Article

Computing Maximal Lyndon Substrings of a String

Frantisek Franek [1,†], Michael Liut [2,*,†]

[1] Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada; franek@mcmaster.ca
[2] Department of Mathematical and Computational Sciences, University of Toronto Mississauga, Mississauga, Ontario, Canada michael.liut@utoronto.ca
* Correspondence: michael.liut@utoronto.ca
† These authors contributed equally to this work.

Abstract: There are two reasons to have an efficient algorithm for identifying all 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 maximal Lyndon substrings of the input string, and secondly, Franek et al. showed in 2017 a linear 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. Among the presented were two well-known algorithms for computing the 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 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 that 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 theoretical analysis of these two algorithms and provides empirical comparisons of both their C++ implementations with respect to iterated Duval's algorithm.

Keywords: Combinatorics on Words; String Algorithms; Regularities in Strings; Suffix Sorting; Lyndon Substrings; Lyndon Arrays; Maximal Lyndon Substrings; Tau-Reduction Algorithm; Baier's Sort Algorithm; Iterative Duval Algorithm;

1. Introduction

There are at least two reasons for having an efficient algorithm for identifying all maximal Lyndon substrings of a string: firstly, Bannai et al. introduced in 2015 (arXiv, [1]) and published in 2017, [2], a linear algorithm to compute all runs in a string that depends on knowing all maximal Lyndon substrings of the input string, and secondly, in 2017, Franek et al. in [3] showed a linear 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, [4]), and published in 2016, [5].

The most significant feature of the runs algorithm presented in [2] 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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

which approach may be more efficient: to compute the Lempel-Ziv factorization or to compute all maximal Lyndon substrings. There are several efficient linear algorithms for Lempel-Ziv factorization (e.g. see [6,7] and the references therein).

Interestingly, Kosolobov [8] shows 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 [4], and published in [5], a new algorithm for suffix sorting. Though Lyndon strings are never mentioned in [4,5], it was noticed by Cristoph Diegelmann in a personal communication, [9], 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 [10]: 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(x[1..n], j, Σ, ≺) : integer
  i ← j + 1; max ← 1
  while i ≤ n do
      k ← 0
    while x[j + k] = x[i + k] do
      k ← k + 1
    if x[j + k] ≺ x[i + k] then
      i ← i + k + 1; max ← i − 1
    else
      return max
procedure IDLA(x[1..n], j, Σ, ≺) : integer array
  i ← 1
  while i < n do
    L[i] = MaxLyn(x[i..n], j, Σ, ≺)
    i ← i + 1
  L[n] ← 1
  return L
```

Figure 1. Algorithm *IDLA*

In an overview [10], Franek et al. discussed an algorithm based on an iterative application of Duval's Lyndon factorization algorithm, [11], which we refer here as *IDLA*, and an algorithmic scheme based on Hohlweg and Reutenauer's work [12], 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 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. Its weakness is its quadratic worst-case complexity that becomes a problem for longer strings with long maximal Lyndon substrings as one of our experiments showed.

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*, see Fig. 1, really computes the maximal Lyndon prefix is not obvious and is based on the properties of periods of prefixes, see [11], or Observation 6 and Lemma 11 in [10]. The following lemma characterizes maximal Lyndon substrings in terms of the relationships of the suffixes, and follows from the work of Hohlweg and Reutenauer [12].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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 [10]. Computing the inverse suffix array and applying *NSV* are "naturally" linear, and computing the suffix array can be implemented to be linear, see [10,13] 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, [14]. 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, 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 [15].

In this work, we present two additional algorithms for Lyndon array not discussed in [10]. The C++ source code of the three implementations, *IDLA*, *TRLA*, and *BSLA* is available, see [16]. Note that the procedure *IDLA* is in `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 [17], and reproduced very successfully in all linear algorithms for suffix sorting (e.g. see [13,18] and the references therein). Farach's approach follows Cooley-Tukey algorithm for the fast Fourier transform relying on recursion to lower the quadratic complexity to $O(n \log(n))$ complexity, see [19]. *TRLA* was first introduced by the authors in 2019, see [20,21], and presented as a part of Liut's Ph.D. thesis [22].

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, [15], and also presented as a part of Liut's Ph.D. thesis [22] 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 analyzed. In Section 4, the *BSLA* algorithm is presented and analyzed. In Section 7, the empirical measurements of the performance of *IDLA*, *TRLA*, and *BSLA* are presented on data sets with random strings of various lengths and over various alphabets. The results are presented both in tabular and graphical forms. In Section 6, the conclusion of the research and the future work are presented.

2. Basic Notation and Terminology

For two integers $i \leq j$, the ***range*** $i..j = \{k \, integer : i \leq k \leq j\}$. 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 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 \$, where $\$ \notin \mathcal{A}$. We use the array notation indexing from 1 for strings, thus $\boldsymbol{x}[1..n]$ indicates a string of length n, the first

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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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. By a ***constant alphabet*** we mean a fixed finite alphabet. A string x is over an ***integer alphabet*** if $\mathcal{A}_x \subseteq \{0, 1, ..., |x|\}$. Thus, the class of ***strings over integer alphabets*** $= \{x \mid x \text{ is a string over } \{0, 1, ..., |x|\}\}$. A string x over an integer alphabet is ***tight*** if $\mathcal{A}_x = \{0, 1, ..., k\}$ for some $k \leq |x|$. For instance, $x = 010$ is tight as $\mathcal{A}_x = \{0, 1\}$, while $y = 020$ is not as $\mathcal{A}_y = \{0, 2\}$; i.e. 1 is missing from $\mathcal{A}_y$.

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 0. 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 a ***trivial prefix*** (respectively, ***trivial substring***, ***trivial 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-trivial proper prefix and a non-trivial proper suffix of x. Thus, both ε and x are trivial borders of 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 $x = x[1..n]$ and $y = y[1..n]$, $x \prec y$ if either x is a proper prefix of y, or there is a $j \leq \min\{n, m\}$ so that $x[1] = y[1]$, ..., $x[j-1] = y[j-1]$ 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 $x[1..n]$, $1 \leq i \leq j \leq n$ is a ***maximal Lyndon substring of*** x if it is Lyndon and either $j = n$ or for any $k > j$, $x[i..k]$ is not 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 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. τ-Reduction Algorithm – *TRLA*

The first idea of the algorithm was proposed in Paracha's 2017 Ph.D. thesis, [23]. It follows Farach's approach [17]:

(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.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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 [24], 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 at most $O(n\ log(n))$ steps 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 $\boldsymbol{x}$ can be computed in linear time, the overall execution of algorithm is linear as well, and thus, the average case complexity is linear in the length of the input string. 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. In particular, the values 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 $\underline{0001\underline{\underline{1}}\,\underline{00\underline{\underline{1}}}}$ and would need to compute the missing maximal Lyndon substrings that are underlined in $\underline{\underline{0001\underline{1}00\underline{1}}}$. 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 $\prec$ where $\boldsymbol{x}[n+1] = \$$ and $\$ \prec a$ for any $a \in \mathcal{A}_{\boldsymbol{x}}$. A ***$\tau$-pair*** consists of a pair of adjacent positions from the range $1..n+1$. The τ-pairs are computed by induction:

- the initial τ-pair is $(1,2)$;
- 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 $\boldsymbol{x}[i-1] \succ \boldsymbol{x}[i]$ **and** $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i+1]$ **then**
    the next τ-pair is set to $(i,i+1)$
  else
    the next τ-pair is set to $(i+1,i+2)$

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 Figure 2, for the formal proof see Lemma 2.

Lemma 2. *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) *$|(i_j,i_j+1) \cap (i_\ell,i_\ell+1)| \leq 1$.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

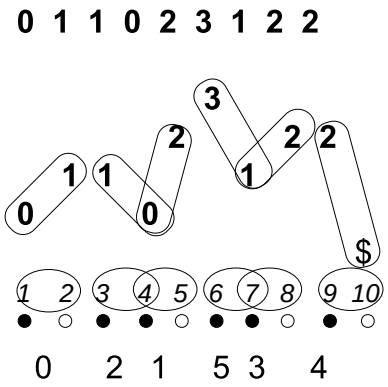


Figure 2. Illustration of τ-reduction of a string **011023122**

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

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

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

□

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 $\lhd$ induced by $\prec$: $(x[i_j], x[i_j+1]) \lhd (x[i_\ell], x[i_\ell+1])$ 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 radix sort with keys of size 2, and assigned letters from a chosen τ-alphabet that is a subset of $\{0, 1, ..., |\tau(x)|\}$ 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 $(0,1)(0,2)(1,0)$ $(1,2)(2,\$)(3,2)$. 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 $(3,1) \to 5$. Note that the assignments respect the order $\lhd$ 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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

integer alphabet. For our running example from Figure 2, $\tau(\boldsymbol{x}) = 021534$. The next lemma justifies calling the above transformation a reduction.

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

Proof. There are two extreme cases; the first is when all the τ-pairs do not overlap at all, then $|\tau(\boldsymbol{x})| = \frac{1}{2}|\boldsymbol{x}|$; and the second 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 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 $\boldsymbol{\tau}$-Reduction

The most important property of τ-reduction is a preservation of maximal Lyndon substrings of $\boldsymbol{x}$, which start at black positions. This means there is a closed formula that gives, for every maximal Lyndon substring of $\tau(\boldsymbol{x})$, a corresponding maximal Lyndon substring of $\boldsymbol{x}$. 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 Lyndon array, the one where the end positions of maximal Lyndon substrings are stored rather than their lengths. More formally:

Theorem 1. *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 4. *Let* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *and let* $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. *Let* $1 \leq i, j \leq n$. *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 $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$.

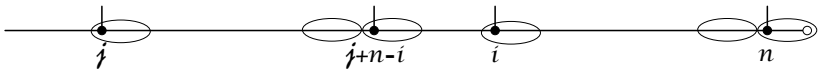
- Case: $\boldsymbol{x}[i..n]$ is a proper prefix of $\boldsymbol{x}[j..n]$.
  Then $j < i$ and so $\boldsymbol{x}[j..j+n-i] = \boldsymbol{x}[i..n]$ and thus $\boldsymbol{x}[i..n]$ is a border of $\boldsymbol{x}[j..n]$.
  - Case: $j+n-i$ is black.
    Since n may be black or white, we need to discuss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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

- Case: n is white.

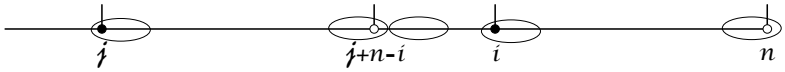
Then the last τ-pair overlapping $j..j+n-i$ must be $(j+n-i-1, j+n-i)$ followed by a τ-pair $(j+n-i, j+n-i+1)$, and the last τ-pair overlapping $i..n$ must be the last τ-pair $(n-1, n)$. Thus, $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n-1)] = \tau(\boldsymbol{x})[t(j)..t(j+n-i-1)]$, 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: 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.



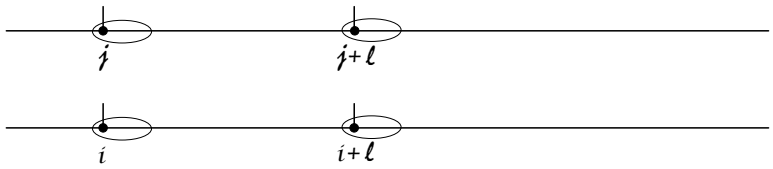
Then n must also be white as the τ-pair overlapping $j..j+n-i$ must be $(j+n-i-1, j+n-i)$ followed by $(j+n-i+1, j+n-i+2)$, and the last τ-pair overlapping $i..n$ must be the very last τ-pair $(n-1, n)$. Then $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n-1)] = \tau(\boldsymbol{x})[t(j)..t(j+n-i-1)]$ which is a proper prefix of $\tau(\boldsymbol{x})[t(j)..m]$.

- Case: $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$.

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: 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.



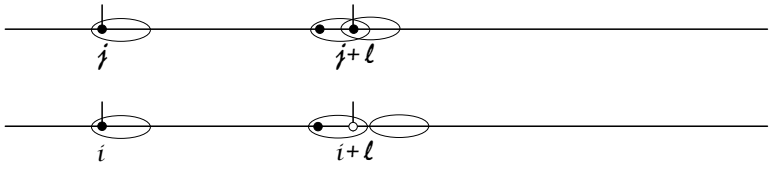
Consider $i+\ell-1$ and $j+\ell-1$. Either they are both black, or both are white.

(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 $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.

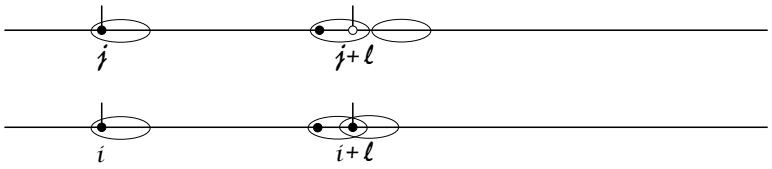
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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



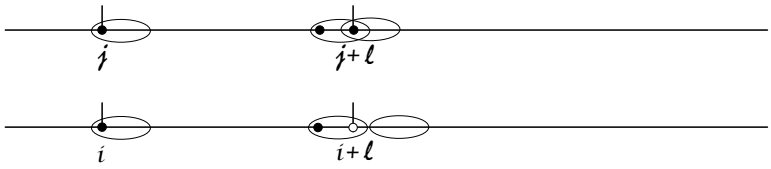
Then both $i+\ell-1$ and $j+\ell-1$ are black, so proceed as in (α).

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



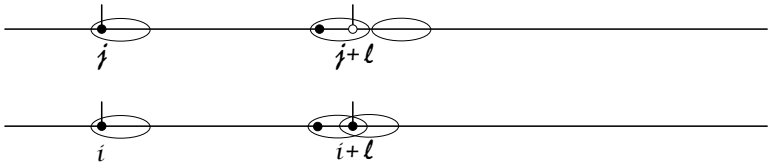
Then both $i+\ell-1$ and $j+\ell-1$ are black, so proceed as in (α).

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

Lemma 5. *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 substring of $\boldsymbol{x}$, and let i be a black position.*

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

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 4. Therefore, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon by Lemma 1.

Now, let us 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.

□

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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

Lemma 6. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \leq i < j \leq n$. 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 where j is black, and the other where j is white.

- Case: j is black.

  Since j is black, $(j, j+1)$ is a τ-pair and $t(j)$ is defined, and by Lemma 5, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon, and hence by Lemma 1, $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[k..m]$ for any $t(i) < k \leq t(j)$. Thus, we must show the maximality, i.e. $\tau(\boldsymbol{x})[t(j)+1..m] \prec \tau(\boldsymbol{x})[t(i)..m]$.

  - Case: $\boldsymbol{x}[j+1] \preceq \boldsymbol{x}[j+2]$.
    Then $\boldsymbol{x}[j] \succ \boldsymbol{x}[j+1] \preceq \boldsymbol{x}[j+2]$ and so $j+1$ is black. It follows that $t(j)+1 = t(j+1)$. By Lemma 4, $\tau(\boldsymbol{x})[t(j+1)..m] \prec \tau(\boldsymbol{x})[t(i)..m]$ 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: $\boldsymbol{x}[j+1] \succ \boldsymbol{x}[j]$.
    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 $\tau(\boldsymbol{x})[t(j)+1..m] \prec \tau(\boldsymbol{x})[t(i)..m]$.

- Case: j is white.
  By Lemma 5, $\tau(\boldsymbol{x})[t(i)..t(j-1)]$ 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 5 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 of Theorem 1.

Proof of Theorem 1.

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: j is black.

  Then by Lemma 6, $\tau(\boldsymbol{x})[t(i)..t(j)]$ 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: j is white.

  Then $j-1$ is black and the $\tau(\boldsymbol{x})[t(j-1)] = p(j-1, j)$. By Lemma 6, $\tau(\boldsymbol{x})[t(i)..t(j-1)]$ 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. $\boldsymbol{x}[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]+1\big)] = \boldsymbol{x}[j] \succ \boldsymbol{x}[i]$.

□

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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

Theorem 1 indicates how to compute the partial $\mathcal{L}'_{\boldsymbol{x}}$ from $\mathcal{L}'_{\tau(\boldsymbol{x})}$. The procedure is given in Figure 3.

```
for i ← 1 to n
  if i = 1 or (x[i−1] ≻ x[i] and x[i] ⪯ x[i+1]) then
    if x[b(L'_τ(x)[t(i)])+1] ⪯ x[i] then
      L'_x[i] ← b(L'_τ(x)[t(i)])
    else
      L'_x[i] ← b(L'_τ(x)[t(i)])+1
  else
    L'_x[i] ← 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 $\mathcal{L}'_{\boldsymbol{x}}[i] = nil$), 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 a length of 1 and so ends in i. Otherwise, let us call such a point *hard*, $\boldsymbol{x}[i..\mathcal{L}'_{\boldsymbol{x}}[i{+}1]]$ is Lyndon and we have to test if the maximal Lyndon substring that immediatly follows is lexicographically bigger. If it is, the two substrings can be concatenated to a bigger Lyndon substring starting at i, 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.

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 Subsection 3.5 for more details and Section 7 for the measurements and graphs.

```
L'_x[n] ← n
for i ← n−1 downto 2
  if L'[i] = nil then
    if x[i] ≻ x[i+1] then
      L'[i] ← i
    else
      if L'[i−1] = i−1 then
        stop ← n
      else
        stop ← L'[i−1]
      L'[i] ← L'[i+1]
      while L'[i] < stop do
        if x[i..L'[i]] ≺ x[L'[i]+1..L'[L'[i]+1]] then
          L'[i] ← L'[L'[i]+1]
        else
          break
```

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

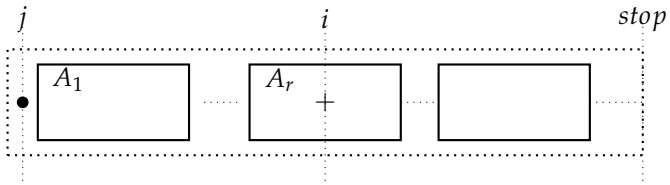
[2] two maximal Lyndon substring are either disjoint or one completely includes the other

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

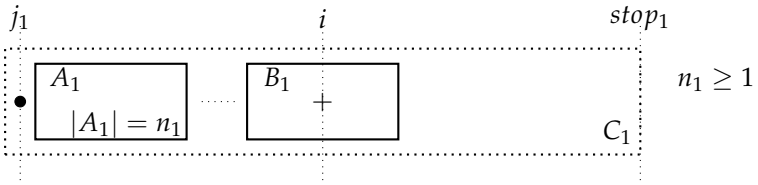
Consider our running example from Figure 2. Since $\tau(x) = 021534$, we have $\mathcal{L}'_{\tau(x)}[1..6] = 6,2,6,4,6,6$ giving $\mathcal{L}'_x[1..9] = 9,\bullet,3,9,\bullet,6,9,\bullet,9$. Computing $\mathcal{L}'_x[8]$ is easy as $x[8] = x[9]$ and so $\mathcal{L}'_x[8] = 8$. $\mathcal{L}'_x[5]$ is more complicate: we can extend the 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

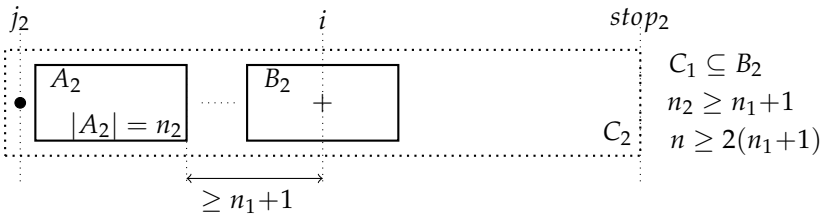
To determine the complexity of the algorithm, we attach to each position i a counter $red[i]$ initialized to 0. When computing a missing value $\mathcal{L}'_x[j]$ with a configuration shown below where $stop = \begin{cases} \mathcal{L}'_x[j-1] & if\ \mathcal{L}'_x[j-1] > j-1 \\ n & otherwise \end{cases}$,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



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 $j_1A1...$ and B_1 during the computation of the missing value $\mathcal{L}'_x[j_1]$ where $stop_1 = \begin{cases} \mathcal{L}'_x[j_1-1] & if\ \mathcal{L}'_x[j_1-1] > j_1-1 \\ n & otherwise \end{cases}$

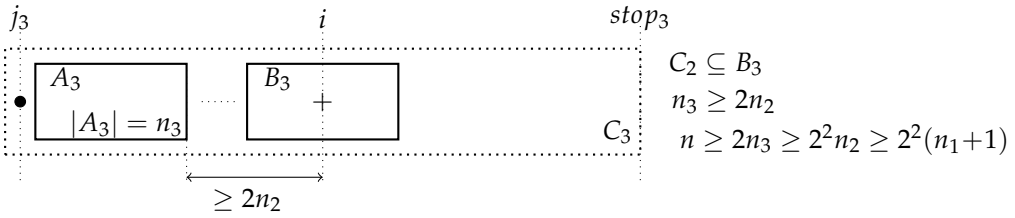


The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 2nd time in the comparison of $j_2A2...$ and B_2 during the computation of the missing value $\mathcal{L}'_x[j_2]$ where $stop_2 = \begin{cases} \mathcal{L}'_x[j_2-1] & if\ \mathcal{L}'_x[j_2-1] > j_2-1 \\ n & otherwise \end{cases}$



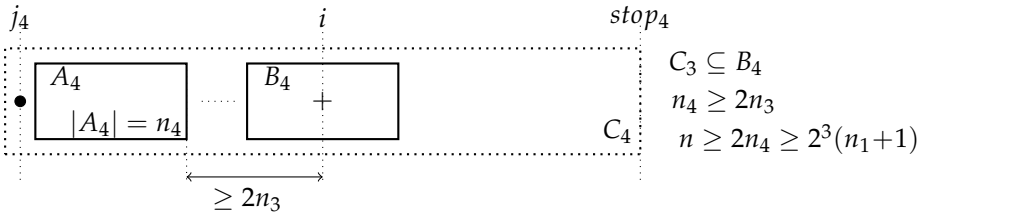
The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 3rd time in the comparison of $j_3A3...$ and B_3 during the computation of the missing value $\mathcal{L}'_x[j_3]$ where $stop_3 = \begin{cases} \mathcal{L}'_x[j_3-1] & if\ \mathcal{L}'_x[j_3-1] > j_3-1 \\ n & otherwise \end{cases}$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1



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 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 that cannot be determined in constant time. We overestimate the number of strings of length n over an alphabet of size C, $2 \leq C \leq n$, that will force non-linear computation, by assuming that every possible $\log(n)$ subset of indices with any possible letters assignment forces worst performance. Thus, there are $C^n - \binom{n}{\log(n)} C^{\log(n)}$ strings that are processed in linear time, say with a constant K_1, and there are $\binom{n}{\log(n)} C^{\log(n)}$ 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

$$\frac{\left(C^n - \binom{n}{\log(n)} C^{\log(n)}\right) Kn + \binom{n}{\log(n)} C^{\log(n)} Kn \log(n)}{C^n} =$$

$$Kn + Kn \frac{\binom{n}{\log(n)} C^{\log(n)}}{C^n} \left(\log(n) - 1\right) \leq$$

$$Kn + Kn \frac{\binom{n}{\log(n)} C^{\log(n)}}{C^n} \log(n) \leq$$

$$Kn + Kn \frac{n^{\log(n)} C^{\log(n)}}{\log(n)! C^n} \log(n) \leq$$

$$Kn + Kn \frac{n^{2\log(n)}}{2^n} \leq$$

$$Kn + Kn = 2Kn$$

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-time 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 data set `extreme_trla` of binary strings is created using the recursive formula $\boldsymbol{u}_{k+1} = 00\boldsymbol{u}_k 0\boldsymbol{u}_k$, using the first 100 shortest binary Lyndon words as the start $\boldsymbol{u}_0$. The moment

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

the size $\boldsymbol{u}_k$ exceeds the required length of the string, the recursion stops and the string is trimmed to the required length. For `extreme_trla1` data set, we used the same approach with the formula $\boldsymbol{u}_{k+1} = \mathtt{000}\boldsymbol{u}_k\mathtt{00}\boldsymbol{u}_k$ and for `extreme_trla2` data set, we used the formula $\boldsymbol{u}_{k+1} = \mathtt{0000}\boldsymbol{u}_k\mathtt{00}\boldsymbol{u}_k$.

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`, [16]) requires $3n$ integers of space for a string of length n. So the first call to *TRLA* requires $3n$, the next recursive call requires at most $3\frac{2}{3}n$, the next recursive call requires at most $3(\frac{2}{3})^2n$, ... Thus, $3n + 3\frac{2}{3}n + 3(\frac{2}{3})^2n + 3(\frac{2}{3})^3n + \ldots = 3n(1 + \frac{2}{3} + (\frac{2}{3})^2 + (\frac{2}{3})^3 + (\frac{2}{3})^4 + \ldots) = 3n\frac{1}{1-\frac{2}{3}} = 9n$. However, it should be possible to bring it down to $6n$ integers.

4. The Algorithm *BSLA*

The 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. 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 $\boldsymbol{x}$ 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. Notation and Basic Notions of BSLA

For the sake of simplicity, we fix a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ for the whole Subsection 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 1. *An ordered list of groups* $\langle G_k, G_{k-1}, \ldots, G_2, G_1\rangle$ *is a* ***group configuration*** *if*

- (C_1) $G_k \cup G_{k-1} \cup \ldots \cup G_2 \cup G_1 = 1..n$;
- (C_2) $G_j \cap G_\ell = \emptyset$ *for any* $1 \le \ell < j \le k$;
- (C_3) $con(G_k) \prec con(G_{k-1}) \prec \ldots \prec con(G_2) \prec con(G_1)$;
- (C_4) *For any* $j \in 1..k$, $con(G_j)$ *is a Lyndon substring of* $\boldsymbol{x}$.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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 $\boldsymbol{prev}$ 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 $\sim$ 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 7. *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 $[i]_\sim = \{j \in G \mid j_1 \leq j \leq j_2\}$.*

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 $prev(j) = prev(j_1) = prev(j_2)$.

□

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 $\boldsymbol{valence}$ by $\boldsymbol{val(j)} = |[i]_\approx|$. *In simple terms, $val(i)$ is the number of elements from $[i]_\sim$ that are $\approx i$.* Thus, $1 \leq val(i) \leq |G|$.

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 $\boldsymbol{x}$. 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 2. *A group G is $\boldsymbol{complete}$ 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 $\boldsymbol{t\text{-}complete}$, $1 \leq t \leq k$, if

- (C_5) *the groups $G_t, ..., G_1$ are complete;*
- (C_6) *the mapping prev is $\boldsymbol{proper}$ 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 $\boldsymbol{proper}$:*
  (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 $con(gr(j)) \preceq con(G_t)$;

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

(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 Subsection 4.2; all the definitions, lemmas and theorems apply and refer to this $\boldsymbol{x}$.

Lemma 9. *Let* $\mathcal{A}_{\boldsymbol{x}} = \{a_1, ..., a_k\}$ *and* $a_1 \prec a_2 \prec ... \prec a_k$. *For* $1 \leq \ell \leq k$, *define* $G_\ell = \{i \in 1..n : \boldsymbol{x}[i] = a_{k+1-\ell}\}$ *with context* $a_{k+-\ell}$. *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 $C(i) = \boldsymbol{x}[i] = \boldsymbol{x}[j] = C(j)$.

□

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 might only partition the groups that precede it, i.e. the groups $G_k, ..., G_{t+1}$, while 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 $con(gr(prev(j_{\ell,k})))con(gr(j_{\ell,k}))^{val(j_{\ell,k})}$. (*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 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 2 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.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

Theorem 2. *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 $1 \leq \ell \leq v$, $i = prev(i_\ell)$, and $C(i_\ell) = con(G_t)$, $v = val(i_\ell)$, 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 $C'(i)$ and $C'(j)$ for some $1 \leq i < j \leq n$.

- 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 $C'(i) \cap C'(j) \neq \emptyset$, then $C(i) \cap C(j) \neq \emptyset$, 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 $C'(i) \cap C'(j) \neq \emptyset$, 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 $C(i) \cap C(j_1) \neq \emptyset$, 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 $C(i) \cap C(j_2) \neq \emptyset$, ..., ultimately giving $C(j)C(j_1)...C(j_w) \subseteq C(i)$. So (C_8) for $Conf'$ holds.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

- 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 $1 \leq \ell \leq v$. 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 $C'(i) \cap C'(j) \neq \emptyset$. Consider $\mathcal{D} = \{i_\ell \mid 1 \leq \ell \leq v \ \&\ C(j) \cap C(i_\ell) \neq \emptyset\}$.
    Assume that $\mathcal{D} \neq \emptyset$:
      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.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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 $C'(i_1) = ... = C'(i_v) = con'(H_{t+1})$ and $j = prev'(i) = prev'(i_1) = ... = prev'(i_v)$ 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 $(C_7(a))$, $con'(H_{t+1}) \prec con'(gr'(i_1)) = con'(H_{t+1})$, 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}.

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 $k \geq i + |con'(H_{t+1})|$, $\boldsymbol{x}[i..k]$ is a maximal Lyndon substring of $\boldsymbol{x}$.

Either $k = n$ and so $con'(gr'(k)) = \boldsymbol{x}[k]$ 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 $i + |con'(H_{t+1})| \leq j_1 \leq k$ and $C'(j_1)$ is a suffix of $\boldsymbol{x}[i..k]$.Take the smallest j_1 such. If $j_1 = i + |con'(H_{t+1})|$, then $C'(i) \prec C'(j_1)$ as $\boldsymbol{x}[i..k] = C'(i)C'(j_1)$ is Lyndon. By $(C_7(b))$, $C'(j_1) \preceq con'(H_{t+1})$, 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 $\boldsymbol{x}[j_1 - 1]$. 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 $gr'(j_1 - 1)$, 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 $con'(gr'(j_1 - 1)) = \boldsymbol{x}[j_1 - 1]$. 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 $\boldsymbol{x}[i..j_1 - 1]$. Take the smallest such j_2. If $C'(j_2) \prec C'(j_1)$, then by $(C_7(b))$, $C'(j_1) \preceq con'(H_{t+1})$, a contradiction. Hence $C'(j_2) \succeq C'(j_1)$. If $j_2 = i + i + |con'(H_{t+1})|$, then $\boldsymbol{x}[i..k] = C'(i)C'(j_2)C'(j_1)$ and so by $(C_7(b))$, $C'(j_2) \preceq con'(H_{t+1})$, a contradiction. Hence $i + |con'(H_{t+1})| < j_2$.

The same argument done for j_2 can be now done for j_3. We end up with $i + |con'(H_{t+1})| \leq j_3 < j_2 < j_1 \leq k$ 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 $i + |con'(H_{t+1})| < j_3$. These argument can be repeated only finitely many times, and we obtain $i + |con'(H_{t+1})| = j_\ell < j_{\ell-1} < ... < j_2 < j_1 \leq k$ 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.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

The four claims show that all the conditions (C_1) ... (C_8) are satisfied for $Conf'$, and that proves Theorem 2.

□

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 3.
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 9.

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 2 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 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.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

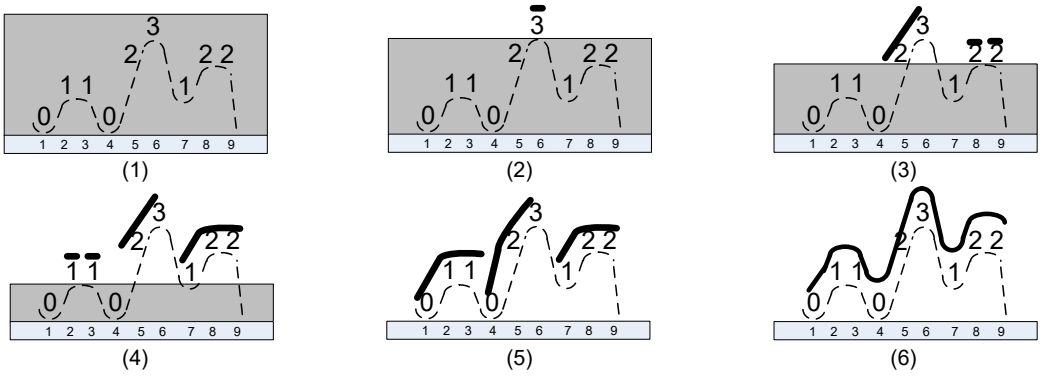


Figure 5. The water draining method for 011023122

In Figure 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.
(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 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 maximal Lyndon substrings of the string 011023122.

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

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

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

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

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

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

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

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

Figure 6. Group Refinement for 011023122

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

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 are indicated by the bold font.

4.4. The Complexity of BSLA

The computation of the initial configuration can be done in linear time as the input string is tight. The tightness of the input string is used to guarantee the linearity of computing the initial value of ***prev***. Since all groups are non-empty, there can never be more groups than n. Theorem 2 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 all worst case complexity of *BSLA* is linear in the length of the input string.

5. Measurements

Initially, computations were performed on the Department of Computing and Software's ***moore*** server; Memory: 32GB (DDR4 @ 2400 MHz), CPU: 8 x Intel Xeon E5-2687W v4 @ 3.00GHz, 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 was produced and computed independently on the University of Toronto Mississauga's ***octolab*** cluster; Memory: 8 x 32GB (DDR4 @ 3200 MHz), CPU: 8 x AMD Ryzen Threadripper 1920X (12-Core) @ 4.00GHz, OS: Ubuntu 16.04.6 LTS (gcc version 5.4.0). The results of both were extremely similar, those reported herein are those generated using the ***moore*** server. All the programs were compiled without any additional level of optimization[3]. 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 1, for strings of length 5×10^6), resulting in a higher precision. 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. We used 4 categories of randomly generated data sets:

(1) `bin`
random tight binary strings over the alphabet $\{0, 1\}$
(2) `dna`
random tight 4-ary (kind of random DNA) over the alphabet $\{0, 1, 2, 3\}$
(3) `eng`
random tight 26-ary strings (kind of random English) over the alphabet $\{0, 1, ..., 25\}$
(4) `int`
random tight strings over integer alphabet $\{0, 1, ..., n\}$.

Each data set contains 100 randomly generated strings of the same length. For each category, there were data sets for length 10, 50, 10^2, 5×10^2, ..., 10^5, 5×10^5, 10^6, and 5×10^6. The minimum, average, and maximum time for each data set was 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 here the averages.

Tables 1, 2, 3, 4 and the graphs in Figures 7, 8, 9, 10 from Section 7 clearly indicate that the performance of the three algorithms is linear and virtually indistinguishable. We expected *IDLA* and *TRLA* to exhibit linear behaviour on random strings as such strings tend to have. However, we did not expect the results to be so close.

[3] i.e. neither `-O1`, nor `-O2`, nor `-O3` flag were specified for the compilation

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

The data set `extreme_idla` contains individual strings 0123...n of the required lengths designed to force *IDLA* into its worst quadratic performance. Table 5 and the graph in Figure 11 from Section 7 show this clearly.

In Subsection 3.5, we described how the three data sets `extreme_trla`, `extreme_trla1`, and `extreme_trla2` are generated and why. The results of experimenting with these data sets do not suggest that the worst-case complexity for *TRLA* is $\mathcal{O}(n\ \log(n))$. Yet again, the performances of the three algorithms are linear and virtually indistinguishable, see Tables 6, 7 and the graphs in Figures 12, 13 from Section 7.

6. Conclusion and Future Work

We presented two novel algorithms for computing maximal Lyndon substrings. The first one, *TRLA*, 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 maximal Lyndon substrings, a fact used significantly in the design of the algorithm. Interestingly, it seem to slightly outperform *BSLA*, at least on the data sets 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 have not indicated 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 average times for the data sets discussed in the previous section. For better understanding of the data, we present them in both, tabular and graphic form.

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	3.876×10^{-1}	1.600	9.269×10^{-1}
50	4.213	1.028×10	5.762
100	1.111×10	2.143×10	1.233×10
500	8.270×10	1.141×10^2	6.414×10
1000	1.872×10^2	2.324×10^2	1.438×10^2
5000	1.233×10^3	1.216×10^3	8.155×10^2
10000	2.758×10^3	2.537×10^3	1.735×10^3
50000	1.669×10^4	1.222×10^4	9.702×10^3
100000	3.523×10^4	2.447×10^4	2.039×10^4
500000	2.031×10^5	1.249×10^5	1.409×10^5
1000000	4.345×10^5	2.687×10^5	3.764×10^5
5000000	2.479×10^6	1.547×10^6	2.775×10^6

Table 1. Average times for data set `bin`
(10^6 *clock ticks per second*)

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

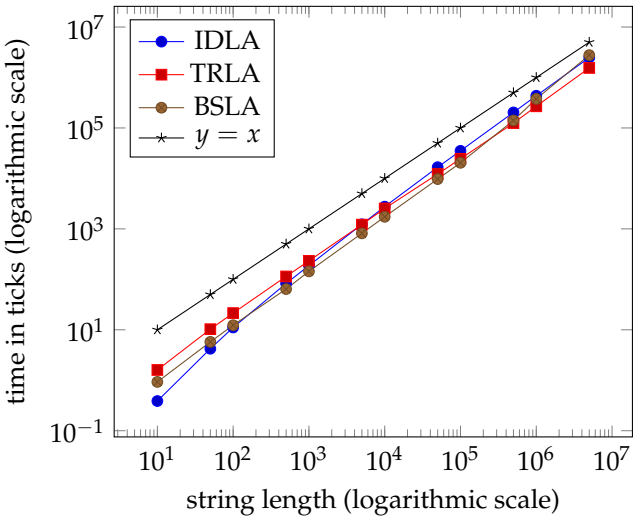


Figure 7. Average times for data set `bin`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	3.576×10^{-1}	1.611	9.803×10^{-1}
50	3.484	1.008×10	6.196
100	8.711	2.084×10	1.268×10
500	6.313×10	1.117×10^2	6.833×10
1000	1.413×10^2	2.332×10^2	1.488×10^2
5000	8.861×10^2	1.191×10^3	8.605×10^2
10000	1.931×10^3	2.396×10^3	1.872×10^3
50000	1.133×10^4	1.266×10^4	1.070×10^4
100000	2.418×10^4	2.575×10^4	2.218×10^4
500000	1.385×10^5	1.309×10^5	1.677×10^5
1000000	2.904×10^5	2.670×10^5	4.840×10^5
5000000	1.645×10^6	1.913×10^6	3.907×10^6

Table 2. Average times for data set `dna`
(10^6 *clock ticks per second*)

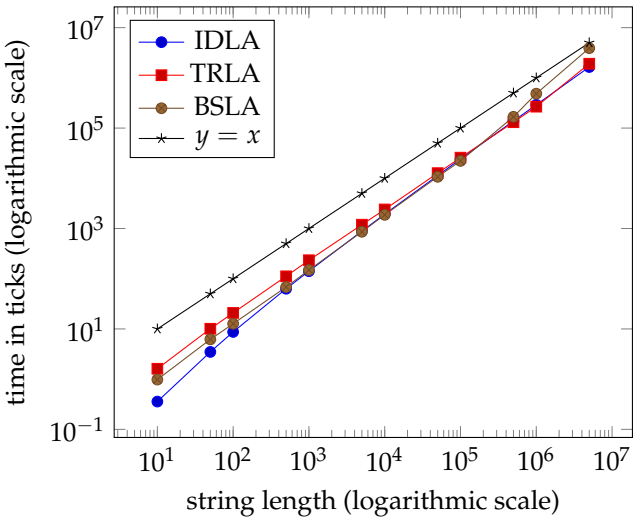


Figure 8. Average times for data set `dna`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	3.556×10^{-1}	1.602	9.021×10^{-1}
50	3.128	9.909	5.729
100	7.293	2.022×10	1.266×10
500	4.976×10	1.095×10^{2}	6.917×10
1000	1.104×10^{2}	2.313×10^{2}	1.478×10^{2}
5000	6.745×10^{2}	1.167×10^{3}	8.647×10^{2}
10000	1.459×10^{3}	2.371×10^{3}	1.968×10^{3}
50000	8.433×10^{3}	1.240×10^{4}	1.274×10^{4}
100000	1.786×10^{4}	2.649×10^{4}	2.703×10^{4}
500000	1.028×10^{5}	1.392×10^{5}	2.267×10^{5}
1000000	2.157×10^{5}	3.210×10^{5}	6.819×10^{5}
5000000	1.208×10^{6}	2.538×10^{6}	5.873×10^{6}

Table 3. Average times for data set `eng`
(10^6 *clock ticks per second*)

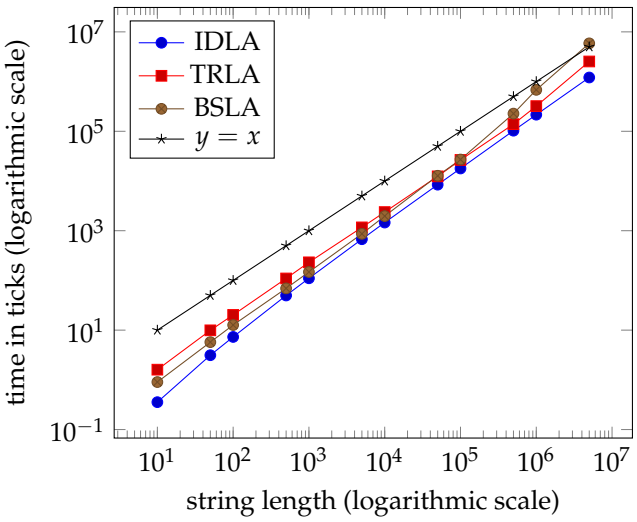


Figure 9. Average times for data set `eng`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	3.601×10^{-1}	1.591	9.091×10^{-1}
50	3.128	9.843	5.269
100	7.341	2.014×10	1.074×10
500	4.893×10	1.103×10^{2}	5.863×10
1000	1.055×10^{2}	2.315×10^{2}	1.206×10^{2}
5000	6.450×10^{2}	1.188×10^{3}	7.191×10^{2}
10000	1.368×10^{3}	2.463×10^{3}	1.609×10^{3}
50000	8.106×10^{3}	1.446×10^{4}	9.722×10^{3}
100000	1.727×10^{4}	2.899×10^{4}	2.097×10^{4}
500000	9.774×10^{4}	1.725×10^{5}	2.136×10^{5}
1000000	2.078×10^{5}	4.550×10^{5}	6.558×10^{5}
5000000	1.159×10^{6}	4.876×10^{6}	5.316×10^{6}

Table 4. Average times for data set `int`
(10^6 *clock ticks per second*)

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

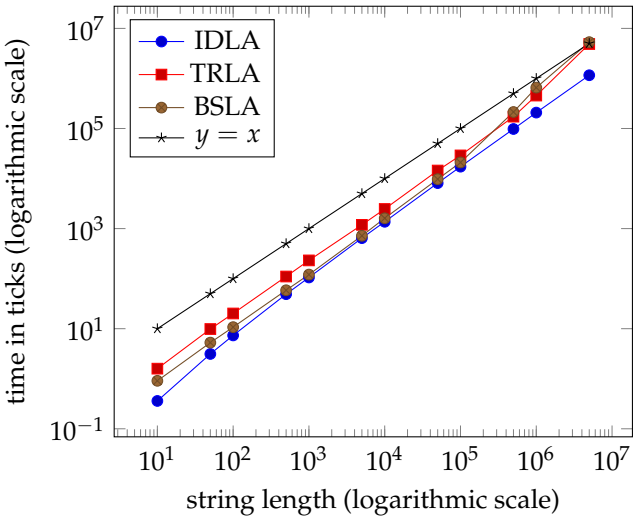


Figure 10. Average times for data set `int`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	7.900×10^{-1}	1.410	5.900×10^{-1}
50	1.830×10	8.300	3.000
100	7.190×10	1.580×10	6.000
500	1.778×10^{3}	7.350×10	2.900×10
1000	7.105×10^{3}	1.440×10^{2}	5.900×10
5000	1.776×10^{5}	7.200×10^{2}	2.950×10^{2}
10000	7.111×10^{5}	1.550×10^{3}	5.900×10^{2}
50000	1.784×10^{7}	7.900×10^{3}	3.000×10^{3}
100000	7.130×10^{7}	1.580×10^{4}	6.000×10^{3}
500000	1.783×10^{9}	8.200×10^{4}	3.200×10^{4}
1000000	7.137×10^{9}	1.670×10^{5}	7.200×10^{4}
5000000	1.813×10^{11}	8.900×10^{5}	4.350×10^{5}

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

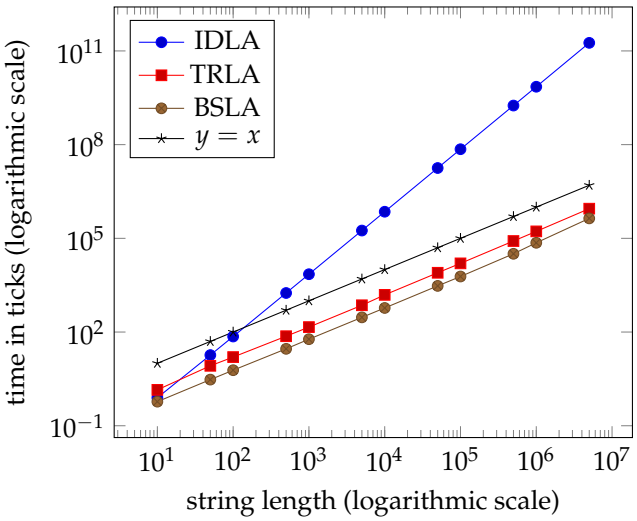


Figure 11. Average times for data set `extreme_idla`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	4.588×10^{-1}	1.628	1.010
50	4.987	1.039×10	6.527
100	1.275×10	2.179×10	1.315×10
500	9.033×10	1.101×10^2	6.418×10
1000	2.060×10^2	2.222×10^2	1.295×10^2
5000	1.319×10^3	1.171×10^3	7.662×10^2
10000	2.896×10^3	2.394×10^3	1.600×10^3
50000	2.209×10^4	1.263×10^4	8.598×10^3
100000	3.965×10^4	2.567×10^4	1.752×10^4
500000	2.233×10^5	1.349×10^5	1.027×10^5
1000000	4.734×10^5	2.759×10^5	2.950×10^5
5000000	2.632×10^6	1.458×10^6	2.174×10^6

Table 6. Average times for data set `extreme_trla`
(10^6 *clock ticks per second*)

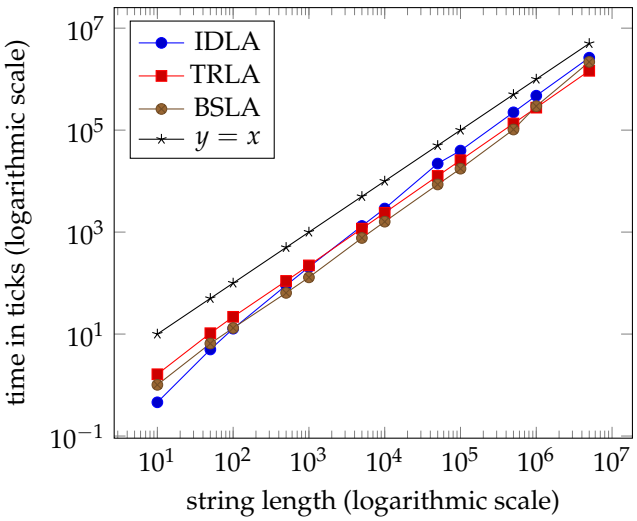


Figure 12. Average times for data set `extreme_trla`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	5.040×10^{-1}	1.600	1.013
50	5.910	1.042×10	6.689
100	1.460×10	2.145×10	1.338×10
500	1.146×10^2	1.126×10^2	6.514×10
1000	2.662×10^2	2.284×10^2	1.298×10^2
5000	1.694×10^3	1.205×10^3	7.408×10^2
10000	3.734×10^3	2.477×10^3	1.648×10^3
50000	2.276×10^4	1.310×10^4	9.294×10^3
100000	4.901×10^4	2.796×10^4	1.917×10^4
500000	2.928×10^5	1.465×10^5	1.174×10^5
1000000	6.199×10^5	3.000×10^5	3.443×10^5
5000000	3.432×10^6	1.642×10^6	2.671×10^6

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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 September 2020 doi:10.20944/preprints202009.0557.v1

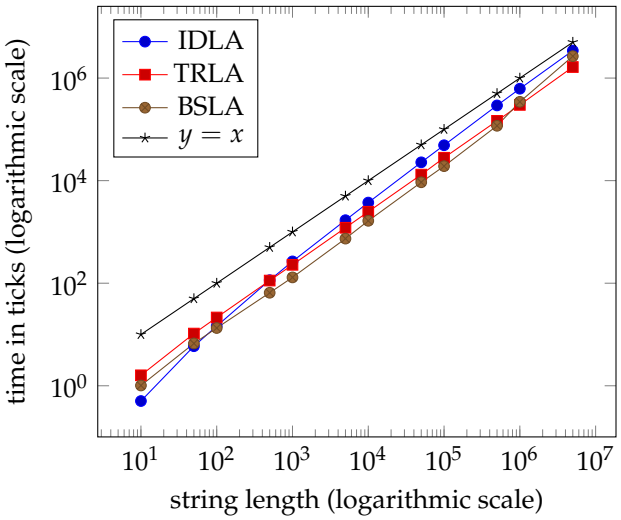


Figure 13. Average times for data set `extreme_trla1`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

string length	time in ticks IDLA	time in ticks TRLA	time in ticks BSLA
10	5.041×10^{-1}	1.683	1.020
50	6.160	1.020×10	6.684
100	1.526×10	2.090×10	1.334×10
500	1.367×10^2	1.074×10^2	6.543×10
1000	3.202×10^2	2.135×10^2	1.302×10^2
5000	2.024×10^3	1.145×10^3	7.421×10^2
10000	4.500×10^3	2.257×10^3	1.654×10^3
50000	2.728×10^4	1.172×10^4	9.623×10^3
100000	5.941×10^4	2.362×10^4	2.027×10^4
500000	3.639×10^5	1.262×10^5	1.266×10^5
1000000	7.719×10^5	2.571×10^5	3.695×10^5
5000000	4.263×10^6	1.323×10^6	2.994×10^6

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

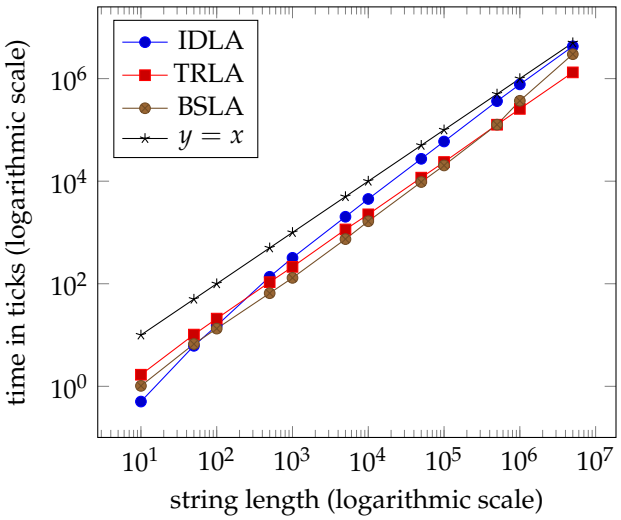


Figure 14. Average times for data set `extreme_trla2`
(10^6 *clock ticks per second*)
for reference the $y = x$ *curve is shown*

Author Contributions: Conceptualization, Frantisek Franek and Michael Liut; Data curation, Frantisek Franek and Michael Liut; Formal analysis, Frantisek Franek and Michael Liut; Funding acquisition, Frantisek Franek; Investigation, Frantisek Franek and Michael Liut; Methodology, Frantisek Franek and Michael Liut; Project administration, Frantisek Franek; Resources, Frantisek Franek and Michael Liut; Software, Frantisek Franek and Michael Liut; Supervision, Frantisek Franek; Validation, Frantisek Franek and Michael Liut; Visualization, Frantisek Franek and Michael Liut; Writing – original draft, Frantisek Franek and Michael Liut; Writing – review & editing, Frantisek Franek and Michael Liut.

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. Bannai, H.; I, T.; Inenaga, S.; Nakashima, Y.; Takeda, M.; Tsuruta, K. The "Runs" Theorem. Available at https://arxiv.org/abs/1406.0263.
2. Bannai, H.; I, T.; Inenaga, S.; Nakashima, Y.; Takeda, M.; Tsuruta, K. The "Runs" Theorem. *SIAM J. COMPUT.* **2017**, *46*, 1501–1514.
3. Franek, F.; Paracha, A.; Smyth, W. The linear equivalence of the suffix array and the partially sorted Lyndon array. Proc. Prague Stringology Conference, 2017, pp. 77–84.
4. Baier, U. Linear-time suffix sorting — a new approach for suffix array construction. M.Sc. Thesis, University of Ulm, Ulm, Germany.
5. Baier, U. Linear-time suffix sorting — a new approach for suffix array construction. 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016); .Grossi, R.; Lewenstein, M., Eds.; Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2016; Vol. 54, *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 1–12.
6. Chen, G.; Puglisi, S.; Smyth, W. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science* **2013**, *1*, 605–623.
7. Crochemore, M.; Ilie, L.; Smyth, W. A simple algorithm for computing the Lempel-Ziv factorization. Proc. 18th Data Compression Conference, 2008, pp. 482–488.
8. Kosolobov, D. Lempel-Ziv factorization may be harder than computing all runs. Proceedings of 32 International Symposium on Theoretical Aspects of Computer Science – STACS 2015, 2015, pp. 582–593.
9. Digelmann, C. Personal communication.
10. Franek, F.; Sohidull Islam, A.; Sohel Rahman, M.; Smyth, W. Algorithms to compute the Lyndon array. Proceedings of Prague Stringology Conference 2016, 2016, pp. 172–184.
11. Duval, J.P. Factorizing words over an ordered alphabet. *J. Algorithms* **1983**, *4*, 363–381.
12. Hohlweg, C.; Reutenauer, C. Lyndon words, permutations and trees. *Theoretical Computer Science* **2003**, *307*, 173–178. doi:http://dx.doi.org/10.1016/S0304-3975(03)00099-9.
13. Nong, G.; Zhang, S.; Chan, W.H. Linear suffix array construction by almost pure induced-sorting. 2009 Data Compression Conference, 2009, pp. 193–202.
14. Louza, F.; Smyth, W.; Manzini, G.; Telles, G. Lyndon array construction during Burrows–Wheeler inversion. *Journal of Discrete Algorithms* **2018**, *50*, 2–9.
15. Franek, F.; Liut, M.; Smyth, W. On Baier's sort of maximal Lyndon substrings. Proceedings of Prague Stringology Conference 2018, 2018, pp. 63–78.
16. C++ code for IDLA, TRLA and BSLA algorithms. Available at: https://github.com/MichaelLiut/Computing-LyndonArray.
17. Farach, M. Optimal suffix tree construction with large alphabets. Proc. 38th IEEE Symp. Foundations of Computer Science. IEEE, 1997, pp. 137–143.

18. Nong, G. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.* **2013**, *31*, 1–15.
19. Cooley, J.; Tukey, J. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation* **1965**, *19*, 297–301.
20. Franek, F.; Liut, M. Computing maximal Lyndon substrings of a string, AdvOL Report 2019/2, McMaster University. Available at: http://optlab.mcmaster.ca//component/option,com_docman/task,cat_view/gid,77/Itemid,92.
21. Franek, F.; Liut, M. Algorithms to compute the Lyndon array revisited. Proc. of Prague Stringology Conference 2019, 2019, pp. 16–28.
22. Liut, M. Computing Lydon Arrays. Ph.D. Thesis, McMaster University, Hamilton, Ontario, Canada.
23. Paracha, A. Lyndon factors and periodicities in strings. Ph.D. Thesis, McMaster University, Hamilton, Ontario, Canada.
24. Kärkkäinen, J.; Sanders, P. Simple linear work suffix array construction. Proceedings of the 30th international conference on Automata, languages and programming; Springer–Verlag: Berlin, Heidelberg, 2003; ICALP'03, pp. 943–955.