Algorithms to Compute the Lyndon Array Revisited

Frantisek Franek and Michael Liut

Department of Computing and Software
McMaster University, Hamilton, Canada
{franek/liutm}@mcmaster.ca

Abstract. The motivation for having an efficient algorithm for identifying all maximal Lyndon substrings of a string comes from the work of Bannai et al. on the Runs Conjecture. In 2015, they resolved the conjecture by considering Lyndon roots of runs and they also presented a unique linear algorithm for computing all runs. The uniqueness of the algorithm lies in the fact that it relies on the knowledge of all maximal Lyndon substrings, while all other linear algorithms for runs rely on Lempel-Ziv factorization of the input string. A Lyndon array is a data structure that encodes the information of all maximal Lyndon substrings of a string. In a 2016 Prague Stringology Conference paper, Franek et al. discussed various known algorithms for computing the Lyndon array. In 2015, in his Masters' thesis, Baier designed a linear algorithm for suffix sorting. Inspired by Phase II of Baier's algorithm, in a 2017 Prague Stringology Conference paper, Franek et al. discussed the linear co-equivalency of sorting suffixes and sorting maximal Lyndon substrings. As noticed by C. Diegelmann, the first phase of Baier's algorithm identifies and sorts all maximal Lyndon substrings of the input string. Based on Phase I of Baier's algorithm, in a 2018 Prague Stringology Conference paper, Franek et. al presented an elementary (in the sense of not relying on a pre-processed global data structure) linear algorithm for identifying and sorting all maximal Lyndon substrings. This paper revisits the subject of algorithms for the Lyndon array and closes off the series of our Prague Stringology Conference contributions on the topic – it provides a simple overview of all currently known algorithms including the new development since 2016. In particular, it presents a detailed analysis of a new algorithm *TRLA*, and comparative measurements of three of the algorithms.

Keywords: string, suffix, suffix array, Lyndon array, Lyndon string, maximal Lyndon substring

1 Introduction

There are at least two reasons for having an efficient algorithm for identifying all maximal Lyndon substrings in a string: firstly, Bannai et al. introduced in 2015 (arXiv, [3]), and published in 2017, [4], a linear algorithm to compute all runs in a string that relies on knowing all maximal Lyndon substrings of the string, and secondly, in 2017, Franek et al. in [13] showed a linear co-equivalence of sorting suffixes and sorting maximal Lyndon substrings, based on a novel suffix sorting algorithm introduced by Baier in 2015 (Master's thesis, [1]), and published in 2016, [2].

The most significant feature of the runs algorithm presented in [4] is that it relies on knowing all maximal Lyndon substrings of the input string for some order of the alphabet and for the inverse of that order, while all other linear algorithms for runs rely on Lempel-Ziv factorization of the input string. Thus, computing runs became yet another application of Lyndon words. It also raised the issue which approach

Frantisek Franek, Michael Liut: Algorithms to Compute the Lyndon Array Revisited, pp. 16–28.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8

may be more efficient: to compute the Lempel-Ziv factorization or to compute all maximal Lyndon substrings. Interestingly, Kosolobov argues that computing Lempel-Ziv factorization may be harder than computing all runs, however his archive paper [17] was not followed up with. There are several efficient linear algorithms for Lempel-Ziv factorization; for example see [5,7] and the references therein.

Baier introduced in [1], and published in [2], a new algorithm for suffix sorting. Though Lyndon strings are never mentioned there, it was noticed by Cristoph Diegelmann in a personal communication, [8], that Phase I of 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 [14]: 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 position* i.

Our research group has presented in the Prague Stringology Conference a series of three papers on the topic of maximal Lyndon substrings:

(1) In 2016, [14] presented an overview of then-current algorithms for computing the Lyndon array.
(2) In 2017, [13] presented the linear co-equivalency of sorting suffixes and sorting maximal Lyndon substrings.
(3) In 2018, [12] presented an elementary[1] linear algorithm to identify and sort all maximal Lyndon substrings, inspired by Phase I of Baier's algorithm.

This paper completes the series and briefly recapitulates the algorithms presented in [14] and then presents the development since 2016 not described in [14], in particular a novel algorithm *TRLA* for computing the Lyndon array based on τ-reduction, and empirical comparisons of three of the algorithms: *IDLA*, *BSLA*, and *TRLA*.

The structure of the paper is as follows. In Section 2, the basic notations and notions are presented. Section 3 contains a brief recapitulation of *IDLA*, the *Iterated Duval algorithm for Lyndon array*. Section 4 contains a brief recapitulation of *RDLA*, the *Recursive Duval algorithm for Lyndon array*. Section 5 contains a brief recapitulation of *SSLA*, the *algorithmic scheme of suffix sorting followed by Next Smaller Value*. Section 6 introduces *BWLA*, the 2018 *algorithmic scheme for computing the Lyndon array via inversion of Burrows-Wheeler transform*. Section 7 contains a brief recapitulation of *RGLA*, the *ranges based algorithm for Lyndon array*. Section 8 contains a brief recapitulation of *BSLA*, the *Baier's sort Phase I inspired algorithm*. Section 9 contains a detailed description and analysis of *TRLA*, the recursive algorithm based on τ-reduction. In Section 10, the empirical measurements of the performance of *IDLA*, *BSLA*, and *TRLA* are presented on various datasets with random strings of various lengths and over various alphabets. The results are presented in a graphical form. In Section 11, the conclusion of the research is presented, and the necessary future work described.

2 Basic notation and terminology

For two integers $i \leq j$, the ***range*** $i..j = \{k \text{ integer} : i \leq k \leq j\}$. An ***alphabet*** is a finite or infinite set of ***symbols*** (equivalently called letters). We assume

[1] not needing a pre-processed global structure such as suffix array

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 \$. We use the array notation indexing from 1 for strings, thus $\boldsymbol{x}[1..n]$ indicates a string of length n, the first symbol is the symbol with index 1, i.e. $\boldsymbol{x}[1]$, the second symbol is the symbol with index 2, i.e. $\boldsymbol{x}[2]$, etc. Thus, $\boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]\cdots\boldsymbol{x}[n]$. For a \$-terminated string $\boldsymbol{x}$ of length n, $\boldsymbol{x}[n+1] = \$$. The ***alphabet of string*** $\boldsymbol{x}$, denoted as $\mathcal{A}_{\boldsymbol{x}}$, is the set of all distinct alphabet symbols occurring in $\boldsymbol{x}$. By a ***constant alphabet*** we mean a fixed finite alphabet. A string $\boldsymbol{x}$ is over an ***integer alphabet*** if $\mathcal{A}_{\boldsymbol{x}} \subseteq \{0, 1, \ldots, |\boldsymbol{x}|\}$. Thus, the class of ***strings over integer alphabets*** $= \{\boldsymbol{x} \mid \boldsymbol{x}$ *is a string over* $\{0, 1, \ldots, |\boldsymbol{x}|\}\}$. A string $\boldsymbol{x}$ over an integer alphabet is ***tight*** if $\mathcal{A}_{\boldsymbol{x}} = \{0, 1, \ldots, k\}$ for some $k \leq |\boldsymbol{x}|$. Thus, for instance $\boldsymbol{x} = 010$ is tight as $\mathcal{A}_{\boldsymbol{x}} = \{0, 1\}$, while $\boldsymbol{y} = 020$ is not as $\mathcal{A}_{\boldsymbol{y}} = \{0, 2\}$ – i.e. 1 is missing from $\mathcal{A}_{\boldsymbol{y}}$.

We use a bold font to denote strings, thus $\boldsymbol{x}$ denotes a string, while x denotes some other mathematical entity such as an integer. The ***empty string*** is denoted by $\boldsymbol{\varepsilon}$ and has length 0. The ***length*** or ***size*** of string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is n. The length of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. For two strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..m]$, the ***concatenation*** $\boldsymbol{xy}$ is a string $\boldsymbol{u}$ where $\boldsymbol{u}[i] = \begin{cases} \boldsymbol{x}[i] \ for\ i \leq n, \\ \boldsymbol{y}[i-n] \ for\ n < i \leq n+m. \end{cases}$
If $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{u}$ is a ***prefix***, $\boldsymbol{v}$ a ***substring***, and $\boldsymbol{w}$ a ***suffix*** of $\boldsymbol{x}$. If $\boldsymbol{u}$ (respectively $\boldsymbol{v}$, $\boldsymbol{w}$) is empty, then it is called a ***trivial prefix*** (respectivly ***trivial substring***, ***trivial suffix***), if $|\boldsymbol{u}| < |\boldsymbol{x}|$ (respectively $|\boldsymbol{v}| < |\boldsymbol{x}|$, $|\boldsymbol{w}| < |\boldsymbol{x}|$) then it is called a ***proper prefix*** (respectively ***proper substring***, ***proper suffix***). If $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a ***rotation*** or a ***conjugate*** of $\boldsymbol{x}$; if either $\boldsymbol{u} = \boldsymbol{\varepsilon}$ or $\boldsymbol{v} = \boldsymbol{\varepsilon}$, then the rotation is called ***trivial***. A non-empty string $\boldsymbol{x}$ is ***primitive*** if there is no string $\boldsymbol{y}$ and no integer $k \geq 2$ so that $\boldsymbol{x} = \boldsymbol{y}^k = \underbrace{\boldsymbol{yy}\cdots\boldsymbol{y}}_{k\ times}$.

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

Let $\prec$ be a total order of an alphabet $\mathcal{A}$. The order is extended to all finite strings over the alphabet $\mathcal{A}$: for $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..n]$, $\boldsymbol{x} \prec \boldsymbol{y}$ if either $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$, or there is a $j \leq \min\{n, m\}$ so that $\boldsymbol{x}[1] = \boldsymbol{y}[1]$, $\ldots$, $\boldsymbol{x}[j-1] = \boldsymbol{y}[j-1]$ and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$. This total order induced by the order of the alphabet is called the ***lexicographic*** order of all non-empty strings over $\mathcal{A}$. We write $\boldsymbol{x} \preceq \boldsymbol{y}$ if either $\boldsymbol{x} \prec \boldsymbol{y}$ or $\boldsymbol{x} = \boldsymbol{y}$. A string $\boldsymbol{x}$ over $\mathcal{A}$ is ***Lyndon*** for a given order $\prec$ of $\mathcal{A}$ if $\boldsymbol{x}$ is strictly lexicographically smaller than any non-trivial rotation of $\boldsymbol{x}$. A substring $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}[1..n]$, $1 \leq i \leq j \leq n$ is a ***maximal Lyndon substring of*** $\boldsymbol{x}$ if it is Lyndon and either $j = n$ or for any $k > j$, $\boldsymbol{x}[i..k]$ is not Lyndon. The ***Lyndon array*** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an integer array $\mathcal{L}[1..n]$ so that $\mathcal{L}[i] = j$ where $j \leq n-i$ is a maximal integer such that $\boldsymbol{x}[i..i+j-1]$ is Lyndon. Alternatively, we can define it as an integer array $\mathcal{L}'[1..n]$ so that $\mathcal{L}'[i] = j$ when $\boldsymbol{x}[i..j]$ is a maximal Lyndon substring. 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 Iterated Duval algorithm - *IDLA*

This algorithm was presented in [14]. The algorithm is based on Duval's work on Lyndon factorization, [9]. The procedure `maxLyn(`$\boldsymbol{x}$`)` returns the length of the maximal Lyndon prefix of the string $\boldsymbol{x}$. In Duval's factorization algorithm, `maxLyn` is then applied to the position immediately after the maximal Lyndon prefix. Here, we apply `maxLyn` iteratively to every suffix of $\boldsymbol{x}$. Why and how `maxLyn` works, and its complexity can be found in [14], including the pseudo-code. The `C++` implementation can be found in the file `lynarr.hpp`, [6]. Note that *IDLA* is referred to as *Iterated MaxLyn* in [14]. The worst-case complexity of $IDLA(\boldsymbol{x})$ is $\mathcal{O}(|\boldsymbol{x}|^2)$. The algorithm works in-place, so the storage requirements are just the storage for the string and the storage for the Lyndon array. The alphabet of the input string need not be sorted, but must be ordered. The alphabet is *sorted* if the alphabet is in the form of an ordered list, so that for each letter you can access in constant time the immediately preceding letter and the immediately succeeding letter. The alphabet is *ordered* if there is a partial order on the alphabet so that comparison of any two letters can be computed in constant time.

4 Recursive Duval algorithm - *RDLA*

This algorithm was presented in [14]. The algorithm is also based on Duval's algorithm for Lyndon factorization which is applied recursively: if $\boldsymbol{x}[1..i_1]\boldsymbol{x}[i_1{+}1..i_2]\cdots\boldsymbol{x}[i_k{+}1..n]$ is a Lyndon factorization of $\boldsymbol{x}$, the algorithm is recursively applied to $\boldsymbol{x}[2..i_1]$, to $\boldsymbol{x}[i_1{+}2..i_2]$, ..., to $\boldsymbol{x}[i_k{+}2..n]$, and so on. The correctness of the algorithm follows from the correctness of Duval's algorithm. The alphabet of the input string need not be sorted, but must be ordered. The algorithm works in the worst-case complexity of $\mathcal{O}(|\boldsymbol{x}|^2)$, and in the special case of the binary alphabet of $\boldsymbol{x}$, it is $\mathcal{O}(|\boldsymbol{x}|\ log(|\boldsymbol{x}|))$, see [14]. Storage requirements are the same as for *IDLA*, plus the additional storage for the stack controlling the recursion.

5 Algorithmic scheme based on suffix sorting - *SSLA*

This scheme was presented in [14]. It is based on Lemma 1 which follows from Hohlweg and Reutenauer's work [14,15]. The lemma characterizes maximal Lyndon substrings in terms of the relationships of the suffixes.

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 $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$.*

Therefore, the Lyndon array of $\boldsymbol{x}$ is the *NSV* (Next Smaller Value) array of the inverse suffix array. The scheme is as follows: sort the suffixes, from the resulting suffix array compute the inverse suffix array, and then apply *NSV* to the inverse suffix array. Computing the inverse suffix array and applying *NSV* are "naturally" linear and computing the suffix array can be implemented to be linear, see [14,20] and the references therein. The time and space characteristics of the whole scheme are dominated by the time and space characteristics of the first step – the computation of the suffix array. For linear suffix sorting, the input strings must be over constant or integer alphabets.

6 Algorithmic scheme based on Burrows-Wheeler transform – *BWLA*

This scheme was not presented in [14] as it was introduced in 2018, see [18]. The algorithm is linear and computes the Lyndon array from a given Burrows-Wheeler transform of the input string. 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. As for *SSLA*, the execution and space characteristics of the scheme are dominated by the computation of the suffix array.

7 An algorithm based on ranges – *RGLA*

This algorithm was discussed in [14] where it was referred to as *NSV**.
In case of a constant alphabet, ranges can be compared in constant time if the Parikh vector for each range is pre-computed, which can be done in linear time. An increasing range is a maximal substring $\boldsymbol{x}[i..j]$ so that $\boldsymbol{x}[k] \preceq \boldsymbol{x}[\ell]$ for every $i \leq k < \ell \leq j$, while a decreasing range is a maximal substring $\boldsymbol{x}[i..j]$ so that $\boldsymbol{x}[k] \succ \boldsymbol{x}[\ell]$ for every $i \leq k < \ell \leq j$. The algorithm emulates the classic stack implementation of *NSV*. The time and space complexity of the algorithm was not given in [14], but the time complexity is at worst $\mathcal{O}(n^2)$, though it was indicated in the paper it could possibly be $\mathcal{O}(n\ log(n))$, where n is the length of the input string. An informal analysis of the correctness of the algorithm was provided.

8 Baier's suffix sort Phase I inspired algorithm – *BSLA*

Introduced in 2018, this algorithm was not discussed in [14], however, it was presented in [12]. There it was referred to as *Baier's sort* and the code was available as `bls`, see [6]. For the interested reader, a simpler and more elegant description and analysis of the correctness of the algorithm can be found in [11]. Our `C++` implementation can be found at [6] as `BSLA`, though based on the ideas of Phase I of Baier's suffix sort, our implementation necessarily differs from Baier's.

The input strings for *BSLA* are tight strings over integer alphabets. Note that this requirement does not significantly 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 tight string also gives 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 complete, too. 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; it will be shown that the group for the largest alphabet symbol is complete, so it is a proper start for the refinement process.

Each group is assigned a specific substring of the input string referred to as the ***context*** of the group; it has the property that for every i in the group, the group's context occurs at the position i. 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 the additional property that the contexts of complete groups are maximal occurrences in $\boldsymbol{x}$. Hence, when the refinement is complete, the contexts of all the groups in the list represent all maximal Lyndon substrings of $\boldsymbol{x}$.

The process of refinement is rather technical and we refer the interested reader to the original presentation in [12], or a better presentation in [11]. The complexity of the algorithm is linear in the length of the input string. The space requirements are relatively high; our `C++` implementation, see [6], uses $12n$ integers of working memory. We refer to the algorithm as *elementary*, as no global data structure needs to be pre-processed, as is the case for *SSLA* and *BWLA*.

9 τ-reduction algorithm – *TRLA*

Introduced in 2017, this algorithm was not presented in [14]. The first idea of the algorithm was proposed in Paracha's 2017 Ph.D. thesis [21]. It follows Farach's approach used in his remarkable linear algorithm for suffix tree construction [10], and reproduced very successfully in all linear algorithms for suffix sorting, see for instance [19,20] and the references therein. The scheme for computing the Lyndon array works as follows:

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

The input strings are \$-terminated strings over integer alphabets. The reduction computed in (1) is important. All linear algorithms for suffix array computations use the proximity property of suffixes: comparing $\boldsymbol{x}[i..n]$ and $\boldsymbol{x}[j..n]$ can be done by comparing $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$, and if they are the same, comparing $\boldsymbol{x}[i+1..n]$ with $\boldsymbol{x}[j+1..n]$. For instance, in the first linear algorithm for suffix array by Kärkkäinen and Sanders, [16], obtaining the sorted suffixes for positions $i \equiv 0 \ (mod\ 3)$ and $i \equiv 1 \ (mod\ 3)$ via the recursive call is sufficient to determine the order of suffixes for $i \equiv 2 \ (mod\ 3)$ positions, and then to merge both lists together. However, there is no such proximity property for maximal Lyndon substrings, so the reduction itself must have a property that helps determine some of the values of the Lyndon array of $\boldsymbol{x}$ from the Lyndon array of $\boldsymbol{y}$ and compute the rest. We present such a reduction that we call τ-reduction, and it may be of some general interest as it preserves order of some suffixes and hence, by Lemma 1, some maximal Lyndon substrings.

The algorithm computes $\boldsymbol{y}$ as a τ-reduction of $\boldsymbol{x}$ in step (1) in linear time and in step (3) it expands the Lyndon array of the reduced string computed by step (2) to an incomplete Lyndon array of the original string also in linear time. However, it computes the missing values of the incomplete Lyndon array in $\Theta(n\ log(n))$ time resulting in the overall worst-case complexity of $\Theta(n\ log(n))$. If the missing values of the incomplete Lyndon array of $\boldsymbol{x}$ were computed in linear time, the overall algorithm would be linear as well. Since for τ-reduction, the size of $\tau(\boldsymbol{x})$ is at most $\frac{2}{3}|\boldsymbol{x}|$, we eventually obtain, through the recursion of step (2) applied to $\tau(\boldsymbol{x})$, a partially filled Lyndon array of the input string; the array is about $\frac{1}{2}$ to $\frac{2}{3}$ full and for every position

i with an unknown value, the values at positions $i-1$ and $i+1$ are known and $\boldsymbol{x}[i-1] \preceq \boldsymbol{x}[i]$. In particular, the value at position 1 and position n are both known. So, a lot of information is provided by the recursive step. For instance, given the string 00011001, via the recursive call we would identify the maximal Lyndon substrings that are underlined in $\underline{000\underline{1}1\,00\underline{1}}$ and would need to compute the missing maximal Lyndon substrings that are underlined in $\underline{00\underline{1}10\underline{01}}$. It is possible that in the future we may come up with a linear procedure to compute the missing values making the whole algorithm linear. We describe the τ-reduction in several steps: first the τ-pairing, then choosing the τ-alphabet, and finally the computation of the τ-reduction of $\boldsymbol{x}$.

9.1 τ-pairing

Consider a \$-terminated string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ whose alphabet $\mathcal{A}_{\boldsymbol{x}}$ is ordered by $\prec$ with $\boldsymbol{x}[n+1] = \$$ and $\$ \prec a$ for any $a \in \mathcal{A}_{\boldsymbol{x}}$. A ***τ-pair*** consists of a pair of 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 illustration see Fig. 1, for formal proof see Lemma 2.

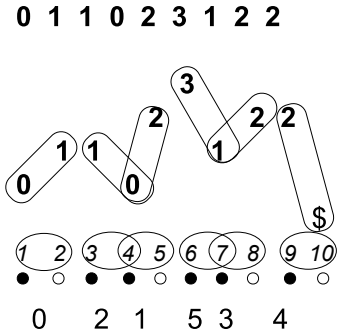

Figure 1. Illustration of τ-reduction of a string **011023122**
The rounded rectangles indicate symbol τ-pairs, the ovals indicate the τ-pairs below are the colour labels of positions, at the bottom is the τ-reduction

Lemma 2. *Let* $(i_1, i_1{+}1)\cdots(i_k, i_k{+}1)$ *be the* τ*-pairs of a strings* $\boldsymbol{x} = \boldsymbol{x}[1..n]$*. Then for any* $j, \ell \in 1..k$

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

Proof. For the proof, please see the online report [11], Observation 3 and Lemma 3. □

9.2 τ-reduction

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

In the example, Fig. 1, 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 $\boldsymbol{x}$ and denoted $\boldsymbol{\tau(x)}$, and it is a \$-terminated string over an integer alphabet. For our running example from Fig. 1, $\tau(\boldsymbol{x}) = 021534$. 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. One extreme case is when all the τ-pairs do not overlap at all, then $|\tau(\boldsymbol{x})| = \frac{1}{2}|\boldsymbol{x}|$. The other extreme case is when all the τ-pairs overlap, then $|\tau(\boldsymbol{x})| = \frac{2}{3}|\boldsymbol{x}|$. Any other case must be in between.

□

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

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

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

In our running example from Fig. 1, $t(1) = 1, t(3) = 2, t(4) = 3, t(6) = 4, t(7) = 5$, 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 $p(9,10) = 4$.

9.3 Properties preserved by τ-reduction

The most important property of τ-reduction is the preservation of maximal Lyndon substrings of $\boldsymbol{x}$ that start at black positions. By that we mean the fact 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 4. *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 5. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i, j \le n$. 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. For the proof, please see the online report [11], Lemma 6. □

Lemma 6 shows that τ-reduction preserves the Lyndon property of certain Lyndon substrings.

Lemma 6. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i < j \le n$. Let $\boldsymbol{x}[i..j]$ be a Lyndon susbtsring of $\boldsymbol{x}$, and let i be a black position.*

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

Proof. For the proof, please see the online report [11], Lemma 7. □

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

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

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. For the proof, please see the online report [11], Lemma 8. □

Now we are ready to tackle the proof Theorem 4 as we promised; please, see the online report [11] for the proof.

```
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 2. Computing partial Lyndon array of the input string

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

Theorem 4 indicates how to compute the partial $\mathcal{L}'_{\boldsymbol{x}}$ from $\mathcal{L}'_{\tau(\boldsymbol{x})}$. The procedure is given in Fig. 2.

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

This is the **while** loop in the procedure given in Fig. 3 that gives it the $\mathcal{O}(n\,log(n))$ complexity as we will show later. At the first, it may seem that it might actually give it $\mathcal{O}(n^2)$ complexity, but the "doubling of size" trims it effectively down to $\mathcal{O}(n\,log(n))$; see Section 9.5.

```
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 3. Computing missing values of the Lyndon array of the input string

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

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

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

9.5 The complexity of *TRLA*

The complexity of *TRLA* is $\Theta(n\ log(n))$ for a string of length n. The detailed analysis can be found in the online report [11], Section 3.5. The analysis involves two steps: the first step is showing that the complexity is $\mathcal{O}(n\ log(n))$, and the second steps gives a *scheme (F)* of generating binary strings that force $\Theta(n\ log(n))$ execution.

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`, [6]) 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 $3(\frac{2}{3})^2 n$, ..., thus, $3n + 3\frac{2}{3}n + 3(\frac{2}{3})^2 n + 3(\frac{2}{3})^3 n + \ldots = 3n(1 + \frac{2}{3} + (\frac{2}{3})^2 + (\frac{2}{3})^3 + (\frac{2}{3})^4 + \ldots) = 3n\frac{1}{1-\frac{2}{3}} = 9n$.

10 Measurements

All the measurements were performed on the ***moore*** server of McMaster University's Department of Computing and Software; Memory: 32GB (DDR4 @ 2400 MHz), CPU: 8 of the Intel Xeon E5-2687W v4 @ 3.00GHz, OS: Linux version 2.6.18-419.el5 (gcc version 4.1.2) (Red Hat 4.1.2-55), further, all the programs were compiled without any additional level of optimization[3]. The CPU time was measured for each of the programs in seconds with a precision of 3 decimal places (i.e. milliseconds). Since the execution time was negligible for short strings, the processing of the same string was repeated several times (the repeat factor varied from 10^6, for strings of length 10, to 1, for strings of length 10^6), resulting in a higher precision (of up to 7 decimal places). Thus, for graphing, the logarithmic scale was used for both, the x-axis representing the length of the strings, and the y-axis representing the time.

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

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

We also tested all three algorithms on datasates containing a single string $01234\ldots n$ referred to as an *extreme_idla* string, which, of course makes *IDLA* exhibit its quadratic complexity, and indeed the results show it; see Fig. 8. The *extreme_trla* strings were generated according to the scheme (F) used in the analysis

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

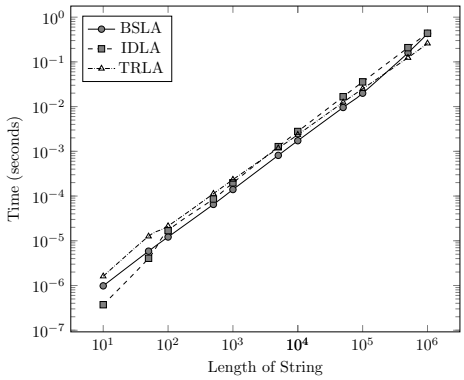

Figure 4. Binary strings

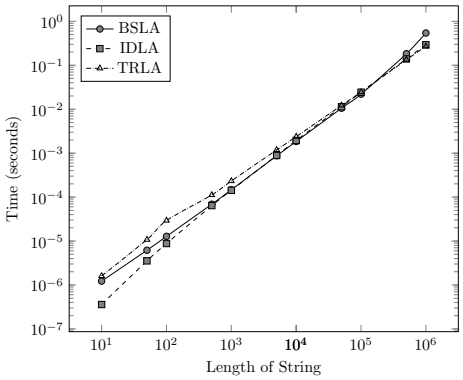

Figure 5. 4-ary strings

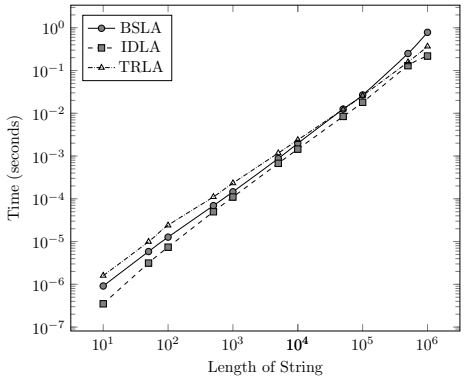

Figure 6. 26-ary strings

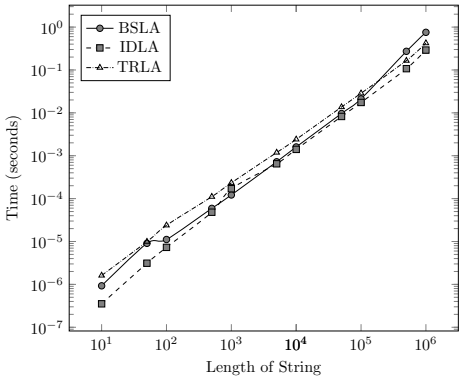

Figure 7. Strings over integer alphabets

of the complexity of *TRLA* in section 9.5. These strings force worst-case execution for *TRLA*. However, even $log(10^6)$ is too small to really highlight the difference, so the results were again very close, see Fig. 9.

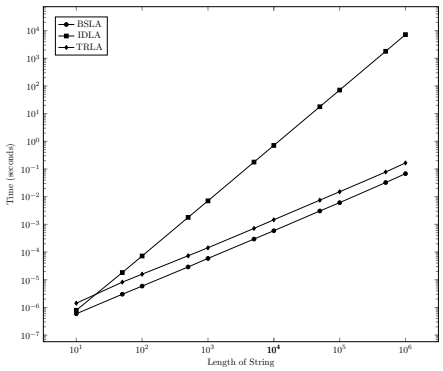

Figure 8. extreme_idla strings

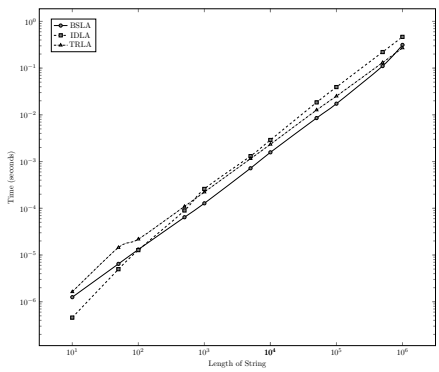

Figure 9. extreme_trla strings

11 Conclusion and Future Work

We presented an overview of current algorithms for computing maximal Lyndon substrings, including new development since the publication of [14]:

- the algorithmic scheme based on the computation of the inverse Burrows-Wheeler transform, *BWLA*, [18];

- the linear algorithm inspired by Phase I of Baier's algorithm, *BSLA*, [11,12]; and
- the novel algorithm based on τ-reduction, *TRLA*.

Then performance of three of the presented algorithms, *IDLA*, *BSLA*, and *TRLA* was compared on various datasets of random strings. The algorithm *TRLA* is mostly of theoretical interest since it has the worst-case complexity $\Theta(n \log(n))$ for strings of length n. Interestingly, on random strings it slightly outperformed *BSLA*, which is linear. Additional effort will go into improving *TRLA*'s complexity in the computation of the missing values. It is imperative that the three algorithms be compared to some efficient *SSLA* or *BWLA* implementation.

References

1. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction.* M.Sc. Thesis, University of Ulm, Ulm, Germany, 2015.
2. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction*, in 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), R. .Grossi and M. Lewenstein, eds., vol. 54 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 1–12.
3. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "Runs" Theorem.* Available at `https://arxiv.org/abs/1406.0263`, 2015.
4. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "Runs" Theorem.* SIAM J. COMPUT., 46 2017, pp. 1501–1514.
5. G. Chen, S. Puglisi, and W. Smyth: *Lempel-Ziv factorization using less time & space.* Mathematics in Computer Science, 1(4) 2013, pp. 605–623.
6. *C++ code for IDLA, TRLA and BSLA algorithms*: Available at `http://www.cas.mcmaster.ca/~franek/research.html`.
7. M. Crochemore, L. Ilie, and W. Smyth: *A simple algorithm for computing the Lempel-Ziv factorization*, in Proc. 18th Data Compression Conference, 2008, pp. 482–488.
8. C. Digelmann: *Personal communication*, 2016.
9. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
10. M. Farach: *Optimal suffix tree construction with large alphabets*, in Proc. 38th IEEE Symp. Foundations of Computer Science, IEEE, October 1997, pp. 137–143.
11. F. Franek and M. Liut: *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`.
12. F. Franek, M. Liut, and W. Smyth: *On Baier's sort of maximal Lyndon substrings*, in Proceedings of Prague Stringology Conference 2018, 2018, pp. 63–78.
13. F. Franek, A. Paracha, and W. Smyth: *The linear equivalence of the suffix array and the partially sorted Lyndon array*, in Proc. Prague Stringology Conference, 2017, pp. 77–84.
14. F. Franek, A. Sohidull Islam, M. Sohel Rahman, and W. Smyth: *Algorithms to compute the Lyndon array*, in Proceedings of Prague Stringology Conference 2016, 2016, pp. 172–184.
15. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 307(1) 2003, pp. 173–178.
16. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proceedings of the 30th international conference on Automata, languages and programming, ICALP'03, Berlin, Heidelberg, 2003, Springer–Verlag, pp. 943–955.
17. D. Kosolobov: *Lempel-Ziv factorization may be harder than computing all runs.* Avaiable at `https://arxiv.org/abs/1409.5641`, 2014.
18. F. Louza, W. Smyth, G. Manzini, and G. Telles: *Lyndon array construction during Burrows–Wheeler inversion.* Journal of Discrete Algorithms, 50 2018, pp. 2–9.
19. G. Nong: *Practical linear-time O(1)-workspace suffix sorting for constant alphabets.* ACM Trans. Inf. Syst., 31(3) 2013, pp. 1–15.
20. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced-sorting*, in 2009 Data Compression Conference, 2009, pp. 193–202.
21. A. Paracha: *Lyndon factors and periodicities in strings.* Ph.D. Thesis, McMaster University, Hamilton, Ontario, Canada, 2017.