was minimal, the results for minimum times and the results for maximum times completely mimicked the results for the average times, so we only present the averages here.

Tables 1–4 and the graphs in Figures 7–10 from Section 7 clearly indicate that the performance of the three algorithms is linear and virtually indistinguishable. We expected *IDL*A and *TRL*A to exhibit linear behaviour on random strings as such strings tend to have many, but short right-maximal Lyndon substrings. However, we did not expect the results to be so close.

Despite the fact that *IDL*A performed in linear time on the random strings, it is relatively easy to force it into its worst quadratic performance. The dataset *extreme_idla* contains individual strings 0123... $n-1$ of the required lengths. Table 5 and the graph in Figure 11 from Section 7 show this clearly.

In Section 3.5, we describe how the three datasets, *extreme_trla*, *extreme_trla1*, and *extreme_trla2*, are generated and why. The results of experimenting with these datasets do not suggest that the worst-case complexity for *TRL*A is $\mathcal{O}(n \log(n))$. Yet again, the performances of the three algorithms are linear and virtually indistinguishable; see Tables 6–8 and the graphs in Figures 12–14 in Section 7.

6. Conclusions and Future Work

We present two novel algorithms for computing right-maximal Lyndon substrings. The first one, *TRL*A, has a simple implementation with a complicated theory behind it. Its average time complexity is linear in the length of the input string, and its worst-case complexity is no worse than $\mathcal{O}(n \log(n))$.

The τ -reduction used in the algorithm is an interesting reduction preserving right-maximal Lyndon substrings, a fact used significantly in the design of the algorithm. Interestingly, it seem to slightly outperform *BSLA*, at least on the datasets used for our experimentations. *BSLA*, the second algorithm, is linear and elementary in the sense that it does not require a pre-processed global data structure. Being linear and elementary, *BSLA* is more interesting, and it is possible that its performance could be more streamlined. However, both the theory and implementation of *BSLA* are rather complex.

On random strings, none of the two algorithms were significantly better than the simple *IDLA*, whose implementation is just a few lines. However, its quadratic worst-case complexity is an obstacle, as our experiments indicated.

Additional effort needs to go into proving *TRLA*'s worst-case complexity. The experiments performed did not indicate that it is not linear even in the worst case. Both algorithms need to be compared to some efficient implementation of *SSLA* and *BWLA*.

7. Results

This section contains the measurements of the average times for the datasets discussed in the previous section. For better understanding of the data, we present them in Tables 1–8 and Figures 7–14. All the graphs include the curve $x = y$ for reference.

Table 1. Average times for dataset bin (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	3.651×10^{-1}	1.582	1.054
50	4.082	1.050×10	6.372
100	1.101×10	2.140×10	1.277×10
500	8.655×10	1.127×10^2	6.786×10
1000	1.975×10^2	2.335×10^2	1.484×10^2
5000	1.278×10^3	1.218×10^3	8.595×10^2
10,000	2.765×10^3	2.423×10^3	1.820×10^3
50,000	1.665×10^4	1.272×10^4	1.018×10^4
100,000	3.606×10^4	2.523×10^4	2.113×10^4
500,000	2.071×10^5	1.338×10^5	1.493×10^5
1,000,000	4.387×10^5	2.717×10^5	4.080×10^5
5,000,000	2.483×10^6	1.561×10^6	3.098×10^6

Figure 7. Average times for dataset bin (10^6 clock ticks per second).

Table 2. Average times for dataset dna (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	3.699×10^{-1}	1.579	1.080
50	3.509	1.037×10	6.627
100	8.898	2.109×10	1.403×10
500	6.403×10	1.123×10^2	7.228×10
1000	1.431×10^2	2.332×10^2	1.544×10^2
5000	8.749×10^2	1.207×10^3	9.039×10^2
10,000	1.912×10^3	2.460×10^3	1.935×10^3
50,000	1.134×10^4	1.280×10^4	1.110×10^4
100,000	2.431×10^4	2.588×10^4	2.316×10^4
500,000	1.383×10^5	1.390×10^5	1.781×10^5
1,000,000	2.916×10^5	2.865×10^5	4.994×10^5
5,000,000	1.643×10^6	1.968×10^6	3.752×10^6

Figure 8. Average times for dataset dna (10^6 clock ticks per second).**Table 3.** Average times for dataset eng (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	3.526×10^{-1}	1.584	9.865×10^{-1}
50	3.162	1.006×10	5.960
100	7.315	2.057×10	1.317×10
500	4.996×10	1.117×10^2	7.245×10
1000	1.112×10^2	2.354×10^2	1.542×10^2
5000	6.722×10^2	1.210×10^3	9.087×10^2
10,000	1.452×10^3	2.427×10^3	2.042×10^3
50,000	8.505×10^3	1.306×10^4	1.301×10^4
100,000	1.802×10^4	2.688×10^4	2.768×10^4
500,000	1.025×10^5	1.428×10^5	2.381×10^5
1,000,000	2.171×10^5	3.253×10^5	7.236×10^5
5,000,000	1.206×10^6	2.599×10^6	6.092×10^6

Figure 9. Average times for dataset eng (10^6 clock ticks per second).

Table 4. Average times for dataset int (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	3.547×10^{-1}	1.645	9.794×10^{-1}
50	3.032	9.992	5.609
100	7.279	2.032×10	1.153×10
500	4.845×10	1.136×10^2	6.184×10
1000	1.057×10^2	2.376×10^2	1.294×10^2
5000	6.428×10^2	1.218×10^3	7.753×10^2
10,000	1.388×10^3	2.544×10^3	1.796×10^3
50,000	8.055×10^3	1.448×10^4	1.088×10^4
100,000	1.710×10^4	2.943×10^4	2.379×10^4
500,000	9.829×10^4	1.825×10^5	2.740×10^5
1,000,000	2.071×10^5	4.827×10^5	7.989×10^5
5,000,000	1.162×10^6	5.143×10^6	6.635×10^6

Figure 10. Average times for dataset int (10^6 clock ticks per second).

Table 5. Average times for dataset *extreme_idla* (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	7.900×10^{-1}	1.440	7.000×10^{-1}
50	1.830×10	8.200	3.600
100	7.190×10	1.590×10	6.800
500	1.778×10^3	7.300×10	3.550×10
1000	7.105×10^3	1.430×10^2	6.900×10
5000	1.776×10^5	7.100×10^2	3.400×10^2
10,000	7.111×10^5	1.550×10^3	6.800×10^2
50,000	1.784×10^7	8.050×10^3	3.400×10^3
100,000	7.130×10^7	1.600×10^4	6.800×10^3
500,000	1.783×10^9	8.200×10^4	3.700×10^4
1,000,000	7.137×10^9	1.660×10^5	7.800×10^4
5,000,000	1.813×10^{11}	8.800×10^5	4.950×10^5

Figure 11. Average times for dataset *extreme_idla* (10^6 clock ticks per second).**Table 6.** Average times for dataset *extreme_trla* (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	4.588×10^{-1}	1.628	1.126
50	4.987	1.039×10	7.112
100	1.275×10	2.179×10	1.439×10
500	9.033×10	1.101×10^2	6.914×10
1000	2.060×10^2	2.222×10^2	1.392×10^2
5000	1.319×10^3	1.171×10^3	7.699×10^2
10,000	2.896×10^3	2.394×10^3	1.652×10^3
50,000	2.209×10^4	1.263×10^4	8.992×10^3
100,000	3.965×10^4	2.567×10^4	1.862×10^4
500,000	2.233×10^5	1.349×10^5	1.091×10^5
1,000,000	4.734×10^5	2.759×10^5	3.104×10^5
5,000,000	2.632×10^6	1.458×10^6	2.298×10^6

Figure 12. Average times for dataset *extreme_trla* (10^6 clock ticks per second).

Table 7. Average times for dataset *extreme_trla1* (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	5.040×10^{-1}	1.600	1.117
50	5.910	1.042×10	7.290
100	1.460×10	2.145×10	1.446×10
500	1.146×10^2	1.126×10^2	6.979×10
1000	2.662×10^2	2.284×10^2	1.379×10^2
5000	1.694×10^3	1.205×10^3	7.853×10^2
10,000	3.734×10^3	2.477×10^3	1.739×10^3
50,000	2.276×10^4	1.310×10^4	9.683×10^3
100,000	4.901×10^4	2.796×10^4	2.009×10^4
500,000	2.928×10^5	1.465×10^5	1.238×10^5
1,000,000	6.199×10^5	3.000×10^5	3.622×10^5
5,000,000	3.432×10^6	1.642×10^6	2.778×10^6

Figure 13. Average times for dataset *extreme_trla1* (10^6 clock ticks per second).

Table 8. Average times for dataset *extreme_trla2* (10^6 clock ticks per second).

String Length	Time in Ticks IDLA	Time in Ticks TRLA	Time in Ticks BSLA
10	5.041×10^{-1}	1.683	1.121
50	6.160	1.020×10	7.257
100	1.526×10	2.090×10	1.441×10
500	1.367×10^2	1.074×10^2	7.117×10
1000	3.202×10^2	2.135×10^2	1.390×10^2
5000	2.024×10^3	1.145×10^3	7.966×10^2
10,000	4.500×10^3	2.257×10^3	1.762×10^3
50,000	2.728×10^4	1.172×10^4	1.012×10^4
100,000	5.941×10^4	2.362×10^4	2.115×10^4
500,000	3.639×10^5	1.262×10^5	1.351×10^5
1,000,000	7.719×10^5	2.571×10^5	3.915×10^5
5,000,000	4.263×10^6	1.323×10^6	3.118×10^6

Figure 14. Average times for dataset *extreme_trla2* (10^6 clock ticks per second).

Author Contributions: Conceptualization, F.F. and M.L.; methodology, F.F. and M.L.; software, F.F. and M.L.; validation, F.F. and M.L.; formal analysis, F.F. and M.L.; investigation, F.F. and M.L.; resources, F.F. and M.L.; data curation, F.F. and M.L.; writing, original draft preparation, F.F. and M.L.; writing, review and editing, F.F. and M.L.; visualization, F.F. and M.L.; supervision, F.F. and M.L.; project administration, F.F. and M.L.; funding acquisition, F.F. Both authors read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Sciences and Research Council of Canada (NSERC) Grant RGPIN/5504-2018.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BSLA Baier’s Sort Lyndon Array
IDLA Iterative Duval Lyndon Array
TRLA Tau Reduction Lyndon Array

References

1. Lyndon, R.C. On Burnside’s Problem. II. *Trans. Am. Math. Soc.* **1955**, *78*, 329–332.
2. Marcus, S.; Sokol, D. 2D Lyndon words and applications. *Algorithmica* **2017**, *77*, 116–133. [[CrossRef](#)]
3. Berstel, J.; Perrin, D. The origins of combinatorics on words. *Eur. J. Comb.* **2007**, *28*, 996–1022. [[CrossRef](#)]

4. Chen, K.; Fox, R.; Lyndon, R. Free differential calculus IV. The quotient groups of the lower central series. *Ann. Math. 2nd Ser.* **1958**, *68*, 81–95. [[CrossRef](#)]
5. Golomb, S. Irreducible polynomials, synchronizing codes, primitive necklaces and cyclotomic algebra. *Comb. Math. Appl.* **1967**, *4*, 358–370.
6. Flajolet, P.; Gourdon, X.; Panario, D. The complete analysis of a polynomial factorization algorithm over finite fields. *J. Algorithms* **2001**, *40*, 37–81. [[CrossRef](#)]
7. Panario, D.; Richmond, B. Smallest components in decomposable structures:exp-log class. *Algorithmica* **2001**, *29*, 205–226. [[CrossRef](#)]
8. Duval, J.P. Factorizing words over an ordered alphabet. *J. Algorithms* **1983**, *4*, 363–381. [[CrossRef](#)]
9. Berstel, J.; Pocchiola, M. Average cost of Duval’s algorithm for generating Lyndon words. *Theor. Comput. Sci.* **1994**, *132*, 415–425. [[CrossRef](#)]
10. Fredricksen, H.; Maiorana, J. Necklaces of beads in k colors and k -ary de Bruijn sequences. *Discret. Math.* **1983**, *23*, 207–210. [[CrossRef](#)]
11. Bannai, H.; Tomohiro, I.; Inenaga, S.; Nakashima, Y.; Takeda, M.; Tsuruta, K. The “Runs” Theorem. *SIAM J. Comput.* **2017**, *46*, 1501–1514. [[CrossRef](#)]
12. Franek, F.; Paracha, A.; Smyth, W. The linear equivalence of the suffix array and the partially sorted Lyndon array. In Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 28–30 August 2017; pp. 77–84.
13. Baier, U. Linear-Time Suffix Sorting—A New Approach for Suffix Array Construction. Master’s Thesis, University of Ulm, Ulm, Germany, 2015.
14. Baier, U. Linear-Time Suffix Sorting—A New Approach for Suffix Array Construction. In Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), Tel Aviv, Israel, 27–29 June 2016; Grossi, R., Lewenstein, M., Eds.; Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2016; Volume 54, pp. 1–12.
15. Chen, G.; Puglisi, S.; Smyth, W. Lempel-Ziv factorization using less time & space. *Math. Comput. Sci.* **2013**, *1*, 605–623.
16. Crochemore, M.; Ilie, L.; Smyth, W. A simple algorithm for computing the Lempel-Ziv factorization. In Proceedings of the 18th Data Compression Conference, Snowbird, UT, USA, 25–27 March 2008; pp. 482–488.
17. Kosolobov, D. Lempel-Ziv factorization may be harder than computing all runs. In Proceedings of the 32 International Symposium on Theoretical Aspects of Computer Science—STACS 2015, Garching, Germany, 4–7 March 2015; pp. 582–593.
18. Digelmann, C. (Frankfurt, Germany). Personal communication, 2016.
19. Franek, F.; Sohiddul Islam, A.; Sohel Rahman, M.; Smyth, W. Algorithms to compute the Lyndon array. In Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, 29–31 August 2016; pp. 172–184.
20. Hohlweg, C.; Reutenauer, C. Lyndon words, permutations and trees. *Theor. Comput. Sci.* **2003**, *307*, 173–178. [[CrossRef](#)]
21. Nong, G.; Zhang, S.; Chan, W.H. Linear suffix array construction by almost pure induced-sorting. In Proceedings of the 2009 Data Compression Conference, Snowbird, UT, USA, 16–18 March 2009; pp. 193–202.
22. Louza, F.; Smyth, W.; Manzini, G.; Telles, G. Lyndon array construction during Burrows–Wheeler inversion. *J. Discret. Algorithms* **2018**, *50*, 2–9. [[CrossRef](#)]
23. Franek, F.; Liut, M.; Smyth, W. On Baier’s sort of maximal Lyndon substrings. In Proceedings of the Prague Stringology Conference 2018, Prague, Czech Republic, 27–28 August 2018; pp. 63–78.
24. C++ Code for IDLA, TRLA and BSLA Algorithms. Available online: <https://github.com/MichaelLiut/Computing-LyndonArray> (accessed on 3 November 2020).
25. Farach, M. Optimal suffix tree construction with large alphabets. In Proceedings of the 38th IEEE Symp. Foundations of Computer Science, Miami Beach, FL, USA, 20–22 October 1997; pp. 137–143.
26. Nong, G. Practical linear-time $O(1)$ -workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.* **2013**, *31*, 1–15. [[CrossRef](#)]
27. Cooley, J.; Tukey, J. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **1965**, *19*, 297–301. [[CrossRef](#)]

28. Franek, F.; Liut, M. Computing Maximal Lyndon Substrings of a String, AdvOL Report 2019/2, McMaster University. Available online: http://optlab.mcmaster.ca/component/option,com_docman/task,cat_view/gid,77/Itemid,92 (accessed on 1 March 2019).
29. Franek, F.; Liut, M. Algorithms to compute the Lyndon array revisited. In Proceedings of the Prague Stringology Conference 2019, Prague, Czech Republic, 26–28 August 2019; pp. 16–28.
30. Liut, M. Computing Lyndon Arrays. Ph.D. Thesis, McMaster University, Hamilton, ON, Canada, 2019.
31. Lothaire, M. *Combinatorics on Words*; Cambridge University Press: Cambridge, UK, 2003.
32. Lothaire, M. *Applied Combinatorics on Words*; Cambridge University Press: Cambridge, UK, 2005.
33. Smyth, B. *Computing Patterns in Strings*; Pearson Addison-Wesley: Boston, MA, USA, 2003.
34. Louza, F.; Gog, S.; Telles, G. *Construction of Fundamental Data Structures for Strings*; Springer: Cham, Switzerland, 2020.
35. Burkhardt, S.; Kärkkäinen, J. Fast Lightweight Suffix Array Construction and Checking. In Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, Michoacan, Mexico, 25–27 June 2003; Springer: Berlin, Heidelberg, 2003; pp. 55–69.
36. Paracha, A. Lyndon Factors and Periodicities in Strings. Ph.D. Thesis, McMaster University, Hamilton, ON, Canada, 2017.
37. Kärkkäinen, J.; Sanders, P. Simple linear work suffix array construction. In Proceedings of the 30th International Conference on Automata, Languages and Programming, Eindhoven, The Netherlands, 30 June–4 July 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 943–955.

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).