Title:

Computing maximal Lyndon substrings of a string

Authors:

F. Franek and M. Liut

AdvOL-Report No. 2019/2

March 2019, Hamilton, Ontario, Canada

Computing maximal Lyndon substrings of a string

F. Franek and M. Liut

Abstract

There are at least two reasons to have an efficient algorithm for identifying all maximal Lyndon substrings in a string: firstly, Bannai et al. introduced in 2015 a linear algorithm to compute all runs in a string that relies on knowing all maximal Lyndon substrings of the input string, and secondly, Franek et al. showed in 2017 a linear co-equivalence of sorting suffixes and sorting 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 maximal Lyndon substrings of the input string. It discussed four different algorithms. Two known algorithms for computing Lyndon array: a quadratic in-place algorithm based on iterated Duval's algorithm for Lyndon factorization, and a linear algorithmic scheme based on linear suffix sorting, computing inverse suffix array and applying NSV (the Next Smaller Value) algorithm. Duval's algorithm works for strings over any ordered alphabet, while for linear suffix sorting, constant or integer alphabet is required. In addition, the overview discussed a recursive version of Duval's algorithm, also with a quadratic complexity, and an algorithm NSV^* with possibly $\mathcal{O}(n\ log(n))$ worst-case complexity based on ranges that can be compared in a constant time for constant alphabet. The authors at that time did not know of Baier's algorithm.

In 2017, Paracha proposed in her Ph.D. thesis an algorithm for Lyndon array. Though the proposed algorithm is not linear, it has $\Theta(n\log(n))$ complexity, it is interesting as it emulates Farach's recursive approach for computing suffix trees in a linear time and introduces a τ-reduction that might be of independent interest. In 2018, Franek et al. presented a linear algorithm to compute Lyndon array of a string inspired by Phase I of Baier's algorithm for suffix sorting.

This work provides the theoretical analysis of the algorithm proposed by Paracha and the linear algorithm based on Baier's Phase I. It also provides empirical comparisons of these two algorithms implemented in C++ using the iterated Duval algorithm as a yardstick.

1 Introduction

There are at least two reasons to have an efficient algorithm for identifying all maximal Lyndon substrings in a string: firstly, Bannai et al. introduced in 2015 (arXiv, [4]) and published in 2017, [5], a linear algorithm to compute all runs in a string that relies on knowing all maximal Lyndon substrings of the string, and secondly, in 2017, Franek et al. in

[12] showed a linear co-equivalence of sorting suffixes and sorting maximal Lyndon substrings, inspired by Phase II of a suffix sorting algorithm introduced by Baier in 2015 (Master's thesis, [2]), and published in 2016, [3].

The most significant feature of the runs algorithm presented in [5] is that it relies on knowing the 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. Thus, computing runs became yet another application of Lyndon words. It also raised the issue which approach may be more efficient: to compute the Lempel-Ziv factorization or to compute all maximal Lyndon substrings. In particular, Kosolobov argues that computing Lempel-Ziv factorization may be harder than computing all runs, however his archive paper [16] was not followed up. There are several efficient linear algorithms for Lempel-Ziv factorization, see e.g. [6, 7] and the references therein.

Baier introduced in [2], and published in [3] a new algorithm for suffix sorting. Though Lyndon strings are never mentioned in [2, 3], it was noticed by Cristoph Diegelmann in a personal communication, [8], that Phase I of the Baier's suffix sort identifies and sorts all maximal Lyndon substrings.

The maximal Lyndon substrings of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ can be best encoded in the so-called ***Lyndon array***, introduced in [13]: an integer array $\mathcal{L}[1..n]$ so that for any $i \in 1..n$, $\mathcal{L}[i] =$ *the length of the maximal Lyndon substring starting at the position* i.

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

Figure 1: Algorithm *IDLA*

In an overview [13], Franek et al. discussed an algorithm based on iterative application of Duval's Lyndon factorization algorithm, [9], which we refer here as *IDLA*, and an algorithmic scheme based on Hohlweg and Reutenauer's work [14], which we refer to as *SSLA*. The authors did not know 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 a constant time for constant alphabet. The correctness of *NSV** and its complexity were discussed there just informally.

The algorithm *IDLA*, see Fig. 1, is simple and in-place, so no space is required except the storage for the string and the storage for 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. The only "blemish" is its quadratic worst-case complexity that might be a problem for longer strings with long maximal Lyndon substrings. In our empirical work, we used *IDLA* as a yardstick for comparison and as a verifier of the results. Note that the reason the procedure *MaxLyn* really computes the maximal Lyndon prefix is not obvious and is based on the properties of periods of prefixes, see [9], or Observation 6 and Lemma 11 in [13]. The following lemma characterizes maximal Lyndon substrings in terms of the relationships of the suffixes, and follows from the work of Hohlweg and Reutenauer, [14].

Lemma 1. *Consider a string* $\boldsymbol{x}[1..n]$ *over an alphabet ordered by* $\prec$*. The substring* $\boldsymbol{x}[i..j]$ *is Lyndon if* $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ *for any* $i < k \leq j$*, and is maximal Lyndon if it is Lyndon and either* $j = n$ *or* $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$*.*

Thus, 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 [13]. Computing the inverse suffix array and applying *NSV* are "naturally" linear, and computing the suffix array can be implemented to be linear, see [13, 19] 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, [17]. Since the Burrows-Wheeler transform is computed in linear time from the suffix array, it is yet another scheme how to obtain the Lyndon array via suffix sorting: compute the suffix array, from the suffix array compute the Burrows-Wheeler transform, and 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 maximal Lyndon substrings brought up the realization that there was an elementary[1] 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 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 [11].

In this work, we present two additional algorithms for Lyndon array not discussed in [13]. The `C++` source code of the three implementations, *IDLA*, *TRLA*, and *BSLA* can be downloaded from or viewed at [1]. Note that the procedure *IDLA* is in `lynarr.hpp` file.

[1]not relying on a pre-processed global data structure such as suffix array or Burrows-Wheeler transform

The first algorithm presented here is *TRLA* – a τ-reduction based Lyndon array algorithm that follows Farach's approach used in his remarkable linear algorithm for suffix tree construction [10], and reproduced very successfully in all linear algorithms for suffix sorting, see e.g. [18, 19] and the references therein.

The second algorithm, *BSLA* – a Baier's sort-based Lyndon array algorithm – is based on the ideas of Phase I of Baier's suffix sort, though our implementation necessarily differs from Baier's. Though *BSLA* was first introduced in [11], here we present a simplified and more polished theoretical analysis of the algorithm and a more refined implementation.

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 analyzed. In Section 4, the *BSLA* algorithm is presented and analyzed. In Section 5, the empirical measurements of the performance of *IDLA*, *TRLA*, and *BSLA* are presented on datasets with random strings of various lengths and over various alphabets. The results are presented in graphical form. In Section 6, the conclusion of the research is presented, and the necessary future work described.

2 Basic notation and terminology

For two integers $i \le j$, the ***range*** $\boldsymbol{i..j} = \{k \textit{ integer} : i \le k \le j\}$. An ***alphabet*** is a finite or infinite sets of ***symbols***, or equivalently called ***letters***. We assume that a sentinel symbol \$ is not in the alphabet and is always assumed to be lexico-graphically 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 $\boldsymbol{x}[1..n]$ indicates a string of length n, the first symbol is the symbol with index 1, i.e. $\boldsymbol{x}[1]$, the second symbol is the symbol with index 2, i.e. $\boldsymbol{x}[2]$, etc. Thus, $\boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]...\boldsymbol{x}[n]$. For a \$-terminated string $\boldsymbol{x}$ of length n, $\boldsymbol{x}[n{+}1] = \$$. The ***alphabet of string*** $\boldsymbol{x}$, denoted as $\mathcal{A}_{\boldsymbol{x}}$, is the set of all distinct alphabet symbols occurring in $\boldsymbol{x}$. By a ***constant alphabet*** we mean a fixed finite alphabet. A string $\boldsymbol{x}$ is over an ***integer alphabet*** if $\mathcal{A}_{\boldsymbol{x}} \subseteq \{0, 1, ..., |\boldsymbol{x}|\}$. Thus, the class of ***strings over integer alphabets*** $= \{\boldsymbol{x} \mid \boldsymbol{x} \textit{ is a string over } \{0, 1, ..., |\boldsymbol{x}|\}\}$. A string $\boldsymbol{x}$ over an integer alphabet is ***tight*** if $\mathcal{A}_{\boldsymbol{x}} = \{0, 1, ..., k\}$ for some $k \le |\boldsymbol{x}|$. Thus, for instance $\boldsymbol{x} = 010$ is tight as $\mathcal{A}_{\boldsymbol{x}} = \{0, 1\}$, while $\boldsymbol{y} = 020$ is not as $\mathcal{A}_{\boldsymbol{y}} = \{0, 2\}$ – i.e. 1 is missing from $\mathcal{A}_{\boldsymbol{y}}$.

We use a bold font to denote strings, thus $\boldsymbol{x}$ denotes a string, while x denotes some other mathematical entity such as an integer. The ***empty string*** is denoted by $\boldsymbol{\varepsilon}$ and has length 0. The ***length*** or ***size*** of string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is n. The length of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. For two strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..m]$, the ***concatenation*** $\boldsymbol{xy}$ is a string $\boldsymbol{u}$ where $\boldsymbol{u}[i] = \begin{cases} \boldsymbol{x}[i] \textit{ for } i \le n, \\ \boldsymbol{y}[i-n] \textit{ for } n < i \le n{+}m. \end{cases}$
If $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{u}$ is a ***prefix***, $\boldsymbol{v}$ a ***substring***, and $\boldsymbol{w}$ a ***suffix*** of $\boldsymbol{x}$. If $\boldsymbol{u}$ (respective $\boldsymbol{v}$, $\boldsymbol{w}$) is empty, then it is called a ***trivial prefix*** (respective ***trivial substring***, ***trivial suffix***), if $|\boldsymbol{u}| < |\boldsymbol{x}|$ (respective $|\boldsymbol{v}| < |\boldsymbol{x}|$, $|\boldsymbol{w}| < |\boldsymbol{x}|$) then it is called a ***proper prefix*** (respective ***proper substring***, ***proper suffix***). If $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a ***rotation*** or a ***conjugate*** of $\boldsymbol{x}$; if either $\boldsymbol{u} = \boldsymbol{\varepsilon}$ or $\boldsymbol{v} = \boldsymbol{\varepsilon}$, then the rotation is called ***trivial***. A non-empty string $\boldsymbol{x}$ is

primitive if there is no string $\boldsymbol{y}$ and no integer $k \geq 2$ so that $\boldsymbol{x} = \boldsymbol{y}^k = \underbrace{\boldsymbol{y}\boldsymbol{y}\cdots\boldsymbol{y}}_{k\ times}$.

A non-empty string $\boldsymbol{x}$ has a non-trivial border $\boldsymbol{u}$ if $\boldsymbol{u}$ is both a non-trivial proper prefix and a non-trivial proper suffix of $\boldsymbol{x}$. Thus, both $\boldsymbol{\varepsilon}$ and $\boldsymbol{x}$ are trivial borders of $\boldsymbol{x}$. A string without a non-trivial border is call ***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 $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..n]$, $\boldsymbol{x} \prec \boldsymbol{y}$ if either $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$, or there is a $j \leq \min\{n, m\}$ so that $\boldsymbol{x}[1] = \boldsymbol{y}[1]$, ..., $\boldsymbol{x}[j-1] = \boldsymbol{y}[j-1]$ and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$. This total order induced by the order of the alphabet is called a ***lexico-graphic*** order of all non-empty strings over $\mathcal{A}$. We denote by $\boldsymbol{x} \preceq \boldsymbol{y}$ if either $\boldsymbol{x} \prec \boldsymbol{y}$ or $\boldsymbol{x} = \boldsymbol{y}$. A string $\boldsymbol{x}$ over $\mathcal{A}$ is ***Lyndon*** for a given order $\prec$ of $\mathcal{A}$ if $\boldsymbol{x}$ is strictly lexico-graphically smaller than any non-trivial rotation of $\boldsymbol{x}$. In particular:

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

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 $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}[1..n]$, $1 \leq i \leq j \leq n$ is a ***maximal Lyndon substring of*** $\boldsymbol{x}$ if it is Lyndon and either $j = n$ or for any $k > j$, $\boldsymbol{x}[i..k]$ is not Lyndon. The ***Lyndon array*** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ 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 $\boldsymbol{x}[i..i+j-1]$ 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 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 $\mathcal{L}[i] = \mathcal{L}'[i]-i+1$.

3 $\boldsymbol{\tau}$-reduction algorithm – ***TRLA***

The first idea of the algorithm was proposed in Paracha's 2017 Ph.D. thesis, [20]. It follows Farach's approach:

(1) reduce the input string $\boldsymbol{x}$ to $\boldsymbol{y}$,
(2) by recursion compute the Lyndon array of $\boldsymbol{y}$,
(3) from the Lyndon array of $\boldsymbol{y}$ compute the Lyndon array of $\boldsymbol{x}$.

The input strings are \$-terminated strings over integer alphabets. The reduction computed in (1) is important. All linear algorithms for suffix array computations use the proximity property of suffixes: comparing $\boldsymbol{x}[i..n]$ and $\boldsymbol{x}[j..n]$ can be done by comparing $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$, and if they are the same, comparing $\boldsymbol{x}[i+1..n]$ with $\boldsymbol{x}[j+1..n]$. For instance, in the first linear algorithm for suffix array by Kärkkäinen and Sanders, [15], obtaining the sorted suffixes for positions $i \equiv 0\ (mod\ 3)$ and $i \equiv 1\ (mod\ 3)$ via the recursive call is sufficient to determine the order of suffixes for $i \equiv 2\ (mod\ 3)$ positions, and then to merge both lists together. However, there is no such proximity property for maximal Lyndon substrings, so the reduction itself must have a property that helps determine some of the values of the Lyndon array of $\boldsymbol{x}$ from the Lyndon array of $\boldsymbol{y}$ and compute the rest. We present such a reduction that we call τ-reduction, and it may be of some general interest as it preserves order of some suffixes and hence, by Lemma 1, some maximal Lyndon substrings.

The algorithm computes $\boldsymbol{y}$ as a τ-reduction of $\boldsymbol{x}$ in step (1) in linear time and 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. However, it computes the missing values of the incomplete Lyndon array in $\Theta(n\ log(n))$ time resulting in the overall worst-case complexity of $\Theta(n\ log(n))$. If the missing values of the incomplete Lyndon array of $\boldsymbol{x}$ were computed in linear time, the overall algorithm would be linear as well. Since for τ-reduction, the size of $\tau(\boldsymbol{x})$ is at most $\frac{2}{3}|\boldsymbol{x}|$, we eventually obtain through the recursion of step (2) applied to $\tau(\boldsymbol{x})$ a partially filled Lyndon array of the input string; the array 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 and $\boldsymbol{x}[i-1] \preceq \boldsymbol{x}[i]$. In particular, the value at position 1 and position n are both known. So, a lot of information is provided by the recursive step. For instance, for 00011001, via the recursive call we would identify the maximal Lyndon substrings that are underlined in $\underline{00\underline{01\underline{1}}\,\underline{00\underline{1}}}$ and would need to compute the missing maximal Lyndon substrings that are underlined in $0\underline{00\underline{1}1}0\underline{01}$. It is possible that in the future we may come up with a linear procedure to compute the missing values making the whole algorithm linear. We describe the τ-reduction in several steps: first the τ-pairing, then choosing the τ-alphabet, and finally the computation of the τ-reduction of $\boldsymbol{x}$.

3.1 $\boldsymbol{\tau}$-pairing

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

- the initial τ-pair is $(1, 2)$;
- if $(1, 2), (i_1, i_1{+}1), (i_2, i_2{+}1), ..., (i_k, i_k{+}1)$ are τ-pairs computed so far, then if $i_k{+}1 \geq n$, stop; otherwise compute the next τ-pair by the following formula:

```
if $i_k{+}2 = n$ then
   the next $\tau$-pair is set to $(n, n{+}1)$
else
   if $\boldsymbol{x}[i_k{+}1] \succ \boldsymbol{x}[i_k{+}2]$ then
      if $\boldsymbol{x}[i_k{+}2] \succ \boldsymbol{x}[i_k{+}3]$ then
         the next $\tau$-pair is set to $(i_k{+}2, i_k{+}3)$
      else
         the next $\tau$-pair is set to $(i_k{+}1, i_k{+}2)$
   else
      the next $\tau$-pair is $(i_k{+}2, i_k{+}3)$
```

In simple terms, if $(i{-}1, i)$ is the last computed τ-pair, then if $\boldsymbol{x}[i{-}1] \succ \boldsymbol{x}[i] \preceq \boldsymbol{x}[i{+}1]$, the next τ-pair is $(i, i{+}1)$, otherwise the next τ-pair is $(i{+}1, i{+}2)$.

Every position of the input string that occurs in some τ-pair as the first element is labeled ***black***, all others are labeled ***white***. Note 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 $\boldsymbol{x}[i{-}1] \succ \boldsymbol{x}[i]$ and $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i{+}1]$. Moreover, a τ-pair can be involved in at most one overlap. For an illustration, see Fig. 2, for the formal proof see Observation 2 and Lemma 3.

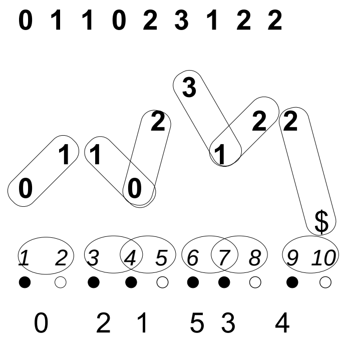


Figure 2: Illustration of τ-reduction of a string **011023122**

The rounded rectangles indicate symbol τ-pairs, the ovals indicate the τ-pairs below are the colour labels of positions, at the bottom is the τ-reduction

Observation 2. *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $n \geq 2$. Let $(i_1, i_1+1)(i_2, i_2+1)...$ (i_k, i_k+1) be the list of τ-pairs of $\boldsymbol{x}[1..n-1]$ ordered in an ascending manner by the first coordinate. Then the τ-pairs of $\boldsymbol{x}[1..n]$ are either*

- $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)(i_k+1, i_k+2)$ *or*
- $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)(i_k+2, i_k+3)$ *or*
- $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)$.

Lemma 3. *Let $(i_1, i_1+1)...(i_k, i_k+1)$ be the τ-pairs of a strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$. 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) $|(i_j, i_j+1) \cap (i_\ell, i_\ell+1)| \leq 1$.

Proof. By induction. Trivially true for $n = 1$. Assume for $n-1$. Consider $\boldsymbol{x}[1..n]$. By induction hypothesis it is true for the τ-pairs $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)$ of $\boldsymbol{x}[1..n-1]$.

- Case the τ-pairs of $\boldsymbol{x}[1..n]$ are $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)(i_k+1, i_k+2)$.
  Then (i_k+1, i_k+2) overlaps only (i_k, i_k+1) and in the point i_k+1.
- Case the τ-pairs of $\boldsymbol{x}[1..n]$ are $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)(i_k+2, i_k+3)$.
  Then (i_k+2, i_k+3) does not overlap any other τ-pair.
- Case the τ-pairs of $\boldsymbol{x}[1..n]$ are $(i_1, i_1+1)(i_2, i_2+1)...(i_k, i_k+1)$.
  Then the assertion obviously holds. □

3.2 τ-reduction

For each τ-pair $(i, i+1)$, we consider the pair of symbols $(\boldsymbol{x}[i], \boldsymbol{x}[i+1])$. We call them ***symbol τ-pairs***. They are in the lexico-graphic order induced by $\prec$: $(\boldsymbol{x}[i_j], \boldsymbol{x}[i_j+1]) \prec (\boldsymbol{x}[i_\ell], \boldsymbol{x}[i_\ell+1])$ if either $\boldsymbol{x}[i_j] \prec \boldsymbol{x}[i_\ell]$, or $\boldsymbol{x}[i_j] = \boldsymbol{x}[i_\ell]$ and $\boldsymbol{x}[i_j+1] \prec \boldsymbol{x}[i_\ell+1]$. They are sorted by a radix sort with keys of size 2, and assigned letters from a chosen τ-alphabet that is a subset of $\{0, 1, ..., |\tau(\boldsymbol{x})|\}$ so that the assignment preserves the order. Because the input string was over an integer alphabet, the radix sort is linear.

In the example, Fig. 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 $(0,1)(0,2)(1,0)$ $(1,2)(2,\$)(3,2)$. Thus we chose as our τ-alphabet $\{0,1,2,3,4,5\}$ and so the symbol τ-pairs are assigned these letters: $(0,1)\to 0$, $(0,2)\to 1$, $(1,0)\to 2$, $(1,2)\to 3$, $(2,\$)\to 4$ and $(3,1)\to 5$.

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 $\boldsymbol{x}$ and denoted $\boldsymbol{\tau(x)}$, and it is a \$-terminated string over integer alphabet. For our running example from Fig. 2, $\tau(\boldsymbol{x}) = 021534$. The next lemma justifies calling the above transformation a reduction.

Lemma 4. *For any string $\boldsymbol{x}$, $\frac{1}{2}|\boldsymbol{x}| \le |\tau(\boldsymbol{x})| \le \frac{2}{3}|\boldsymbol{x}|$.*

Proof. One extreme case is when all the τ-pairs do not overlap at all, then $|\tau(\boldsymbol{x})| = \frac{1}{2}|\boldsymbol{x}|$. The other extreme case is when all the τ-pairs overlap, then $|\tau(\boldsymbol{x})| = \frac{2}{3}|\boldsymbol{x}|$. Any other case must be in between. □

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

$$1..|\tau(\boldsymbol{x})| \underset{\mathrm{t}}{\overset{\mathrm{b}}{\rightleftarrows}} \mathcal{B}(\boldsymbol{x})$$

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

In our running example from Fig. 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 maximal Lyndon substrings of $\boldsymbol{x}$ that start at black positions, i.e. there is a closed formula that gives for every maximal Lyndon substring of $\tau(\boldsymbol{x})$ a corresponding maximal Lyndon substring of $\boldsymbol{x}$ so that all maximal Lyndon substrings of $\boldsymbol{x}$ starting at black positions are given. 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 Lyndon array, the one where the end positions of maximal Lyndon substrings are stored rather than their lengths. More formally:

Theorem 5. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$, let $\mathcal{L}'_{\tau(\boldsymbol{x})}[1..m]$ be the Lyndon array of $\tau(\boldsymbol{x})$, and let $\mathcal{L}'_{\boldsymbol{x}}[1..n]$ be the Lyndon array of $\boldsymbol{x}$.*

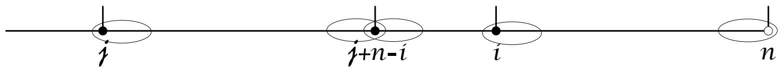
Then for any black $i \in 1..n$, $\mathcal{L}'_{\boldsymbol{x}}[i] = \begin{cases} b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big) & \text{if } \boldsymbol{x}[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)+1] \preceq \boldsymbol{x}[i] \\ b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)+1 & \text{otherwise.} \end{cases}$

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

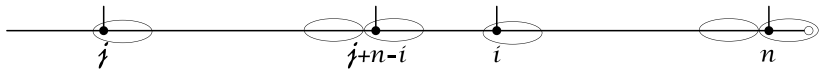
Lemma 6. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i, j \le n$. If i and j are both black positions, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$ implies $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.*

Proof. Since i and j are both black positions, both $t(i)$ and $t(j)$ are defined . Let us assume that $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$.

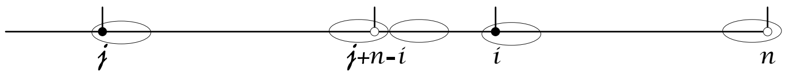
- Case $\boldsymbol{x}[i..n]$ is a proper prefix of $\boldsymbol{x}[j..n]$.
  Then $j < i$ and so $\boldsymbol{x}[j..j{+}n{-}i] = \boldsymbol{x}[i..n]$ and thus $\boldsymbol{x}[i..n]$ is a border of $\boldsymbol{x}[j..n]$.
  - Case $j{+}n{-}i$ is black.
    Since n may be black or white, we need to discuss both cases.
    - Case that n is white.

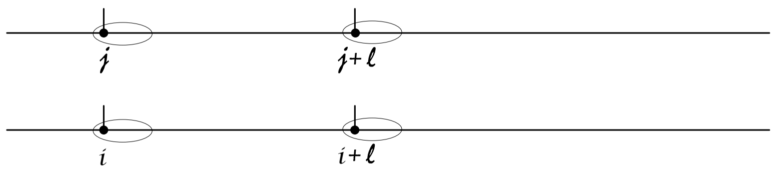
      Then the last τ-pair overlapping $j..j+n-i$ must be $(j+n-i-1, j+n-i)$ followed by a τ-pair $(j{+}n{-}i, j{+}n{-}i{+}1)$, and the last τ-pair overlapping $i..n$ must be the last τ-pair $(n{-}1, n)$. Thus, $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n{-}1)] = \tau(\boldsymbol{x})[t(j)..t(j+n-i-1)]$, and $\tau(\boldsymbol{x})[t(j)..m] = \tau(\boldsymbol{x})[t(j)..t(n-1)]$, and so $\tau(\boldsymbol{x})[t(i)..m]$ is a proper prefix of $\tau(\boldsymbol{x})[t(j)..m]$.
    - Case that n is black.

      Then the last two τ-pairs overlapping $j..j{+}n{-}i$ must be $(j{+}n{-}i{-}1, j{+}n{-}i)$ and $(j{+}n{-}i, j{+}n{-}i{+}1)$, and the last two τ-pairs overlapping $i..n$ must be $(n{-}1, n)$ and $(n, n{+}1)$. Thus, $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n)] \prec \tau(\boldsymbol{x})[t(j)..t(j{+}n{-}i)]$, which is a prefix of $\tau(\boldsymbol{x})[t(j)..m]$.
  - Case $j{+}n{-}i$ is white.

    Then n must also be white as the τ-pair overlapping $j..j+n-i$ must be $(j{+}n{-}i{-}1, j{+}n{-}i)$ followed by $(j{+}n{-}i{+}1, j{+}n{-}i{+}2)$, and the last τ-pair overlapping $i..n$ must be the very last τ-pair $(n{-}1, n)$. Then $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n{-}1)] = \tau(\boldsymbol{x})[t(j)..t(j{+}n{-}i{-}1)]$ which is a proper prefix of $\tau(\boldsymbol{x})[t(j)..m]$.
- Case when $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$.
  Then $\tau(\boldsymbol{x})[t(i)] = p(i, i{+}1)$ and $\tau(\boldsymbol{x})[t(j)] = p(j, j{+}1)$. Since $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$, we have $(\boldsymbol{x}[i], \boldsymbol{x}[i{+}1]) \prec (\boldsymbol{x}[j], \boldsymbol{x}[j{+}1])$ and so $p(i, i{+}1) \prec p(j, j{+}1)$, giving $\tau(\boldsymbol{x})[t(i)] \prec \tau(\boldsymbol{x})[t(j)]$ and so $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.
- Case when for some ℓ, $\boldsymbol{x}[i..i{+}\ell{-}1] = \boldsymbol{x}[j..j{+}\ell{-}1]$ while $\boldsymbol{x}[i{+}\ell] \prec \boldsymbol{x}[j{+}\ell]$.
  - Case both $i{+}\ell$ and $j{+}\ell$ are black.

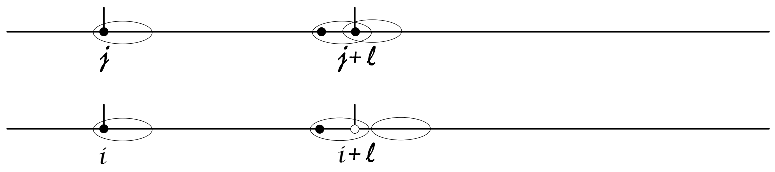
    

Consider $i{+}\ell{-}1$ and $j{+}\ell{-}1$. Either they are both black, or both are white.

(α) Case both $i{+}\ell{-}1$ and $j{+}\ell{-}1$ are black,
then $\tau(\boldsymbol{x})[t(i)..t(i{+}\ell{-}1)] = \tau(\boldsymbol{x})[t(j)..t(j{+}\ell{-}1)]$ and so $\tau(\boldsymbol{x})[t(i)..t(i{+}\ell)] = \tau(\boldsymbol{x})[t(i)..t(i+\ell-1)]\tau(\boldsymbol{x})[i+\ell] = \tau(\boldsymbol{x})[t(j)..t(j+\ell-1)]\tau(\boldsymbol{x})[t(i+\ell)] \prec \tau(\boldsymbol{x})[t(j)..t(j{+}\ell{-}1)]\tau(\boldsymbol{x})[t(j{+}\ell)] = \tau(\boldsymbol{x})[t(j)..t(j{+}\ell)]$ and so $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.

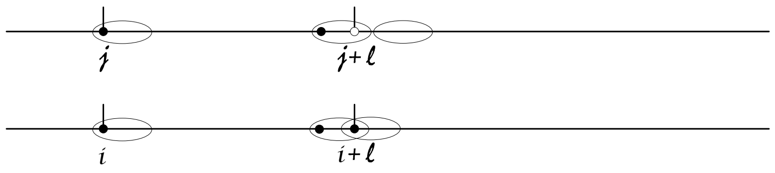
(β) Case both $i{+}\ell{-}1$ and $j{+}\ell{-}1$ are white,
then both $i+\ell-2$ and $j+\ell-2$ are black. So, $\tau(\boldsymbol{x})[t(i)..t(i+\ell-2)] = \tau(\boldsymbol{x})[t(j)..t(j{+}\ell{-}2)]$ and so $\tau(\boldsymbol{x})[t(i)..t(i{+}\ell)] = \tau(\boldsymbol{x})[t(i)..t(i{+}\ell{-}2)]\tau(\boldsymbol{x})[i{+}\ell] = \tau(\boldsymbol{x})[t(j)..t(j+\ell-2)]\tau(\boldsymbol{x})[i+\ell] \prec \tau(\boldsymbol{x})[t(j)..t(j+\ell-2)]\tau(\boldsymbol{x})[j+\ell] = \tau(\boldsymbol{x})[t(j)..t(j{+}\ell)]$ and so $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.

- Case $j{+}\ell$ is black and $i{+}\ell$ is white.



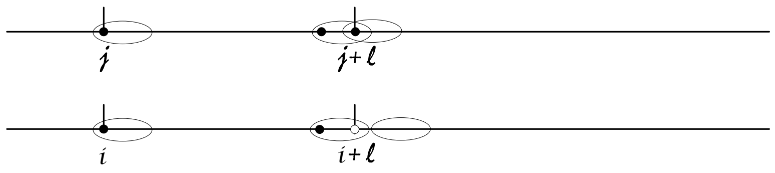
Then both $i{+}\ell{-}1$ and $j{+}\ell{-}1$ are black, so proceed as in (α).

- Case $j{+}\ell$ is white and $i{+}\ell$ is black.



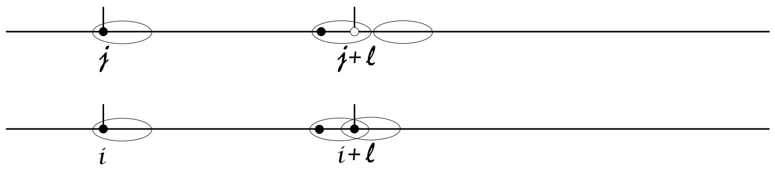
Then both $i{+}\ell{-}1$ and $j{+}\ell{-}1$ are black, so proceed as in (α).

- Case both $j{+}\ell$ and $i{+}\ell$ are white.



Then both $i{+}\ell{-}1$ and $j{+}\ell{-}1$ are black, so proceed as in (α).

- Case $j{+}\ell$ is white and $i{+}\ell$ is black.



Then both $i{+}\ell{-}1$ and $j{+}\ell{-}1$ are black, so proceed as in (α).

□

Lemma 7 shows that τ-reduction preserves the Lyndon property of certain Lyndon substrings.

Lemma 7. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i < j \le n$. Let $\boldsymbol{x}[i..j]$ be a Lyndon susbtsring of $\boldsymbol{x}$, and let i be a black position.*

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

Proof. Let us first assume that j is black.
Let $i_1 = t(i)$, $j_1 = t(j)$ and consider k_1 so that $i_1 < k_1 \le j_1$. Let $k = b(k_1)$. Then $i < k \le j$ and so $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ by Lemma 1. Hence, $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(k)..m]$ by Lemma 6. Therefore, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon by Lemma 1.
Now assume that j is white.
Then $j{-}1$ is black and $\boldsymbol{x}[i..j{-}1]$ is Lyndon, so as in the previous case, $\tau(\boldsymbol{x})[t(i)..t(j{-}1)]$ is Lyndon. □

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

Lemma 8. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i < j \le n$. Let $\boldsymbol{x}[i..j]$ be a maximal Lyndon substring, and let i be a black position.*

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

Proof. Since $\boldsymbol{x}[i..j]$ is maximal Lyndon, $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$ by Lemma 1, giving $\boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[i]$. Since $\boldsymbol{x}[i..j]$ is Lyndon, $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$. Thus, $\boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[i] \prec \boldsymbol{x}[j]$.
We will proceed by discussing two possible cases, one that j is black, and the other that j is white.

- Assume that j is black.
  Since j is black, $(j, j{+}1)$ is a τ-pair and $t(j)$ is defined, and by Lemma 7, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon, and hence by Lemma 1, $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[k..m]$ for any $t(i) < k \le t(j)$. Thus, we must show the maximality, i.e. $\tau(\boldsymbol{x})[t(j){+}1..m] \prec \tau(\boldsymbol{x})[t(i)..m]$.
  - Case when $\boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[j{+}2]$.
    Then $\boldsymbol{x}[j] \succ \boldsymbol{x}[\text{j}{+}1] \preceq \boldsymbol{x}[j{+}2]$ and so $j{+}1$ is black. It follows that $t(j){+}1 = t(j{+}1)$. By Lemma 6, $\tau(\boldsymbol{x})[t(j{+}1)..m] \prec \tau(\boldsymbol{x})[t(i)..m]$ because $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$, thus $\tau(\boldsymbol{x})[t(j){+}1..m] \prec \tau(\boldsymbol{x})[t(i)..m]$.
  - Case when $\boldsymbol{x}[j{+}1] \succ \boldsymbol{x}[j]$.
    Then $\boldsymbol{x}[j] \succ \boldsymbol{x}[i] \succeq \boldsymbol{x}[j{+}1] \succ \boldsymbol{x}[j{+}2]$, then $\tau(\boldsymbol{x})[t(i)] = p(i, i{+}1)$, and $\tau(\boldsymbol{x})[t(j)] = p(j, j{+}1)$, and $\tau(\boldsymbol{x})[t(j){+}1] = p(j{+}2, j{+}3)$. It follows that $\tau(\boldsymbol{x})[t(j)] = p(j, j{+}1) \succ \tau(\boldsymbol{x})[t(i)] = p(i, i{+}1) \succ \tau(\boldsymbol{x})[t(j){+}1] = p(j{+}2, i{+}3)$. Thus, $\tau(\boldsymbol{x})[t(j){+}1] \prec \tau(\boldsymbol{x})[t(i)]$, and so $\tau(\boldsymbol{x})[t(j){+}1..m] \prec \tau(\boldsymbol{x})[t(i)..m]$.
- Assume that j is white.
  By Lemma 7, $\tau(\boldsymbol{x})[t(i)..t(j{-}1)]$ is Lyndon. Since j is white, it follows that $j{-}1$ and $j{+}1$ are black and $t(j{-}1){+}1 = t(j{+}1)$. Since $\boldsymbol{x}[i..n] \succ \boldsymbol{x}[j{+}1..n]$, by Lemma 7 we get $\tau(\boldsymbol{x})[t(i)..m] \succ \tau(\boldsymbol{x})[t(j{+}1)..m] = \tau(\boldsymbol{x})[t(j{-}1){+}1..m]$.

□

Now we are ready to tackle the proof Theorem 5 as we promised.

Proof of Theorem 5. Let $\mathcal{L}'_{\boldsymbol{x}}[i] = j$ where i is black. Then $t(i)$ is defined and $\boldsymbol{x}[i..j]$ is a maximal Lyndon substring of $\boldsymbol{x}$.

- Case when j is black.
  Then by Lemma 8, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is a maximal Lyndon substring of $\tau(\boldsymbol{x})$, hence $\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)] = t(j)$. Therefore, $b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big) = b(t(j)) = j = \mathcal{L}'_{\boldsymbol{x}}[i]$. Since $\boldsymbol{x}[i..j]$ is maximal, $\boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[i]$, i.e. $\boldsymbol{x}[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big){+}1] = \boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[i]$.
- Case when j is white.
  Then $j{-}1$ is black and the $\tau(\boldsymbol{x})[t(j{-}1)] = p(j{-}1, j)$. By Lemma 8, $\tau(\boldsymbol{x})[t(i)..t(j{-}1)]$ is a maximal Lyndon substring of $\tau(\boldsymbol{x})$, hence $\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)] = t(j{-}1)$, so $b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big) = b(t(j{-}1)) = j{-}1$, giving $b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]{+}1\big) = j$. Since $\boldsymbol{x}[i..j]$ is maximal, $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$, i.e. $\boldsymbol{x}[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]{+}1\big)] = \boldsymbol{x}[j] \succ \boldsymbol{x}[i]$.

$\square$

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

Theorem 5 indicates how to compute the partial $\mathcal{L}'_{\boldsymbol{x}}$ from $\mathcal{L}'_{\tau(\boldsymbol{x})}$. The procedure is given in Fig. 3.

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

Figure 3: Computing partial Lyndon array of the input string

How 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 $\mathcal{L}'_{\boldsymbol{x}}[i] = nil$), the Lyndon array $\mathcal{L}'_{\boldsymbol{x}}[i+1..n]$ is completely filled and also $\mathcal{L}'_{\boldsymbol{x}}[i-1]$ is known. Note that $\mathcal{L}'_{\boldsymbol{x}}[i+1]$ is the ending position of the maximal Lyndon substring starting at the position $i{+}1$. If $\boldsymbol{x}[i] \succ \boldsymbol{x}[i{+}1]$, then the maximal Lyndon substring from position $i{+}1$ cannot be extended to the left, and hence the maximal Lyndon substring at the position i has length 1 and so ends in i. Otherwise, $\boldsymbol{x}\big[i..\mathcal{L}'_{\boldsymbol{x}}[i{+}1]\big]$ is Lyndon, and we have to test if we can extend the maximal Lyndon substring right after, and so on. But of course, this is all happening inside the maximal Lyndon substring starting at $i{-}1$ and ending at $\mathcal{L}'_{\boldsymbol{x}}[i{-}1]$ due to Monge property[2] of the maximal Lyndon substrings.

This is the **while** loop in the procedure given in Fig. 4 that gives it the $\mathcal{O}(n\ log(n))$ complexity as we will show later. At the first, it may seem that it might actually give it

[2]two maximal Lyndon susbtrings are either disjoint or one completely includes the other

$\mathcal{O}(n^2)$ complexity, but the "doubling of size" trims it effectively down to $\mathcal{O}(n\ log(n))$, see section 3.5.

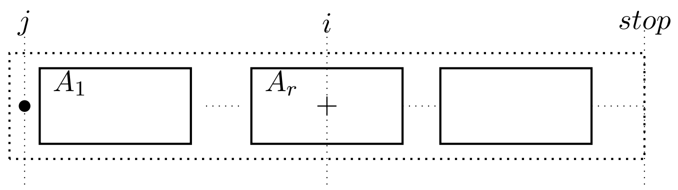
```
$\mathcal{L}'_{\boldsymbol{x}}[n] \leftarrow n$
**for** $i \leftarrow n{-}1$ **downto** 2
  **if** $\mathcal{L}'[i] = nil$ **then**
    **if** $\boldsymbol{x}[i] \succ \boldsymbol{x}[i{+}1]$ **then**
      $\mathcal{L}'[i] \leftarrow i$
    **else**
      **if** $\mathcal{L}'[i{-}1] = i{-}1$ **then**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

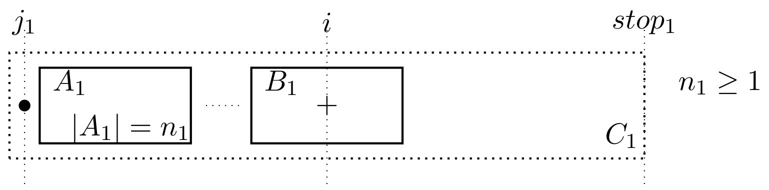
Consider our running example from Fig. 2. Since $\tau(\boldsymbol{x}) = 021534$, we have $\mathcal{L}'_{\tau(\boldsymbol{x})}[1..6] = 6, 2, 6, 4, 6, 6$ giving $\mathcal{L}'_{\boldsymbol{x}}[1..9] = 9, \bullet, 3, 9, \bullet, 6, 9, \bullet, 9$. Computing $\mathcal{L}'_{\boldsymbol{x}}[8]$ is easy as $\boldsymbol{x}[8] = \boldsymbol{x}[9]$ and so $\mathcal{L}'_{\boldsymbol{x}}[8] = 8$. $\mathcal{L}'_{\boldsymbol{x}}[5]$ is more complicate: we can extend the maximal Lyndon substring from $\mathcal{L}'_{\boldsymbol{x}}[6]$ to the left to 23, but no more, so $\mathcal{L}'_{\boldsymbol{x}}[5] = 6$. Computing $\mathcal{L}'_{\boldsymbol{x}}[2]$ is again easy as $\boldsymbol{x}[2] = \boldsymbol{x}[3]$ and so $\mathcal{L}'_{\boldsymbol{x}}[2] = 2$. Thus $\mathcal{L}'_{\boldsymbol{x}}[1..9] = 9, 2, 3, 9, 6, 6, 9, 8, 9$.

3.5 The complexity of *TRLA*

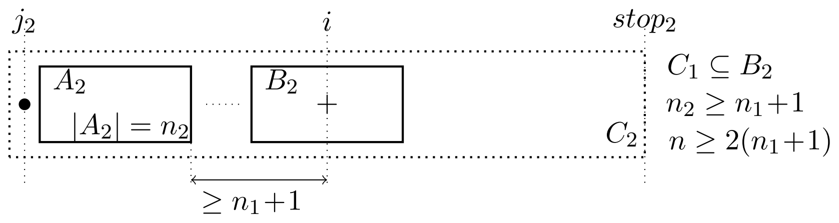
To determine the complexity of the algorithm, we attach to each position i a counter $red[i]$ initialized to 0. When computing a missing value $\mathcal{L}'_{\boldsymbol{x}}[j]$ with a configuration shown below where $stop = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j{-}1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j{-}1] > j{-}1 \\ n & otherwise \end{cases}$, we have to check if A_1 can be extended by j to a Lyndon substring (i.e. if $\boldsymbol{x}[j] \preceq \boldsymbol{x}[j{+}1]$), if so, we have to compare jA_1 with A_2 and if jA_1A_2 is Lyndon (i.e. if $jA_1 \prec A_2$), we must check if $jA_1A_2A_3$ is Lyndon (i.e. if $jA_1A_2 \prec A_3$) ... checking if $jA_1..A_r$ is Lyndon (i.e. if $jA_1A_2..A_{r-1} \prec A_r$) , etc. ... When comparing the Lyndon substring $jA_1..A_{r-1}$ with A_r, at every position i of A_r, we increment the counter $red[i]$. When done, the value of $red[i]$ represents how many times the position i was used in comparisons.



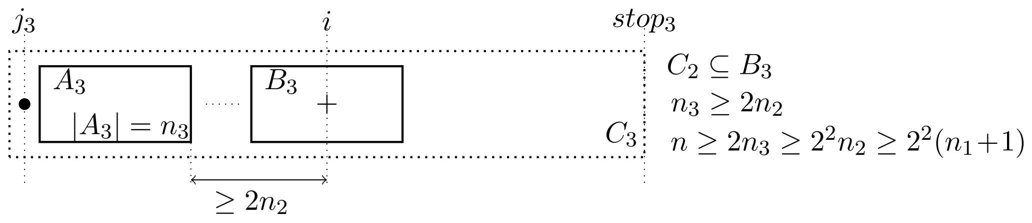
Consider a position i that was used k times for $k \geq 4$, i.e. $red[i]=k$. The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 1st time in the comparison of $j_1 A1...$ and B_1 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_1]$ where $stop_1 = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j_1-1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j_1-1] > j_1-1 \\ n & otherwise \end{cases}$



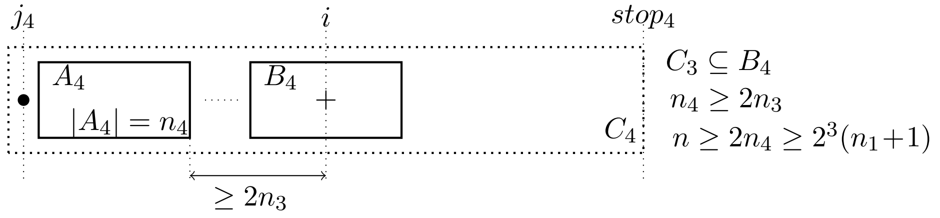
The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 2nd time in the comparison of $j_2 A2...$ and B_2 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_2]$ where $stop_2 = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j_2-1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j_2-1] > j_2-1 \\ n & otherwise \end{cases}$



The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 3rd time in the comparison of $j_3 A3...$ and B_3 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_3]$ where $stop_3 = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j_3-1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j_3-1] > j_3-1 \\ n & otherwise \end{cases}$



The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 4th time in the comparison of $j_4 A4...$ and B_4 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_4]$ where $stop_4 = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j_4-1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j_4-1] > j_4-1 \\ n & otherwise \end{cases}$



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 see that the overall complexity is $\Theta(n\ log(n))$ just consider the following strings:

$$\text{Set } \boldsymbol{u}_0 = 011. \text{ By induction, set } \boldsymbol{u}_k = 00\boldsymbol{u}_{k-1}0\boldsymbol{u}_{k-1}. \tag{F}$$

For any $k \geq 1$, $\boldsymbol{u}_k$ is Lyndon and so is $0\boldsymbol{u}_k$. The second 0 in $\boldsymbol{u}_k$ will be the missing value. Since $0\boldsymbol{u}_{k-1}$ is Lyndon, the algorithm will compare the first occurrence of $0\boldsymbol{u}_{k-1}$ with the second occurrence of $0\boldsymbol{u}_{k-1}$ all the way through before it concludes that $0\boldsymbol{u}_{k-1}$ and $0\boldsymbol{u}_{k-1}$ cannot be joined together. Thus, for $\boldsymbol{u}_k$, the very last $\boldsymbol{u}_0$ will be involved in k comparisons. $|\boldsymbol{u}_1| = 3{+}2|\boldsymbol{u}_0|$, $|\boldsymbol{u}_2| = 3{+}2|\boldsymbol{u}_1| = 3{+}2\cdot 3{+}2^2|\boldsymbol{u}_0|$, ..., $|\boldsymbol{u}_k| = 3{+}2\cdot 3{+}...{+}2^{k-1}\cdot 3{+}2^k|\boldsymbol{u}_0|$. Since $|\boldsymbol{u}_0| = 3$, we get $|\boldsymbol{u}_k| = 3{+}2\cdot 3{+}...{+}2^{k-1}\cdot 3{+}2^k\cdot 3 = (1{+}2{+}2^2{+}2^k)\cdot 3 = 2^{k+1}\cdot 3$. Thus $log(|\boldsymbol{u}_k|) = log(3){+}log(k{+}1)$, i.e. $log\big(\frac{|\boldsymbol{u}_k}{6}\big) = k$ as $log(2) = 1$. For $|\boldsymbol{u}_k| \geq 36$, $k = log\big(\frac{|\boldsymbol{u}_k}{6}\big) \geq \frac{1}{2}log(|\boldsymbol{u}_k|)$. So, the algorithm *TRLA* is forced to perform at least $|\boldsymbol{u}_k|\cdot\frac{1}{2}log(|\boldsymbol{u}_k|)$ steps. It follows that the complexity of *TRLA* is $\Theta(n\ log(n))$.

Note that we could start the scheme (F) with any binary Lyndon string as $\boldsymbol{u}_0$. The scheme (F) was used to generate the strings in the datasets *extreme_trla* to force the worst-case performance of *TRLA*.

4 The algorithm *BSLA*

The input strings for *BSLA* are tight strings over integer alphabets. Note that this requirement does not detract from the applicability of the algorithm as any string over an integer alphabet can easily be transformed in $\mathcal{O}(|\boldsymbol{x}|)$ time to a tight string so that the original string and the transformed string have the same Lyndon array. Thus, computing the Lyndon array for the transformed string gives also the Lyndon array for the original string.

The algorithm is based on a refinement of a list of groups of indices of the input string $\boldsymbol{x}$. 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.

Each group is assigned a specific substring of the input string referred to as the ***context*** of the group. Every index in a group has the property that there is an occurrence of the group's context at that position, or, equivalently, that the suffix starting at that index has the group's context as its prefix. Throughout the process the list of the groups is maintained in an increasing lexico-graphic order by their contexts. Moreover, at every stage, the contexts of all the groups are Lyndon substrings of $\boldsymbol{x}$ with an additional property that the contexts of complete groups are maximal Lyndon substrings. Hence, when the refinement is complete, the contexts of all the groups in the list represent all maximal Lyndon substrings of $\boldsymbol{x}$.

In order to verify the process and prove it correct, and to be able to describe the refinement in technical details, we must formally define several notions and properties in the next section.

4.1 The notation and the basic notions used in the analysis of *BSLA*

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

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

Definition 9. *An ordered list of groups $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$ is a **group configuration** for $\boldsymbol{x}$ if*

(C_1) $G_k \cup G_{k-1} \cup ... \cup G_2 \cup G_1 = 1..n$;

(C_2) $G_j \cap G_\ell = \emptyset$ *for any* $1 \le \ell < j \le k$;

(C_3) $con(G_k) \prec con(G_{k-1}) \prec ... \prec con(G_2) \prec con(G_1)$;

(C_4) *The context of G_j is a Lyndon substring of $\boldsymbol{x}$ for any $j \in 1..k$.*

Note that (C_1) and (C_2) guarantee that $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$ is a disjoint partitioning of $1..n$. For $i \in n$, $\boldsymbol{gr(i)}$ 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, $C(i) = \boldsymbol{x}[i..i+|con(gr(i))|-1]$.
The mapping ***prev*** is defined by $prev(i) = \max\{j < i : con(gr(j)) \prec con(gr(i))\}$ if such j exists, otherwise $prev(i) = nil$.
For a group G from a group configuration, we define an equivalence $\boldsymbol{\sim}$ on G as follows: $i \sim j$ iff $gr(prev(i)) = gr(prev(j))$ or $prev(i) = prev(j) = nil$. The symbol $\boldsymbol{[i]_\sim}$ 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 10. *Let G be a group from a group configuration for $\boldsymbol{x}$. Consider an $i \in G$ such that $prev(i) \neq nil$. Let $j_1 = \min[i]_\sim$ and $j_2 = \max[i]_\sim$. Then $[i]_\sim = \{j \in G \mid j_1 \le j \le j_2\}$.*

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

On each non-trivial class of $\sim$, we define a relation $\boldsymbol{\approx}$ as follows: $i \approx j$ iff $|j–i| = |con(G)|$; *in simple terms it means that the occurrence $C(i)$ of $con(G)$ is immediately followed by the occurrence $C(j)$ of $con(G)$.* The transitive closure of $\approx$ is a relation of equivalence, which we also denote by $\boldsymbol{\approx}$. The symbol $\boldsymbol{[i]_\approx}$ 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 $\boldsymbol{val(j)} = |[i]_\approx|$. *In simple terms, $val(i)$ is the number of elements from $[i]_\sim$ that are $\approx i$.* Thus, $1 \le val(i) \le |G|$.

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

Observation 11. *Let G be a group from a group configuration for $\boldsymbol{x}$. 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. Argue by contradiction. Assume that there is an $j \in [i]_\sim$ so that $j_1 < j < j_2$ so that $j \notin [i]_\approx$. 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. So $i \approx j' \approx j$ and so $j \approx i$, a contradiction. □

Definition 12. *A group G is **complete** if for any $i \in G$, the occurrence $C(i)$ of $con(G)$ is a maximal Lyndon substring of $\boldsymbol{x}$.*
*A group configuration $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$ is **t-complete**, $1 \leq t \leq k$, if*

(C_5) *the groups $G_t, ..., G_1$ are complete;*
(C_6) *the mapping prev is **proper** on G_t:*
for any $i \in G_t$, if $prev(i) \neq nil$ and $v = val(i)$, then there are $i_1, ..., i_v \in G_t$, $i \in \{i_1, ..., i_v\}$, $prev(i) = prev(i_1) = ... = prev(i_v)$, and so that $C(prev(i))C(i_1)...C(i_v)$ is a prefix of $\boldsymbol{x}[j..n]$;
(C_7) *the family $\{C(i) \mid i \in 1..n\}$ is **proper**:*
(a) *if $C(j)$ is proper substring of $C(i)$, i.e. $C(j) \subsetneq C(i)$, then $con(G_t) \prec con(gr(j))$,*
(b) *if $C(i)$ is followed immediately by $C(j)$, i.e. when $i{+}|con(gr(i))| = j$, and $C(i) \prec C(j)$, then $con(gr(j)) \preceq con(G_t)$;*
(C_8) *the family $\{C(i) \mid i \in 1..n\}$ has the **Monge** property, i.e. if $C(i) \cap C(j) \neq \emptyset$, then $C(i) \subseteq C(j)$ or $C(j) \subseteq C(i)$.*

The condition (C_6) is all-important for carrying out the refinement process (see (R_3) below). The conditions (C_7) and (C_8) are necessary for asserting that the condition (C_6) is preserved during the refinement process.

4.2 The refinement

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

Lemma 13. *Let $\{a_1, ..., a_k\}$ be the symbols occurring in $\boldsymbol{x}$ listed in the increasing alphabetical order. For $1 \leq \ell \leq k$, define $G_\ell = \{i \in 1..n : \boldsymbol{x}[i] = a_{k+1-\ell}\}$ with context $a_{k+1-\ell}$. Then $\langle G_k, ..., G_1\rangle$ is a 1-complete group configuration.*

Proof. (C_1), (C_2), (C_3), and (C_4) are straightforward to verify. To verify (C_5), we need to show that G_1 is complete. Any occurrence of a_k in $\boldsymbol{x}$ is a maximal Lyndon substring, so G_1 is complete.
To verify (C_6), consider $j = prev(i)$ and $val(i) = v$ for $i \in G_1$. Consider any $j < \ell < i$. If $\boldsymbol{x}[\ell] \neq a_k$, then $prev(i) < \ell$ which contradicts the definition of $prev$. Hence $\boldsymbol{x}[\ell] = a_k$ and so $\boldsymbol{x}[j{+}1] = ... = \boldsymbol{x}[i] = ...\boldsymbol{x}[j{+}v{+}1] = a_k$ while $\boldsymbol{x}[j] = a_\ell$ for some $\ell < k$. It follows that $\boldsymbol{x}[j..n]$ has $a_\ell(a_k)^v$ as a prefix.
The condition ($C_7(a)$) is trivially satisfied as no $C(i)$ can have a proper substring. If $C(i)$ is

immediately followed by $C(j)$ and $C(i) \prec C(j)$, then $C(i) = \boldsymbol{x}[i]$, $j = i{+}1$, $C(j) = \boldsymbol{x}[i{+}1]$ and $\boldsymbol{x}[i] \prec \boldsymbol{x}[i{+}1]$. Then $con(C(j)) = \boldsymbol{x}[i{+}1] \preceq a_k = con(G_1)$, so $(C_7(b))$ is also satisfied. To verify (C_8), consider $C(i) \cap C(j) \neq \emptyset$. Then $C(i) = \boldsymbol{x}[i] = \boldsymbol{x}[j] = C(j)$. $\square$

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

(R_1) Partition G_t into classes of the equivalence $\sim$.
$G_t = [i_1]_\sim \cup [i_2]_\sim \cup ... \cup [i_p]_\sim \cup X$ where $X = \{i \in G_t : prev(i) = nil\}$ may be possibly empty and $i_1 < i_2 < ... < i_p$.

(R_2) Partition every class $[i_\ell]_\sim$, $1 \leq \ell \leq p$, into classes of the equivalence $\approx$.
$[i_\ell]_\sim = [j_{\ell,1}]_\approx \cup [j_{\ell,2}]_\approx \cup ... \cup [j_{\ell,m_\ell}]_\approx$ where $val(j_{\ell,1}) < val(j_{\ell,2}) < ... < val(j_{\ell,m_\ell})$.

(R_3) So we have a list of classes in this order: $[j_{1,1}]_\approx$, $[j_{1,2}]_\approx$, ... $[j_{1,m_1}]_\approx$, $[j_{2,1}]_\approx$, $[j_{2,2}]_\approx$, ... $[j_{2,m_2}]_\approx$, ..., $[j_{p,1}]_\approx$, $[j_{p,2}]_\approx$, ... $[j_{p,m_p}]_\approx$. This list is processed from left to right. Note that for each $i \in [j_{\ell,k}]_\approx$, $prev(i) \in gr(j_{\ell,k})$ and $val(i) = val(j_{\ell,k})$.
For each $j_{\ell,k}$, move all elements $\{prev(i) : i \in [j_{\ell,k}]_\approx\}$ from the group $gr(prev(j_{\ell,k}))$ into a new group H and place H in the list of groups right after the group $gr(prev(j_{\ell,k}))$ and set its context to $con(gr(prev(j_{\ell,k})))con(gr(j_{\ell,k}))^{val(j_{\ell,k})}$. $\big($*Note, that this "doubling of the contexts" is possible due to* $(C_6)\big)$. 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 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 14 shows that having a t-complete group configuration $\langle G_k, ..., G_{t+1}, G_t, ..., G_1\rangle$ and refining it by G_t, then the resulting system of groups is a $(t{+}1)$-complete group configuration. This allows to carry on the refinement in an iterative fashion.

Theorem 14. *Let $Conf = \langle G_k, ..., G_{t+1}, G_t, ..., G_1\rangle$ 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 $gr()$, $con()$, $C()$, $prev()$, and $val()$ denote the functions for $Conf$, while $gr'()$, $con'()$, $C'()$, $prev'()$, and $val'()$ denote the functions for $Conf'$.

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, and $con(gr(prev(j))) \prec con(gr(j))$, so $con(gr(prev(j)))con(gr(j))^{val(j)}$ is Lyndon as well and thus (C_4) holds. That concludes the proof of Claim 1.

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

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

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

Consider $C'(i)$ and $C'(j)$ for some $1 \le i < j \le n$.

- Case $C'(i) = C(i)$ and $C'(j) = C(j)$.
  - Show that $(C_7(a))$ holds.
    If $C'(j) \subsetneq C'(i)$, then $C(j) \subsetneq C(i)$, 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.
  - Show that (C_8) holds.
    If $C'(i) \cap C'(j) \ne \emptyset$, then $C(i) \cap C(j) \ne \emptyset$, so $C(j) \subseteq C(i)$, and so $C'(j) \subseteq C'(i)$, so (C_8) for *Conf'* holds.
- Case $C'(i) = C(i)$ and $C'(j) = C(j)C(j_1)..C(j_w)$,
  where $w = val(j_1)$, $C(j_1) = ... = C(j_w) = con(G_t)$, and $j_1 \approx ... \approx j_w$.
  - Show that $(C_7(a))$ holds.
    If $C'(j) \subsetneq C'(i)$, then $C(j)C(j_1)..C(j_w) \subsetneq C(i)$, hence $C(j) \subsetneq C(i)$, and so by $(C_7(a))$ for *Conf*, $con(G_t) \prec con(gr(j))$. By t-completeness of *Conf*, $C(j)$ is a maximal Lyndon substring, a contradiction with $C(j)C(j_1)..,C(j_w)$ being Lyndon. This is an impossible case.
  - Show that (C_8) holds.
    If $C'(i) \cap C'(j) \ne \emptyset$, then $C(j) \subseteq C(i)$ by (C_8) for *Conf*. By $(C_7(a))$ for *Conf*, $C(j)$ cannot be a suffix of $C(i)$ as $con(gr(j)) \prec con(G_t)$. Hence $C(i) \cap C(j_1) \ne \emptyset$, and so $C(j)C(j_1) \subseteq C(i)$ and since $C(j_1)$ cannot be a suffix of $C(i)$ as $gr(j_1) = G_t$, it follows that $C(i) \cap C(j_2) \ne \emptyset$, ..., ultimately giving $C(j)C(j_1)...C(j_w) \subseteq C(i)$. So (C_8) for *Conf'* holds.
- Case $C'(i) = C(i)C(i_1)..C(i_v)$ and $C'(j) = C(j)$,
  where $v = val(i_1)$, $C(i_1) = ... = C(i_v) = con(G_t)$, and $i_1 \approx ... \approx i_v$.
  - Show that $(C_7(a))$ holds.
    If $C'(j) \subsetneq C'(i)$, then either $C(j) \subsetneq C(i)$, which implies by $(C_7(a))$ for *Conf* that $con(G_t) \prec con(gr(j))$, giving $con'(H_{t+1}) \prec con'(G_t) = con(G_t) \prec con(gr(j)) = con'(gr'(j))$, or $C(j) \subseteq C(i_\ell)$ for some $1 \le \ell \le v$. If $C(j) = C(i_\ell)$, then $gr(j) = gr(i_\ell) = G_t$, giving $con'(H_{t+1}) \prec con(G_t) = con(gr(j))$. So $(C_7(a))$ for *Conf'* holds.
  - Show that (C_8) holds.
    Let $C'(i) \cap C'(j) \ne \emptyset$. Consider $\mathcal{D} = \{i_\ell \mid 1 \le \ell \le v \;\&\; C(j) \cap C(i_\ell) \ne \emptyset\}$.
    Assume that $\mathcal{D} \ne \emptyset$:

By (C_8) for *Conf*, either $C(j) \subseteq \bigcup_{i_\ell \in \mathcal{D}} C(i_\ell) \subseteq C'(i)$ and we are done, or $\bigcup_{i_\ell \in \mathcal{D}} C(i_\ell) \subseteq C(j)$. Let i_k be the smallest element of $\mathcal{D}$. Since $C(i_k)$ cannot be prefix of $C(j)$, it means that $i_k = i_1$. Since $C(i_1)$ cannot be a prefix of $C(j)$, it means that $C(i) \cap C(j) \neq \emptyset$, and so $C(j) \subseteq C(i)$, which contradicts the fact that $C(j) \subseteq \bigcup_{i_\ell \in \mathcal{D}} C(i_\ell) \subseteq C'(i)$.

Assume that $\mathcal{D} = \emptyset$:

Then $C(i) \cap C(j) \neq \emptyset$, and so by (C_8) for *Conf*, $C(j) \subseteq C(i) \subseteq C'(i)$ as $i < j$.

- Case $C'(i) = C(i)C(i_1)..C(i_v)$ and $C'(j) = C(j)C(j_1)...C(j_w)$, where $v = val(i_1)$, $C(i_1) = ... = C(i_v) = con(G_t)$, and $i_1 \approx ... \approx i_v$, and where $v = val(j_1)$, $C(j_1) = ... = C(j_w) = con(G_t)$, and $j_1 \approx ... \approx j_w$.
  - Show that $(C_7(a))$ holds.
    Let $C'(j) \subsetneq C'(i)$. Then either $C(j) \subseteq C(i)$ and so $con(G_t) \prec con(gr(j))$, implying that $C(j)$ is maximal contradicting $C(j)C(j_1)...C(j_w)$ being Lyndon. Thus $C(j) \subsetneq C(i_\ell)$ for some $1 \leq \ell \leq v$. But then $con(G_t) \prec con(gr(j))$, implying that $C(j)$ is maximal, again a contradiction. This is an impossible case.
  - Show that (C_8) holds.
    Let $C'(i) \cap C'(j) \neq \emptyset$. Let us first assume that $C(i) \cap C(j) \neq \emptyset$. Then $C(j) \subseteq C(i)$. Since $C(j)$ cannot be a suffix of $C(i)$, it follows that $C(i) \cap C(j_1) \neq \emptyset$. Therefore, $C(j)C(j_1) \subseteq C(i)$. Repeating this argument leads to $C(j)C(j_1)...C(j_w) \subseteq C(i)$ and we are done.
    So assume that $C(i) \cap C(j) = \emptyset$. Let $1 \leq \ell \leq v$ be the smallest such that $C(i_\ell) \cap C(j) \neq \emptyset$. Such ℓ must exists. Then $i_\ell \leq j$. If $i_\ell = j$, then either $C(i_\ell)$ is a prefix of $C(j)$ or vice versa, both impossibilities, hence $i_\ell < j$. Repeating the same arguments as for i, we get that $C(j)C(j_1)..C(j_w) \subseteq C(i_\ell)$ 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 $C'(i) \prec C'(j)$.

- Assume that $gr'(j) \in \{G_{t-1}, ..., G_1\}$.
  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 $C'(i) = C(i)C(i_1)...C(i_v)$ 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.*
- Assume that $gr'(j) = G_t$.
  Then the group $gr(i)$ were 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 $val(j) = v+1$.

That 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 $[i]_\approx = \{i_1, ..., i_v\}$,

where $i_1 < i_2 < ... < i_v$. Hence $i_1, ..., i_v \in H_{t+1}$ and $C'(i_1) = ... = C'(i_v) = \mathit{con}'(H_{t+1})$ and $j = \mathit{prev}'(i) = \mathit{prev}'(i_1) = ... = \mathit{prev}'(i_v)$ and so $j < i_1$. It remains to show that $C'(j)C'(i_1)...C'(i_v)$ is a prefix of $\boldsymbol{x}[j..n]$. 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 $(C_7(a))$, $\mathit{con}'(H_{t+1}) \prec \mathit{con}'(\mathit{gr}'(i_1)) = \mathit{con}'(H_{t+1})$, a contradiction.
Thus, $C'(j) \cap C'(i_1) = \emptyset$. Set $j_1 = j + |\mathit{con}'(\mathit{gr}'(j))|$. It follows that $j_1 \leq i_1$. Assume that $j_1 < i_1$. Since $j = \mathit{prev}'(i_1)$ and $j < i_1$, $\mathit{con}'(\mathit{gr}'(j_1)) \succeq \mathit{con}'(\mathit{gr}'(i_1)) = \mathit{con}'(H_{t+1})$. Since $j_1 \notin H_{t+1}$, $\mathit{con}'(\mathit{gr}'(j_1)) \succ \mathit{con}'(H_{t+1})$. 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 $(C_7(a))$, $\mathit{con}'(H_{t+1}) \prec \mathit{con}'(\mathit{gr}'(i_1)) = \mathit{con}'(H_{t+1})$, a contradiction. Thus $C'(j_1) \cap C'(i_1) = \emptyset$. Since $C'(j_1)$ immediately follows $C'(j)$, by $(C_7(b))$, $\mathit{con}'(\mathit{gr}'(j_1)) \preceq \mathit{con}'(H_{t+1})$, a contradiction. Therefore $j_1 = i_1$, and so prev' is proper on H_{t+1}. That 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 $k \geq i+|\mathit{con}'(H_{t+1})|$, $\boldsymbol{x}[i..k]$ is a maximal Lyndon substring of $\boldsymbol{x}$.
Either $k = n$ and so $\mathit{con}'(\mathit{gr}'(k)) = \boldsymbol{x}[k]$ and so $C'(k)$ is a suffix of $\boldsymbol{x}[i..k]$, or $k < n$ and then $\boldsymbol{x}[k+1] \prec \boldsymbol{x}[k]$ since $\boldsymbol{x}[k+1] \preceq \boldsymbol{x}[k]$ implies that $\boldsymbol{x}[i..k+1]$ is Lyndon, a contradiction with the maximality of $\boldsymbol{x}[i..k]$. Consider $C'(k)$, then $C'(k) \subseteq \boldsymbol{x}[i..k]$ and so $C'(k) = \boldsymbol{x}[k]$.
Therefore, there is j_1 so that $i+|\mathit{con}'(H_{t+1})| \leq j_1 \leq k$ and $C'(j_1)$ is a suffix of $\boldsymbol{x}[i..k]$.Take the smallest j_1 such. If $j_1 = i+|\mathit{con}'(H_{t+1})|$, then $C'(i) \prec C'(j_1)$ as $\boldsymbol{x}[i..k] = C'(i)C'(j_1)$ is Lyndon. By $(C_7(b))$, $C'(j_1) \preceq \mathit{con}'(H_{t+1})$, so we have $\mathit{con}'(H_{t+1}) = C'(i) \prec C'(j_1) \preceq \mathit{con}'(H_{t+1})$, a contradiction.
Therefore, $j_1 > i+|\mathit{con}'(H_{t+1})|$. Consider $\boldsymbol{x}[j_1-1]$. If $\boldsymbol{x}[j_1-1] \preceq \boldsymbol{x}[j_1]$, $\boldsymbol{x}[j_1-1..k]$ is Lyndon, and since $\boldsymbol{x}[j_1..k] = C'(j_1)$, $\boldsymbol{x}[j_1-1..k]$ would be a context of $\mathit{gr}'(j_1-1)$, and this contradicts the fact the j_1 was chosen to be the smallest such. Therefore, $\boldsymbol{x}[j_1-1] \succ \boldsymbol{x}[j_1]$ and so $\mathit{con}'(\mathit{gr}'(j_1-1)) = \boldsymbol{x}[j_1-1]$. Thus, there is j_2, $i+|\mathit{con}'(H_{t+1})| \leq j_2 < j_1 \leq k$ and $C'(j_2)$ is a suffix of $\boldsymbol{x}[i..j_1-1]$. Take the smallest such j_2. If $C'(j_2) \prec C'(j_1)$, then by $(C_7(b))$, $C'(j_1) \preceq \mathit{con}'(H_{t+1})$, a contradiction. Hence $C'(j_2) \succeq C'(j_1)$. If $j_2 = i+i+|\mathit{con}'(H_{t+1})|$, then $\boldsymbol{x}[i..k] = C'(i)C'(j_2)C'(j_1)$ and so by $(C_7(b))$, $C'(j_2) \preceq \mathit{con}'(H_{t+1})$, a contradiction. Hence $i+|\mathit{con}'(H_{t+1})| < j_2$.
The same argument done for j_2 can be now done for j_3. We end up with $i+|\mathit{con}'(H_{t+1})| \leq j_3 < j_2 < j_1 \leq k$ and with $C'(j_3) \succeq C'(j_2) \succeq C'(j_1) \succ \mathit{con}'(H_{t+1})$. If $i+|\mathit{con}'(H_{t+1})| = j_3$, then we have a contradiction, so $i+|\mathit{con}'(H_{t+1})| < j_3$. These argument can be repeated only finitely many times, and we obtain $i+|\mathit{con}'(H_{t+1})| = j_\ell < j_{\ell-1} < ... < j_2 < j_1 \leq k$ so that $\boldsymbol{x}[i..k] = C'(i)C'(j_\ell)C'(j_{\ell-1}...C'(j_2)C'(j_1)$, which is a contradiction.
Therefore, our initial assumption that $C'(i)$ is not maximal always leads to a contradiction. That concludes the proof of Claim 4.

The four claims show that all the conditions (C_1) ... (C_8) are satisfied for Conf', and that proves Theorem 14. □

As the last step, we show that when the process of refinement is completed, all maximal

Lyndon substrings of $\boldsymbol{x}$ are identified and sorted via the contexts of the groups of the final configuration.

Theorem 15.
Let $Conf_1 = \langle G^1_{k_1}, G^1_{k_1-1}, ..., G^1_2, G^1_1\rangle$ with $gr_1()$, $con_1()$, $C_1()$, $prev_1()$, and $val_1()$ be the initial 1-complete group configuration from Lemma 13.

Let $Conf_2 = \langle G^2_{k_2}, G^2_{k_2-1}, ..., G^2_2, G^2_1\rangle$ 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 $Conf_3 = \langle G^3_{k_3}, G^3_{k_3-1}, ..., G^3_2, G^3_1\rangle$ 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 $Conf_r = \langle G^r_{k_r}, G^r_{k_r-1}, ..., G^r_2, G^r_1\rangle$ 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 $\boldsymbol{x}[i..k]$ is a maximal Lyndon substring of $\boldsymbol{x}$ iff $\boldsymbol{x}[i..k] = C_r(i) = con_r(gr_r(i))$.

Proof. That all the groups of $Conf_r$ are complete follows from Theorem 14 and hence every $C_r(i)$ is a maximal Lyndon string. Let $\boldsymbol{x}[i..k]$ be a maximal Lyndon substring of $\boldsymbol{x}$. Consider $C_r(i)$, since it is maximal, it must be equal to $\boldsymbol{x}[i..k]$. □

4.3 Motivation for the refinement

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

(a) lower the water level by one
(b) extend the existing Lyndon substrings
the revealed letters are used to extend the existing Lyndon substrings where possible, or became Lyndon substrings of length 1 otherwise;
(c) 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 Fig. 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 Fig. 5, we illustrate the process:

(1) We start with the string 011023122 and a full tank of water.
(2) We drain one level, only 3 is revealed, nothing to extend, nothing to consolidate.
(3) 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 1, nothing to consolidate.

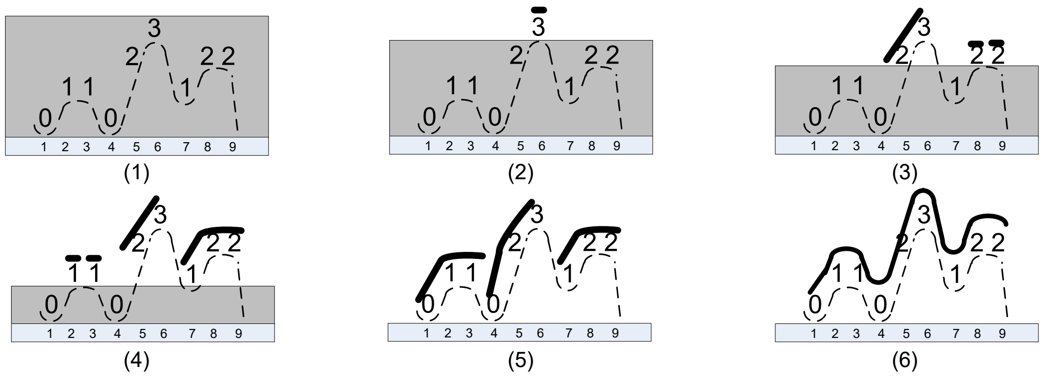


Figure 5: The water draining method for 011023122

(4) We drain one more level and three 1's are revealed, the first two 1's form Lyndon substrings 1 of length 1, the third 1 extends 22 to 122, 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.

So, during the process, the following maximal Lyndon substrings were identified: 3 at position 6, 23 at position 5, 2 at positions 8, 9, 1 at positions 2, 3, 122 at position 7, 023 at position 4, and finally 011023122 at position 1. Note that all positions are accounted for, we really got all maximal Lyndon substrings of the string 011023122.

Here we present an illustrative example for a string 011023122, where the arrows represent the *prev* mapping shown only on the group used for the refinement which is indicated by the bold font.

$$\begin{array}{ccccccccc} \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{2} & \mathbf{3} & \mathbf{1} & \mathbf{2} & \mathbf{2} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}$$

$$\begin{tikzcd} G_0=\{1,4\} & G_1=\{2,3,7\} & G_2=\{5,8,9\} & \boldsymbol{G_3=\{6\}} \arrow[l, bend left] \end{tikzcd}$$

$$\begin{tikzcd} G_0=\{1,4\} & G_1=\{2,3,7\} & G_2=\{8,9\} & \boldsymbol{G_{23}=\{5\}} \arrow[lll, bend left] & G_3=\{6\} \end{tikzcd}$$

$$\begin{tikzcd} G_0=\{1\} & G_{023}=\{4\} & G_1=\{2,3,7\} & \boldsymbol{G_2=\{8,9\}} \arrow[l, bend left] \arrow[l, bend left=50] & G_{23}=\{5\} & G_3=\{6\} \end{tikzcd}$$

$$\begin{tikzcd} G_0=\{1\} & G_{023}=\{4\} & G_1=\{2,3\} & \boldsymbol{G_{122}=\{7\}} \arrow[ll, bend left] & G_2=\{8,9\} & G_{23}=\{5\} & G_3=\{6\} \end{tikzcd}$$

$$\begin{tikzcd} G_0=\{1\} & G_{023122}=\{4\} & \boldsymbol{G_1=\{2,3\}} \arrow[ll, bend left] \arrow[ll, bend left=50] & G_{122}=\{7\} & G_2=\{8,9\} & G_{23}=\{5\} & G_3=\{6\} \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] G_{011} = \{1\} & \boldsymbol{G_{023122} = \{4\}} \arrow[l, bend left] & G_1 = \{2,3\} & G_{122} = \{7\} & G_2 = \{8,9\} & G_{23} = \{5\} & G_3 = \{6\} \end{tikzcd}$$

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

5 Measurements

All the measurements were performed on the ***moore*** server of the Department of Computing and Software; Memory: 32GB (DDR4 @ 2400 MHz), CPU: 8 of the Intel Xeon E5-2687W v4 @ 3.00GHz, OS: Linux version 2.6.18-419.el5 (gcc version 4.1.2) (Red Hat 4.1.2-55), further all the programs were compiled without any level of optimization.

The CPU time was measured for each of the programs in seconds with the precision of 3 decimal places (i.e. milliseconds). 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 1 for strings of length 10^6), resulting in much higher precision (of up to 7 decimal places). Thus, for graphing, the logarithmic scale was used for both, the x-axis representing the length of the strings, and the y-axis representing the time.

There were 4 categories of datasets: random binary strings over the alphabet $\{0,1\}$, random 4-ary strings (kind of random DNA) over the alphabet $\{0,1,2,3\}$, random 26-ary strings (kind of random English) over the alphabet $\{0,1,...,25\}$, and random strings over integer alphabet $\{01,2,...,k\}$ where $k \leq n$ and n is the length of the string. Each of the dataset contained 500 randomly generated strings of the same length. For each category, there were datasets for length 10, 50, 10^2, $5{\cdot}10^2$, ..., 10^5, $5{\cdot}10^5$, and 10^6. The average time for each dataset was computed and used in the following graphs.

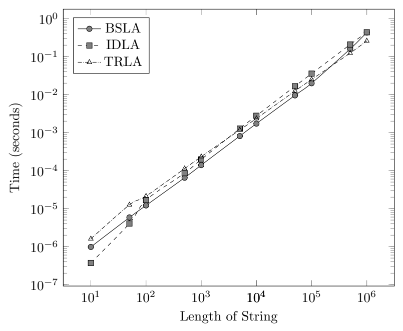


Figure 6 Binary strings

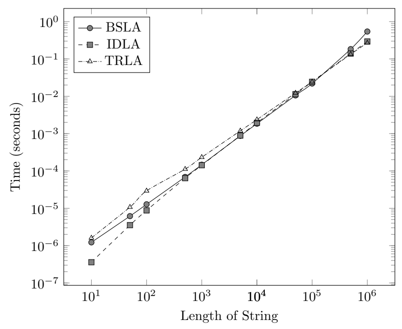


Figure 7 4-ary strings

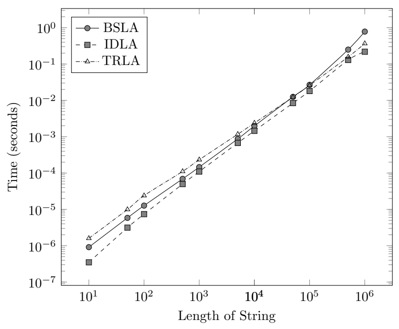


Figure 8 26-ary strings

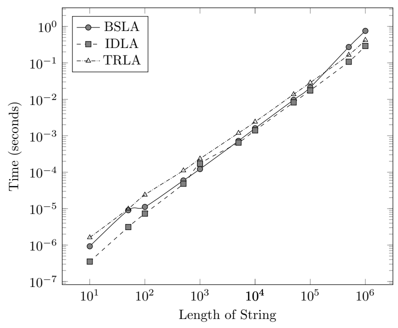


Figure 9 strings over integer alphabet

As the graphs clearly indicate, the performance of the three algorithms is virtually indistinguishable. We expected *IDLA* and *TRLA* to exhibit linear behaviour on random strings as such strings tend to have almost all maximal Lyndon factors short with respect the length of the strings. However, we did not expect the results to be so close.

We also tested all three algorithms on datasates containing a single string 01234...n referred to as a *extreme_idla* string, which, of course makes *IDLA* exhibit its quadratic complexity, and indeed the results show it:

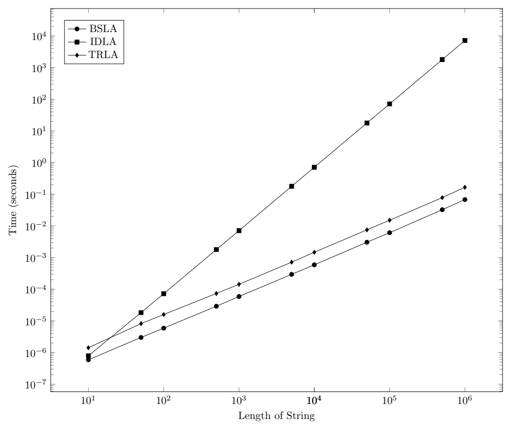


Figure 10 extreme_idla strings

The *extreme_trla* strings were generated according to the scheme (F) in Section 3.5. These string force worst-case execution for *TRLA*. However, even $log(10^6)$ is too small to really illustrate the difference, so the results were again very close:

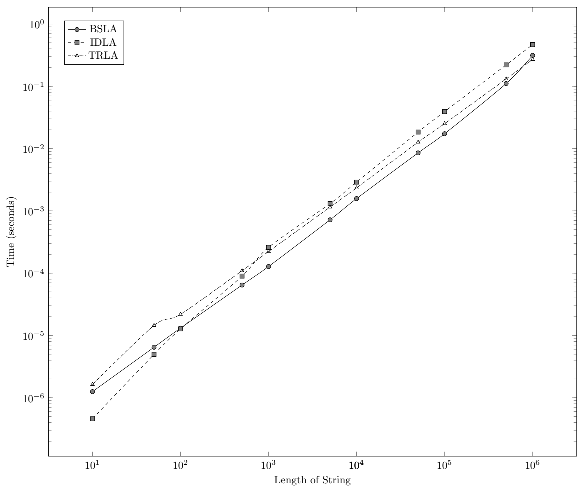


Figure 10 extreme_trla strings

6 Conclusion and Future Work

We presented two novel algorithms for computing maximal Lyndon substrings. The first one, *TRLA*, is mostly of theoretical interest since it has the worst-case complexity $\Theta(n\log(n))$ for strings of length n. Interestingly, on random strings it slightly outperformed *BSLA*, the second algorithm, which is linear and elementary in the sense that it does not require a preprocessed global data structure. Being linear and elementary, *BSLA* is more interesting and it is possible that its performance could be more streamlined. Additional effort will go into improving *TRLA*'s complexity in the computation of the missing values. Both algorithms need to be compared to some efficient implementation of *SSLA* and *BWLA*.

References

[1] `C++` code for IDLA, TRLA and BSLA algorithms. Available at `http://www.cas.mcmaster.ca/~franek/research.html`.

[2] U. Baier. Linear-time suffix sorting — a new approach for suffix array construction. M.Sc. Thesis, University of Ulm, Ulm, Germany, 2015.

[3] U. Baier. Linear-time suffix sorting — a new approach for suffix array construction. In R .Grossi and M. Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "Runs" Theorem. Available at `https://arxiv.org/abs/1406.0263`, 2015.

[5] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The “Runs” Theorem. *SIAM J. COMPUT.*, 46:1501–1514, 2017.

[6] G. Chen, S.J. Puglisi, and W.F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2013.

[7] M. Crochemore, L. Ilie, and W.F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *Proc. 18th Data Compression Conference*, pages 482–488, 2008.

[8] C. Digelmann. Personal communication. 2016.

[9] J-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[10] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th IEEE Symp. Foundations of Computer Science*, pages 137–143. IEEE, October 1997.

[11] F. Franek, M. Liut, and W.F. Smyth. On Baier's sort of maximal Lyndon substrings. In *Proceedings of Prague Stringology Conference 2018*, pages 63–78, 2018.

[12] F. Franek, A. Paracha, and W.F. Smyth. The linear equivalence of the suffix array and the partially sorted Lyndon array. In *Proc. Prague Stringology Conference*, pages 77–84, 2017.

[13] F. Franek, A.S.M. Sohidull Islam, M. Sohel Rahman, and W.F. Smyth. Algorithms to compute the Lyndon array. In *Proceedings of Prague Stringology Conference 2016*, pages 172–184, 2016.

[14] C. Hohlweg and C. Reutenauer. Lyndon words, permutations and trees. *Theoretical Computer Science*, 307(1):173–178, 2003.

[15] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th international conference on Automata, languages and programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer–Verlag.

[16] D. Kosolobov. Lempel-Ziv factorization may be harder than computing all runs. Avaiable at `https://arxiv.org/abs/1409.5641`, 2014.

[17] F.A. Louza, W.F. Smyth, G. Manzini, and G.P. Telles. Lyndon array construction during Burrows–Wheeler inversion. *Journal of Discrete Algorithms*, 50:2–9, 2018.

[18] G. Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):1–15, 2013.

[19] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202, 2009.

[20] A. Paracha. Lyndon factors and periodicities in strings. Ph.D. Thesis, McMaster University, Hamilton, Ontario, Canada, 2017.