Computing Lyndon Arrays

Michael Adam Liut

Doctoral Thesis

Computing Lyndon Arrays

author
Michael Adam Liut

last edit
September 26, 2019

institution
Department of Computing and Software
Faculty of Engineering
McMaster University

Acknowledgements

This journey would not have been possible without the support of my family, supervising professors and mentors, and friends.

To my family: thank you for inspiring me to perform and achieve my best. To my parents and brother: your unwavering support and constant encouragement were my backbone and will never be forgotten.

To my Ph.D. supervisors Dr. Antoine Deza and Dr. Franya Franek: thank you for your wisdom and guidance. You have both been a pillar throughout my university experience and two mentors who I look up to and aspire to be. Without you both, this would have not been possible.

To my Ph.D. supervisory committee: thank you for keeping me on track and continuously in pursuit of academic excellence.

To my friends, labmates, and colleagues: thank you for being my mainstay and allowing me to continually perform at my peak.

I would also like to thank:

- the Advanced Optimization Laboratory (AdvOL);
- the Department of Computing and Software, the Faculty of Engineering, and McMaster University;
- the National Sciences and Engineering Research Council of Canada; and
- Dr. Paul Vrbik for the use of his LaTeX template.

Contents

Abstract

There are at least two reasons to have an efficient algorithm for identifying all maximal Lyndon substrings in a string: first, in 2015, Bannai et al. introduced a linear algorithm to compute all runs in a string that relies on knowing all maximal Lyndon substrings of the input string, and second, in 2017, Franek et al. showed a linear co-equivalence of sorting suffixes and sorting maximal Lyndon substrings of a string (inspired by a novel suffix sorting algorithm of Baier).

In 2016, Franek et al. presented a brief overview of algorithms for computing the Lyndon array that encodes the knowledge of maximal Lyndon substrings of the input string. It discussed four different algorithms. Two known algorithms for computing 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 the inverse suffix array, and applying the NSV (*Next Smaller Value*) algorithm. The overview also discusses a recursive version of Duval's algorithm with a quadratic complexity and an algorithm emulating the NSV approach with a possible $\mathcal{O}(n\ log(n))$ complexity. The authors at that time did not know of Baier's algorithm. In 2017, Paracha proposed in her Ph.D. thesis an algorithm for the Lyndon array. The proposed algorithm was interesting as it emulated Farach's recursive approach for computing suffix trees in linear time and introduced τ-reduction; which might be of independent interest.

This was the starting point of this Ph.D. thesis. The primary aim is: (a) developing, analyzing, proving correct, and implementing in `C++` a linear algorithm for computing the Lyndon array based on Baier's suffix sorting; (b) analyzing, proving correct, and implementing in `C++` the algorithm proposed by Paracha; and (c) empirically comparing the performance of these two algorithms with the iterative version of Duval's algorithm.

Problem Statement

Design, analyze, implement, and test algorithms to compute maximal Lyndon substrings of a string in linear or subquadratic time without the involvement of two-stage computation[a].

[a] i.e. when a data structure not directly related to maximal Lyndon substrings is computed in the first stage, then the maximal Lyndon substrings are computed from it in the second stage

Overview

The following is a high-level overview of what novel work has been completed in the research effort for this thesis and represents new contributions to the field of stringology:

1. Baier's Sort inspired Lyndon Array algorithm (BSLA)

   BSLA, chapter 5, discuses the creation, development, and implementation of a new algorithm that is elementary in the sense of not needing any previous computation of some global structure such as a suffix array. Further, the algorithm is formally proven correct, implemented in C++, rigorously tested, and its execution measured.

   A preliminary version of the research for this thesis was reported in [15] of which the author of this thesis was the principal co-author. This thesis contains a significantly improved and simplified analysis of the correctness. The implementation for this thesis is also significantly improved from the preliminary implementation for [15] which focused on just the proof of the concept.

2. $\tau-$Reduction Lyndon Array algorithm (TRLA)

   TRLA, chapter 6, is improved from its initial design by Paracha. A significantly refined and corrected analysis is presented in this thesis, including the complexity analysis. A deep analysis of the $\tau-$reduction is presented as it may be of independent research interest. The implementation of *TRLA* in C++ is completely new and never before reported. The algorithm was rigorously tested, and its execution measured.

3. Results and empirical analysis

   Both *BSLA* and *TRLA* have been implemented in `C++`, alongside one of the standard algorithms *IDLA* (see chapter 4). All three algorithms have been empirically tested against one another and analyzed in-depth. Results were provided on a variety of datasets. The results are reported both in the form of graphs for a quick visualization, also in terms of tables containing the raw data. The methodology of the experiments and a description of the experimental setup are presented.

   Moreover, not described in this thesis is the software testbed for the measurements as well as the software for production of the test datasets. All the experiments were also used as validation tests for the correctness of the implementations of *BSLA* and *TRLA* as the results as computed by these two programs were compared to the result computed independently by *IDLA*.

4. A survey of all algorithms for computing maximal Lyndon substrings

   The impetus for the initial research for this thesis was paper [17] and is described in Chapter 3. This lead to a paper [14] of which the author of this thesis was a principal co-author and represents an up-to-date survey of Lyndon array algorithms.

1 Introduction

There are two paths which one can take when computing all the runs of a string in linear time: the first is relying on Lempel-Ziv factorization [20] and the second is computing all maximal Lyndon substrings [4]. There are several efficient linear algorithms for Lempel-Ziv factorization for strings over constant and integer alphabets; for example [8, 10] and the references therein. In 2015, Bannai et al. introduced a linear algorithm to compute all the runs in a string [4]. This algorithm, published in 2017 [5], relies on knowing all maximal Lyndon substrings of the string with respect to an order of the alphabet and all maximal Lyndon substrings with respect to the inverse of that order. Thus, computing runs became another application of Lyndon words.

These two approaches raise the question, which approach may be more efficient: to compute the Lempel-Ziv factorization or to compute all maximal Lyndon substrings? Interestingly enough, Dmitry Kosolobov argues that computing Lempel-Ziv factorization may be harder than computing all the runs [21] for general alphabets, however, there was no substantial followup to this research and the paper was never published; it was only uploaded to `arXiv`.

The work of Bannai et al. awoke an interest in efficient computing of maximal Lyndon substrings of a string. In 2016, Franek et al., [17], presented an overview of then-known algorithms for computing maximal Lyndon substrings and introduced the notion of the Lyndon array. Unknown to the authors at that time was the work of Baier, [2, 3].

In 2017, Franek et al. demonstrated linear co-equivalence of sorting suffixes and sorting maximal Lyndon substrings [16], based on a novel suffix sorting algorithm introduced by Baier [2] in 2015, and published

in 2016 [3]. Though Lyndon strings were not discussed by Baier at all, it was noticed by Cristoph Diegelmann in a personal communication [11] that Phase I of Baier's suffix sort identifies and sorts all maximal Lyndon substrings. The fact that the "sorting of suffixes" is, in a sense, equivalent to the "sorting of maximal Lyndon substrings" increased the interest in efficient computing of maximal Lyndon substrings.

This was the starting point of the research for this thesis. The result of which is two algorithms for computing the Lyndon array: a linear algorithm *BSLA* based on Baier's method and a recursive $\mathcal{O}(n\ log(n))$ algorithm *TRLA* emulating Farach's method. The algorithms are formally described, analyzed, and the executions of their `C++` implementations are compared.

This thesis is structured as follows: in chapter 2 basic notion, terminology, and facts are presented; in chapter 3 the background of the problem is given; in chapter 4 the Iterated Duval algorithm for Lyndon Array (*IDLA*) is described and discussed; in chapter 5 Baier's Sort (Phase I) inspired Lyndon Array algorithm (*BSLA*) is described and analyzed in depth; in chapter 6 a detailed description and analysis of the recursive τ-Reduction algorithm for Lyndon Array (*TRLA*) is conducted; in chapter 7 the empirical measurements and results of the performance of *IDLA*, *BSLA*, and *TRLA* are presented on various datasets with random strings of various lengths and over various alphabets; and finally in chapter 8 the conclusion of the research is presented and the future work described.

2 Notation and Basic Facts

Some fundamental notions, definitions, facts, and string algorithms can be found in the following references [22, 23, 28]. For ease of access, this chapter includes those that are directly related to the work herein.

For two integers $i \leq j$, the ***range*** $i..j = \{k \; integer : i \leq k \leq j\}$. An ***alphabet*** is a finite or infinite set of ***symbols*** (equivalently called ***letters***). We assume that a sentinel symbol \$ is not in the alphabet and is always assumed to be smaller than all symbols in the alphabet. 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 utilize array notation, commencing at 1, for indexing strings. Therefore, $\boldsymbol{x}[1..n]$ indicates a string of length n, the first symbol is the symbol with index 1, $\boldsymbol{x}[1]$, the second symbol is the symbol with index 2, $\boldsymbol{x}[2]$, etc. The strings is a concatenation of the one-symbol strings, i.e. $\boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]...\boldsymbol{x}[n]$, and for a \$-terminated string $\boldsymbol{x}$ of length n, $\boldsymbol{x}[n+1] = \$$.

The ***alphabet of string*** $\boldsymbol{x}$, denoted as $\mathcal{A}_{\boldsymbol{x}}$, is the set of all distinct alphabet symbols occurring in $\boldsymbol{x}$. By a ***constant alphabet*** we mean a fixed finite alphabet. A string $\boldsymbol{x}$ is over an ***integer alphabet*** if $\mathcal{A}_{\boldsymbol{x}} \subseteq \{0, 1, ..., |\boldsymbol{x}|\}$. Thus, the class of ***strings over integer alphabets*** $= \{\boldsymbol{x} \mid \boldsymbol{x}$ *is a string over* $\{0, 1, ..., |\boldsymbol{x}|\}\}$. A string $\boldsymbol{x}$ over an integer alphabet is ***tight*** if $\mathcal{A}_{\boldsymbol{x}} = \{0, 1, ..., k\}$ for some $k \leq |\boldsymbol{x}|$. For instance $\boldsymbol{x} = 010$ is tight as $\mathcal{A}_{\boldsymbol{x}} = \{0, 1\}$ while $\boldsymbol{y} = 020$ is not tight because $\mathcal{A}_{\boldsymbol{y}} = \{0, 2\}$; 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 $\boldsymbol{u}$ (respective $\boldsymbol{v}$, $\boldsymbol{w}$) is empty, then it is called a ***trivial prefix*** (respective ***trivial substring***, ***trivial suffix***), if $|\boldsymbol{u}| < |\boldsymbol{x}|$ (respective $|\boldsymbol{v}| < |\boldsymbol{x}|$, $|\boldsymbol{w}| < |\boldsymbol{x}|$) then it is called a ***proper prefix*** (respective ***proper substring***, ***proper suffix***). If $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a ***rotation*** or a ***conjugate*** of $\boldsymbol{x}$; if either $\boldsymbol{u} = \boldsymbol{\varepsilon}$ or $\boldsymbol{v} = \boldsymbol{\varepsilon}$, then the rotation is called ***trivial***. A non-empty string $\boldsymbol{x}$ is ***primitive*** if there are 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***.

A ***period*** of a string $\boldsymbol{x}[1\,..\,n]$ is an integer p so that $\boldsymbol{x}[i] = \boldsymbol{x}[i+p]$ for any $1 \leq i,\ i+p \leq n$.

> For example, $\boldsymbol{x}[1\,..\,7] = abababa$ has a period 2, as $\boldsymbol{x}[1] = \boldsymbol{x}[3] = \boldsymbol{x}[5] = \boldsymbol{x}[7] = a$ and $\boldsymbol{x}[2] = \boldsymbol{x}[4] = \boldsymbol{x}[6] = b$. It also has a period 4, as $\boldsymbol{x}[1] = \boldsymbol{x}[5] = a$, $\boldsymbol{x}[2] = \boldsymbol{x}[6] = b$, and $\boldsymbol{x}[3] = \boldsymbol{x}[7] = a$.

A substring $\boldsymbol{x}[i\,..\,j]$ is a ***repetition*** if there is a period p of $\boldsymbol{x}[i\,..\,j]$ such that $j-i+1 = kp$ for some integer $k \geq 2$. The substring $\boldsymbol{x}[i\,..\,i+p-1]$ is properly called ***generator*** of the repetition, but is often referred to as the period as well. Therefore, period frequently means both the generator and the length of the generator, but it is usually quite clear from the context which meaning is correct.

> For example, if $\boldsymbol{x}[1\,..\,12] = aabcabcabcab$, then $\boldsymbol{x}[2\,..\,7] = abcabc$ is a repetition of period 3 with the generator abc, $\boldsymbol{x}[2\,..\,10] = abcabcabc$ is a repetition of period 3 with the generator abc, while $\boldsymbol{x}[7\,..\,12] = cabcab$ is a repetition of period 3 with the generator cab. Another example is $\boldsymbol{x}[1\,..\,2] = aa$, a repetition of period 1 and the generator a. It should be noted that there are several other repetitions in $\boldsymbol{x}$.

The substring $\boldsymbol{x}[i\,..\,j]$ is a ***run with period p*** and ***exponent e*** if it is a maximal repetition – i.e. neither $\boldsymbol{x}[i-1\,..\,j]$ nor $\boldsymbol{x}[i\,..\,j+1]$ are repetitions, its period p is the least of all periods of $\boldsymbol{x}[i\,..\,j]$ (and so the generator $\boldsymbol{x}[i\,..\,i+p-1]$

is primitive), and $e = (j-i+1)/p \geq 2$.

For example, $\boldsymbol{x}[1..10] = abcdbcdbca$ has a run $\boldsymbol{x}[2..9] = bcdbcdbc$ with a period 3 as $\boldsymbol{x}[1..9] = abcdbcdbc$ is not a repetition with a period 3, and $\boldsymbol{x}[2..10] = bcdbcdbca$ is not a repetition with a period 3.

A binary relation $\prec$ is a ***total order*** on an alphabet $\mathcal{A}$ if it is antireflexive: $a \not\prec a$ for any $a \in \mathcal{A}$, and antisymetric: if $a \prec b$, then $b \not\prec a$ for any $a, b \in \mathcal{A}$, and transitive: if $a \prec b$ and $b \prec c$, then $a \prec c$, for any $a, b, c \in \mathcal{A}$. This relationship can thus be considered an order of strings of length 1. The order is extended to all finite strings over the alphabet $\mathcal{A}$: for $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..n]$, $\boldsymbol{x} \prec \boldsymbol{y}$ if either $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$, or there is a $j \leq \min\{n, m\}$ so that $\boldsymbol{x}[1] = \boldsymbol{y}[1]$, ..., $\boldsymbol{x}[j-1] = \boldsymbol{y}[j-1]$ and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$. This total order induced by the order of the alphabet is called the ***lexicographic*** order of all non-empty strings over $\mathcal{A}$. Typically, the same symbol is used to denote the lexicographic order as the order of the alphabet that induces it. Thus, $\prec$ may denote the order of the alphabet $\mathcal{A}$ as well as the lexicographic order of strings over $\mathcal{A}$. It is usually clear from the context: $x \prec y$ is the alphabet order if x and y are alphabet symbols, or the lexicographic order if x and y are strings. We denote by $\boldsymbol{x} \preceq \boldsymbol{y}$ if either $\boldsymbol{x} \prec \boldsymbol{y}$ or $\boldsymbol{x} = \boldsymbol{y}$. For a \$-terminated string, we always assume that \$ is lexicographically smaller than all symbols in the alphabet.

From the definition of $\prec$, the notation $\boldsymbol{x} \prec \boldsymbol{y}$ can mean two things, either $\boldsymbol{x}$ is a proper non-trivial prefix of $\boldsymbol{y}$, or there is j, $1 \leq j \leq \min(|\boldsymbol{x}|, |\boldsymbol{y}|)$ so that $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$ while $\boldsymbol{x}[i] = \boldsymbol{y}[i]$ for any $1 \leq i < j$. Sometimes we need just to consider the second possibility and thus we occasionally utilize a technical notation $\prec_\bullet$ defined by: $\boldsymbol{x} \prec_\bullet \boldsymbol{y}$ iff $\boldsymbol{x} \prec \boldsymbol{y}$ and $\boldsymbol{x}$ is not a prefix of $\boldsymbol{y}$. Thus, $\boldsymbol{x} \prec_\bullet \boldsymbol{y}$ iff there is $1 \leq j \leq min(|\boldsymbol{x}|, |\boldsymbol{y}|)$ so that $\boldsymbol{x}[j] \prec \boldsymbol{x}[j]$ while $\boldsymbol{x}[i] = \boldsymbol{y}[i]$ for any $1 \leq i < j$. In simple terms, $\boldsymbol{x} \prec_\bullet \boldsymbol{y}$ means that we can find a position j in $\boldsymbol{x}$ and $\boldsymbol{y}$ where they differ for the first time and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$. The great advantage of $\prec_\bullet$ is the following property not true in general for $\prec$:

Observation 2.1. If $\boldsymbol{x} \prec_\bullet \boldsymbol{y}$, then $\boldsymbol{xz} \prec_\bullet \boldsymbol{yt}$ for any string $\boldsymbol{z}$ and any string $\boldsymbol{t}$.

For example, $ab \prec abac$ and $ab \not\prec_\bullet abac$, while $aba \prec abbc$ and $aba \prec_\bullet abbc$. Note, that $aba\boldsymbol{z} \prec_\bullet abbc\boldsymbol{t}$ for any string $\boldsymbol{z}$ and any string $\boldsymbol{t}$.

An ***ordered alphabet*** is an alphabet with a total order so that compar-

isons of any two alphabet symbols can be computed in constant time. A ***sorted alphabet*** is an ordered alphabet with the additional requirement that for each alphabet symbol, the immediately preceding symbol and the immediately succeeding symbol can be computed in constant time – in practical terms it means that the alphabet is given as an ordered doubly-linked list or as an ordered array. Note that a constant alphabet can be sorted in constant time and an integer alphabet $1..n$ can be sorted in $\mathcal{O}(n)$ time. To sort a general alphabet $\mathcal{A}$ takes $\mathcal{O}(|\mathcal{A}|\ log(|\mathcal{A}|))$ time.

In this thesis, the sorting of suffixes of a string is mentioned many times. Typically, this is performed in the form of computing the suffix array of the input string. Formally, an integer array $s[1..n]$ is a ***suffix array*** of $\boldsymbol{x}$ if it is a permutation of $1..n$ and for any $i, j \in 1..n$, $\boldsymbol{x}[s[i]..n] \prec \boldsymbol{x}[s[j]..n] \Leftrightarrow i < j$. Therefore, $\boldsymbol{x}[s[1]..n]$, $\boldsymbol{x}[s[2]..n]$, ..., $\boldsymbol{x}[s[n]..n]$ is a complete list of all suffixes of $\boldsymbol{x}$ in an ascending lexicographic order. The ***inverse suffix array*** $s^{-1}[1..n]$ is an integer array so that $s^{-1}[i] = j \Leftrightarrow s[j] = i$.

For example, consider a string $\boldsymbol{x}[1..7] = abbcaad$. The suffixes are: $\boldsymbol{x}[1..7] = abbcaad$, $\boldsymbol{x}[2..7] = bbcaad$, $\boldsymbol{x}[3..7] = bcaad$, $\boldsymbol{x}[4..7] = caad$, $\boldsymbol{x}[5..7] = aad$, $\boldsymbol{x}[6..7] = ad$, $\boldsymbol{x}[7..7] = d$. Now, let us list them in an ascending lexicographic order: $\boldsymbol{x}[5..7] = aad$, $\boldsymbol{x}[1..7] = abbcaad$, $\boldsymbol{x}[6..7] = ad$, $\boldsymbol{x}[2..7] = bbcaad$, $\boldsymbol{x}[3..7] = bcaad$, $\boldsymbol{x}[4..7] = caad$, $\boldsymbol{x}[7..7] = d$.

Thus, the suffix array of $\boldsymbol{x}$ is $s = [5, 1, 6, 2, 3, 4, 7]$. Since $s[1] = 5$, $s^{-1}[5] = 1$; since $s[2] = 1$, $s^{-1}[1] = 2$; since $s[3] = 6$, $s^{-1}[6] = 3$; since $s[4] = 2$, $s^{-1}[2] = 4$; since $s[5] = 3$, $s^{-1}[3] = 5$; since $s[6] = 4$, $s^{-1}[4] = 6$; and since $s[7] = 7$, $s^{-1}[7] = 7$. Thus, the inverse suffix array is $s^{-1} = [2, 4, 5, 6, 1, 3, 7]$.

A string $\boldsymbol{x}$ over $\mathcal{A}$ is ***Lyndon with respect to the order*** $\prec$ ***of*** $\mathcal{A}$ if $\boldsymbol{x}$ is strictly lexicographically smaller than any non-trivial rotation of $\boldsymbol{x}$. Note, that if the order of the alphabet is changed, strings that were Lyndon with respect to the old order do not have to be Lyndon with respect to the new order, and vice versa. Typically, the order is clear from the context, so we do not need to specify *with respect to the order* $\prec$. A string of length 1 is Lyndon with respect to any order; we refer to it as a ***trivial*** Lyndon string.

2.1 Basic Properties of Lyndon Strings

Proposition 2.2. Let x be a string over $\mathcal{A}$. Then
x is Lyndon $\Rightarrow$ x is unbordered $\Rightarrow$ x is primitive

PROOF. A string of length 1 is Lyndon, unbordered, and primitive. Thus, we assume that x of length ≥ 2 is Lyndon with respect to $\prec$.

- x is Lyndon $\Rightarrow$ x is unbordered.
  Assume that x has a border. For the smallest border u for x, there is a non-empty v so that $x = uvu$. Let $r, k \geq 1$ be the largest integers so that there is a non-empty v with $x = u^r v u^k$. Since x is Lyndon, $u^r v u^k \prec u^{r+1} v u^{k-1}$. Thus, $vu^k \prec uvu^{k-1}$, and so $vu \prec uv$. On the other hand, since x is Lyndon, $u^r v u^k \prec u^{r-1} v u^{k+1}$. Thus, $uvu^k \prec vu^{k+1}$, and so $uv \prec uv$. Therefore, $vu \prec uv$ and $uv \prec uv$, a contradiction. It follows that x must be unbordered.

- x is unbordered $\Rightarrow$ x is primitive
  Assume that string x is not primitive. It follows that for some u, $x = u^k$ for some integer $k \leq 2$. Then u is a border of x, a contradiction.

■

Note that the reverse implications do not hold: *aba* is primitive but neither unbordered, nor Lyndon, while *acaab* is unbordered, but not Lyndon. There are several conditions that are equivalent with being Lyndon described in the following propositions.

Proposition 2.3. Let x be a string over $\mathcal{A}$ of length ≥ 2. Then
x is Lyndon $\Leftrightarrow$ for any non-trivial proper suffix v of x, $x \prec v$.

PROOF.

- x is Lyndon $\Rightarrow$ for any non-trivial proper suffix v of x, $x \prec v$
  Let $x = uv$ where $u, v \neq \varepsilon$. Assume that $x \not\prec v$. Then either v is a prefix of x, or $v \prec\bullet x$. In the former case, u would be a border of x, a contradiction. In the latter case, $vu \prec x$, a contradiction.

- for any non-trivial proper suffix v of x, $x \prec v \Rightarrow x$ is Lyndon
  Let $x = uv$. Then $x \prec v$ and so $x \prec vu$. Thus, x is Lyndon.

■

Proposition 2.4. Let x be a string over $\mathcal{A}$ of length ≥ 2. Then
x is Lyndon $\Leftrightarrow$ for any non-empty u, v so that $x = uv$, $u \prec v$.

PROOF.

- x is Lyndon $\Rightarrow$ for any non-empty u, v so that $x = uv$, $u \prec v$
  Let us assume that $u \not\prec v$. Then either v is a prefix of u or $v \prec\bullet u$. In the former case, if v is a prefix of u, then it gives x a border; which is inherently not possible. In the latter, $v \prec\bullet u$ gives $vu \prec uv$, contradicting the Lyndoness of x. Thus, $u \prec v$.

- for any non-empty u, v so that $x = uv$, $u \prec v \Rightarrow x$ is Lyndon
  Let $x = uv$. Let $r \geq 0$ be maximal such that $v = u^r w$ for some w, and so $x = u^{r+1}w$. Then $u^{r+1} \prec w$. It follows that either u^{r+1} is a prefix of w, which contradicts the maximality of r, or $u^{r+1} \prec\bullet w$. Therefore, $u^{r+1} \prec\bullet w \Rightarrow u \prec\bullet w \Rightarrow u^{r+1} \prec\bullet u^r w \Rightarrow u^{r+1}w \prec\bullet u^r w \Rightarrow u^{r+1}w \prec\bullet u^r wu \Rightarrow uv \prec vu$. Thus, x is Lyndon.

■

The previous propositions indicate that a Lyndon string can be factorized in many ways. So-called ***standard factorization*** of a Lyndon string x is a factorization $x = uv$ where both u and v are also Lyndon and v is as long as possible. The next lemma asserts that standard factorization always exists:

Lemma 2.5. For any non-trivial Lyndon string x, there are non-empty Lyndon strings u and v so that x=uv.

PROOF. There are many sources with a proof of the lemma, for instance [6].

■

2.2 Basic Properties of Lyndon Substrings

It is natural to ask whether a substring $\boldsymbol{x}[i\,..\,j]$ of a string $\boldsymbol{x}[1\,..\,n]$ is Lyndon or not. But we can also investigate the context in which they occur. The first property we define is *maximality*. A substring $\boldsymbol{x}[i\,..\,j]$ is a ***maximal Lyndon substring*** if it is Lyndon and for any $j < k \leq n$, $\boldsymbol{x}[i\,..\,k]$ is not Lyndon.

If a Lyndon substring is a suffix of $\boldsymbol{x}[1\,..\,n]$, then it is trivially maximal. But a Lyndon substring can be maximal and not be a suffix. For instance, $\boldsymbol{x}[1\,..\,6] = acbaab$, the substring $\boldsymbol{x}[1\,..\,3] = acb$ is Lyndon and it cannot be extended: $acba$ is not Lyndon (it has a border a), $acbaa$ is not Lyndon (it has a border a), and $acbaab$ is not Lyndon (as $acb \not< aab$ contradicts Proposition 2.4). Note that acb will be the maximal Lyndon substring of $\boldsymbol{xy}$ for any $\boldsymbol{y}$, i.e. in a certain sense it is *non-extensible*, while aab is a maximal Lyndon substring of $\boldsymbol{x}$, but not a maximal Lyndon substring of $\boldsymbol{xy}$ if, for instance, $\boldsymbol{y} = b$.

The information of all maximal Lyndon substrings is succinctly captured by the notion of Lyndon array introduced in [17]. 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 $\mathcal{L}[i] = \mathcal{L}'[i]-i+1$.

Consider $\boldsymbol{x}[1\,..\,6] = aababb$. Since $aababb$ is Lyndon, it is maximal, and so $\mathcal{L}[1] = 6$, or $\mathcal{L}'[1] = 6$. The maximal Lyndon substring starting at position 2 is $ababb$, so $\mathcal{L}[2] = 5$ and $\mathcal{L}'[2] = 6$. The maximal Lyndon substring starting at position 3 is b, so $\mathcal{L}[3] = 1$ and $\mathcal{L}'[3] = 3$. The maximal Lyndon substring starting at position 4 is abb, so $\mathcal{L}[4] = 3$ and $\mathcal{L}'[4] = 6$. The maximal Lyndon substring starting at position 5 b, so $\mathcal{L}[5] = 1$ and $\mathcal{L}'[5] = 5$. The maximal Lyndon substring starting at position 6 b, so $\mathcal{L}[6] = 1$ and $\mathcal{L}'[6] = 6$. Therefore, $\mathcal{L} = [6,5,1,3,1,1]$ while $\mathcal{L}' = [6,6,3,6,5,6]$.

From the work of Hohlweg and Reutenauer, [18] follows the next lemma describing a very important relationship between Lyndon, maximal Lyndon substrings of a string, and the order of the suffixes of that string.

Lemma 2.6. Consider a string $x = x[1..n]$. Then for any $1 \leq i \leq j \leq n$

(1) $x[i..j]$ is Lyndon $\Leftrightarrow$ for any $i < k \leq j$, $x[i..n] \prec x[k..n]$

(2) $x[i..j]$ is maximal Lyndon $\Leftrightarrow$ for any $i < k \leq j$, $x[i..n] \prec x[k..n]$ and either $j = n$ or $x[j+1..n] \prec x[i..n]$.

PROOF. See Lemma 15 and its proof in Appendix 2 of [17].

■

3 Background of the Problem

In Chapter 1, it was explained that the interest in maximal Lyndon substrings of a string initially piqued after the debut of Bannai et al.'s algorithm, [4, 5], on computing all the runs in a string over the period of 2015-2017. The algorithm relies on the knowledge of all maximal Lyndon substrings of the input string with respect to a given order of the alphabet and the inverse of the given order. To compute all such maximal Lyndon substrings, the authors presented what in this thesis is called *SSLA* and which is described below. This inspired one of McMaster University's research groups to present a brief but comprehensive overview of all algorithms known for computing the Lyndon array, [17]. The layout of this chapter follows the layout of this paper and includes additional algorithms not known at that time.

3.1 Brute Force

A naive brute force approach *LynArr* is given in Fig. 3.1. For each position $i \in 1..n$, the procedure *MaxLyn* returns the length of the maximal Lyndon substring starting at the position i. The procedure *MaxLyn* tries all possible end points j and uses the *IsLyndon* procedure to check whether $\boldsymbol{x}[i..j]$ is Lyndon. The variable *max* stores the so far attained maximal length. The procedure *IsLyndon* tries and compares all possible rotations of $\boldsymbol{x}[i..j]$ to see if $\boldsymbol{x}[i..j]$ is Lyndon. The comparison of $\boldsymbol{x}[i..j]$ and its rotation is performed by the procedure *Lex*. The complexity of *Lex* is $O(n)$, Thus, the complexity of the procedure *IsLyndon* is $O(n^2)$, the complexity of the procedure *MaxLyndon* is $O(n^3)$, and the complexity of the procedure *LynArr* is $O(n^4)$.

This is a typical problem in the field of algorithms on strings. The brute force approach is simple and its complexity is polynomial with a small degree. It is only for large strings, like those used in DNA processing or web searches, where even a small degree of the polynomial complexity matters; typically, it should be better than quadratic complexity. Thus, this research focused on sub-quadratic algorithms, i.e. linear algorithms and $\mathcal{O}(n\ log(n))$ algorithms, where n is the length of the input string.

3.2 Iterative Duval Algorithm – IDLA

The simplest and the most elegant algorithm for computing maximal Lyndon substrings is based on Duval's algorithm for Lyndon factorization, [12]. Since the algorithm referred to as *IDLA* was so essential to this research, an entire chapter has been devoted to it (see chapter 4).

3.3 Recursive Duval Algorithm - RDLA

RDLA is also based on Duval's algorithm for Lyndon factorization which, is applied recursively rather than iteratively:

> if $\boldsymbol{x}[1..i_1]\ \boldsymbol{x}[i_1{+}1..i_2]$ ... $\boldsymbol{x}[i_k{+}1..n]$ 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 $\boldsymbol{x}[i_k{+}2..n]$.

The correctness of the algorithm is consequent from the correctness of Duval's original algorithm. The alphabet of the input string need not be sorted, but must be ordered. *RDLA* has a 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 [17]. Storage requirements are the same as for *IDLA*, plus the additional storage for the stack controlling the recursion.

```
procedure Lex(x[1..n], y[1..m], Σ, ≺) boolean
  i ← 1
  while i ≤ n and i ≤ m and x[i] ≺ y[i] do
    i ← i + 1
  if i ≤ n then return FALSE
  return TRUE

procedure IsLyndon(x[1..n], i, j, Σ, ≺) : boolean
  i ← 1
  while i ≤ n do
      y[i] ← x[i]
      i ← i + 1
  y[1..n] ← y[2..n]y[1]
  i ← 2
  while i ≤ n and lex(x[1..n], y[1..n], Σ, ≺) do
    i ← i + 1
    y[1..n] ← y[2..n]y[1]
  if i > n then
    return TRUE
  else
    return FALSE

procedure MaxLyn(x[1..n], i, Σ, ≺) : integer
  max ← 1
  j ← i + 1
  while j ≤ n do
    if IsLyndon(x, i, j, Σ, ≺) then
      max ← j − i + 1
    j ← j + 1
  return max

procedure LynArr(x[1..n], Σ, ≺) : integerarray[1..n]
  i ← 1
  while i ≤ n do
    L[i] ← MaxLyn(x, i, Σ, ≺)
    i ← i + 1
  return L[1..n]
```

Figure 3.1: Brute force algorithm

3.4 Algorithmic Scheme Based on Suffix Sorting - SSLA

The suffix sorting algorithmic scheme is based on Lemma 2.6 which characterizes maximal Lyndon substrings in terms of the relationships of the suffixes. Therefore, the Lyndon array of x is the *NSV* (Next Smaller Value) array of the inverse suffix array. The scheme is as follows:

(1) sort the suffixes, i.e. compute the suffix array;

(2) from the suffix array compute the inverse suffix array; and

(3) 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 [17, 26] and the references therein. The time and space characteristics of the whole scheme are dominated by the time and space characteristics of step (1) – the computation of the suffix array. For linear suffix sorting, the input strings must be over constant or integer alphabets.

3.5 Algorithmic Scheme Based on Burrows-Wheeler Transform – BWLA

The algorithmic scheme based on Burrows-Wheeler transform was not presented in [17] as it was introduced in 2018, see [24]. The algorithm is linear and computes the Lyndon array from a given Burrows-Wheeler transform of the input string. In some sense, it is a "byproduct" of the computation of the inverse of the Burrows-Wheeler transform. Since the only known linear computation of Burrows-Wheeler transform is from the suffix array, it is yet another scheme of how to obtain the Lyndon array via suffix sorting:

(1) compute the suffix array;

(2) from the suffix array compute the Burrows-Wheeler transform; and

(3) compute the Lyndon array during the inversion of the Burrows-Wheeler transform.

As for *SSLA*, the execution and space characteristics of the whole scheme are dominated by the computation of the suffix array.

3.6 An Algorithm Based on Ranges – RGLA

This algorithm was discussed in [17] where it was referred to as *NSV**. The algorithm is emulating the stack-based implementation of *NSV* working with ranges. In case of a constant alphabet, ranges can be compared in constant time if the Parikh vector for each range is pre-computed. 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 $i \leq k < \ell \leq j$. A more efficient version based on pre-computation of Parikh vectors for ranges is also presented. The time and space complexity of the algorithms was not given in [17], but based on the algorithms construction, the time complexity should be at worst $\mathcal{O}(n^2)$. In [17] it was indicated that the time complexity could possibly be $\mathcal{O}(n\ log(n))$, where n is the length of the input string, however, no formal analysis or proof were included and there has been no followup research concerning these two algorithms.

4 Iterated Duval Algorithm (IDLA)

The Iterated Duval algorithm for Lyndon Array, hereinafter referred to as *IDLA*, is based on the work of Jean-Pierre Duval [12]. His linear algorithm was designed for Lyndon factorization of a string. The fundamental procedure we refer to as *MaxLyn* identifies the maximal Lyndon prefix of a string, which can easily be adopted for computing Lyndon array [17]. With respect to Lyndon factorization, Duval iteratively applied *MaxLyn* to the suffix starting at the position immediately following the end of the maximal Lyndon prefix identified. In *IDLA*, *MaxLyn* is applied to every suffix of the input string.

The algorithm *IDLA*, see Fig. 4.1, is a simple and in-place algorithm. This means that aside from the original string and the Lyndon array, no additional storage is required. The algorithm *IDLA* is completely independent of the alphabet of the string and does not require the alphabet to be sorted, just to be ordered; i.e. a pairwise constant-time comparison of alphabet symbols must be provided.

One caveat with this algorithm is that *IDLA* has an $\mathcal{O}(|x|^2)$ worst-case complexity, where x is the input string, which may be problematic for longer strings. However, note that a brute force approach to determining if a prefix is Lyndon is quadratic; it must be checked against all its rotations. The elegance of Duval's design is that it can perform this operation in linear time, bringing the overall complexity of *IDLA* to quadratic. The fact that *MaxLyn* truly computes the length of the maximal Lyndon prefix is not obvious and follows from a deep understanding of the aperiodic nature of Lyndon words.

Since *IDLA* is so essential for this work as it was used both as a yardstick for performance comparisons and a result verifier of *TRLA* and *BSLA*,

```
procedure MaxLyn(x[1..n], j, Σ, ≺) : integer
                    ; max = length of the maximal Lyndon prefix of x[j..n]
  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], Σ, ≺) : integer array
                                  ; x string over alphabet Σ ordered by ≺
  i ← 1
  while i < n do
    L[i] = MaxLyn(x[1..n], i, Σ, ≺)
    i ← i + 1
  L[n] ← 1
  return L
```

Figure 4.1: Algorithm *IDLA*

it is important to describe *IDLA* to some degree even though it had been covered in both [17] and [27].

As mentioned in Chapter 2, there are several string conditions equivalent with the string being Lyndon. In Lemma 4.2, we present another one, but first a definition of the prefix table of a string.

Definition 4.1. For a string $x = x[1..n]$, ***prefix table*** $\pi[1..n]$ is an integer array in which for every $i \in 1..n$, $\pi[i]$ is the length of the longest substring beginning at position i of x that matches a prefix of x.

The recent results on the prefix table [7, 9] confirm its relevance for string algorithms. The following lemma is simple as $1+\pi[i]$ and $i+\pi[i]$ are the first positions at which $x[1..n]$ and $x[i..n]$ differ.

Lemma 4.2. $\boldsymbol{x}$ is a Lyndon string if and only if for every i such that $2 \leq i \leq n$ and $i+\pi[i] \leq n$, it follows that $\boldsymbol{x}[1+\pi[i]] \prec \boldsymbol{x}[i+\pi[i]]$.

PROOF. $\boldsymbol{x}[1\,..\,n]$ is Lyndon $\Leftrightarrow$ for any i such that $2 \leq i \leq n$, $\boldsymbol{x}[1\,..\,n] \prec \boldsymbol{x}[i\,..\,n]$. Since $\boldsymbol{x}[1\,..\,\pi[i]] = \boldsymbol{x}[i\,..\,i+\pi[i]-1]$, it follows that $\boldsymbol{x}[1+\pi[i]] \prec \boldsymbol{x}[i+\pi[i]]$.

■

Lemma 4.3. Suppose that for some position i in a Lyndon string $\boldsymbol{x}[1\,..\,n]$, $\pi[i] \geq 2$. Then for every j such that $i < j < i+\pi[i]$, $\pi[j] \leq i+\pi[i]-j$.

PROOF. If $i+\pi[i] = n+1$, then $\boldsymbol{x}$ would have a border, and hence could not be Lyndon. Thus, $i+\pi[i] \leq n$. Arguing by contradiction, assume that for some j, $\pi[j] > i+\pi[i]-j$. It follows that:

$$\boldsymbol{x}[1\,..\,i+\pi[i]-j+1] = \boldsymbol{x}[j\,..\,i+\pi[i]], \tag{1}$$

while $\boldsymbol{x}[j-i+1\,..\,\pi[i]] = \boldsymbol{x}[j\,..\,i+\pi[i]-1]$. Since $\boldsymbol{x}$ is Lyndon, therefore, $\boldsymbol{x}[1+\pi[i]] \prec \boldsymbol{x}[i+\pi[i]]$ by Lemma 4.2, and so:

$$\boldsymbol{x}[j-i+1\,..\,1+\pi[i]] \prec \boldsymbol{x}[j\,..\,i+\pi[i]]. \tag{2}$$

From (1) and (2) we see that $\boldsymbol{x}[1\,..\,\pi[i]+1]$ has suffix $\boldsymbol{x}[j-i+1\,..\,\pi[i]+1]$ satisfying $\boldsymbol{x}[j-i+1\,..\,\pi[i]+1] \prec \boldsymbol{x}[1\,..\,i+\pi[i]-j+1]$, $\boldsymbol{x}[1\,..\,\pi[i]+1]$ has suffix $\boldsymbol{x}[j-i+1\,..\,\pi[i]+1]$ that is lexicographically smaller than prefix $\boldsymbol{x}[1\,..\,i+\pi[i]-j+1]$, contradicting the assumption that $\boldsymbol{x}$ is Lyndon.

■

The result of Lemma 4.3 forms the basis of the procedure *MaxLyn* in Figure 4.1 that computes the *maximum* length of the longest Lyndon prefix. The efficiency of *MaxLyn* is a consequence of the instruction $i \leftarrow i+k+1$ that skips over positions in the range $i+1\,..\,i+k-1$, effectively assuming that for every position j in that range, $j + \pi[j] \leq i + k$.

4.1 Implementation Notes

The implementation of *IDLA* is straightforward, simple, and in-place (no space is required except the storage of the string and the Lyndon array). The algorithm is implemented in `C++` and for strings whose alphabet symbols can be non-negative 64-bit signed integers; including 0.More precisely, the `C++` data type of the alphabet symbols is `long`, but only non-negative values are permitted. To that end, the class `Lstring` implements such a string as a `long` array with an additional attribute `len` that stores the length of the string. For additional details, the code has been made publicly available [1]. The `C++` source file containing the procedure `idla` and some other related procedures is named `lynarr.hpp`.

Note that the possible alphabet is finite in the theoretical sense, as there are only $2^{63}+1$ possible alphabet symbols. However, since the lengths of the test strings by far do not exceed the length $2^{63}+1$, we can pretend that we are working with integer alphabets.

5 Baier's Sort Inspired Algorithm (BSLA)

This Lyndon array algorithm, hereinafter referred to as *BSLA*, is inspired by the technique Baier designed for phase I of his suffix sorting algorithm [2, 3]. In that phase, all suffixes with the same maximal Lyndon prefix are identified and grouped together. The input strings for *BSLA* are tight strings over integer alphabets. It is important to note that the requirement of the tightness of the input string does not significantly detract from the applicability of the algorithm as any string x over an integer alphabet can easily be transformed in $\mathcal{O}(|x|)$ time to a tight string; see procedure *Tight* at Fig. 5.1. The original string x and the transformed string have the same Lyndon array since they are *isomorphic* (see definition 5.1), thus, the Lyndon array computed by *BSLA* for the transformed tight string is the Lyndon array of the original string x as well.

Definition 5.1. For strings x and y, let $\mathcal{A}(x) = \{a_1, .., a_n\}$ and $\mathcal{A}(y) = \{b_1, .., b_n\}$. Let $f : \mathcal{A}(x) \implies \mathcal{A}(y)$ be a bijection such that $a_i \prec a_j$ iff $f(a_i) \prec f(a_j)$. If $f(x) = f(x[1]) .. f(x[n]) = y$, then x and y are isomorphic.

It is quite clear, a straightforward observation, that the procedure *Tight* is linear in the length of the input string x; as it consists of a series of traversals of an array of length $|x|$ or traversals of the string x. The only exception is the one loop containing a nested loop, its inner loop is to set a new value for $free$, and it traverses from the old $free$ to the new $free$. Therefore, the loop is performed in $2|x|$ steps. Further, the procedure requires a working space in the form of an array of size $|x|$. The storage for the Lyndon array to be computed by *BSLA* can be used for this step, so *Tight* does not require any additional space, if used.

```
procedure Tight(x[1..n], frq[1..n])
  i ← 1                                              ; initialize frq[]
  while i ≤ n do
    frq[i] ← 0
    i ← i + 1
  i ← 1                          ; traverse x and set occurrences of symbols
  while i ≤ n do
    frq[x[i]] ← 1
    i ← i + 1
  free ← −1                                          ; find first unused symbol
  i ← 1
  while i ≤ n do
    if frq[x[i]] = 0 then
      free ← i
      break
  if free = −1 then return                           ; x already tight
  i ← 1
  while i ≤ n do
    if frq[i] = 0 then continue
    if i < free then continue
    frq[i] ← free − i
    j ← free + 1                                     ; update free
    while j ≤ i do
      if frq[j] ≤ 0 then
        free ← j
        break
  i ← 1                                              ; traverse x and modify it
  while i ≤ n do
    if frq[x[i]] = 1 continue
    x[i] ← x[i] + frq[x[i]]
  return
```

Figure 5.1: Algorithm *Tight*

BSLA is based on a refinement of a list of groups of indices of the input string $\boldsymbol{x}$. The initial list of groups consists of the groups of indices with the same alphabet symbol. The refinement is driven by a group that is already complete (see Definition 5.5) and completes the immediately preceding group. In turn, this newly completed group is used as the driver of the next round of the refinement. Hence, the refinement proceeds from right-to-left until all of the groups in the list are complete. For the intuition behind the refinement process, please see section 5.3 below.

Each group is assigned a specific substring of the input string, this is referred to as the ***context*** of the group. There is a tight relationship between the context and the indices in the group: for every index i of the group, there is an occurrence of the context at the position i; for a precise definition see Section 5.1. Throughout the process the list of 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. Therefore, 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, prove that it is correct, and to describe the refinement in technical detail, several properties must be introduced and notions formally defined.

5.1 Notation and Notions for Analysis of BSLA

For simplicity's sake, we fix a tight string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over an integer alphabet for Section 5.1 in its entirety; all the definitions and observations refer to this $\boldsymbol{x}$.

A ***group*** $\boldsymbol{G}$ 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 5.2. An ordered list of groups $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$ is a **group configuration**, if all four conditions (C_1), (C_2), (C_3), and (C_4) hold, where:

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

(C_2) $G_j \cap G_\ell = \emptyset$ for any $j \neq \ell$;

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

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

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 the range $1\,..\,n$.

For a given group configuration $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$, we introduce two functions, gr and $prev$. The conditions (C_1) and (C_2) guarantee that for every $i \in 1\,..\,n$, there is a unique G_t in the configuration so that $i \in G_t$. Let $gr(i)$ denote this unique group, i.e. $gr(i) = G_t$. Using the notion of gr, $C(i) = \boldsymbol{x}[i\,..\,i+|con(gr(i))|-1]$. 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 each group G from a group configuration $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$, we define 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 equivalence class of $\sim$ 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:

Proposition 5.3. Let G be a group from a group configuration $\langle G_k, G_{k-1}, ..., G_2, G_1\rangle$. 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 $\approx$ 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 an equivalence relation, which we also denote by $\approx$. The symbol $[\boldsymbol{i}]_\approx$ denotes the equivalence class of $\approx$ containing i, i.e. $[i]_\approx = \{j \in [i]_\sim \mid j \approx i\}$.

For each j from a non-trivial $[i]_\sim$, we define the ***valence*** by $\boldsymbol{val(j)} = |[i]_\approx|$. *In simple terms, $val(j)$ is the number of elements from $[j]_\sim$ that are $\approx j$.* Thus, $1 \leq val(j) \leq |G|$. Interestingly, if G is viewed as an ordered set of indices, then $[i]_\approx$ is a subinterval of the interval $[i]_\sim$:

Proposition 5.4. 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 5.5. A group G is ***complete*** if for any $i \in G$, the occurrence $C(i)$ of $con(G)$ is a maximal Lyndon substring of $\boldsymbol{x}$.

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

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

5.2 The Refinement

For simplicity's sake, we fix a tight string $x = x[1..n]$ over an integer alphabet for Section 5.2 in its entirety; all the definitions, lemmas, and theorems refer to this x.

Lemma 5.6. Let $\mathcal{A}_x = \{a_1, ..., a_k\}$ and $a_1 \prec a_2 \prec ... \prec a_k$. For $1 \leq \ell \leq k$, define $G_\ell = \{i \in 1..n : x[i] = a_{k+1-\ell}\}$ with context $a_{k+1-\ell}$. Then $\langle G_k, ..., G_1 \rangle$ is a 1-complete group configuration.

PROOF. (C_1), (C_2), (C_3), and (C_4) are straightforward to verify. To verify (C_5), we need to show that G_1 is complete. Any occurrence of a_k in 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 $x[\ell] \neq a_k$, then $prev(i) < \ell$ which contradicts the definition of $prev$. Hence $x[\ell] = a_k$ and so $x[j+1] = ... = x[i] = ... x[j+v+1] = a_k$ while $x[j] = a_\ell$ for some $\ell < k$. It follows that $x[j..n]$ 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 $C(i) = x[i]$, $j = i+1$, $C(j) = x[i+1]$ and $x[i] \prec x[i+1]$. Then $con(C(j)) = x[i+1] \preceq a_k = con(G_1)$, so $(C_7(b))$ is also satisfied.

To verify (C_8), consider $C(i) \cap C(j) \neq \emptyset$. Then $C(i) = x[i] = x[j] = C(j)$.

■

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. The refinement by G_t consists of three steps (R_1), (R_2), and (R_3) described below.

(R_1) The group G_t is partitioned into equivalence classes of $\sim$:
Thus, $G_t = [i_1]_\sim \cup [i_2]_\sim \cup ... \cup [i_p]_\sim \cup X$, where $X = \{i \in G_t : prev(i) = nil\}$, which may be empty, and $i_1 < i_2 < ... < i_p$.

(R_2) Every class $[i_\ell]_\sim$, $1 \leq \ell \leq p$, is then partitioned into equivalence classes of $\approx$:
Thus, $[i_\ell]_\sim = [j_{\ell,1}]_\approx \cup [j_{\ell,2}]_\approx \cup ... \cup [j_{\ell,m_\ell}]_\approx$, 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 target group $gr(prev(j_{\ell,k}))$ into a new group H and place H in the list of groups right after the target 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 5.7 shows that having a t-complete group configuration $\langle G_k, ..., G_{t+1}, G_t, ..., G_1\rangle$ and refining it by G_t, then the resulting system of groups is a $(t+1)$-complete group configuration. This allows to carry on the refinement in an iterative fashion.

Theorem 5.7. 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.

$\therefore$ concluding 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.

- 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 are done.
    Let us 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 $val(j) = v+1$.

$\therefore$ concluding the proof of Claim 2.

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

Proof of Claim 3.
Let $j = prev'(i)$ and $i \in H_{t+1}$ with $val'(i) = v$. Then, $|[i]_{\approx}| = v$ and so $[i]_{\approx} = \{i_1, ..., i_v\}$, where $i_1 < i_2 < ... < i_v$. Hence, $i_1, ..., i_v \in H_{t+1}$ and $C'(i_1) = ... = C'(i_v) = con'(H_{t+1})$ 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}.

$\therefore$ concluding 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 made for j_2 can now be made for j_3. This results in $i+|con'(H_{t+1})| \leq j_3 < j_2 < j_1 \leq k$ and $C'(j_3) \succeq C'(j_2) \succeq C'(j_1) \succ con'(H_{t+1})$. If $i+|con'(H_{t+1})| = j_3$, then it is a contradiction, so $i+|con'(H_{t+1})| < j_3$. These arguments 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.

Thus, the initial assumption that $C'(i)$ is not maximal always leads to a contradiction.

$\therefore$ concluding the proof of Claim 4.

The four claims show that all the conditions (C_1) ... (C_8) are satisfied for $Conf'$, and that proves Theorem 5.7.

■

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 5.8.
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 $1\overline{7}$complete group configuration from Lemma 5.6.

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 $2\overline{7}$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 $3\overline{7}$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 $r\overline{7}$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 5.7 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 $\boldsymbol{x}[i\,..\,k]$.

■

5.3 Intuition Behind the Refinement Process

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

(a) lower the water level by one

(b) extend the existing Lyndon substrings
the revealed letters are used to extend the existing Lyndon substrings where possible, or became Lyndon substrings of length 1 otherwise;

(c) consolidate the new Lyndon substrings
processed from the right, if several Lyndon substrings are adjacent and can be joined to a longer Lyndon substring, they are joined.

The diagram in Fig. 5.2 and the description that follows it illustrate the method for a string 011023122. The input string is visualized as a curve and the height at each point is the value of the letter at that position.

In Fig. 5.2, 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.

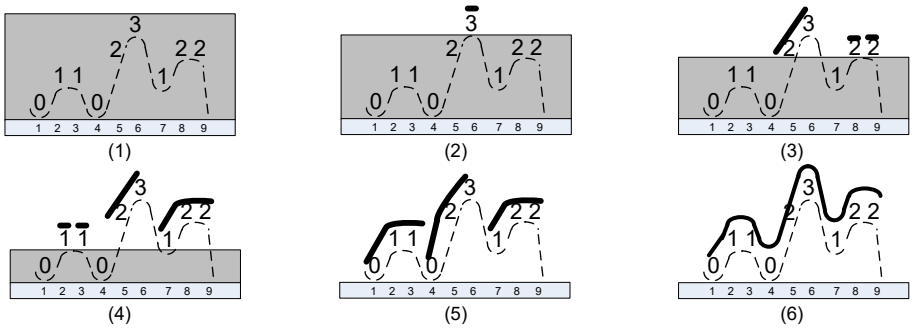


Figure 5.2: Water draining process for 011023122

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

In Figure 5.3, below, an illustrative example for the string 011023122 is presented, where the arrows represent the *prev* mapping shown only on the group used for the refinement which is indicated by the bold font. The group contexts are shown as indices of the groups, thus, G_0 is a group with context 0, or $G_{011023122}$ is a group with context 011023122.

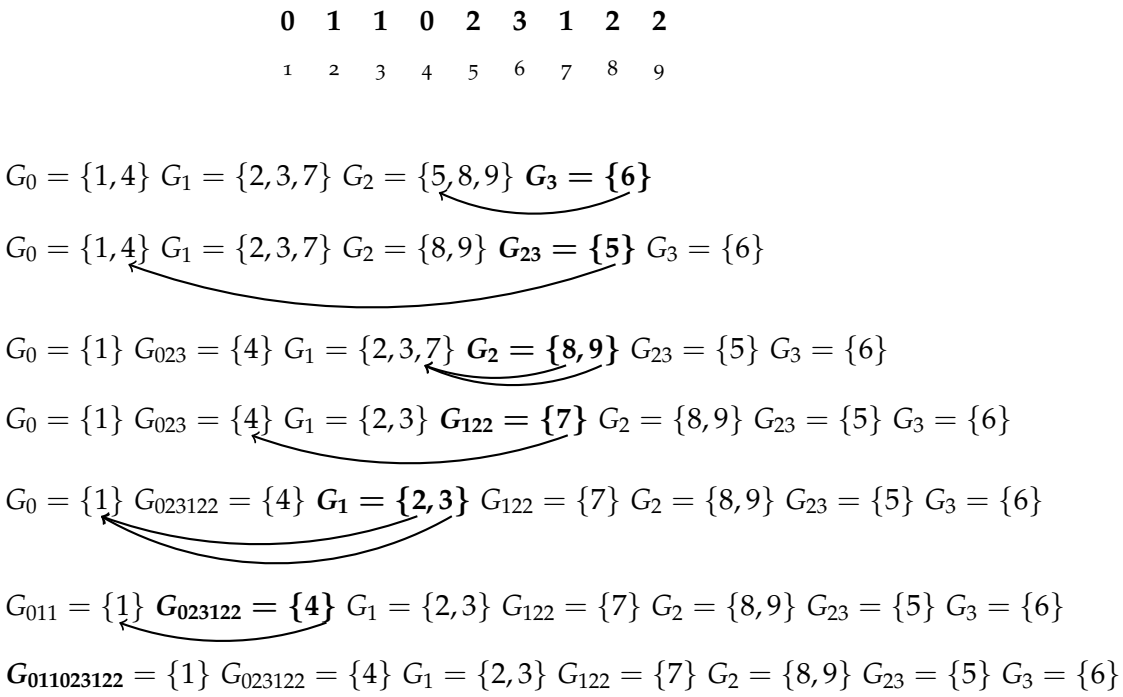


Figure 5.3: Illustration of the refinement process

5.4 Implementation Notes

The *BSLA* algorithm was implemented in `C++` and made publicly available [1]. First it is important to note that this implementation only works on tight strings (see procedure *Tight*, Fig. 5.1, for explanation of how to convert a non-tight string to an isomorphic tight format).

The most important aspect of the implementation is the design of the data structures representing the group configuration. During one step of the refinement process, some groups remain unchanged, but some groups are partitioned. The data structures selected for the representation of the group configuration must be highly dynamic.

We decided to represent a group as a doubly-linked list and similarly, a group configuration as a doubly-linked list. To prevent costly dynamic memory allocation and deallocation during the programs execution, the groups and the group configuration are represented by static integer arrays. This is possible because the group configuration is always partitioning the index set $1\,..\,n$ for an input string $x[1\,..\,n]$, thus, the space requirement does not change. At the onset of the program's execution, these arrays are allocated together once, as a single block, and they are never deallocated. There are 8 arrays of length n, with the following variable names: `Gstart`, `Gprev`, `Gnext`, `Gcntxt`, `Gval`, `Cprev`, `Cnext`, and `Gmemb`.

The groups are indexed by values from $1\,..\,n$ which is adequate since there are never more than n groups.

- `Gstart` emulates the pointer to the first element of the group (i.e. the group's start), thus, `Gstart[i]=j` means that `j` is the first element of the group with index `i`.
- `Gprev` emulates the link to the previous element of the group, thus, `Gprev[i]=j` means that the previous element in the group to which `i` belongs is `j`.
- `Gnext[i]=j` means that the next element in the group to which `i` belongs is `j`.
- `Gcntxt` contains the length of the context for the group, thus, `Gcntxt[i]=j` means that the context of the group with index `i` has length `j`. To find the value of the context is straightforward – take any element of that group, say `k`, then x`[k..j]` is the context.
- `Gval` represent the valence of elements, thus, `Gval[i]=j` means that the valence of the element `i` is `j`.
- `Cprev` emulates the pointer to the previous group in the configuration, thus, `Cprev[i]=j` means that the group with index `i` is preceded in the configuration by the group with index `j`.
- `Cnext` emulates the pointer to the next group in the configuration, thus, `Cnext[i]=j` means that the group with index `i` is succeeded in the configuration by the group with index `j`.
- `Gmem` stores the membership of elements, thus, `Gmem[i]=j` means that the element `i` belongs to the group with index `j`.

The data structure allows the *creation of an empty group*, *deletion of an empty group*, and the *movement of an element from one group to another* to all be constant time operations. Additionally, three auxiliary arrays (`auxA`, `auxB`, and `auxC`) are needed for operations such as sorting. Finally, an array `prev` holds the values of the *prev* function.

The computation of the initial configuration is straightforward. Since the input string is tight, a simple traversal of the input string determines the alphabet of the string, so the groups can be created empty, and then filled with another traversal of the input string. During the refinement of a single target group, updating `prev` is easily computed in time proportional to the size of the group driving the refinement, and so for one step of the refinement, the update of `prev` is linear. However, it is important to compute the values of the initial `prev` in linear time. Though the computation of the initial `prev` in linear time is not complicated, it is not straightforward either. This process uses a stack approach similar to the stack implementation of *NSV* made possible by the tightness of the input strings.

Fig. 5.4 provides the code for the stack implementation of the computation of `prev`'s initial value. Note that the program does not require the additional structure `prev_stack` as the auxiliary array `auxA` is not needed yet, and so its storage is used to house the stack `prev_stack`.

```
prev[0] ← nil
empty prev_stack
push 0 onto prev_stack
i ← 1
while i < n do
  if x[i1] < x[i] then
    prev[i] ← i1
    push i onto prev_stack
  elseif x[i1] = x[i] then
    prev[i] ← prev[i1]
    pop prev_stack
    push i onto prev_stack
  elseif x[i1] > x[i] then
    while TRUE do
      j ← top of prev_stack
      if j = nil then
        push i onto prev_stack
        prev[i] ← nil
        break
      elseif x[j] > x[i] then
        pop prev_stack
      elseif x[j] = x[i] then
        pop prev_stack
        push i onto prev_stack
        prev[i] ← prev[j]
        break
      else
        prev[i] ← j
        push i onto prev_stack
        break
```

Figure 5.4: Initial setting of `prev`

6 τ-Reduction Algorithm for Lyndon Array (TRLA)

This algorithm was initially presented, conceptually, as part of Asma Paracha's 2017 Ph.D. thesis [27]. It follows Martin Farach-Colton's approach used in his remarkable linear algorithm for suffix tree construction [13], and reproduced very successfully in all linear algorithms for suffix sorting, for instance see [25, 26] and the references therein. The scheme for computing the Lyndon array works as follows:

1. reduce the input string $\boldsymbol{x}$ to its τ-reduction $\boldsymbol{y}$;
2. by recursion compute the Lyndon array of $\boldsymbol{y}$; and
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 the suffix $\boldsymbol{x}[i+1..n]$ with the suffix $\boldsymbol{x}[j+1..n]$. For instance, in the first linear algorithm for suffix array by Kärkkäinen and Sanders, [19], 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 2.6, 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 $\underline{\underline{000\underline{1}1}\,\underline{00\underline{1}}}$ and would need to compute the missing maximal Lyndon substrings that are underlined in $\underline{000\underline{1}1\underline{00}\underline{1}}$. 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 $\boldsymbol{x}$.

6.1 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 ***$\tau$-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 $(1, 2)$;
- let $(i-1, i)$ be the last τ-pair computed:

  if $i = n-1$ **then**
  the next τ-pair is set to $(n, n+1)$
  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. 6.1, for the formal proof see Lemma 6.1.

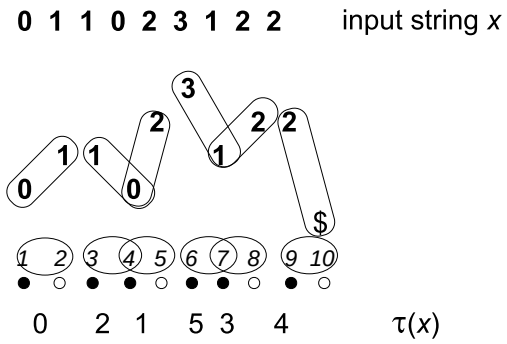


Figure 6.1: τ-reduction of string **011023122**
The rounded rectangles indicate symbol τ-pairs, the ovals indicate the τ-pairs below are the colour labels of positions, at the bottom is the τ-reduction

Lemma 6.1. 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* $m \neq j, \ell$, $|(i_j, i_j{+}1) \cap (i_m, i_m{+}1)| = 0$; and

(2) $|(i_j, i_j{+}1) \cap (i_\ell, i_\ell{+}1)| \leq 1$.

PROOF. By induction. Trivially true for $|\boldsymbol{x}| = 1$ as $(1, 2)$ is the only τ-pair. Further, let's assume that it is true for $|\boldsymbol{x}| \leq n{-}1$.

- 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 $\boldsymbol{x}[1..n{-}1]$, 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 $\boldsymbol{x}[1..n{-}1]$, 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 $\boldsymbol{x}[1..n{-}2]$, 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 $(i_1, i_1{+}1)...(i_k, i_k{+}1)$.

■

6.2 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 by a radix sort with keys of size 2, and assigned letters from a chosen τ-alphabet that is a subset of $\{0, 1, ..., |\tau(\boldsymbol{x})|\}$ so that the assignment preserves the order. Because the input string was over an integer alphabet, the radix sort is linear.

In the example, Fig. 6.1, 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 symbol's 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. 6.1, $\tau(\boldsymbol{x}) = 021534$. The next lemma justifies calling the above transformation a reduction.

Lemma 6.2. For any string x of size at least 2, $\frac{1}{2}|x| \leq |\tau(x)| \leq \frac{2}{3}|x|$.

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

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

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

Additionally, p is defined as the mapping symbol of the τ-pairs to the τ-alphabet.

In the running example from Fig. 6.1: $t(1) = 1$, $t(3) = 2$, $t(4) = 3$, $t(6) = 4$, $t(7) = 5$, and $t(9) = 6$, while $b(1) = 1$, $b(2) = 3$, $b(3) = 4$, $b(4) = 6$, $b(5) = 7$, and $b(6) = 9$. Further, the letter mapping yields $p(1, 2) = 0$, $p(3,4) = 2$, $p(4,5) = 1$, $p(6,7) = 5$, $p(7,8) = 3$, and $p(9,10) = 4$.

6.3 Properties Preserved by τ-reduction

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

Theorem 6.3. Let $x = x[1..n]$, let $\mathcal{L}'_{\tau(x)}[1..m]$ be the Lyndon array of $\tau(x)$, and let $\mathcal{L}'_x[1..n]$ be the Lyndon array of x. Then for any black $i \in 1..n$,

$$\mathcal{L}'_x[i] = \begin{cases} b\big(\mathcal{L}'_{\tau(x)}[t(i)]\big) & \text{if } x[b\big(\mathcal{L}'_{\tau(x)}[t(i)]\big)+1] \preceq x[i] \\ b\big(\mathcal{L}'_{\tau(x)}[t(i)]\big)+1 & \text{otherwise.} \end{cases}$$

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

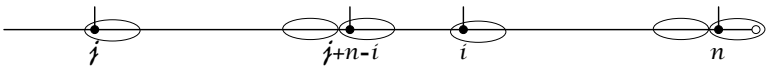
Lemma 6.4. Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \leq i, j \leq n$. If i and j are both black positions, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$ implies $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.

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.
    - Case that n is white.

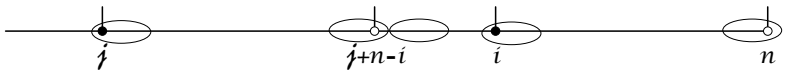
      Then the last τ-pair overlapping $j..j+n-i$ must be $(j+n-i-1, j+n-i)$ followed by a τ-pair $(j+n-i, j+n-i+1)$, and the last τ-pair overlapping $i..n$ must be the last τ-pair $(n-1, n)$. Thus, $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n-1)] = \tau(\boldsymbol{x})[t(j)..t(j+n-i-1)]$,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

    - Case that n is black.



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

- Case $j+n-i$ is white.



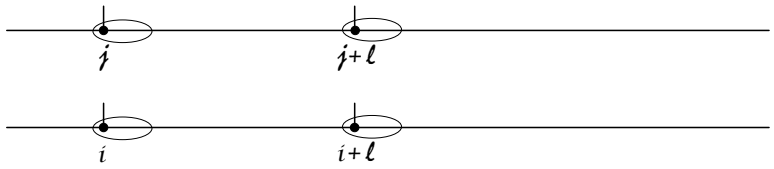
Then n must also be white as the τ-pair overlapping $j..j+n-i$ must be $(j+n-i-1, j+n-i)$ followed by $(j+n-i+1, j+n-i+2)$, and the last τ-pair overlapping $i..n$ must be the very last τ-pair $(n-1, n)$. Then $\tau(\boldsymbol{x})[t(i)..m] = \tau(\boldsymbol{x})[t(i)..t(n-1)] = \tau(\boldsymbol{x})[t(j)..t(j+n-i-1)]$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

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

Then $\tau(\boldsymbol{x})[t(i)] = p(i, i+1)$ and $\tau(\boldsymbol{x})[t(j)] = p(j, j+1)$. Since $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$, we have $(\boldsymbol{x}[i], \boldsymbol{x}[i+1]) \prec (\boldsymbol{x}[j], \boldsymbol{x}[j+1])$ and so $p(i, i+1) \prec p(j, j+1)$, giving $\tau(\boldsymbol{x})[t(i)] \prec \tau(\boldsymbol{x})[t(j)]$ and so $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.

- Case when for some ℓ, $\boldsymbol{x}[i..i+\ell-1] = \boldsymbol{x}[j..j+\ell-1]$ while $\boldsymbol{x}[i+\ell] \prec \boldsymbol{x}[j+\ell]$.

- Case both $i+\ell$ and $j+\ell$ are black.

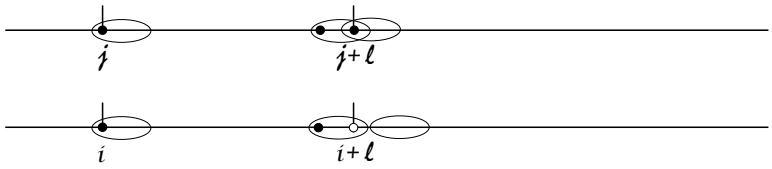


Consider $i+\ell-1$ and $j+\ell-1$. Either they are both black or they are both white.

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

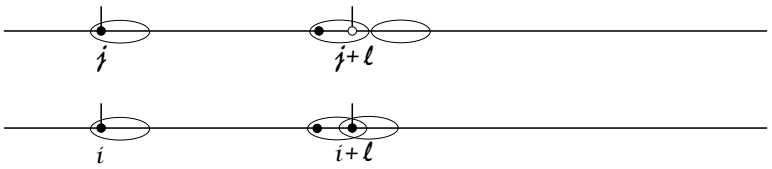
(β) Case both $i+\ell-1$ and $j+\ell-1$ are white, then both $i+\ell-2$ and $j+\ell-2$ are black. So, $\tau(\boldsymbol{x})[t(i)..t(i+\ell-2)] = \tau(\boldsymbol{x})[t(j)..t(j+\ell-2)]$ and so $\tau(\boldsymbol{x})[t(i)..t(i+\ell)] = \tau(\boldsymbol{x})[t(i)..t(i+\ell-2)]\tau(\boldsymbol{x})[i+\ell]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

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



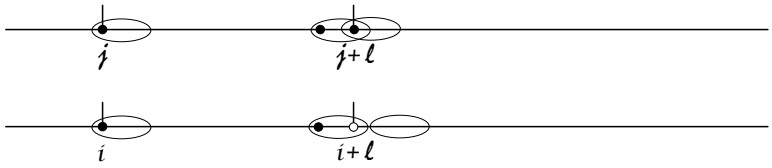
Then both $i+\ell-1$ and $j+\ell-1$ are black, so proceed as in (α).

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



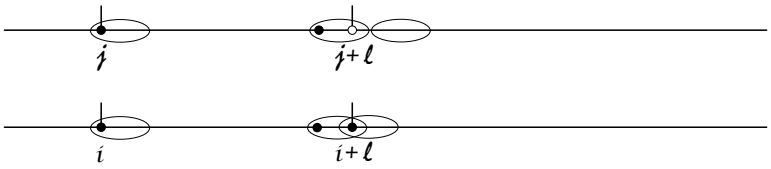
Then both $i+\ell-1$ and $j+\ell-1$ are black, so proceed as in (α).

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



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

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



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

■

Lemma 6.5 shows that τ-reduction preserves the Lyndon property of certain Lyndon substrings.

Lemma 6.5. 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 Lyndon susbtsring of $\boldsymbol{x}$, and let i be a black position.

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

PROOF.
Let us first assume that j is black.

Let $i_1 = t(i)$, $j_1 = t(j)$ and consider k_1 so that $i_1 < k_1 \leq j_1$. Let $k = b(k_1)$. Then $i < k \leq j$ and so $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ by Lemma 2.6. Hence, $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(k)..m]$ by Lemma 6.4. Therefore, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon by Lemma 2.6.

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.

■

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

Lemma 6.6. 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.

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

PROOF. Since $\boldsymbol{x}[i..j]$ is maximal Lyndon, $\boldsymbol{x}[j+1..n] \prec \boldsymbol{x}[i..n]$ by Lemma 2.6, giving $\boldsymbol{x}[j+1] \preceq \boldsymbol{x}[i]$. Since $\boldsymbol{x}[i..j]$ is Lyndon, $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$. Thus, $\boldsymbol{x}[j+1] \preceq \boldsymbol{x}[i] \prec \boldsymbol{x}[j]$.

We will proceed by discussing two possible cases, one that j is black, and the other that j is white.

- Assume that j is black.

  Since j is black, $(j, j+1)$ is a τ-pair and $t(j)$ is defined, and by Lemma 6.5, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon, and hence by Lemma 2.6, $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[k..m]$ 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 when $\boldsymbol{x}[j+1] \preceq \boldsymbol{x}[j+2]$.

    Then $\boldsymbol{x}[j] \succ \boldsymbol{x}[æ+1] \preceq \boldsymbol{x}[j+2]$ and so $j+1$ is black. It follows that $t(j)+1 = t(j+1)$. By Lemma 6.4, $\tau(\boldsymbol{x})[t(j+1)..m] \prec \tau(\boldsymbol{x})[t(i)..m]$ 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 when $\boldsymbol{x}[j+1] \succ \boldsymbol{x}[j]$.

    Then $\boldsymbol{x}[j] \succ \boldsymbol{x}[i] \succeq \boldsymbol{x}[j+1] \succ \boldsymbol{x}[j+2]$, then $\tau(\boldsymbol{x})[t(i)] = p(i, i+1)$, and $\tau(\boldsymbol{x})[t(j)] = p(j, j+1)$, and $\tau(\boldsymbol{x})[t(j)+1] = p(j+2, j+3)$. It follows that $\tau(\boldsymbol{x})[t(j)] = p(j, j+1) \succ \tau(\boldsymbol{x})[t(i)] = p(i, i+1) \succ \tau(\boldsymbol{x})[t(j)+1] = p(j+2, i+3)$. Thus, $\tau(\boldsymbol{x})[t(j)+1] \prec \tau(\boldsymbol{x})[t(i)]$, and so $\tau(\boldsymbol{x})[t(j)+1..m] \prec \tau(\boldsymbol{x})[t(i)..m]$.

- Assume that j is white.

  By Lemma 6.5, $\tau(\boldsymbol{x})[t(i)..t(j-1)]$ 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 6.5 we get $\tau(\boldsymbol{x})[t(i)..m] \succ \tau(\boldsymbol{x})[t(j+1)..m] = \tau(\boldsymbol{x})[t(j-1)+1..m]$.

■

At this point, presentation of the proof Theorem 6.3 is possible.

Proof of Theorem 6.3. Let $\mathcal{L}'_{\boldsymbol{x}}[i] = j$ where i is black. Then $t(i)$ is defined and $\boldsymbol{x}[i..j]$ is a maximal Lyndon substring of $\boldsymbol{x}$.

- Case when j is black.

  Then by Lemma 6.6, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is a maximal Lyndon substring of $\tau(\boldsymbol{x})$, hence $\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)] = t(j)$. Therefore, $b(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]) = b(t(j)) = j = \mathcal{L}'_{\boldsymbol{x}}[i]$. Since $\boldsymbol{x}[i..j]$ is maximal, $\boldsymbol{x}[j+1] \preceq \boldsymbol{x}[i]$, i.e. $\boldsymbol{x}[b(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)])+1] = \boldsymbol{x}[j+1] \preceq \boldsymbol{x}[i]$.

- Case when j is white.

  Then $j-1$ is black and the $\tau(\boldsymbol{x})[t(j-1)] = p(j-1, j)$. By Lemma 6.6, $\tau(\boldsymbol{x})[t(i)..t(j-1)]$ is a maximal Lyndon substring of $\tau(\boldsymbol{x})$, hence $\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)] = t(j-1)$, so $b(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]) = b(t(j-1)) = j-1$, giving $b(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]+1) = j$. Since $\boldsymbol{x}[i..j]$ is maximal, $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$, i.e. $\boldsymbol{x}[b(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]+1)] = \boldsymbol{x}[j] \succ \boldsymbol{x}[i]$.

■

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

Theorem 6.3 indicates how to compute the partial $\mathcal{L}'_{\boldsymbol{x}}$ from $\mathcal{L}'_{\tau(\boldsymbol{x})}$. The procedure is given in Fig. 6.2.

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. Further, $\mathcal{L}'_{\boldsymbol{x}}[i+1]$ is the final

```
for i ← 1 to n
  if i = 1 or (x[i1] ≻ 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 6.2: Computing partial Lyndon array of the input string

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, hence the maximal Lyndon substring at the position i has length 1 and so ends at position i. Otherwise, $\boldsymbol{x}[i..\mathcal{L}'_{\boldsymbol{x}}[i+1]]$ is Lyndon, and we must 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[1] of the maximal Lyndon substrings.

This is the **while** loop in the procedure given in Fig. 6.3 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 6.5.

Consider the running example from Fig. 6.1. 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 $\mathcal{L}'_{\boldsymbol{x}}[8]$ 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, $\mathcal{L}'_{\boldsymbol{x}}[1..9] = 9,2,3,9,6,6,9,8,9$.

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

```
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'[i1] = i1 then
        stop ← n
      else
        stop ← L'[i1]
      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
```

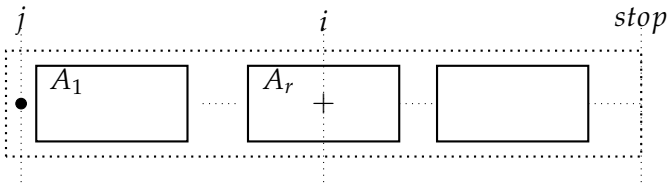
Figure 6.3: Computing missing values of the Lyndon array of the input string

6.5 The Complexity of TRLA

To determine the complexity of the algorithm, we attach to each position i a counter $red[i]$ initialized to 0. When computing a missing value $\mathcal{L}'_{\boldsymbol{x}}[j]$ with a configuration shown below where

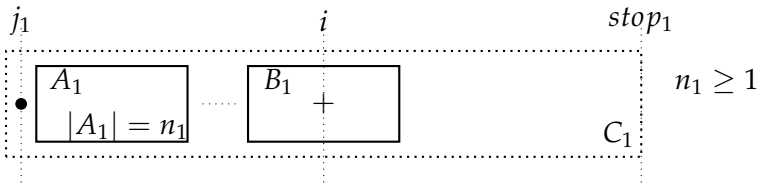
$$stop = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j-1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j-1] > j-1 \\ n & otherwise \end{cases},$$

we have to check if A_1 can be extended by j to a Lyndon substring (i.e. if $\boldsymbol{x}[j] \preceq \boldsymbol{x}[j+1]$), if so, we have to compare jA_1 with A_2 and if jA_1A_2 is Lyndon (i.e. if $jA_1 \prec A_2$), we must check if $jA_1A_2A_3$ is Lyndon (i.e. if $jA_1A_2 \prec A_3$) ... checking if $jA_1..A_r$ is Lyndon (i.e. if $jA_1A_2..A_{r-1} \prec A_r$) , etc. ... When comparing the Lyndon substring $jA_1..A_{r-1}$ with A_r, at every position i of A_r, we increment the counter $red[i]$. When done, the value of $red[i]$ represents how many times the position i was used in comparisons.



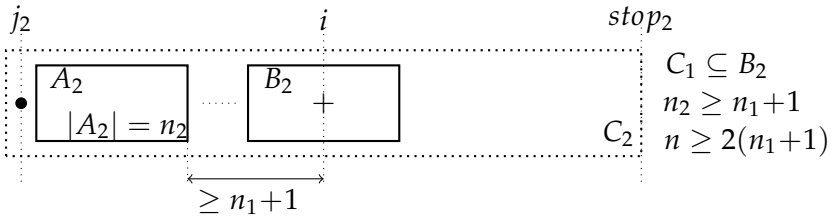
Consider a position i that was used k times for $k \geq 4$, i.e. $red[i] = k$. The next diagram indicates the configuration when the counter $red[i]$ was incremented for the 1st time in the comparison of $j_1A1...$ and B_1 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_1]$ where

$$stop_1 = \begin{cases} \mathcal{L}'_{\boldsymbol{x}}[j_1-1] & if\ \mathcal{L}'_{\boldsymbol{x}}[j_1-1] > j_1-1 \\ n & otherwise \end{cases}.$$



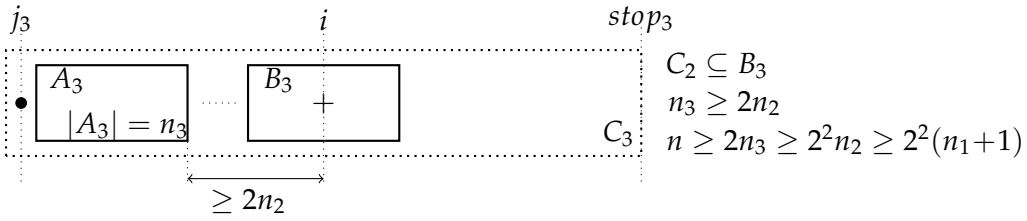
The next diagram indicates the configuration when the counter $red[i]$ was incremented for the second time in the comparison of $j_2A2...$ and B_2 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_2]$ where

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



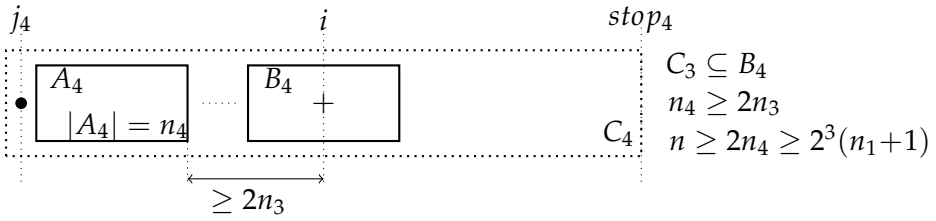
The next diagram indicates the configuration when the counter $red[i]$ was incremented for the third time in the comparison of $j_3A3...$ and B_3 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_3]$ where

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



The next diagram indicates the configuration when the counter $red[i]$ was incremented for the fourth time in the comparison of $j_4A4...$ and B_4 during the computation of the missing value $\mathcal{L}'_{\boldsymbol{x}}[j_4]$ where

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



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

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

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

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

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

6.6 Implementation Notes

The *TRLA* algorithm was implemented in `C++` and made publicly available [1]. The `C++` code for *TRLA* is in the source file `trla.cpp`, utilizing `Tau.hpp` which contains the class `Tau`. The class provides the code for the computation of the τ-reduction and three auxiliary arrays that after the computation of the τ-reduction is completed store the representation of the mappings b and t mapping black positions of the input string to the positions of the τ-reduction and vice-versa. These mappings are required for the computation of the partial Lyndon array of the input string.

The `Tau` method `translate` computes the τ-reduction. The code in `C++` realizes the recursive call to *TRLA* with the τ-reduction, computation of the partial Lyndon array of the input string, and the procedure `FillGaps` computes the missing values.

The space complexity of this C++ implementation is bounded by $9n$ integers. This upper bound is derived from the fact that a `Tau` object (see `Tau.hpp`, [1]) 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, $3n + 3\frac{2}{3}n + 3(\frac{2}{3})^2 n + 3(\frac{2}{3})^3 n + ... = 3n(1 + \frac{2}{3} + (\frac{2}{3})^2 + (\frac{2}{3})^3 + (\frac{2}{3})^4 + ...) = 3n\frac{1}{1-\frac{2}{3}} = 9n$. However, it should be possible to bring it down to $6n$ integers.

7 Empirical Testing and Results

All the measurements were performed on ***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[1]. This was strategically done as to not bias the programs in any way, as the optimization may have favoured one program more than another. The CPU time was measured for each of the programs in seconds. 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 $\{0,1,...,25\}$, 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.

The data is randomly generated once and stored. It should be noted that the data is checked for tightness as part of the generation process. Further, this data is stored and used for the computation of all of the algorithms. This is important as we do not want to use new random data each

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

time, rather the same random data on the different algorithms. Comparisons against random tight 4-ary strings versus random DNA sequence generators, as well as, random tight 26-ary strings versus random bible text or random novel text generators were next to indistinguishable.

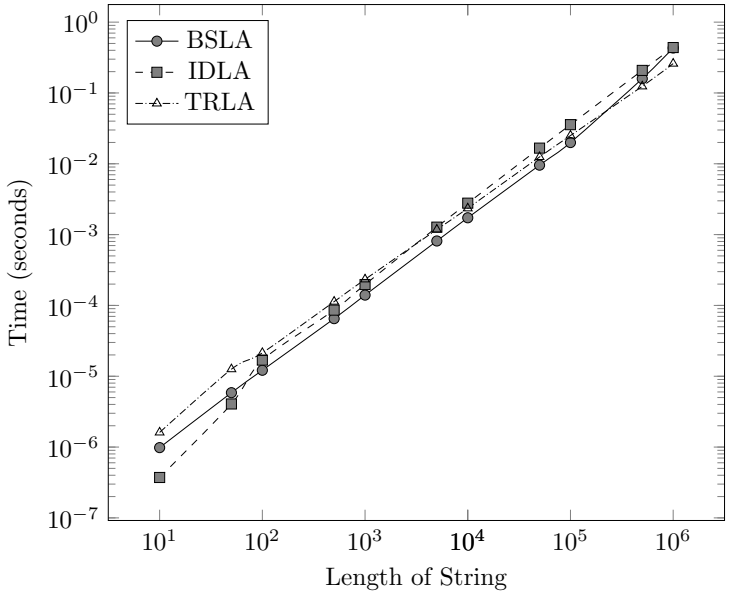


Figure 1. Average times of *IDLA*, *BSLA*, and *TRLA* on random binary datasets.

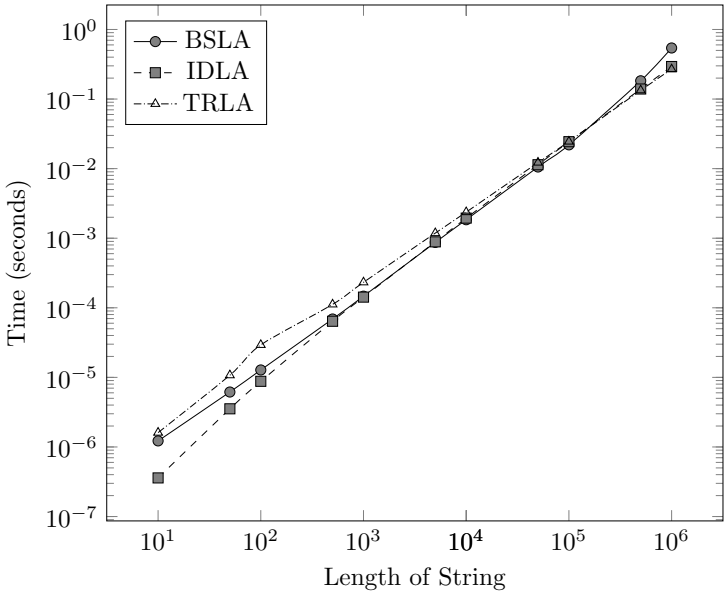


Figure 2. Average times of *IDLA*, *BSLA*, and *TRLA* on random 4-ary datasets.

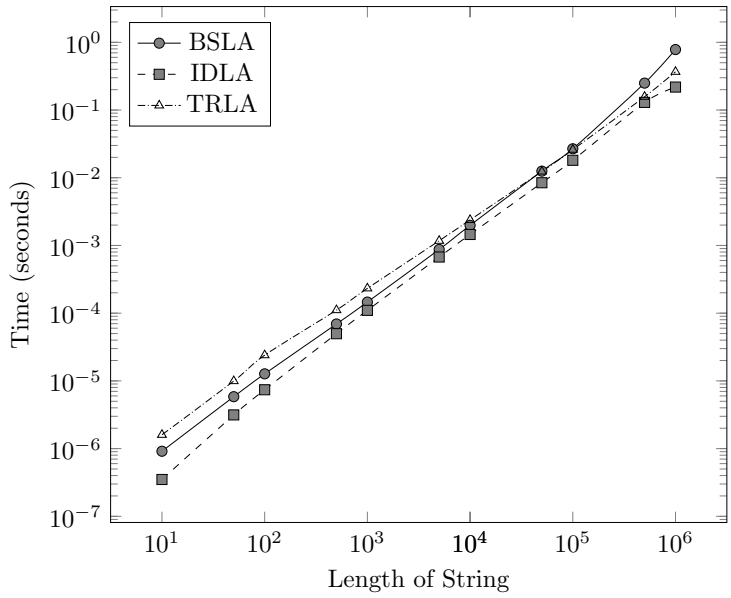


Figure 3. Average times of *IDLA*, *BSLA*, and *TRLA* on random 26-ary datasets.

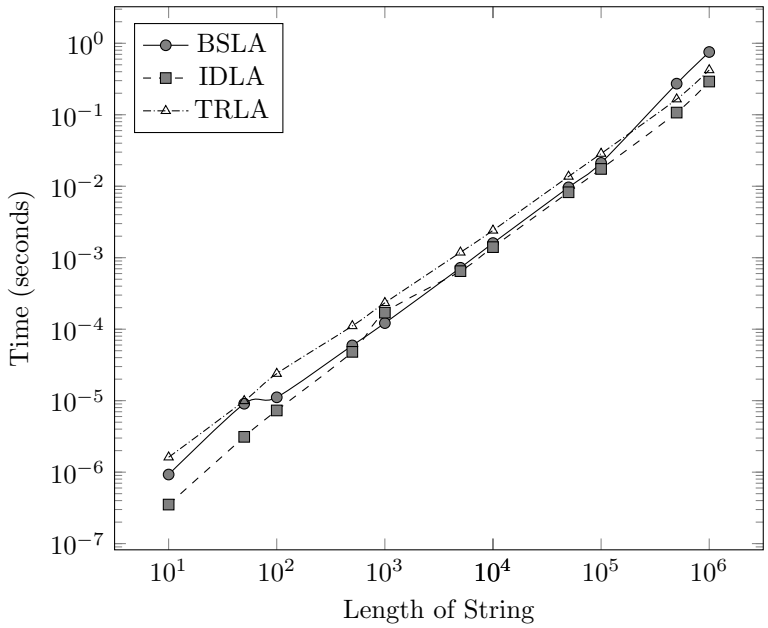


Figure 4. Average times of *IDLA*, *BSLA*, and *TRLA* on random integer datasets.

As the graphs clearly indicate, the performance of the three algorithms is virtually indistinguishable. It was expected that *IDLA* and *TRLA* would exhibit linear behaviour on random strings; as such strings tend to have almost all maximal Lyndon substrings short with respect to the length of

the strings. However, from the empirical results, it was not expected to be so close. Further, the following graphs were produced to identify potential outliers or trends in the data/computation, but are clearly inconsequential.

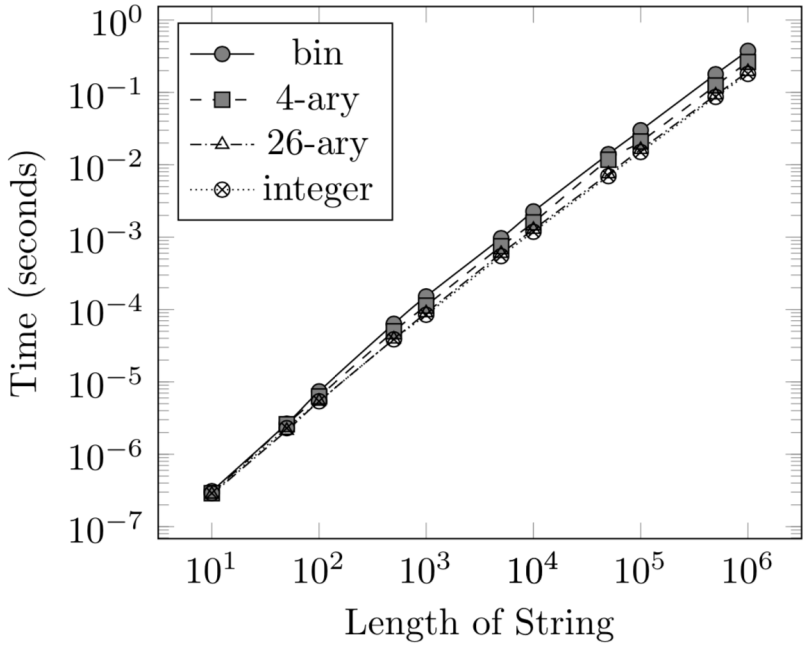


Figure 5. IDLA minimum

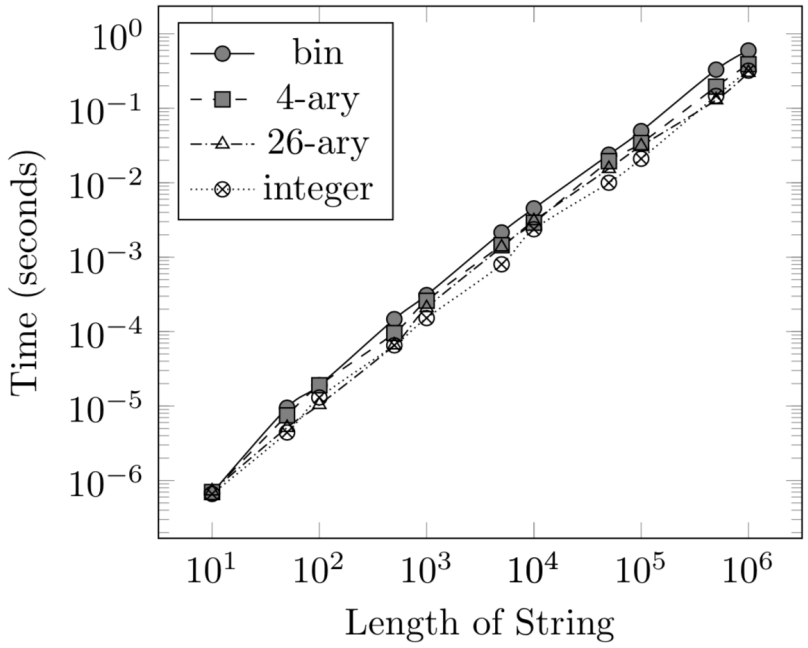


Figure 6. IDLA maximum

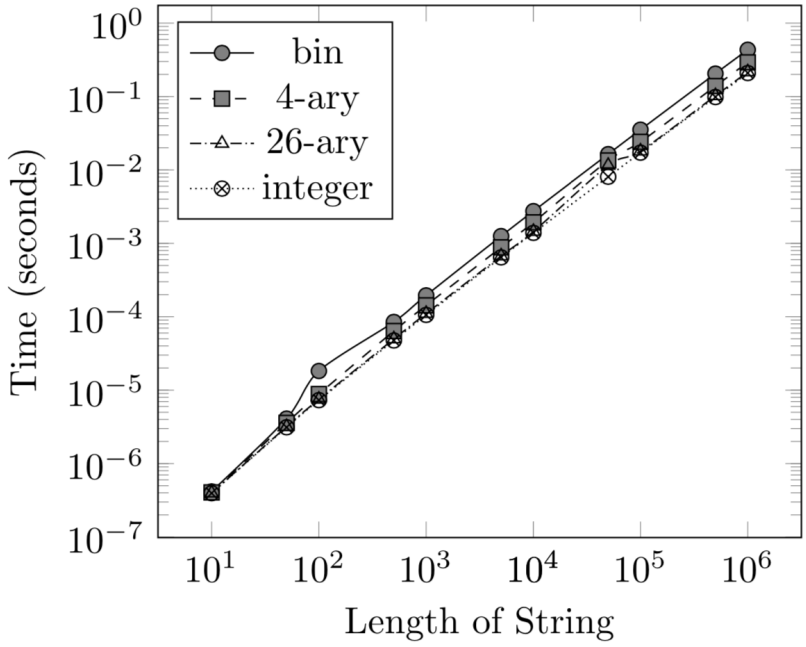


Figure 7. IDLA average

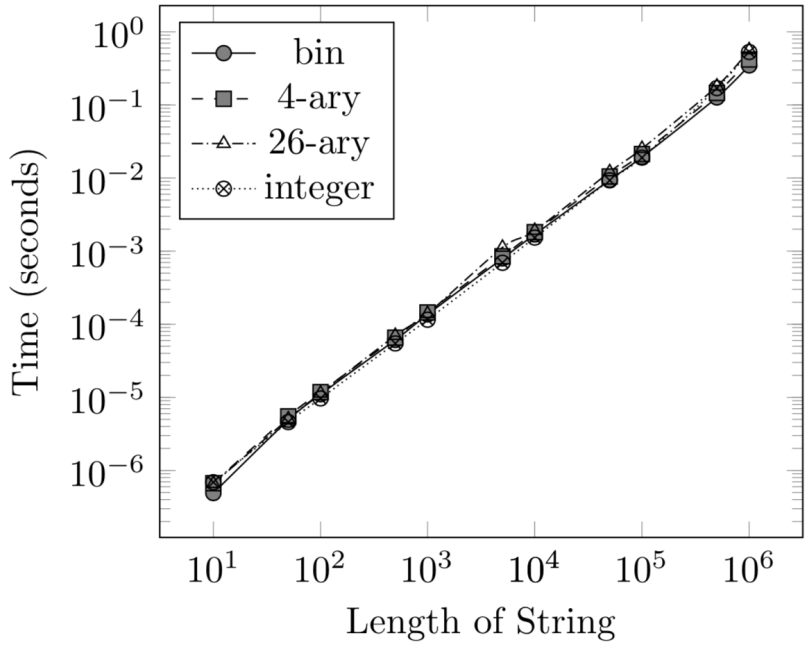


Figure 8. BSLA minimum

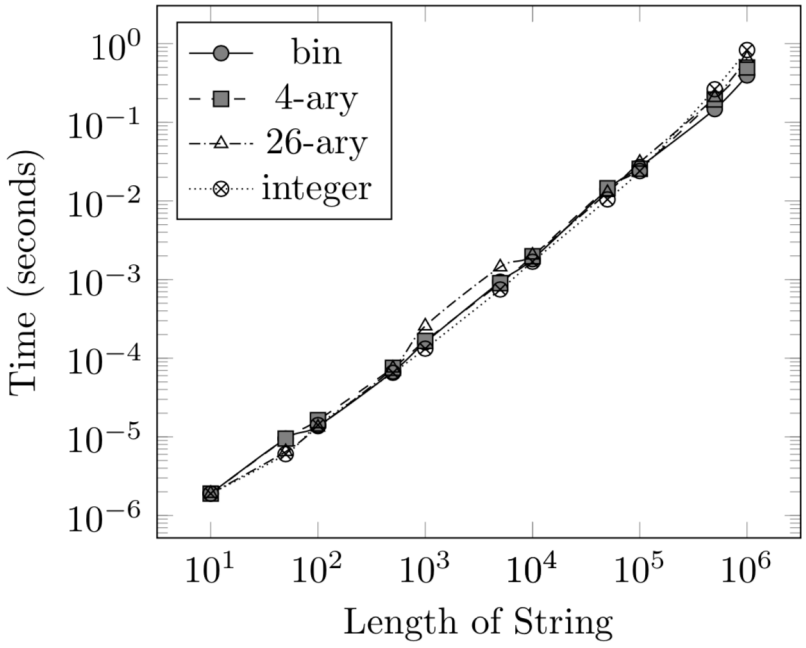


Figure 9. BSLA maximum

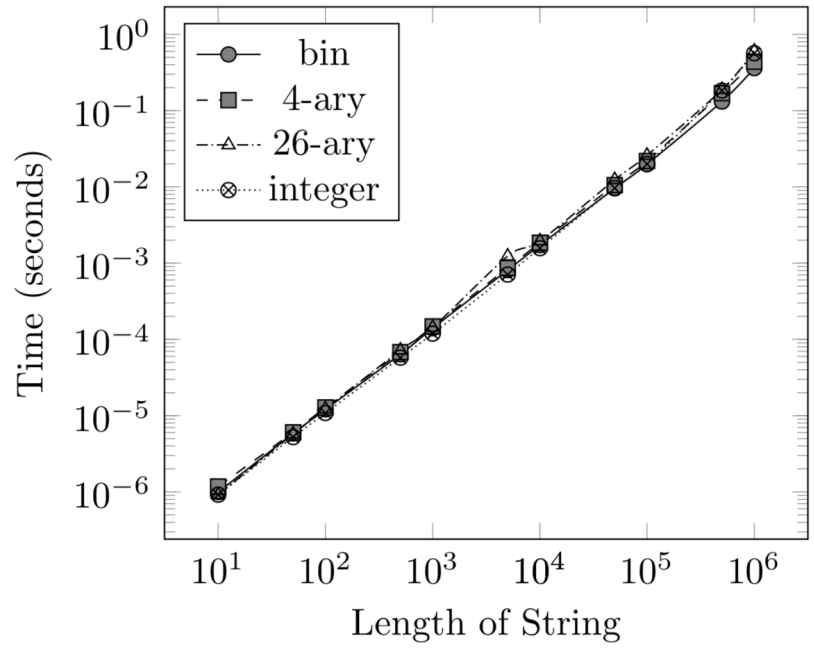


Figure 10. BSLA average

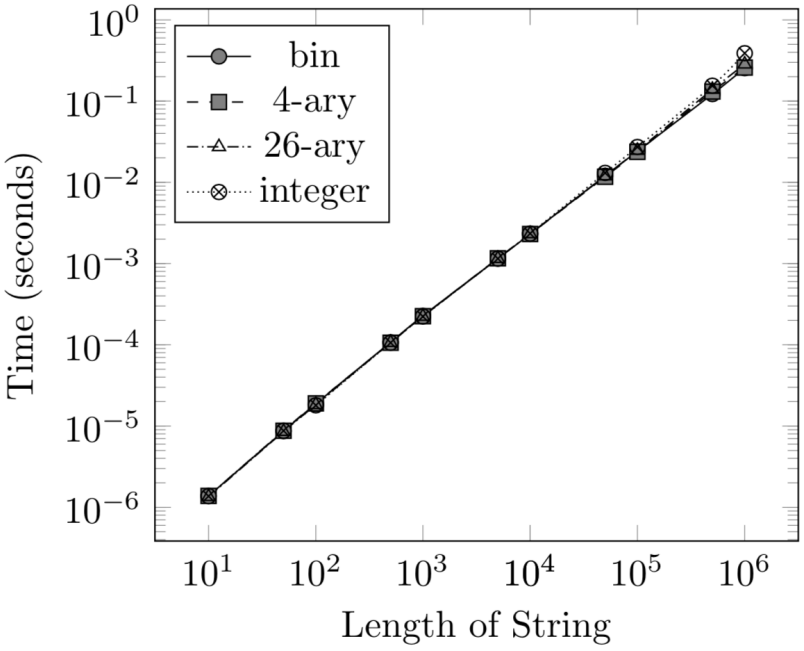


Figure 11. TRLA minimum

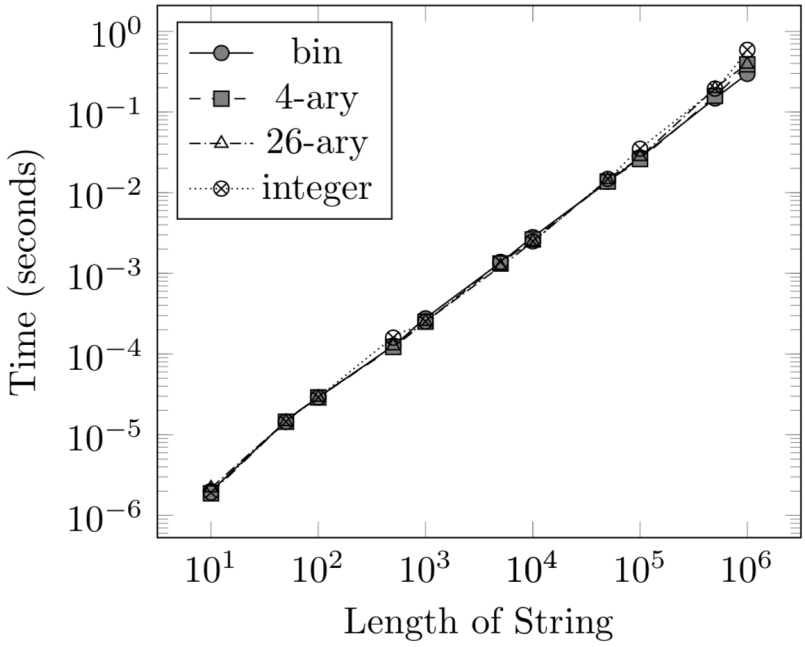


Figure 12. TRLA maximum

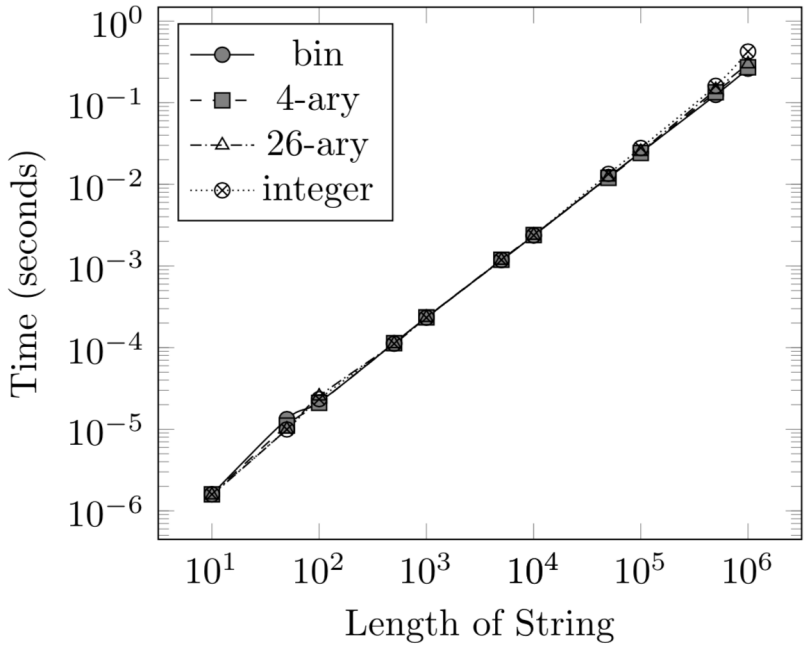


Figure 13. TRLA average

Figs. 5, 8, 11, all illustrate the minimum time for each dataset for BSLA, IDLA, and TRLA respectively. Whereas Figs. 6, 9, 12, depict the maximum time for each datasets, while Figs. 7, 10, 13 depict the average time for each dataset. It should be noted that performance was a measurement of computational time and given that a single string could be a potential outlier/single point of failure in the empirical analysis, the average of 500 strings were computed (see Figs. 7, 10, 13). However, it is reassuring to see that there doesn't appear to be any abnormal data points (see Table 7.1, Table 7.2, Table 7.3, and Table 7.4 for easy comparison of the averages).

Further, all three algorithms were tested on datasets containing a single string 01234...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. 14. The *extreme trla* strings were generated according to the scheme (F) in Section 6.5. These strings force the 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. 15. Additionally, Table 7.5 and Table 7.6 have been provided for easy readability of the data.

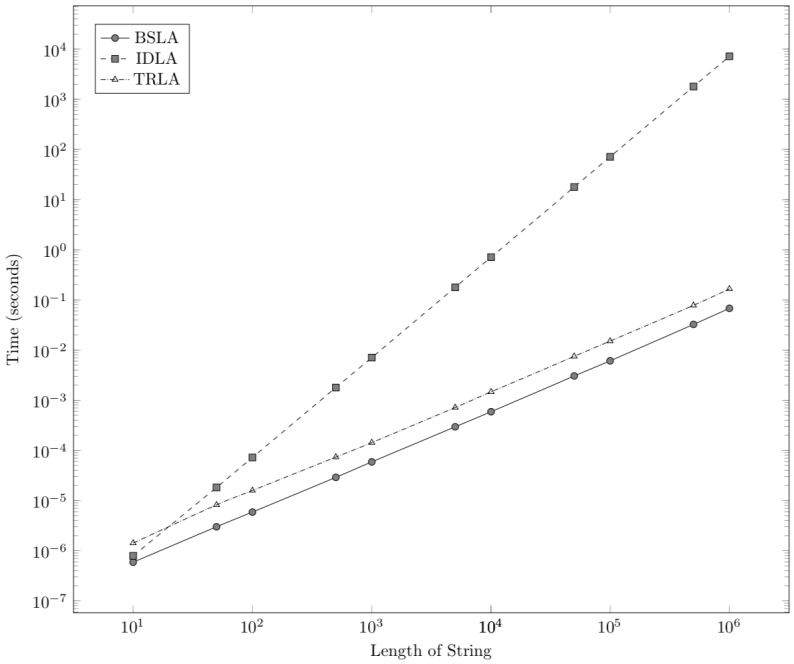


Figure 14. Average times of *IDLA*, *BSLA*, and *TRLA* on extreme *IDLA* strings

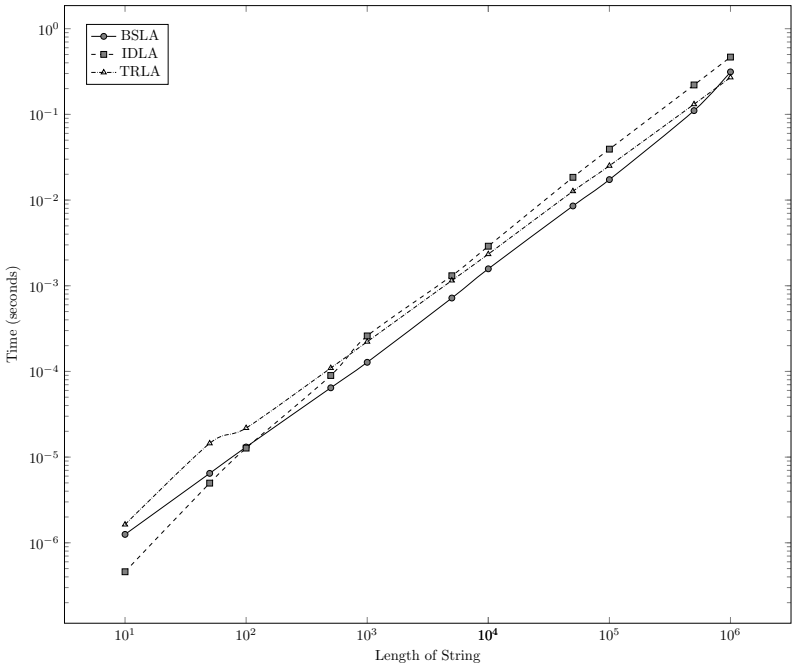


Figure 15. Average times of *IDLA*, *BSLA*, and *TRLA* on extreme *TRLA* strings

Length	Random binary strings		
	IDLA	*BSLA*	*TRLA*
10	4.18×10^{-7}	9.93×10^{-7}	1.59×10^{-6}
50	4.10×10^{-6}	5.84×10^{-6}	1.33×10^{-5}
100	1.83×10^{-5}	1.21×10^{-5}	2.14×10^{-5}
500	8.51×10^{-5}	6.37×10^{-5}	1.14×10^{-4}
1,000	1.96×10^{-4}	1.40×10^{-4}	2.34×10^{-4}
5,000	1.26×10^{-3}	8.09×10^{-4}	1.18×10^{-3}
10,000	2.75×10^{-3}	1.72×10^{-3}	2.36×10^{-3}
50,000	1.64×10^{-2}	9.56×10^{-3}	1.23×10^{-2}
100,000	3.53×10^{-2}	1.98×10^{-2}	2.47×10^{-2}
500,000	2.05×10^{-1}	1.31×10^{-1}	1.25×10^{-1}
1,000,000	4.34×10^{-1}	3.61×10^{-1}	2.61×10^{-1}

measurements are in seconds

Table 7.1: Average times of *IDLA*, *BSLA*, and *TRLA* on random binary datasets.

Length	Random 4-ary strings		
	IDLA	*BSLA*	*TRLA*
10	4.05×10^{-7}	1.18×10^{-6}	1.59×10^{-6}
50	3.58×10^{-6}	6.05×10^{-6}	1.11×10^{-5}
100	8.85×10^{-6}	1.28×10^{-5}	2.10×10^{-5}
500	6.39×10^{-5}	6.81×10^{-5}	1.13×10^{-4}
1,000	1.43×10^{-4}	1.48×10^{-4}	2.35×10^{-4}
5,000	8.83×10^{-4}	8.65×10^{-4}	1.19×10^{-3}
10,000	1.93×10^{-3}	1.85×10^{-3}	2.39×10^{-3}
50,000	1.35×10^{-2}	1.05×10^{-2}	1.20×10^{-2}
100,000	2.41×10^{-2}	2.18×10^{-2}	2.42×10^{-2}
500,000	1.39×10^{-1}	1.69×10^{-1}	1.34×10^{-1}
1,000,000	2.92×10^{-1}	4.38×10^{-1}	2.72×10^{-1}

measurements are in seconds

Table 7.2: Average times of *IDLA*, *BSLA*, and *TRLA* on random 4-ary datasets.

Length	Random 26-ary strings		
	IDLA	*BSLA*	*TRLA*
10	3.97×10^{-7}	9.17×10^{-7}	1.56×10^{-6}
50	3.18×10^{-6}	5.74×10^{-6}	9.99×10^{-6}
100	7.51×10^{-6}	1.22×10^{-5}	2.54×10^{-5}
500	4.98×10^{-5}	7.21×10^{-5}	1.11×10^{-3}
$1,000$	1.11×10^{-4}	1.43×10^{-4}	2.33×10^{-4}
$5,000$	6.73×10^{-4}	1.23×10^{-3}	1.17×10^{-3}
$10,000$	1.45×10^{-3}	1.95×10^{-3}	2.37×10^{-3}
$50,000$	1.15×10^{-3}	1.24×10^{-2}	1.22×10^{-2}
$100,000$	1.81×10^{-2}	2.58×10^{-2}	2.51×10^{-2}
$500,000$	1.02×10^{-1}	1.88×10^{-1}	1.44×10^{-1}
$1,000,000$	2.16×10^{-1}	6.00×10^{-1}	2.99×10^{-1}

measurements are in seconds

Table 7.3: Average times of *IDLA*, *BSLA*, and *TRLA* on random 26-ary datasets.

Length	Random integer strings		
	IDLA	*BSLA*	*TRLA*
10	3.95×10^{-7}	9.21×10^{-7}	1.59×10^{-6}
50	3.12×10^{-6}	5.24×10^{-6}	9.95×10^{-6}
100	7.31×10^{-6}	1.08×10^{-5}	2.34×10^{-5}
500	4.75×10^{-5}	5.76×10^{-5}	1.12×10^{-4}
$1,000$	1.06×10^{-4}	1.19×10^{-4}	2.33×10^{-4}
$5,000$	6.44×10^{-4}	7.08×10^{-4}	1.19×10^{-3}
$10,000$	1.39×10^{-3}	1.57×10^{-3}	2.37×10^{-3}
$50,000$	8.07×10^{-3}	9.63×10^{-3}	1.34×10^{-2}
$100,000$	1.72×10^{-2}	2.00×10^{-2}	2.79×10^{-2}
$500,000$	$9.8\bar{3} \times 10^{-2}$	$1.8\bar{3} \times 10^{-1}$	1.61×10^{-1}
$1,000,000$	2.09×10^{-1}	5.65×10^{-1}	4.24×10^{-1}

measurements are in seconds

Table 7.4: Average times of *IDLA*, *BSLA*, and *TRLA* on random integer datasets.

Length	*IDLA* extreme strings		
	IDLA	*BSLA*	*TRLA*
10	7.90×10^{-7}	5.90×10^{-7}	1.42×10^{-6}
50	1.83×10^{-5}	3.00×10^{-6}	8.20×10^{-6}
100	7.22×10^{-5}	5.81×10^{-6}	1.59×10^{-5}
500	1.79×10^{-3}	2.81×10^{-5}	7.35×10^{-5}
1,000	7.08×10^{-3}	5.90×10^{-5}	1.43×10^{-4}
5,000	1.78×10^{-1}	2.96×10^{-4}	7.15×10^{-4}
10,000	7.12×10^{-1}	5.81×10^{-4}	1.47×10^{-3}
50,000	1.78×10^{1}	3.05×10^{-3}	7.50×10^{-3}
100,000	7.12×10^{1}	$6.1\bar{9} \times 10^{-3}$	1.51×10^{-2}
500,000	1.79×10^{3}	3.25×10^{-2}	7.8×10^{-2}
1,000,000	7.18×10^{3}	6.80×10^{-2}	1.66×10^{-1}

measurements are in seconds

Table 7.5: Average times of *IDLA*, *BSLA*, and *TRLA* on *IDLA* extreme strings.

Length	*TRLA* extreme strings		
	IDLA	*BSLA*	*TRLA*
10	4.59×10^{-7}	1.25×10^{-6}	1.63×10^{-6}
50	4.97×10^{-6}	6.46×10^{-6}	1.45×10^{-5}
100	1.27×10^{-5}	1.31×10^{-5}	2.18×10^{-5}
500	8.97×10^{-5}	6.44×10^{-5}	1.10×10^{-4}
1,000	2.60×10^{-4}	1.28×10^{-4}	2.22×10^{-4}
5,000	1.31×10^{-3}	7.19×10^{-4}	1.15×10^{-3}
10,000	2.90×10^{-3}	1.58×10^{-3}	2.34×10^{-3}
50,000	1.84×10^{-2}	8.54×10^{-3}	1.27×10^{-2}
100,000	3.93×10^{-2}	1.73×10^{-2}	2.51×10^{-2}
500,000	2.21×10^{-1}	1.11×10^{-1}	1.31×10^{-1}
1,000,000	4.67×10^{-1}	3.13×10^{-1}	2.70×10^{-1}

measurements are in seconds

Table 7.6: Average times of *IDLA*, *BSLA*, and *TRLA* on *TRLA* extreme strings.

8 Conclusion

Given that computing maximal Lyndon substrings is one of two ways to compute all the runs of a string in linear time, it is only reasonable to assume that improving on the efficiency of such algorithms is important and relevant. Further, as there were only two "standard" algorithms for computing the Lyndon array, it is essential to use them as a benchmark against any future advances within this field.

This thesis has analyzed, formally proven correct, implemented, and demonstrated the performance of two novel algorithms: *BSLA* and *TRLA*. Specifically, these algorithms have been developed both theoretically and programatically for empirical analysis and comparison against *IDLA*; one of the two standard algorithms for computing maximal Lyndon substrings of a string.

The first of the algorithms designed in the course of this thesis research, *BSLA* is an important advancement in the field, as the algorithm is both linear and elementary. By that it is meant that it does not require any pre-processing of a global data structure. For example, *SSLA* or *BWLA* are currently the only other linear algorithms to compute the Lyndon array, but they both have to first compute the suffix array which, in a sense, is completely unrelated to the task. *BSLA* does not need to do that.

The second of the algorithms designed in the course of this thesis research is *TRLA*. It has the worst-case complexity of $\mathcal{O}(n\ log(n))$ for an input string of length n. Despite this, the algorithm is of interest to the stringology community for two reasons: it is quite likely that an improvement of the computation of the missing values can lead to an overall linear complexity, and the τ-reduction developed for the algorithm can be of interest elsewhere; for instance it could be used for a different approach for linear suffix sorting.

Empirical tests performed revealed that both *BSLA* and *TRLA* are good algorithmic alternatives to *IDLA* as generally they both outperformed it; for a particular class of strings significantly so. However, an unexpected surprise was how negligible the differences were for random strings. As a result of the testing it seems that it tells us more about random strings than the three algorithms. Random strings tend to have a lot of very short maximal Lyndon substrings, and that is what makes all three algorithm perform very similarly. Testing using more heavily biased strings is needed. To truly map the performance, both *BSLA* and *TRLA* need to be compared to the fastest implementation of *SSLA*.

8.1 Future Work

There are several avenues which can be pursued in this area. Some of which are:

1. Test the performance of *BSLA*, and *TRLA* against a suffix sorting algorithm with next-smaller-value (*SSLA*) and on an algorithmic scheme based on Burrows-Wheeler transform (*BWLA*).

2. Formally analyze, and prove correct, an algorithm based on ranges (*RGLA*). Furthermore, investigate possible connections to deBruijn sequences.

3. Test *TRLA* against larger 'extreme' strings to exhibit a more distinguishable worst-case upon empirical comparison with BLSA and *IDLA*.

4. Optimize and reduce the space requirement of current *BSLA* implementation.

5. Optimize and reduce the space requirement of current *TRLA* implementation.

6. Investigate the claim [21] that computing Lempel-Ziv factorization may be harder than computing all runs for general alphabets in a framework of working with general alphabets as opposed to the current state of affairs when only constant and integer alphabets are considered.

Bibliography

[1] C++ code for IDLA, BSLA, and TRLA algorithms. Available at: https://github.com/MichaelLiut/Computing-LyndonArray.

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

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

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

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

[6] J. Berstel and D. Perrin. The origins of combinatorics on words. European Journal of Combinatorics, 28(3):996 – 1022, 2007.

[7] W. Bland, G. Kucherov, and W.F. Smyth. Prefix table construction and conversion. In Thierry Lecroq and Laurent Mouchard, editors, Combinatorial Algorithms, pages 41–53, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

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

[9] M. Christodoulakis, P.J. Ryan, W. F. Smyth, and S. Wang. Indeterminate strings, prefix arrays and undirected graphs. Theoretical Comput. Sci., 600:34–48, 2015.

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

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

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

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

[14] F. Franek and M. Liut. Algorithms to compute the Lyndon array revisited. In Proc. of Prague Stringology Conference 2019, pages 16–28, 2019.

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

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

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

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

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

[20] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. FOCS40, pages 596–604, 1999.

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

[22] M. Lothaire. Combinatorics on words. Cambridge University Press, 2003.

[23] M. Lothaire. Applied Combinatorics on Words. Cambridge University Press, 2005.

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

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

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

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

[28] B. Smyth. Computing patterns in strings. Pearson Addison-Wesley, 2003.