LYNDON FACTORS AND PERIODICITIES IN STRINGS

Asma Paracha

LYNDON FACTORS AND PERIODICITIES IN STRINGS

By

ASMA PARACHA

A Thesis Submitted to the Department of Computing and Software and the School of Graduate Studies of McMaster University in Partial Fulfilment of the Requierement for the Degree of

Doctor of Philosophy

Ph.D. Thesis — McMaster University

Department of Computing and Software — Hamilton, Ontario, Canada

TITLE:	Lyndon Factors and Periodicities in Strings
AUTHOR:	Asma Paracha
	M.A.Sc. McMaster University
	M.Sc. Sir Syed University of Engineering and Technology
	B.E. NED University of Engineering and Technology
SUPERVISOR:	Dr. Frantisek Franek
NUMBER OF PAGES:	VI, 90, IX

Abstract

Strings are very simple yet very applicable data structures. Their applicability ranges from modelling DNA, to modelling protein sequences, to information retrieval, to web page searches, and many more. Due to their simplicity, there are few structural properties that could be exploited for analysis of string algorithms and their auxiliary data structures. Thus, from the beginning, researchers paid utmost attention to periodic properties of strings, such as runs which are maximal fractional periodicities.Though first conjectured in 1999 by Kolpakov and Kucherov, the runs conjecture that there are fewer runs than the length of the string was only settled in 2015 by Bannai et al. via specific Lyndon roots referred to as L-roots. This method allows mapping of runs to the starting points of its L-roots that form mutually disjoint subsets of the indices of the string. This relationship between runs and maximal Lyndon factors (substrings) of a string is not coincidental, as Bannai et al. used the knowledge of all maximal Lyndon factors with respect to an order and its inverse to compute all runs in linear time. Thus, computing the all maximal Lyndon factors efficiently becomes of importance.

In this thesis, we review the fundamental properties of Lyndon strings, including the famous Lyndon factorization and its linear solution due to Duval. In addition to that we explore a new and conceptually simple data structure called Lyndon array and its relationship to the suffix array. Finally, we discuss 2015 Baier's algorithm for sorting suffixes that identifies and sorts in phase 2 the maximal Lyndon factors in $\mathcal{O}(n \log(|\Sigma|))$ steps for a string of length n over an alphabet Σ. We examine the fact that Baier's algorithm sorts the suffixes by sorting the maximal Lyndon factors, and present a different, potentially faster algorithm for phase 2. Our goal was to gather all the relevant well known and some unpublished facts about Lyndon strings and their relationship to runs. In addition we present a novel $\mathcal{O}(n\,log(n))$ recursive algorithm for computing Lyndon arrays that may be competitive with Baier's for strings with large alphabets.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Frantisek Franek, for his invaluable guidance, generous support and continuous encouragement to my research studies and my life.

My special thanks go to the other members of the supervisory committees: Dr. Antoine Deza and Dr. Ryzsard Janicki for their support and guidance throughout my Ph.D. studies.

To my family and friends.

Table of Contents

Chapter 1

Introduction

1.1 Maximum-number-of-runs problem

Repetitions, or *tandem repeats*, in strings is one of the most basic and well studied characteristics of strings, with various theoretical and practical applications. In *combinatorics on words*, the study of strings began with an investigation of periodic properties of strings and periodicity of various kinds is still an intensive research focus in several application areas such as data compression, pattern matching, and computational biology. One of intensely studied problem concerns the maximum number of runs.

A notion of ***run*** succinctly captures the idea of a maximal fractional repetition. A ***repetition*** in $\boldsymbol{x}$ starting at position i is a substring $\boldsymbol{u}^r = \boldsymbol{x}[i..i + r|\boldsymbol{u}| - 1]$, $r \geq 2$, where $\boldsymbol{x}[j] = \boldsymbol{x}[j + k|\boldsymbol{u}|]$ for every $k \in 0..r - 1$ and every $j \in i..|\boldsymbol{u}| - 1$. We call $\boldsymbol{u}$ the ***generator*** or ***root***, $|\boldsymbol{u}|$ the ***period*** of

the repetition, and r the ***exponent***. We refer to a repetition where $r = 2$ as a ***square***, and a repetition where $r = 3$ as a ***cube***. Such a repetition can be represented by an integer tuple (start, end, period), i.e. $(i, i+r|\boldsymbol{u}|-1, |\boldsymbol{u}|)$. A ***maximal repetition*** is a repetition that can neither be extended left nor right, i.e. a repetition $\boldsymbol{u}^r = \boldsymbol{x}[i..i+r|\boldsymbol{u}|-1]$ such that neither $\boldsymbol{x}[i-|\boldsymbol{u}|..r|\boldsymbol{u}|-1]$ nor $\boldsymbol{x}[i..i + (r + 1)|\boldsymbol{u}| - 1]$ is a repetition $\boldsymbol{u}^{r+1}$.

Main [1] showed that we need to compute ***runs***, i.e. ***maximal fractional periodicities*** in order to capture all the repetitions in a string $\boldsymbol{x}$. A **run** $\boldsymbol{u}$ in a string $\boldsymbol{x}[1..n]$ is a substring $\boldsymbol{u}^r\boldsymbol{t} = \boldsymbol{x}[i..i + r|\boldsymbol{u}| + r - 1]$, where $\boldsymbol{u}^r$ is a repetition, $\boldsymbol{t}$ is a proper prefix of $\boldsymbol{u}$, and no repetition of period $|\boldsymbol{u}|$ begins at position $i - 1$ of $\boldsymbol{x}$ or ends at position $i + r|\boldsymbol{u}| + |\boldsymbol{t}|$. Moreover, it is required that $\boldsymbol{u}$ is primitive. The string $\boldsymbol{u}$ is called the *generator* or the *root* of the run, $\boldsymbol{t}$ is the tail; such a run can be generally represented by a 3-tuple (start, end, period), i.e. $(i, i + r|\boldsymbol{u}| + |\boldsymbol{t}|, |\boldsymbol{u}|)$. For example, in the string *baaba aba abb*, the underlined run is encoded by $(3, 10, 3)$, and its root *aba* is repeated twice, with the last repeat being incomplete (*ab* only).

The critical observation that a run encapsulates $\boldsymbol{t}$ adjacent maximal repetitions with the same period implies that there are at most as many runs as repetitions. Further, by computing all runs we are implicitly computing all repetitions [2].

Crochemore [3] showed in 1981 that the order of the number of maximal repetitions in a string of length n is $\Theta(n \log(n))$. In 1999, Kolpakov and Kucherov [4] showed that the order of the maximum number of runs over all

strings of length n, denoted as $\rho(n)$, is $\mathcal{O}(n)$, without determining an explicit constant, and conjectured that $\rho(n) \leq n$. Rytter [5, 6] determined such a constant in 2006, and the following years witnessed a tightening of the lower and upper bounds for the limit of $\rho(n)/n$, see [7, 8, 9, 10, 11, 12, 13, 14, 15].

In 2015, the conjecture was proven by Bannai et al. [16] who showed that $\rho(n) \leq n - 1$, and $\rho(n) \leq n - 3$ for $n \geq 5$. Their approach maps each run to the starting positions of its L-roots. For two distinct runs, the sets of the starting positions of their respective L-roots are mutually disjoint subsets of the indeces of the string, and so $\rho(n) \leq n$. *L-roots* of a run are all Lyndon non-trivial right cyclic shifts of the root.

Deza and Franek investigated $\rho_d(n)$, the largest number of runs over all strings of length n with exactly d distinct symbols. Similarities between $\rho_d(n)$ and the largest diameter $\Delta(d, n)$ over all polytopes of dimension d having n facets triggered the formulation of the d-step conjecture for strings stating that $\rho_d(n) \leq n - d$, see [17]. The proposed d-step approach proved that the following statements are equivalent $\boldsymbol{\rho_d(n) \leq n - d}$ **for all** $\boldsymbol{d}$ **and** $\boldsymbol{n}$ and $\boldsymbol{\rho_d(2d) \leq d}$ **for all** $\boldsymbol{d}$, and that $\rho_d(2d)$ is achieved for all d by, up to relabelling, a unique string. Considering binary strings, based on the method of L-roots, Fischer et al. [18] showed that $\rho_2(n) \leq \lceil 22n/23 \rceil$. While it is widely believed that $\rho_{d+1}(n) \leq \rho_d(n)$, and thus that $\rho(n) = \rho_2(n)$, no such results are known. In [19], Deza and Franek refined Bannai et al.'s method and proved the main d-step conjecture and highlighted the structural properties of run-maximal strings. Besides strengthening by one the upper

bound to $\rho(n) \leq n - 4$ for $n \geq 9$, these structural properties may provide preliminary substantiation for the hypothesis that $\rho(n) \leq n - \lceil \log_2 n \rceil$.

Previously, the Main/Kolpakov-Kucherov algorithm was the only known linear-time algorithm for computing all the runs in a given string $\boldsymbol{x}$. The algorithm was complex and depend for its worst-case behaviour on the use of Farachs algorithm, which is also complex and not space-efficient, for linear-time computation of suffix trees [2]. Since 2003, several recursive linear time algorithms for sorting suffixes emerged, all based on Farach's approach, see e.g. [20, 21, 22].

Computation of Lyndon array has recently become of interest since Bannai et al. showed that it could be used to efficiently compute all the runs in a given string. ***Lyndon array*** $\lambda = \lambda_{\boldsymbol{x}}[1..n]$ of a given nonempty string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ gives at each position i the length (or, equivalently, the end position) of the longest Lyndon word starting at i. In essence, in 2003 Hohlweg and Reutenauer characterized maximal Lyndon words in a string $\boldsymbol{x}$ [23], and showed that the Lyndon array of $\boldsymbol{x}$ is the NSV (next smaller value) array of the inverse suffix array. The NSV algorithm can be implemented using a stack and performs in linear time. Since suffix array can be computed in linear time (see e.g. [20]), Lyndon array can be computed in linear time. Thus, computing Lyndon array boils down to sorting suffixes, yet another application of the prolific suffix atrrays.

Since 2003 three worst-case linear-time suffix array construction algorithms, [21], have been available for use in the computation of the LZ factor-

ization , but even after the substitution of suffix arrays for suffix trees in the all-runs algorithm, significant complications remain. For instance, the algorithm still requires at least $13n$ bytes of space for the input string of length n. Further, it appears that due to their recursive nature the linear-time algorithms are not in practice the fastest suffix array construction algorithms available [24].

For more details and additional results concerning runs in strings we refer to [16] and references therein.

1.2 Preliminaries

Definition 1.1. *An* alphabet $\sum$ *is a set of* symbols *(or* characters*). A finite* string $\boldsymbol{x}$ *over the alphabet* $\sum$ *is a finite sequence of symbols from the alphabet. The number of the symbols in the sequence is the* length *of the string;* $|\boldsymbol{u}|$ *denotes the length of the string* $\boldsymbol{u}$*. A string with no symbols, i.e. of length 0, is referred to as an* empty string *and is denoted by the symbol* $\boldsymbol{\varepsilon}$*. In this thesis,* word *is a synonym for* string*.*

Alphabet	Strings
{a}	$\boldsymbol{\varepsilon}$,a,aa,aaa
{a,b}	$\boldsymbol{\varepsilon}$,ab,aab,abab,bba

Table 1.1: Examples of alphabets and strings

Definition 1.2. *The notation* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *indicates a string of length* n*. The* [ ] *indexing (starting from* 1*) is used to indicate a* substring *where the*

indices are the starting and ending positions: $\boldsymbol{x}[i..j]$ *indicates the substring of* $\boldsymbol{x}$ *from the position* i *to the position* j*. A* concatenation *of* $\boldsymbol{u} = \boldsymbol{u}[1..n]$ *and* $\boldsymbol{v}[1..m]$ *denoted as* $\boldsymbol{uv}$ *is defined as* $\boldsymbol{uv} = \boldsymbol{x}[1..n+m]$ *where* $\boldsymbol{x}[i] = \boldsymbol{u}[i]$ *for any* $i \in 1..n$ *and* $\boldsymbol{x}[i] = \boldsymbol{v}[i-n]$ *for any* $i \in n+1..n+m$*.*

String	Length	Substrings
aab	3	{a,ab }
ababaab	7	{ab,baab,abaab}
abababababbaa	11	{ab,ab,bbaa}

Table 1.2: Examples of strings and substrings

Let $\boldsymbol{u} = ababaab$ and $\boldsymbol{v} = abababababbaa$ then $\boldsymbol{uv} = abababaababababbaa$.

Definition 1.3. *If* $\boldsymbol{x} = \boldsymbol{uvw}$*, then* $\boldsymbol{u}$*,* $\boldsymbol{v}$*, and* $\boldsymbol{w}$ *are* factors *(*substrings, subwords*) of* $\boldsymbol{x}$*, and furthermore,* $\boldsymbol{u}$ *is a* prefix *and* $\boldsymbol{w}$ *is a* suffix *of* $\boldsymbol{x}$*. If* $\boldsymbol{u} \neq \boldsymbol{\varepsilon}$*, then* $\boldsymbol{u}$ *is a* non-trivial prefix*, while* $\boldsymbol{\varepsilon}$ *is a* trivial prefix *of any string. If* $\boldsymbol{u} \neq \boldsymbol{x}$*, then* $\boldsymbol{u}$ *is a* proper prefix *of* $\boldsymbol{x}$*. Similarly if* $\boldsymbol{v} \neq \boldsymbol{\varepsilon}$*, it is referred to as a* non-trivial suffix*, while* $\boldsymbol{\varepsilon}$ *is a* trivial suffix *of any string. If* $\boldsymbol{u} \neq \boldsymbol{x}$*,* $\boldsymbol{u}$ *is referred to as a* proper prefix*, similarly for* proper suffix*. If* $\boldsymbol{x} = \boldsymbol{uv} = \boldsymbol{wu}$ *for some* $\boldsymbol{u}$*,* $\boldsymbol{v}$*, and* $\boldsymbol{w}$*, then* $\boldsymbol{u}$ *is a* border of $\boldsymbol{x}$*. Note that every nonempty string has an empty border. A string with no non-trivial border is called* unbordered*.*

Definition 1.4. *If* $\boldsymbol{x} = \boldsymbol{uv}$ *for some* $\boldsymbol{u}$ *and* $\boldsymbol{v}$*, then* $\boldsymbol{vu}$ *is said to be the a* rotation *(or a* conjugate*) of* $\boldsymbol{x}$*. A rotation* $\boldsymbol{vu}$ *of* $\boldsymbol{uv}$ *is said to be* non-trivial *if both* $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *are nonempty.*

String	Prefix	Suffix	Border
ccababcc	cc	cc	cc
ccabab	cc	ab	$\boldsymbol{\varepsilon}$
1236767123	123	123	123

Table 1.3: Examples of prefixes, suffixes and borders

String	Conjugate
aba	baa
aababab	ababaab

Table 1.4: Examples of strings and their conjugates

Definition 1.5. *Let* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *and consider a substring* $\boldsymbol{x}[i..j]$, $1 \leq i < j < n$. *The substring* $\boldsymbol{x}[i..j]$ *is a* trivial right cyclic shift *of* $\boldsymbol{x} = \boldsymbol{x}[i..j]$. *The substring* $\boldsymbol{x}[i+1..j+1]$ *is a* right cyclic shift *of the substring* $\boldsymbol{x}[i..j]$ *iff* $\boldsymbol{x}[i] = \boldsymbol{x}[j+1]$. *The right cyclic shifts of higher order are defined recursively:* $\boldsymbol{x}[i+k..j+k]$ *is a* right cyclic shift *of* $\boldsymbol{x}[i..j]$ *iff* $\boldsymbol{x}[i+k-1..j+k-1]$ *is a right cyclic shift of* $\boldsymbol{x}[i..j]$ *and* $\boldsymbol{x}[i+k..j+k]$ *is a right cyclic shift of* $\boldsymbol{x}[i+k-1..j+k-1]$.

Thus

Observation. $\boldsymbol{x}[i+k..j+k]$ *is a right cyclic shift of* $\boldsymbol{x}[i..j]$ *iff for any* $0 \leq k_1 < k$, $\boldsymbol{x}[i+k_1..j+k_1]$ *is a right cyclic shift of* $\boldsymbol{x}[i..j]$ *and* $\boldsymbol{x}[i+k..j+k]$ *is a right cyclic shift of* $\boldsymbol{x}[i+k_1..j+k_1]$.

A *left cyclic shift* is defined similarly, so

Observation. $\boldsymbol{x}[i-k..j-k]$ *is a left cyclic shift of* $\boldsymbol{x}[i..j]$ *iff for any* $0 \leq k_1 < k$, $\boldsymbol{x}[i-k_1..j-k_1]$ *is a left cyclic shift of* $\boldsymbol{x}[i..j]$ *and* $\boldsymbol{x}[i-k..j-k]$ *is a left cyclic shift of* $\boldsymbol{x}[i-k_1..j-k_1]$.

Example 1.6. *All right cyclic shifts of the prefix abba of string* $\boldsymbol{x} = \underline{abbaabc}$.

Moreover, if $\boldsymbol{x}[i+k..j+k]$ is a right cyclic shift of $\boldsymbol{x}[i..j]$, then $\boldsymbol{x}[i..j]$ is a left cyclic shift of $\boldsymbol{x}[i+k..j+k]$ and vice versa. Note that a left or right cyclic shift of $\boldsymbol{x}[i..j]$ is a rotation of $\boldsymbol{x}[i..j]$.

Definition 1.7. *A concatenation of* k *copies of a string* $\boldsymbol{u}$ *is denoted as* $\boldsymbol{u}^k$ *and referred to as a* power of order k, $k \geq 1$. *Thus,* $\boldsymbol{u}^k = \underbrace{\boldsymbol{u} \cdots \boldsymbol{u}}_{k \text{ times}}$. *A power of order 2,* $\boldsymbol{u}^2$, *is referred to as a* square, *a power of order 3,* $\boldsymbol{u}^3$, *as a* cube. *A string* $\boldsymbol{x}$ *is* primitive *if it is not a power of order* k *for any* $k \geq 2$.

String	Square	Cube
aabca	...aabcaaabca...	...aabcaaabcaaabca...
a	aa	aaa
ab	...ababa...	...ababab

Table 1.5: Examples of non-primitive strings

Definition 1.8. *A very important notion is that of a* period *of a string:* p *is a period of* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *iff* $\boldsymbol{x}[i] = \boldsymbol{x}[i+p]$ *for any* $1 \leq i \leq n-p$.

Note that if p is a period of $\boldsymbol{x}$, then $\boldsymbol{x} = \boldsymbol{u}^k\boldsymbol{v}$ where $|\boldsymbol{u}| = p$, $k \geq 1$, and $\boldsymbol{v}$ is a proper, not necessarily non-trivial, prefix of $\boldsymbol{u}$. It follows, that if p is a minimal period of $\boldsymbol{x}$, then $\boldsymbol{x}[1..p]$ must be primitive. Further note that any rotation and so any cyclic shift of a primitive string must again be primitive. A string $\boldsymbol{x}$ is *periodic* if for some p, $1 \leq p < |\boldsymbol{x}|$, p is a period of $\boldsymbol{x}$, and if it is not periodic, it is said to be *aperiodic*.

Observation. (*a*) *For every* p, $1 \leq p < |\boldsymbol{u}|$, p *is a period of* $\boldsymbol{u}$ *iff* $\boldsymbol{u}$ *has a border of size* p.
(*b*) $\boldsymbol{u}$ *is aperiodic iff* $\boldsymbol{u}$ *is unbordered.*
(*c*) *if* $\boldsymbol{u}$ *has a border, then* $\boldsymbol{u}$ *has a border of size* $< \frac{1}{2}|\boldsymbol{u}|$.

Proof. (*a*) and (*b*) are straightforward. We shall show (*c*). Let $\boldsymbol{u_1}$ be a border of $\boldsymbol{u}$. If $|\boldsymbol{u_1}| < \frac{1}{2}|\boldsymbol{u}|$, then we are done. If the size of $\boldsymbol{u_1}$ is too big, the prefix $\boldsymbol{u_1}$ of $\boldsymbol{u}$ and the suffix $\boldsymbol{u_1}$ of $\boldsymbol{u}$ overlap – let $\boldsymbol{u_2}$ be their intersection. Then $\boldsymbol{u_2}$ is again a border of $\boldsymbol{u}$ and is smaller that $\boldsymbol{u_1}$. Either the size of $\boldsymbol{u_2}$ is sufficiently small, and we are done, or it is still too big and we can repeat the process. At each step we are producing a smaller border, so in finitely many steps we must produce a border that does not overlap. □

With the basic definitions concerning strings, we can define more complex structures and notions: we already defined a run and its roots. We can now see that a run (s, e, p) can be viewed also as a right cyclic shift of its root $x[s..s+p-1]$ at least by p positions: $\boldsymbol{ba\overbrace{\underline{aab}aabab}^{}b} \rightarrow \boldsymbol{ba\overbrace{a\underline{aba}abab}^{}b} \rightarrow \boldsymbol{ba\overbrace{aa\underline{baa}bab}^{}b} \rightarrow \boldsymbol{ba\overbrace{aab\underline{aab}ab}^{}b} \rightarrow \boldsymbol{ba\overbrace{aaba\underline{aba}b}^{}b} \rightarrow \boldsymbol{ba\overbrace{abaa\underline{bab}}^{}b}$.

Definition 1.9. *A* repetition (s, e, p) *in* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *is the substring* $\boldsymbol{x}[s..e]$ *so that* $\boldsymbol{x}[s..e] = \boldsymbol{x}[s..s+p-1]^k$ *for some integer* $k \geq 2$, *where* $\boldsymbol{x}[s..s+p-1]$ *is the* root *of the repetition. If moreover the repetition cannot be extended left, i.e. either* $s < p+1$ *or* $\boldsymbol{x}[s-p..s-1] \neq \boldsymbol{x}[s..s+p-1]$, *and if the repetition cannot be extended right, i.e. either* $e > n-p$ *or* $\boldsymbol{x}[e+1..e+p] \neq \boldsymbol{x}[s..s+p-1]$, *then we speak of* maximal repetition.

The maximum number of maximal repetitions in any string $\boldsymbol{x}$ is known to be $\Theta(n \log(n))$ [15]. A maximal repetition again can be seen as a right cyclic shift of its root, but in comparison to runs, the number of shifts must be a multiple of the size of the root, the period.

String	Repetitions	Runs
...abbaabba...	{ abba,abb,bba}	abbaabba
dabcabcabcad	{ abc,bca,cab }	abcabcabc

Table 1.6: Examples of repetitions and runs

Definition 1.10. *An* ***order*** *or* ***ordering*** $\preceq$ *of a alphabet* $\sum$ *is a total ordering of* $\sum$ *as a set,i.e it is a binary relation pocessing the following properities:*

- *Reflexive:* $a \preceq a$ *for any* $a \epsilon \sum$
- *Antisymmetric: For any* a , $b \epsilon \sum$ $a \preceq b$ *and* $b \preceq a$ *implies* $a = b$
- *Transitive: If* $a \prec b \prec c$*, then* $a \prec c$
- *Total: For any a,b,c* $\epsilon \sum$*,* $a \preceq b$ *or* $b \preceq a$

The order $\prec$ is typically extended to a total *lexicographic* order of all strings over the alphabet $\sum$ by a simple rule that $x \prec y$ iff either x is a proper prefix of y or $x[i] \prec y[i]$ while $x[1..i-1] = y[1..i-1]$, while $x \preceq y$ iff $x \prec y$ or $x = y$.

Definition 1.11. *For a string* $\boldsymbol{x} = \boldsymbol{uv}$*,* $\boldsymbol{vu}$ *is a* rotation *of* x*. If* $\boldsymbol{v}$ *and* $\boldsymbol{u}$ *are both non-empty, the rotation is said to be* non-trivial*. A string is* Lyndon *if it is strictly lexicographically smaller than any of its non-trivial rotations.*

Alphabet	Order	Lexicographic order
{a,b}	a $\prec$ b	aa $\prec$ aab $\prec$ aba $\prec$ b $\prec$ baa
{a,b,c}	a $\prec$ b $\prec$ c	ab $\prec$ abb $\prec$ abbc $\prec$ abc $\prec$ acbc
{a,b,c}	a $\prec$ c $\prec$ b	acbc $\prec$ ab $\prec$ abc $\prec$ abb $\prec$ abbc

Table 1.7: Examples of lexicographic order of strings over a given alphabet

The investigation of Lyndon strings, in the *combinatorics on words* community called Lyndon *words*, was initiated by Lyndon who was looking for a suitable description of generators of free Lie algebras, [25].

x:Word	Rotations	Lexicographical Order ($\prec$)	Lyndon
abb	abb,bba,bab	abb $\prec$ bab $\prec$ bba	x is Lyndon
aba	aba,baa,aab	aab $\prec$ aba $\prec$ baa	x is not Lyndon
abab	abab,baba,abab,baba	abab $\prec$ baba	x is not Lyndon

Table 1.8: Examples of Lydon words

One of the basic properties of Lyndon words is that every word is uniquely factorisable as a non-increasing concatenation of Lyndon words. The following theorem, though not stated in [26] explicitly, follows from the work presented there:

Theorem 1.12 (Lyndon decomposition theorem, Chen+Fox+Lyndon, 1958)**.** *For any string* $\boldsymbol{x}$*, there are unique Lyndon strings* $\boldsymbol{u}_1, ..., \boldsymbol{u}_k$ *so that* $\boldsymbol{u}_{i+1} \prec \boldsymbol{u}_i$ *for any* $1 \leq i < k$*, and* $\boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_k$ *.*

As there exists a bijection between Lyndon words over an alphabet of cardinality k and irreducible polynomials over $\boldsymbol{F_k}$, [27], lots of results are known

about this factorization: the average number of factors, the average length of the longest factor [28] and of the shortest [29]. Several algorithms deal with Lyndon factorization. Duval gives in [30] an algorithm that computes, in linear time and in-place, the factorization of a word into Lyndon words. In [31] an algorithm generating all Lyndon words up to a given length in lexicographical order is presented. This algorithm runs in a constant average time.

As far as we know, the first to consider the problem of computation of maximal Lyndon factors (substrings) at every position of a string were Hohlweg and Reutenauer, [23]. From their work it follows that

Theorem 1.13. *For a string* $\boldsymbol{x}[1..n]$, $\boldsymbol{x}[i..i+k]$ *is a maximal Lyndon factor of* $\boldsymbol{x}$ *iff* $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[i+j..n]$ *for any* $1 < j \leq k$, *while* $\boldsymbol{x}[k+1..n] \prec \boldsymbol{x}[i..n]$.

Definition 1.14. *An integer array* $L_{\boldsymbol{x}}[1..n]$ *is called* Lyndon array *of string* $\boldsymbol{x}$ *if* $L_{\boldsymbol{x}}[i]$ = *the length of the maximal Lyndon factor of* $\boldsymbol{x}$ *starting at the position* i.

Lyndon Factors:

a b b a b a b a a a b a

Lyndon array:

3 1 1 2 1 2 1 4 3 2 1 1

From [23], it follows that the Lyndon array of a string can be computed from the inverse suffix array in linear time using the stack-based NSV (Next

Smaller Value) algorithm. For a string $\boldsymbol{x}[1..n]$, the *suffix array* gives the lexicographic order of suffixes of $\boldsymbol{x}$, i.e. $s_{\boldsymbol{x}}[i] = j$ if the suffix $\boldsymbol{x}[i..n]$ is the j-th suffix in the lexicographic order. The *inverse suffix array* $s^{-1}_{\boldsymbol{x}}[j] = i$ iff $s_{\boldsymbol{x}}[i] = j$. Therefore, the complexity of computing the Lyndon array boils down to the complexity of sorting suffixes as computation of the inverse suffix array from the suffix array can be simply done in $\Theta(n)$ time and space.

i	SA[i]	ISA[i]	$S_{SA[i]}$
1	12	7	\$
2	11	6	a\$
3	8	11	auga\$
4	10	9	ga\$
5	5	5	issauga\$
6	2	10	ississauga\$
7	1	8	mississauga\$
8	7	3	sauga\$
9	4	12	sissauga\$
10	6	4	ssauga\$
11	3	2	ssissauga\$
12	9	1	uga\$

Table 1.9: Suffix Array and Inverse Suffix Array of string $\boldsymbol{s}$=*mississauga*\$.

Suffixes of a string of length n can be sorted simply and in-place by brute force in $\mathcal{O}(n^2)$ – thus there is a simple $\mathcal{O}(n^2)$ algorithm to compute the Lyndon array that requires $\mathcal{O}(n)$ memory. With some extra effort and more memory, they can be naturally sorted in $\mathcal{O}(n \log(n))$ via an iterative algorithm, so there is an $\mathcal{O}(n \log(n))$ algorithm to compute the Lyndon array.

In 2003 three linear-time suffix sorting algorithms were introduced.The

two most practical were by Kärkkäinen and Sanders, [21], and by Ko and Aluru, [21].

In 2015, [24, 32], Baier presented $\mathcal{O}(n \log(|\Sigma|))$ algorithm for sorting suffixes for strings over the alphabet Σ. Hence, for a bounded or sorted alphabet, it is a linear-time algorithm, and it is the first such algorithm which is non-recursive. Recently, we proposed in [33] a potentially faster algorithm for phase 2 of Bair's method that computes Lyndon array from a partially sorted Lyndon grouping array in linear time. We discuss this in Chapter 5.

Construction of suffix array is computationally hard. Lots of work has been done to make this more time and space efficient. Various approaches use the relation between suffixes of the same string and rely on three main techniques:

Prefix doubling

Sort the suffixes by their first character establishing a *sorted context*, followed by a more fine grained ordering by using the lexicographic rank of the suffixes implying *context doubling / prefix doubling.*The technique has asymptotic time complexity of $\mathcal{O}(n \log(n))$.

Recursion

The suffixes of the input string $\boldsymbol{x}$ are devided into two groups G_1 and G_2; a new string $\boldsymbol{y}$ is built so that the order of its suffixes corresponds to the order of suffixes in G_1, its suffixes are sorted recursively. From the order of the suffixes in G_1, the order of suffixes in G_2 can be computed in linear time.

Then the two groups are merged together in linear time. Since the length of $\boldsymbol{y}$ is an integer fraction of the lengths of $\boldsymbol{x}$, the overall complexity is $\mathcal{O}(n)$.

Induced Copying

The idea of typing suffixes is the same as in recursive algorithms, but instead of using recursion, efficient string sorting techniques are used to obtain the order of specially typed suffixes.

Chapter 2

Lyndon Words and Lyndon roots of runs

2.1 Lyndon Words

Lyndon words are named after mathematician *Roger Lyndon* who introduced them 1954 under the name of *standard lexicographic sequences*. There are several mutually equivalent properties of Lyndon words:

Lemma 2.1. *For any non-trivial non-empty word* $\boldsymbol{x}$*, the following are equivalent:*

(*a*) $\boldsymbol{x}$ *is Lyndon*

(*b*) $\boldsymbol{x} \prec \boldsymbol{x}[i..n]$ *for any* $2 \leq i \leq n$

(*c*) $\boldsymbol{x}[1..i] \prec \boldsymbol{x}[i{+}1..n]$ *for any* $1 \leq i < n$

(*d*) *There exists* $1 \leq i < n$ *so that* $\boldsymbol{x}[1..i]$ *and* $\boldsymbol{x}[i{+}1..n]$ *are Lyndon.*
The standard factorization *of* $\boldsymbol{x}$ *is such a pair of Lyndon words* $\boldsymbol{x}[1..i]$, $\boldsymbol{x}[i{+}1..n]$ *is the longest.*

Lemma 2.2. $\boldsymbol{u}$ *is Lyndon* $\overset{\Rightarrow}{\not\Leftarrow}$ $\boldsymbol{u}$ *is unbordered* $\overset{\Rightarrow}{\not\Leftarrow}$ $\boldsymbol{u}$ *is primitive*

Proof. Lyndon $\Rightarrow$ *unbordered* – assume that $\boldsymbol{u}$ has a non-overlapping border $\boldsymbol{v}$. Than $\boldsymbol{u} = \boldsymbol{vwv}$. Let $r \geq 0$ be maximal such that $\boldsymbol{w} = \boldsymbol{v}^r\boldsymbol{w_1}$. Then $\boldsymbol{u} = \boldsymbol{v}^{\boldsymbol{r+1}}\boldsymbol{w_1v}$. Then $\boldsymbol{v}^{r+2}\boldsymbol{w_1}$ is lexicographically smaller than $\boldsymbol{v}^{r+1}\boldsymbol{w_1v}$, a contradiction with $\boldsymbol{u} = \boldsymbol{v}^{r+1}\boldsymbol{w_1v}$ being Lyndon.
Lyndon $\not\Leftarrow$ *unbordered* – for instance *cab* is unbordered, but it is not Lyndon.
unbordered $\Rightarrow$ *primitive* – assume that $\boldsymbol{u}$ is not primitive, then $\boldsymbol{u} = \boldsymbol{v}^r$ for some $\boldsymbol{v}$ and some $r \geq 2$. Then $\boldsymbol{v}$ is a border of $\boldsymbol{u}$.
unbordered $\not\Leftarrow$ *primitive* – for instance, *aba* is primitive, but it has a non-trivial border *a*. □

The fact that the standard factorization of a Lyndon word can be extended to a factorization of an arbitrary word follows from the Chen, Fox, and Lyndon paper 1958 [26], though it is not stated there explicitly. It is also often referred to as *standard Lyndon factorization.*

Theorem 2.3 (Lyndon decomposition theorem, Chen+Fox+Lyndon, 1958)**.** *For any string* x*, there are unique Lyndon strings* $u_1, ..., u_k$ *so that* $u_{i+1} \prec u_i$ *for any* $1 \leq i < k$*, and* $x = u_1u_2...u_k$.

In 1983, Duval [30] proposed an elegant liner time algorithm to compute

the Lyndon factorization of a word. In the following we briefly describe the main ideas of the algorithm.

Let $\mathcal{A}$ be a finite alphabet totally ordered by $\prec$, and let $\preceq$ denote the lexicographical ordering induced on $\mathcal{A}^*$. The conjugacy class of a word $\boldsymbol{w}$ is the set of all words $\boldsymbol{uv}$ such that $\boldsymbol{w} = \boldsymbol{vu}$, i.e. all rotations of $\boldsymbol{w}$. Then Lyndon words are all minimal elements in the conjugacy classes.

For example if $\mathcal{A} = \{0, 1\}$ with $0 \prec 1$, then the 14 Lyndon words of length at most 5 in lexicographic ordering are: 0, 00001, 0001, 00011, 001, 00101, 0111, 00111, 01, 01011, 011, 0111, 01111, 1 .

Denote by a and z the minimal and the maximal letter in the alphabet $\mathcal{A}$, and by $v(\boldsymbol{b})$ the letter following $\boldsymbol{b} \neq \varepsilon$ in the total ordering of $\mathcal{A}$. If $\boldsymbol{w}$ is a word of the form $\boldsymbol{w} = \boldsymbol{u}bz^h$, with $b \neq z$, then we denote by $P(\boldsymbol{w})$ the word $\boldsymbol{u}v(b)$.

Consider a fixed integer n. Duval's algorithm computes, from a given Lyndon word $\boldsymbol{w}$, the next Lyndon word $N(\boldsymbol{w})$ of length at most n in two steps.

Algorithm

Input: An integer n, and a Lyndon word $\boldsymbol{w} \neq z$ of length at most n.

Step 1: Compute the word $\boldsymbol{v} = D(\boldsymbol{w}) = \boldsymbol{w}^h\boldsymbol{w'}$, where $h \leq 1$ and $\boldsymbol{w'}$ is a proper prefix of $\boldsymbol{w}$ defined by $n = h|\boldsymbol{w}| + |\boldsymbol{w'}|$.

Step 2: Compute the word $P(\boldsymbol{v})$.

output: $P(D(\boldsymbol{w}))$.

Duval proved that $N(\boldsymbol{w}) = P(D(\boldsymbol{w}))$. The implementation of the algorithm

is quite straightforward. [34] We will revisit the algorithm when discussing computation of Lyndon arrays.

Applications of Lyndon Words

There are many applications of Lyndon words in algebra and combinatorics, such as:

1. They are used to describe the generators of the *free Lie algebras* – the original motivation Lyndon investigated them.
2. They are used as a *special case of Hall sets*.
3. Lyndon words have applications to semigroups, pattern matching, and representation theory of certain algebras.

All of these applications make use of the *combinatorial properties of Lyndon words*, in particular the factorization theorem. Lyndon word naming classifies highly periodic strings by the conjugacy of their periods and uses the Lyndon word as the class representative.Once the Lyndon word naming has been performed, a string can be represented by the name of its period's class and its $LWpos$, the position at which the Lyndon word first occurs in the string. For example: the strings $T_1 = abbaabbaabbaabbaab$ and $T_2 = aabbaabbaabbaabbaa$ are in the same class and the class representative is $aabb$. $LWpos$ of T_1 is 3 while $LWpos$ of T_2 is 0 since it begins with the Lyndon word that represents its period [35].

Lyndon bracketing is another significant mechanism for Lyndon factorization.

Definition 2.4. *Let $L_k(n)$: set of k-ary Lyndon words of length n.*

If w is in $L_k(n)$ then $\gamma(w)$ is called ***Lyndon Bracketing*** *of* $\boldsymbol{w}$ *given by:*

$$\gamma(w) = \begin{cases} w & if \quad |w| = \quad 1 \\ [\gamma(l), \gamma(m)] & otherwise \ where \quad \sigma(w) = \quad (l, m) \end{cases}$$

Example 2.5. $[a, [a, [b]]]$

2.2 Periodicity

If $\boldsymbol{x}[i] = \boldsymbol{x}[i{+}p]$ for all $i \in [1..|\boldsymbol{x}|{-}p]$, then $\mathbf{x}$ has a period $\mathbf{p}$. Every period of a string corresponds to a border. Below are some of the most frequently used lemmas concerning periodicities.

Lemma 2.6. *(Lothaire 2002, Section 8.1.1). If* $\boldsymbol{v}$ *is a border of* $\boldsymbol{w}$ *, then* $\boldsymbol{w}$ *has a period* $|\boldsymbol{w}|{-}|\boldsymbol{v}|$*. Conversely, if* $\boldsymbol{w}$ *has a period p , then* $\boldsymbol{w}[1..|\boldsymbol{w}|{-}p]$ *is a border.*

For example the string:

	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x}$ =	a	b	a	a	b	a	b	a	a	b

has borders *abaab* and *ab*, hence corresponding periods 5 and 8 respectively. The analysis of periodicity often involves strings of more than one period, or periodic strings that overlap.

Lemma 2.7. *(Lothaire 2002, Section 8.1.1). If* $\boldsymbol{x}$ *has periods p and q such that* $q < p \leq |\boldsymbol{x}|$*, then the border of* $\boldsymbol{x}$ *of length* $|\boldsymbol{x}|{-}q$ *has a period* $p{-}q$*.*

Lemma 2.8. *(Lothaire 2002, Section 8.1.3). If $\boldsymbol{x}$ has a period p and there exists a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ with $p \leq |\boldsymbol{u}|$ that has a period q, where q divides p, then $\boldsymbol{x}$ has a period q.*

The following Periodicity Lemma is quite well-known with many consequences. For instance, it played a major role in the investigation of FS-double-squares in [36].

Lemma 2.9. *Periodicity Lemma (Fine and Wilf 1965; Lothaire 2005). If $\boldsymbol{x}$ has periods p and q, and $p+q \leq |\boldsymbol{x}|+\gcd(p,q)$, then $\boldsymbol{x}$ also has a period $\gcd(p,q)$.*

For example, the string:

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{x}$ =	a	b	a	a	b	a	a	b	a	a	b	a	a

has length $n = 13$, and periods $p = 6$ and $q = 9$. Since $d = gcd(p, q) = 3$ and $p + q = 15 < n + d = 16$, the Periodicity Lemma allows us to infer that the string also has a period $d = 3$.

Periodicity Lemma is one of the most important results in combinatorics on words featuring in many correctness proofs of string algorithms.

Lemma 2.10. *(Lothaire 2002, Lemma 8.1.2). If $\boldsymbol{x} = \boldsymbol{uvw}$, and $\boldsymbol{uv}$ and $\boldsymbol{vw}$ have period $p \leq v$, then $\boldsymbol{x}$ has period p.*

Lemma 2.11. *(Simpson 2007, Section 1). If $\boldsymbol{x} = \boldsymbol{uvw}$, where $\boldsymbol{uv}$ has period p, $\boldsymbol{vw}$ has period q, and $p + q \leq v + gcd(p, q)$, then $\boldsymbol{x}$ has period gcd(p, q).*

Lemma 2.12. *(Synchronization Principle). If a string $\boldsymbol{x}$ is primitive, and $\boldsymbol{u}$ is a proper prefix of $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu} \neq \boldsymbol{x}$, or equivalently, $\boldsymbol{x}$ is not equal to any of its non-trivial rotations.*

Proof. Assume by contradiction that $\boldsymbol{x} = \boldsymbol{uv} = \boldsymbol{vu}$ for non empty strings $\boldsymbol{u}$ and $\boldsymbol{v}$. Without loss of generality, suppose that $|\boldsymbol{u}| < |\boldsymbol{v}|$. Since $\boldsymbol{u}$ and $\boldsymbol{v}$ are prefixes of $\boldsymbol{x}$, $\boldsymbol{u}$ is a prefix of $\boldsymbol{v}$ and vu is a factor of $\boldsymbol{v}^2$. Since $\boldsymbol{v}$ is a prefix of $\boldsymbol{uv}$, $\boldsymbol{v} = \boldsymbol{u}^n\boldsymbol{u'}$ for a prefix $\boldsymbol{u'}$ of $\boldsymbol{u}$ and an integer n, and $\boldsymbol{uv} = \boldsymbol{u}^{n+1}\boldsymbol{u'}$ is a factor of $\boldsymbol{u}^{n+2}$. By Periodicity Lemma, $\boldsymbol{x}$ has a period $gcd(|\boldsymbol{u}|, |\boldsymbol{v}|)$ and is not primitive, hence a contradiction. □

2.3 Periodicitiy And Lyndon Roots

When periodicities in words are considered, then two notions are central: the period, which gives the least amount by which a word has to be shifted in order to overlap with itself, and the shortest border, which denotes the least (nonempty) overlap of a word with itself. Both of these notions are related to each other, for example the length of the shortest border of a word $\boldsymbol{w}$ is not larger than the period of $\boldsymbol{w}$, and hence, the period of an unbordered word is its length, moreover, the shortest border itself is always unbordered.

Deeper dependencies between the period of a word and its unbordered factors have been investigated for decades, see for instance [37]. These two properities of words plays important role in string searching algorithms, data compression, codes and also in computational biology for sequence assembly

of superstrings and in serial data communications systems.

As discussed previously, a run can be considered a system of at least $|\boldsymbol{r}|$ right cyclic shifts of its root $\boldsymbol{r}$. Since the root $\boldsymbol{r}$ is primitive, one of the shifts must be Lyndon. So, every non-trivial right cyclic shift of the root of a run is called *Lyndon root*. Note that a run can have more than one Lyndon root: Lyndon roots are underscored $\underline{ab}\,\underline{ab}\,\underline{ab}\,a$. Lyndon roots had been investigated for some time and played the major role in bounding the number of *cubic runs*, i.e. runs where the root repeats three times or more, [38].

In a cubic run of period p, consider the first Lyndon root. The Lyndon root starts in the root. The next p right cyclic shifts of the first Lyndon root is again a Lyndon root. Thus a cubic run has a Lyndon square whose both parts are Lyndon roots of the the run. Assign the run the end point of the first Lyndon root and call it a *handle* of the run. Now we can argue that each handle is unique, i.e. for two different runs the handles must be different. Assume that two different runs have the same handle. Since the runs overlap significantly, they have to have different periods as otherwise they would be the same run. Let r_1 denote the run with the larger period, and r_2 the one with the shorter period. Since they both have the same handle, we have $\overbrace{a..\underbrace{a...\boldsymbol{b}a...b}_{r_2}..b}^{r_1}$ two Lyndon squares. Thus, the left part of the shorter square (for r_2) is a border of the left part of the longer square (for r_1), a contradiction with the fact that it must be Lyndon.

Thus, Lyndon roots were considered for runs before Bannai et al. [16], however there was always a problem with the fact that two interleaved runs

can have Lyndon roots starting in the same positions: consider the two runs *baba* (overlined) and *abaaba* (underlined) $\overline{b\underline{abaaba}}$. The Lyndon root of *baba* is $b\underline{ab}aaba$ and the Lynbdon roots of *abaaba* are $b\underline{aba}\,\underline{aba}$.

Definition 2.13. *Consider a string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$. *The substring* $\boldsymbol{x}[i..j]$ *is a* maximal Lyndon factor at position i *if* $\boldsymbol{x}[i..j]$ *is Lyndon and for every* k, $j < k \leq n$, $\boldsymbol{x}[i..k]$ *is not Lyndon.*
The substring $\boldsymbol{x}[i..j]$ *is a* non-extendible Lyndon factor at position i *if it is a maximal Lyndon factor in* $\boldsymbol{xy}$ *for any* $\boldsymbol{y}$.

Note that *ab* is a maximal Lyndon factor at position 4 of a string *abbab*. But it is not non-extendible since the maximal Lyndon factor at position 4 of a string *abbabb* is *abb*. On the other hand consider *ab* at position 1 of the string *abaa*, it is non-extendible Lyndon factor. The reason it is non-extensible is that the string *aa* following *ab* is lexicographically smaller than *ab* so by Lemma 2.1 no Lyndon word could start with *abaa*. The Bannai et al. resolved the problem of Lyndon roots of different runs sharing the same starting position by providing a mechanism to make every Lyndon root non-extendible.

For this argument, we consider that each string is terminated by a lexicographically smallest sentinel symbol \$. Furthermore we fix a total order of the alphabet $\prec$. Let $\prec^{-1}$ denote the reverse order. Consider a run (s, e, p) in a string $\boldsymbol{x}[1..n]$. The substring $\boldsymbol{x}[e{-}p{+}1..e]$ is the last right cyclic shift of the root. Compare the two symbols $\boldsymbol{x}[e{-}p{+}1]$ and $\boldsymbol{x}[e{+}1]$, note that the latter could be the sentinel symbol \$. Also not that they must be different,

if they were not, we could shift the root one more position to the right, so the run could not end in e. If $\boldsymbol{x}[e-p+1] \prec \boldsymbol{x}[e+1]$, we chose $\prec^{-1}$, otherwise we choose $\prec$. We only consider the Lyndon roots with our chosen order. If we chose $\prec^{-1}$, it was because $\boldsymbol{x}[e+1] \prec \boldsymbol{x}[e-p+1]$ i.e. $\boldsymbol{x}[e-p+1] \prec^{-1} \boldsymbol{x}[e+1]$ and so the Lyndon roots of the run with respect to $\prec^{-1}$ are non-extendible. If we chose $\prec$, it was because $\boldsymbol{x}[e-p+1] \prec \boldsymbol{x}[e+1]$ and so again all the Lyndon roots of the run with respect to $\prec$ are non-extendible. As a technicality, though important, if the run starts with a Lyndon root, that root is not considered. This assumption does not cause any problems since when a run starts with a Lyndon root, there is another Lyndon root available.

The Lyndon roots selected by the above rules are thus referred to as *L-roots* of the run. It is now easy to see that an L-root of a run r_1 and an L-root of a distinct run r_2 cannot start at the same position: if they started with the same position, they must be Lyndon with respect to the same order, either $\prec$ or $\prec^{-1}$, but then they are both non-extendible, but one is shorter, a contradiction. Hence each run can be assigned the starting positions of all its L-roots and the subsets of string indeces are disjoint. It follows that the maximum number of runs, i.e. $\rho(n)$ is bounded by the length of the string.

Bannai et al. presented a linear algorithm to compute all runs in a string if Lyndon array of the string with respect to $\prec$ and $\prec^{-1}$ are given. They noted that it is the only linear algorithm to compute all runs without first computing the Lempel-Ziv factorization of the string. Thus, for the algorithm to be "competetive", one needs to be able to compute efficiently and in linear

time the Lyndon array for a given string and the order of its alphabet.

In [23] Hohlweg and Reutenauer gave a global characterization of maximal Lyndon factors of a string.

Lemma 2.14. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string over an alphabet $\mathcal{A}$ that is totally ordered by $\prec$. A substring $\boldsymbol{x}[i..j]$, $1 \leq i \leq j \leq n$ is a maximal Lyndon factor of $\boldsymbol{x}$ with respect to $\prec$ iff $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ for any $i < k \leq j$ and $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$.*

The same lemma can be re-phrased to make use of suffix and inverse suffix arrays.

Lemma 2.15. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string over an alphabet $\mathcal{A}$ that is totally ordered by $\prec$ and let $suf^{-1}[1..n]$ be the inverse suffix array of $\boldsymbol{x}$. A substring $\boldsymbol{x}[i..j]$, $1 \leq i \leq j \leq n$ is a maximal Lyndon factor of $\boldsymbol{x}$ iff $suf^{-1}[i] < suf^{-1}[k]$ for any $i < k \leq j$ and $suf^{-1}[j{+}1] < suf^{-1}[i]$.*

Chapter 3

Existing Algorithms to calculate Lyndon Arrays

According [16], the number of all the runs in a given string $\boldsymbol{x}$ is is computable in linear time from the Lyndon arrays $\lambda_{\boldsymbol{x}}^{\prec}$ and $\lambda_{\boldsymbol{x}}^{\prec^{-1}}$. In this chapter we will present four different algorithms for computing $\lambda_{\boldsymbol{x}}$. On occasions, it may be better to consider the array $\mathcal{L}_{\boldsymbol{x}}[i] = i + \lambda_{\boldsymbol{x}} - 1$ instead of $\lambda_{\boldsymbol{x}}$. Note that $\mathcal{L}_{\boldsymbol{x}}[i]$ is the index of the last character of the maximal Lyndon factor starting at i, i.e. $\boldsymbol{x}[i..\mathcal{L}[i]]$ is a maximal Lyndon factor of $\boldsymbol{x}$ and has a length of $\lambda_{\boldsymbol{x}}[i]$.

3.1 Background Information

Here we make various observations that apply to the algorithms presented later in the chapter.

Observation 3.1. *Let* $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ *be the Lyndon decompostion* [26, 30] *of* $\boldsymbol{x}$*, with Lyndon words* $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$*. Then every Lyndon word* $\boldsymbol{x}[i..\mathcal{L}[i]]$ *of length* $\boldsymbol{\lambda}[i]$ *is a substring of some* $\boldsymbol{w_h}$*,* $h \in 1..k$*.*

Proof. For some $h \in 1..k-1$, consider $\boldsymbol{w_h}$ with nonempty proper suffix $\boldsymbol{v_h}$, and for some $t \in 1..k-h$, consider $\boldsymbol{w_{h+t}}$ with nonempty prefix $\boldsymbol{u_{h+t}}$. Since $\boldsymbol{w_h}$ is a Lyndon word, $\boldsymbol{w_h} < \boldsymbol{v_h}$, and by lexorder, $\boldsymbol{u_{h+t}} \leq \boldsymbol{w_{h+t}}$. Thus $\boldsymbol{v_h} > \boldsymbol{w_h} \geq \boldsymbol{w_{h+t}} \geq \boldsymbol{u_{h+t}}$, and so $\boldsymbol{v_h w_{h+1} \cdots w_{h+t-1} u_{h+t}}$ cannot be a Lyndon word for any choice of h or t. □

Therefore, to compute $\lambda_{\boldsymbol{x}}$ or $\mathcal{L}_{\boldsymbol{x}}$ it suffices to consider separately each distinct element $\boldsymbol{w_h}$ in the Lyndon decomposition of $\boldsymbol{x}$. Hence, without loss of generality suppose $\boldsymbol{x}$ is a Lyndon word and write it in the form $\boldsymbol{x_1 x_2 \cdots x_m}$, where for each $r \in 1..m$, $|\boldsymbol{x_r}| = \ell_r$ and

$$\boldsymbol{x_r}[1] \leq \boldsymbol{x_r}[2] \leq \cdots \leq \boldsymbol{x_r}[\ell_r], \tag{3.1}$$

while for $1 \leq r < m$,

$$\boldsymbol{x_r}[\ell_r] > \boldsymbol{x_{r+1}}[1]. \tag{3.2}$$

We call $\boldsymbol{x_r}$ a ***range*** in $\boldsymbol{x}$ and the boundary between $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$ a ***drop***.

Observation 3.2. *Let i be a position in $\boldsymbol{x}$ that corresponds to a position $j \in 1..\ell_r$ within a range $\boldsymbol{x_r}$; that is $i = \sum_{r'=1}^{r-1} \ell_{r'} + j$.*

(a) *If $\boldsymbol{x_r}[j] = \boldsymbol{x_r}[\ell_r]$, then $\mathcal{L}[i] = i$.*

(b) *Otherwise, $\mathcal{L}[i] = i'$, where i' is the final position in some range $\boldsymbol{x_{r'}}$, $r' \geq r$; that is, $i' = \sum_{s=1}^{r'} \ell_s$.*

Proof. (a) is an immediate consequence of (3.1) and (3.2). To prove (b), suppose that $\boldsymbol{x}[i..\mathcal{L}[i]]$ is a maximum-length Lyndon word, where $\mathcal{L}[i]$ falls within range r' but $\mathcal{L}[i] < i'$. Since by (3.1) $\boldsymbol{x}[\mathcal{L}(i)] \leq \boldsymbol{x}[\mathcal{L}[i]{+}1]$, there are two consecutive Lyndon words $\boldsymbol{x}[i..\mathcal{L}[i]], \boldsymbol{x}[\mathcal{L}[i]+1]$ that by the Lyndon decomposition theorem [26] can be merged into a single Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]{+}1]$. Thus $\boldsymbol{x}[i..\mathcal{L}[i]]$ is not maximum-length, a contradiction. □

We see then that if $\boldsymbol{x_r}[j] < \boldsymbol{x_r}[\ell_r]$, then $\boldsymbol{x_r}[j..\ell_r]$ is a Lyndon word:

$$\begin{array}{rcccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \boldsymbol{x} = & a & a & a & b\,| & a & a & b\,| & a & b\,| & a & a & b & b \\ \mathcal{L} = & 13 & 13 & 4 & 4 & 9 & 7 & 7 & 9 & 9 & 13 & 13 & 12 & 13 \end{array} \tag{3.3}$$

More generally, a "reverse engineering" result [39]:

Observation 3.3. *Let $\mathcal{L}_{\boldsymbol{x}}$ be the Lyndon array of a string $\boldsymbol{x}[1..n]$ on $\Sigma = \{a, b\}$, with $\boldsymbol{x} \neq b^m a^{n-m}$ for any $m \in 0..n$. Then $\boldsymbol{x}$ is determined uniquely by $\mathcal{L}_{\boldsymbol{x}}$.*

Proof. Let n' be the smallest index such that for every $i \in n'..n$, there exists no $j < i$ such that $\mathcal{L}[j] = i$; if there is no such n', let $n' = n+1$. If $n' \leq n$, $\boldsymbol{x}[n'..n] \leftarrow a^{n-n'+1}$. By observation 3.2, for every $i < n'$ such that $\mathcal{L}[i] = i$, $\boldsymbol{x}[i] \leftarrow b$. For all other i, $\boldsymbol{x}[i] \leftarrow a$. □

The division of $\boldsymbol{x}$ into ranges has the useful consequence that ranges can be compared without necessarily requiring that all positions be compared:

Observation 3.4. *Suppose strings $\boldsymbol{x}$ and $\boldsymbol{y}$ are expressed in terms of their ranges: $\boldsymbol{x} = \boldsymbol{x_1 x_2 \cdots x_m}$, $\boldsymbol{y} = \boldsymbol{y_1 y_2 \cdots y_n}$. Suppose further that for some least integer $r \in 1..\min(m,n)$, $\boldsymbol{x_r} < \boldsymbol{y_r}$ (respectively, $\boldsymbol{x_r} > \boldsymbol{y_r}$). Then $\boldsymbol{x} < \boldsymbol{y}$ (respectively, $\boldsymbol{x} > \boldsymbol{y}$).*

Proof. If $\boldsymbol{x_r} < \boldsymbol{y_r}$, then either

(a) $\boldsymbol{x_r}$ is a nonempty proper prefix of $\boldsymbol{y_r}$; or

(b) there is some least position j such that $\boldsymbol{x_r}[j] < \boldsymbol{y_r}[j]$.

In case (a), if $r = m$, then $\boldsymbol{x}$ is actually a prefix of $\boldsymbol{y}$, so that $\boldsymbol{x} < \boldsymbol{y}$, while if $r < m$, then by (3.2), $\boldsymbol{x_{r+1}}[1] < \boldsymbol{y_r}[|\boldsymbol{x_r}|+1]$, and again $\boldsymbol{x} < \boldsymbol{y}$. In case (b) the result is immediate. The proof for $\boldsymbol{x_r} > \boldsymbol{y_r}$ is similar. □

Also useful for our algorithms is the following simple property:

Observation 3.5. *Within range $\boldsymbol{x_r}$, let i^* denote the greatest value of $i \in 1..\ell_r - 1$ such that $\boldsymbol{x_r}[i] < \boldsymbol{x}[\ell_r]$. Then for every $i \in 2..i^*$, $\mathcal{L}[i-1] \geq \mathcal{L}[i]$.*

3.2 Basic Algorithms

3.2.1 Folklore – Iterated MaxLyn Algorithm

For a string $\boldsymbol{x}$ of length n, ***prefix table*** $\pi[1..n]$ is an integer array in which for every $i\epsilon 1\,..\,n$, $\pi[i]$ is the length of the longest substring beginning at position i of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. Given a nonempty string $\boldsymbol{x}$ on alphabet Σ, let us define $\boldsymbol{x'} = \boldsymbol{x}\$$, where the sentinel $\$ < \mu$ for every letter $\mu \in \Sigma$.

Observation 3.6. *$\boldsymbol{x}$ is a Lyndon word if and only if for every $i \in 2\,..\,n$, $\boldsymbol{x'}[1+k] < \boldsymbol{x'}[i+k]$, where $k = \pi[i]$.*

This result forms the basis of the algorithm given in Figure 3.1 that computes the length $\max \in 1\,..\,n-\ell+1$ of the longest Lyndon factor at a given position j in $\boldsymbol{x}[1..n]$. Its efficiency 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, $j+\pi[j] \leq i+k$. the following lemma justifies this assumption.

Lemma 3.7. *Suppose that for some position i in a Lyndon word $\boldsymbol{x}[1..n]$, $k = \pi[i] \geq 2$. Then for every $j \in i+1\,..\,i+k-1$, $\pi[j] \leq i+k-j$.*

Proof. The result certainly holds for $i+k = n+1$, so we consider $i+k \leq n$. Assume that for some $j \in i+1\,..\,i+k-1$, $\pi[j] > i+k-j$. It follows that

$$\boldsymbol{x}[1\,..\,i+k-j+1] = \boldsymbol{x}[j\,..\,i+k], \tag{3.4}$$

while $\boldsymbol{x}[j-i+1\,..\,k] = \boldsymbol{x}[j\,..\,i+k-1]$. Since $\boldsymbol{x}$ is Lyndon, therefore $\boldsymbol{x}[1+k] \prec \boldsymbol{x}[i+k]$, and so we find that

$$\boldsymbol{x}[j-i+1\,..\,1+k] \prec \boldsymbol{x}[j\,..\,i+k]. \tag{3.5}$$

From (3.4) and (3.5) we see that $\boldsymbol{x}[1..k+1]$ has suffix $\boldsymbol{x}[j-i+1..k+1]$ satisfying $\boldsymbol{x}[j-i+1..k+1] \prec \boldsymbol{x}[1..i+k-j+1]$, $\boldsymbol{x}[1..k+1]$ has suffix $\boldsymbol{x}[j-i+1..k+1] \prec$ prefix $\boldsymbol{x}[1..i+k-j+1]$, contradicting the assumption that $\boldsymbol{x}$ is Lyndon. □

Simply repeating MaxLyn at every position j of $\boldsymbol{x}$ gives a simple, fast $\mathcal{O}(n^2)$ time and $\mathcal{O}(1)$ additional space algorithm to compute $\boldsymbol{\lambda_x}$.

```
procedure MaxLyn(x[1..n], j, Σ, ≺) : integer
i ← j + 1; max ← 1
while i ≤ n do
  k ← 0
  while x'[j + k] = x'[i + k] do
    k ← k + 1
  if x'[j + k] ≺ x'[i + k] then
    i ← i + k + 1; max ← i − 1
  else
    return max
```

Figure 3.1: Algorithm MaxLyn

Recent work on the prefix table [40, 41] has confirmed its importance as a data structure for string algorithms. In this context it is interesting to find that Lyndon words $\boldsymbol{x}$ can be characterized in terms of $\pi_{\boldsymbol{x}}$:

Observation 3.8. *Suppose* $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ *is a string on alphabet* Σ*. Then* $\boldsymbol{x}$ *is a Lyndon word over* Σ *if and only if for every* $i \in 2\,..\,n$*,*

(*a*) $i + \pi_{\boldsymbol{x}}[i] < n + 1$*; and*

(*b*) *for every* $j \in i + 1\,..\,i + \pi_{\boldsymbol{x}}[i] - 1$, $j + \pi_{\boldsymbol{x}}[j] \leq i + \pi_{\boldsymbol{x}}[i]$.

3.2.2 Recursive Duval Factorization

By observation 3.1, whenever $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a Lyndon word, we know that $\lambda_{\boldsymbol{x}}[1] = n$. Thus computing the Lyndon decomposition $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$, $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$, allows us to assign $\boldsymbol{\lambda}[i_j] = |\boldsymbol{w_j}|$, where i_j is the first position of $\boldsymbol{w_j}$, $j = 1, 2, \ldots, k$.

Algorithm RDuval make use of the above given observation recursively, by assigning $\boldsymbol{\lambda}[i_j] \leftarrow |\boldsymbol{w_j}|$, then removing the first letter i_j from each $\boldsymbol{w_j}$ to form $\boldsymbol{w'_j}$, to which the Lyndon decomposition is applied in the next recursive step. This process continues until each Lyndon word is reduced to a single letter.

The time required for RDuval is bounded above by n times the maximum depth of the recursion, thus $\mathcal{O}(n^2)$ in the worst case — consider, for example, the string $\boldsymbol{x} = a^{n-1}b$. However, to estimate expected behaviour, we can make use of a result of Bassino *et al.* [42].

Given a Lyndon word $\boldsymbol{w}$, the factorization $\boldsymbol{w} = \boldsymbol{uv}$ is called the ***standard factorization*** of $\boldsymbol{w}$ if $\boldsymbol{u}$ and $\boldsymbol{v}$ are both Lyndon words and $\boldsymbol{v}$ is of maximum size. They then show that if $\boldsymbol{w}$ is a binary string ($\Sigma = \{a, b\}$), the average

length of $\boldsymbol{v}$ is asymptotically $3|\boldsymbol{w}|/4$. Thus each recursive application of RDuval yields a left Lyndon factor of expected length $|\boldsymbol{w}|/4$ and a remainder of length $3|\boldsymbol{w}|/4$ to be factored further. It follows that the expected number of recursive calls of RDuval is $\mathcal{O}(\log_{4/3} n)$. The following lemma is a proof of the time complexity for binary strings

Lemma 3.9. *On binary strings RDuval executes in $\mathcal{O}(n \log_{4/3} n)$ time on average.*

Example 3.10. *For*

		1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x}$	=	a	a	b	a	a	b	b	a	b	b	a	b
$\boldsymbol{\lambda}$	=	12	2	1	9	3	1	1	3	1	1	2	1

the factors considered are first 1–12, then

- *2–3 and 4–12 in the first level of recursion;*
- *3, 5–7, 8–10 and 11–12 in the second level;*
- *6–7 and 9–10 in the third;*
- *7 and 10 in the fourth.*

The positions are assigned as follows: $\boldsymbol{\lambda}[1] \leftarrow 12; \boldsymbol{\lambda}[2] \leftarrow 2, \boldsymbol{\lambda}[4] \leftarrow 9; \boldsymbol{\lambda}[3] \leftarrow 1, \boldsymbol{\lambda}[5] \leftarrow 3, \boldsymbol{\lambda}[8] \leftarrow 3, \boldsymbol{\lambda}[11] \leftarrow 2; \boldsymbol{\lambda}[6] \leftarrow 1, \boldsymbol{\lambda}[9] \leftarrow 1; \boldsymbol{\lambda}[7] \leftarrow 1, \boldsymbol{\lambda}[10] \leftarrow 1.$

3.2.3 NSV Applied to the Inverse Suffix Array

Definition 3.11. *Given a string* $\boldsymbol{x}[1..n]$ *on an ordered alphabet* Σ, $NSV = NSV_{\boldsymbol{x}}[1..n]$ *is the* ***next smaller value array*** *of* $\boldsymbol{x}$ *if and only if for every* $i \in 1..n$, $NSV[i] = j$, *where*

(*a*) *for every* $h \in 1..j-1$, $\boldsymbol{x}[i] \leq \boldsymbol{x}[i+h]$; *and*

(*b*) *either* $i+j = n+1$ *or* $\boldsymbol{x}[i] > \boldsymbol{x}[i+j]$.

Example 3.12.

	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x} =$	3	8	7	10	2	1	4	9	6	5
$NSV_{\boldsymbol{x}} =$	4	1	2	1	1	5	4	1	1	1

$\text{NSV}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ time using a stack. We are focusing on the main observation touched in [23], that $\boldsymbol{\lambda_x}$ can be computed merely by applying NSV to the inverse suffix array $\text{ISA}_{\boldsymbol{x}}$. Below is a very simple $\Theta(n)$-time, $\mathcal{O}(n)$-space algorithm for this calculation

procedure NSVISA($\boldsymbol{x}[1\,..\,n]$) : $\boldsymbol{\lambda_x}[1\,..\,n]$
Compute $SA_{\boldsymbol{x}}$ (see [43, 44])
Compute $\text{ISA}_{\boldsymbol{x}}$ from $\text{SA}_{\boldsymbol{x}}$ in place (see [44])
$\boldsymbol{\lambda_x} \leftarrow \text{NSV}(\text{ISA}_{\boldsymbol{x}})$ (in place)

Figure 3.2: Apply NSV to $\text{ISA}_{\boldsymbol{x}}$

Presented here are the lemmas and the proof of the observation .

Theorem 3.13. *For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and total order $\prec$, let $ISA = ISA^{\prec}_{\boldsymbol{x}}$. Then for every $i \in 1..n$, the substring $\boldsymbol{x}[i..j]$ is a maximal Lyndon factor with respect to $\prec$ if and only if*

(a) *for every $h \in i{+}1..j$, $ISA[j] < ISA[h]$; and*

(b) *either $j = n$ or $ISA[j{+}1] < ISA[i]$.*

The following well-known result is needed to prove Lemma 3.15:

Lemma 3.14 (Duval, Lemma 1.6, [30])**.** *Suppose $\boldsymbol{x} \in \Sigma^+$, where Σ is an alphabet totally ordered by $\prec$. Let $\boldsymbol{x} = \boldsymbol{u}^r\boldsymbol{u_1}b$, where $\boldsymbol{u}$ is nonempty, $r \geq 1$, $\boldsymbol{u_1}$ a possibly empty proper prefix of $\boldsymbol{u}$, and the letter $b \neq \boldsymbol{u}[|\boldsymbol{u_1}|{+}1]$.*

(a) *If $b \prec \boldsymbol{u}[|\boldsymbol{u_1}|{+}1]$, then $\boldsymbol{u}$ is a maximal Lyndon prefix of $\boldsymbol{xy}$ for any $\boldsymbol{y}$;*

(b) *if $b \succ \boldsymbol{u}[|\boldsymbol{u_1}|{+}1]$, then $\boldsymbol{x}$ is Lyndon with respect to $\prec$.*

For a given string $\boldsymbol{x}[1..n]$, let $\boldsymbol{s_x}(i) = \boldsymbol{x}[i..n]$ denote the suffix of $\boldsymbol{x}$ beginning at position i. When clear from context we write just $\boldsymbol{s}(i)$.

Lemma 3.15. *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over alphabet Σ with ordering $\prec$. Let $\boldsymbol{x}[i\,..\,j]$ be the maximal Lyndon factor of $\boldsymbol{x}$ starting at i. Then $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(k)$ for every $k \in i{+}1..j$ and either $j = n$ or $\boldsymbol{s_x}(j{+}1) \prec \boldsymbol{s_x}(i)$.*

Proof. Because $\boldsymbol{x}[i\,..\,j]$ is Lyndon, therefore for any $i < k \leq j$, $\boldsymbol{x}[i\,..\,j] \prec \boldsymbol{x}[k\,..\,j]$ and so $\boldsymbol{s}(i) \prec s(k)$. If $j = n$, we are done. So we may assume $j < n$, and we want to show that $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)$. Suppose then that $\boldsymbol{s}(j{+}1) \not\prec \boldsymbol{s}(i)$. Since $\boldsymbol{s}(i)$ and $\boldsymbol{s}(j{+}1)$ are distinct, it follows that $\boldsymbol{s}(i) \prec \boldsymbol{s}(j{+}1)$. If we let $d = lcp(\boldsymbol{s}(i), \boldsymbol{s}(j{+}1)) + 1$, two cases arise:

(a) $0 \leq d \leq j - i$.

Here $i \leq i + d \leq j$. Thus $\boldsymbol{x}[i\,..\,i{+}d{-}1] = \boldsymbol{x}[j{+}1\,..\,j{+}d]$ and $\boldsymbol{x}[i{+}d] \prec \boldsymbol{x}[j{+}1{+}d]$, and so for $j < k \leq j{+}1{+}d$, $\boldsymbol{x}[i\,..\,j{+}1{+}d] \prec \boldsymbol{x}[k\,..\,j{+}1{+}d]$. Since $\boldsymbol{x}[i\,..\,j]$ is Lyndon, $\boldsymbol{x}[i\,..\,j] \prec \boldsymbol{x}[k\,..\,j]$ and so $\boldsymbol{x}[i\,..\,j{+}1{+}d] \prec \boldsymbol{x}[k\,..\,j{+}1{+}d]$ for any $i < k \leq j$. Thus $\boldsymbol{x}[i\,..\,j{+}1{+}d]$ is Lyndon, contradicting the assumption that $\boldsymbol{x}[i\,..\,j]$ is the maximal Lyndon factor starting at i.

(b) $0 < j - i \leq d$.

Let $d = r(j - i) + d_1$, where $0 \leq d_1 < j - i$. Then $r \geq 1$ and $\boldsymbol{x}[i\,..\,j{+}1{+}d] = \boldsymbol{u}^r \boldsymbol{u_1} b$ where $\boldsymbol{u} = \boldsymbol{x}[i\,..\,j]$,

$$\boldsymbol{u_1} = \boldsymbol{x}[j{+}r(j{-}i){+}1\,..\,j{+}r(j{-}i){+}d_1{-}1] = \boldsymbol{x}[j{+}r(j{-}i){+}1\,..\,j{+}d{-}1]$$

is a prefix of $\boldsymbol{x}[i\,..\,j]$, and $\boldsymbol{x}[i{+}d] \prec \boldsymbol{x}[j{+}1{+}d]$, so that by Lemma 3.14 (b), $\boldsymbol{x}[i\,..\,j{+}1{+}d]$ is Lyndon, contradicting the assumption that $\boldsymbol{x}[i\,..\,j]$ is the maximal Lyndon factor starting at i.

Thus $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)$, as required. □

Lemma 3.16 describes the property of being a maximal Lyndon factor of a string $\boldsymbol{x}$ in terms of relationships between corresponding suffixes.

Lemma 3.16. *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over an alphabet Σ with an ordering $\prec$. A substring $\boldsymbol{x}[i\,..\,j]$ is a maximal Lyndon factor of $\boldsymbol{x}$ with respect to $\prec$ if and only if $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(k)$ for every $k \in i + 1..j$ and either $j = n$ or $\boldsymbol{s_x}(j{+}1) \prec \boldsymbol{s_x}(i)$.*

Proof. Let (A) denote $\{\boldsymbol{x}[i\,..\,j]$ *is a maximal Lyndon factor of* $\boldsymbol{x}\}$ and let (B) denote $\{\boldsymbol{s}(i) \prec \boldsymbol{s}(k)$ *for any* $1 \leq k \leq j$ *and* $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)\}$. Then (A) $\Rightarrow$ (B) follows from Lemma 3.15, so we need to prove that (B) $\Rightarrow$ (A).

Suppose then that (B) holds, and let $\boldsymbol{x}[i\,..\,k]$ be the maximal Lyndon factor of $\boldsymbol{x}$ starting at position i. If $k < j$, then by Lemma 3.15, $\boldsymbol{s}(k{+}1) \prec \boldsymbol{s}(i)$, a contradiction since $k{+}1 \leq j$. If $k > j$, then by Lemma 3.15, $\boldsymbol{s}(i) \prec \boldsymbol{s}(j{+}1)$ because $j{+}1 \leq k$, which again gives us a contradiction. Thus $k = j$ and $\boldsymbol{x}[i\,..\,j]$ is a maximal Lyndon factor of $\boldsymbol{x}$. □

Now we reformulate Lemma 3.16 in terms of the inverse suffix array ISA of $\boldsymbol{x}$ using the relationship that $\boldsymbol{s}(i) \prec s(j) \Longleftrightarrow \mathrm{ISA}[i] < \mathrm{ISA}[j]$, thus yielding Theorem 3.13, as required. Hence the Lyndon array can be computed in a simple three-step algorithm, as shown in Figure 3.2, that executes in $\Theta(n)$ time and uses only one additional array of integers.

3.2.4 NSV*: A Variant of NSV Algorithm

The algorithm uses an approach to the computation of $\lambda_{\boldsymbol{x}}$ that makes use of a variant of the NSV idea. The processing identifies ranges in a single left-to-right scan of the given string $\boldsymbol{x}$, making use of two range comparison routines, COMP and MATCH. COMP compares adjacent individual ranges $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$, returning $\delta_1 \in \{-1, 0, +1\}$ according to whether $\boldsymbol{x_r} < \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} = \boldsymbol{x_{r+1}}$, or $\boldsymbol{x_r} > \boldsymbol{x_{r+1}}$. MATCH similarly returns $\delta_2 \in \{-1, 0, +1\}$ for adjacent *sequences* of ranges. The whole algorithm is based on the idea

encapsulated in Lemma 3.16. It processes the string $\boldsymbol{x}$ from left to right using a stack. At each iteration, the top of the stack (say, j) is compared with the current index (say, i). In particular, we need to compare $\boldsymbol{s_x}(i)$ with $\boldsymbol{s_x}(j)$, where $\boldsymbol{s_x}(i) = \boldsymbol{x}[i..n]$ denotes the suffix of the input string $\boldsymbol{x}$ beginning at position i. As long as $\boldsymbol{s_x}(i) \succeq \boldsymbol{s_x}(j)$, NSV* pushes the current index and continues to the next. When $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(j)$, it pops the stack and puts appropriate values in the corresponding indices of $\lambda_{\boldsymbol{x}}$. As noted above, ranges are employed to expedite these suffix comparisons.

Two auxiliary integer arrays, ***nextequal*** and ***period*** are maintained. Whenever a suffix of a previous range at position j equals the current range at position i, the algorithm assigns $\boldsymbol{nextequal}[j] \leftarrow i$ before i is pushed onto the stack. Then when a later MATCH yields $\delta_2 = 0$, the value of ***period*** — that is, the extent of the following periodicity — may need to be set or adjusted, as shown in the following example:

Example 3.17.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\boldsymbol{x}$	=	a	a	a	b	a	a	b	a	a	b	a	a	b	a	b
nextequal	=	0	5	0	0	8	0	0	11	0	0	0	14	0	0	0
period	=	0	12	0	0	9	0	0	6	0	0	0	4	0	0	0

A straightforward implementation of COMP and MATCH could require a number of letter comparisons equal to the length of the shortest of the two sequences of ranges being matched, so that in the worst case $\mathcal{O}(n^2 \log n)$

time would be required. However, by performing $\Theta(n)$-time preprocessing, we can compare two ranges in $\mathcal{O}(\sigma)$ time, where $\sigma = |\Sigma|$ is the alphabet size. Another approach to this suffix comparison problem is under way, which also achieves run time $\mathcal{O}(n \log n)$ by maintaining a simple data structure requiring $\mathcal{O}(n \log n)$ space.

Chapter 4

Our Novel Algorithm

There are several linear algorithms to sort the suffixes of a string. Currently, the fastest and most efficient one is due to Nong, Zhang, and Chan, [20]. There one can find the references to other linear time algorithms for suffix sorting. An interesting question is whether Lyndon array can be computed without fully sorting all the suffixes.

4.1 Background Information

Weiner [45] was the first one to introduce suffix tree and showed that it can be constructed in $\mathcal{O}(n)$ time for a constant sized alphabet. This construction and its analysis are nontrivial.

A constant sized alphabet is an alphabet of size at most $\boldsymbol{C}$ where $\boldsymbol{C}$ is a fixed constant. A binary alphabet is an example of a constant sized alphabet

if the fixed constant $\boldsymbol{C \geq 2}$. Generally, it takes $\mathcal{O}(m \log(m))$ to sort an alphabet of size m. As Weiner's algorithm requires sorting of the alphabet of the input string, if the alphabet were not of a constant size, the algorithm would in fact execute in $\mathcal{O}(n \log(n))$ time in the worst case.

Since then, considerable efforts had been made in the design of linear time suffix tree constructions, but, prior to Farach, all algorithms developed had been variants of the original approach of Weiner [45]. The main question was how to deal with arbitrarily large alphabets and this problem was successfully addressed by Farach [46]. Below is the general scheme of this approach:

1. Given string $\boldsymbol{X}$ of length $\boldsymbol{n}$. Compute a reduced string $\boldsymbol{Y}$ of size $\frac{1}{2}n$ in at most $C_1 \frac{1}{2}n$ steps for a constant C_1.
2. Make a recursive call to $\boldsymbol{Y}$ that works in $C \frac{1}{2}n$ steps and returns a suffix tree for $\boldsymbol{Y}$.
3. Compute $\boldsymbol{T_o}$, odd suffix tree of $\boldsymbol{X}$ from suffix tree of $\boldsymbol{Y}$ in $C_2 \frac{1}{2}n$ steps for a constant C_2.
4. From $\boldsymbol{T_o}$ compute the even suffix tree $\boldsymbol{T_e}$ of $\boldsymbol{X}$ in $C_3 \leq \frac{1}{2}n$ steps for a constant C_3.
5. Merge the two suffix trees into one in $C_4 \frac{1}{2}n$ steps for a constant C_4.
6. Return the tree.

As long as $C_1 + C_2 + C_3 + C4 \leq C$, then the algorithm works in at most Cn steps. Or, in another words, as long as all steps 1, 3-6 are linear, the overall algorithm is linear.

Suffix tree is a powerful data structure with numerous applications in computational biology, [47] and elsewhere, [48, 49]. It can be constructed in linear time in the length of the string [46]. On the other hand, suffix array [50, 51] is the lexicographically sorted array of the suffixes of a string. It contains much the same information as the suffix tree, in a more implicit form while being a simpler and a more compact data structure for many applications. Due to the more explicit structure and the direct linear-time construction algorithms, theoreticians tend to prefer suffix trees over suffix arrays. Practitioners, on the other hand, often use suffix array, because they are more space-efficient and simpler to implement, [22]. Keeping that in mind, below is a scheme for suffix array construction using Farach's approach.

1. Given string $\boldsymbol{X}$ of length $\boldsymbol{n}$. Divide the suffixes of $\boldsymbol{x}$ to two disjoint groups, G_1 and G_2 in at most $C_1\, n$ steps, for some constant C_1.
2. Compute a reduced string $\boldsymbol{Y}$ of size $\leq \frac{p}{q}n$ for some $1 \leq p < q$ in at most $C_2\, \frac{p}{q}n$ steps, for some constant C_2.
3. Make a recursive call to $\boldsymbol{Y}$ that at most in $C\, \frac{p}{q}n$ steps computes the suffix array for $\boldsymbol{Y}$.
4. Compute suffix array $\boldsymbol{G_1}$ from suffix array of $\boldsymbol{Y}$ in at most $C_3\, \frac{p}{q}n$ steps for some constant C_3.
5. From the suffix array of $\boldsymbol{G_1}$ compute the suffix array of $\boldsymbol{G_2}$ in at most $C_4\, \frac{p}{q}n$ steps for some constant C_4.
6. Merge $\boldsymbol{G_1}$ and $\boldsymbol{G_2}$ in at most $C_5\, n$ steps for some constant C_5.

7. Return the suffix array.

As long as all steps 1-2, 4-5 are linear, the overall algorithm is linear, or, more precisely, as long as $\frac{C_1q+C_2p+C_3p+C_4p+C_5q}{q-p} \leq C$, the algorithm executes in at most $C\,n$ steps.

Our novel algorithm to compute Lyndon array of a string is based on this approach. This approach had been successfully followed by all linear algorithms for suffix sorting to date, except the latest one by Baier[24]. The worst-case complexity of our algorithm is $\Theta(|\boldsymbol{x}|\ \log(|\boldsymbol{x}|))$.

Below are the basic notions and ideas used in our proposed algorithm.

Definition 4.1. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string and let $\mathcal{A}(\boldsymbol{x})$, the string's alphabet be ordered by $\prec$.*

(1) *A position $i \in 1..n$ is **first-ascending** if either*

($1a$) *$i = 1$ and $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i{+}1]$, or*

($1b$) *$\boldsymbol{x}[i-1] \succ \boldsymbol{x}[i] \preceq \boldsymbol{x}[i{+}1]$;*

(2) *A position $i \in 1..n$ is **ascending** if either $\boldsymbol{x}[i{-}1] \preceq \boldsymbol{x}[i]$ or $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i{+}1]$; all other positions are called **descending**.*

(3) *A range $j..k$, $1 \leq i \leq j \leq n$ is an **ascending range** if every position $i \in j..k$ is ascending. It is called a **maximal ascending range** if either $j = 1$ or $j{-}1$ is a descending position,and either $k = n$ or $k{+}1$ is a descending position.*

(4) *A range $j..k$, $1 \leq i \leq j \leq n$ is a **descending range** if every position $i \in j..k$ is descending. It is called a **maximal descending range** if*

$j-1$ is an ascending position, and $k+1$ is an ascending position.

(5) *A range $i..j$ is an **AD range** if there is a $i \leq k \leq j$ such that $i..k$ is a maximal ascending range, $k+1..j$ is either a maximal descending range if $k < j$, and either $j = n$ or $j+1$ is a first-ascending position.*

Classification of ascending and descending positions and ranges depends on the order $\prec$ of the alphabet $\mathcal{A}(\boldsymbol{x})$. 4.1 illustrates the classification of positions.

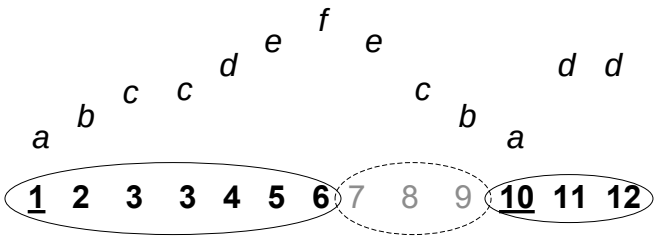

Figure 4.1: Illustration of position classification for a string ***abccdefecbadd***
Ascending positions are indicated in bold, first-ascending positions are underlined, descending positions are in gray, maximal ascending ranges are circled, maximal descending range is circled in a dotted line. The natural order of the symbols is assumed.

Observation 4.2. *1. A string always starts with an ascending range.*

2. An ascending range of length 1 consists of a single first-ascending position and can occurr only at the beginning or the end of a string.

3. A string may not have any descending range; for instance all binary strings consist entirely of ascending ranges.

4.2 τ-pairing

The ***τ-pairing*** consists of partitioning $1..n$ into AD-ranges; each AD-range is then partition into disjoint pairs of positions. If the last position in the AD-range could not be paired, then it is paired with the first-ascending position of the next AD-range, or with the position $n{+}1$ containing \$, if it is the very last position of a string. The τ-pair consists of the pair of symbols at those two positions. The position in which a τ-pair starts is labeled ***black***, the position in which a τ-pair ends is labeled ***white***. The symbol $\tau 2(\boldsymbol{x})$ denotes the set of all τ-pairs of $\boldsymbol{x}$, while $\mathcal{B}(\boldsymbol{x})$ denotes the set of all black positions of $\boldsymbol{x}$ (black and black-and-white), $\mathcal{W}(\boldsymbol{x})$ denotes the set of all white positions of $\boldsymbol{x}$ (white and black-and-white), while $\neg\mathcal{B}(\boldsymbol{x})$ denotes the set of non-black positions, i.e. the pure white ones, and $\neg\mathcal{W}(\boldsymbol{x})$ denotes the set of non-white poisitions, i.e. the pure black ones.

Most of the τ-pairs do not overlap; if two τ-pairs overlap, they overlap in a position which is first-ascending, and which is both labeled as black and white. Moreover, a τ-pair can be involved in at most one overlap. For

an illustration, see Fig. 4.2. In this example, the set of τ-pairs $\tau 2(\boldsymbol{x}) = \{(a,b),(b,a)\}$.

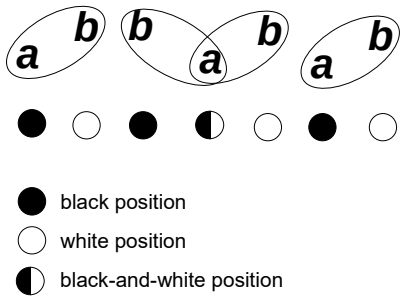

Figure 4.2: Illustration of τ-pairs of a string ***abbabab***
The ovals indicate the τ-pairs

The τ-pairs are ordered lexicographically based on the ordering $\prec$ of $\mathcal{A}(\boldsymbol{x})$: $(a,b) \lhd (c,d)$ iff $a \prec c$ or ($a = c$ and $b \prec d$). $(a,b) \trianglelefteq (c,d)$ iff $(a,b) \lhd (c,d)$ or ($a = c$ and $b = d$).

In our example, the order of $\tau 2(\boldsymbol{x})$ is simple: $(a,b) \lhd (b,a)$.

4.3 τ-alphabet

Let $\boldsymbol{x}$ be a string and let $\mathcal{A}(\boldsymbol{x})$ be ordered by $\prec$. We create a new alphabet $\tau(\mathcal{A}(\boldsymbol{x}))$ by assigning each τ-pair from the set $\tau 2(\boldsymbol{x})$ a distinct symbol not from $\mathcal{A}(\boldsymbol{x})$, i.e $\mathcal{A}(\boldsymbol{x}) \cap \tau(\mathcal{A}(\boldsymbol{x})) = \emptyset$ and $|\tau(\mathcal{A}(\boldsymbol{x}))| = |\tau 2(\boldsymbol{x})|$.

For a τ-pair (a,b), let $r(a,b)$ be the corresponding element from $\tau(\mathcal{A}(\boldsymbol{x}))$, while for $B \in \tau(\mathcal{A}(\boldsymbol{x}))$, $p(B) = (a,b)$ is the corresponding τ-pair, i.e.

$p(r(a,b)) = (a,b)$ and $r(p(B)) = B$. Thus,

$$\tau 2(\boldsymbol{x}) \underset{\text{p}}{\overset{\text{r}}{\rightleftarrows}} \tau(\mathcal{A}(\boldsymbol{x}))$$

The order of the alphabet $\tau(\mathcal{A}(\boldsymbol{x}))$ is induced by the order $\lhd$ of the $\tau 2(\boldsymbol{x})$ and thus we will us the same symbol for the ordering of the alphabet $\tau(\mathcal{A}(\boldsymbol{x}))$ and for the lexicographic ordering of strings over $\tau(\mathcal{A}(\boldsymbol{x}))$, thus for $A, B \in \tau(\mathcal{A}(\boldsymbol{x}))$, $A \lhd B$ iff $p(A) \lhd p(B)$.

4.4 τ-reduction

For a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ with an alphabet $\mathcal{A}(\boldsymbol{x})$ ordered by $\prec$, the string $\tau(\boldsymbol{x})$ is a string over the alphabet $\tau(\mathcal{A}(\boldsymbol{x}))$ that is created by replacing each τ-pair of $\boldsymbol{x}$ with the corresponding symbol from $\tau(\mathcal{A}(\boldsymbol{x}))$.

For any $i \in 1..|\tau(\boldsymbol{x})|$, $b(i) = j$ where j is the starting (and hence black) position in $\boldsymbol{x}$ of the τ-pair corresponding to the symbol in $\tau(\boldsymbol{x})$ at position i, while $t(j)$ assigns each black position of $\boldsymbol{x}$ the position in $\tau(\boldsymbol{x})$ where the corresponding τ-symbol is, i.e. $b(t(j)) = j$ and $t(b(i)) = i$. Thus,

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

$\tau(\boldsymbol{x})$ is defined as concatenation of symbols from $\tau(\mathcal{A}(\boldsymbol{x}))$ assigned to all τ-pairs of $\boldsymbol{x}$, thus, $\mathcal{A}(\tau(\boldsymbol{x})) = \tau(\mathcal{A}(\boldsymbol{x}))$. For a substring $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}$ where i and j are both black positions, $\tau(\boldsymbol{x}[i..j])$ is the concatenation of symbols from $\tau(\mathcal{A}(\boldsymbol{x}))$ assigned to consecutive τ-pairs of $\boldsymbol{x}$ starting with the τ-pair $(\boldsymbol{x}[i], \boldsymbol{x}[i{+}1])$ and ending with the τ-pair $(\boldsymbol{x}[j], \boldsymbol{x}[j{+}1])$.

Example 4.3. *In Fig. 4.3 the τ-reduction of **abbabab** is illustrated. We assign the τ-symbol A to τ-pair (a, b) and the τ-symbol B to τ-pair (b, a). So, $\tau(abbabab) = ABAA$*

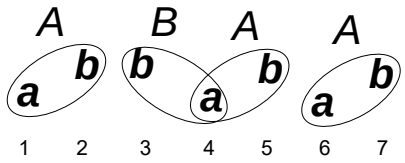

Figure 4.3: Illustration of τ-reduction of a string ***abbabab***

$r =$

τ-*symbol*	A	B
τ-*pair*	(a, b)	(b, a)

$p =$

τ-*pair*	(a, b)	(b, a)
τ-*symbol*	A	B

$b =$

positions of $\tau(\boldsymbol{x})$	*1*	*2*	*3*	*4*
poitions of $\boldsymbol{x}$	*1*	*3*	*4*	*6*

$t =$

positions of $\boldsymbol{x}$	*1*	*3*	*4*	*6*
positions of $\tau(\boldsymbol{x})$	*1*	*2*	*3*	*4*

The most important properties of τ-reduction is that

(a) $\frac{1}{2}|\boldsymbol{x}| \leq |\tau(\boldsymbol{x})| \leq \frac{2}{3}|\boldsymbol{x}|$

Proof. There are two extremes. One, when all τ-pairs are disjoint. Then $|\tau(\boldsymbol{x})| = \frac{1}{2}|\boldsymbol{x}|$ as there are $\frac{1}{2}|\boldsymbol{x}|$ such τ-pairs if $|\boldsymbol{x}|$ is even, or $1+\frac{1}{2}|\boldsymbol{x}|$ of $|\boldsymbol{x}|$ is odd. The other extreme is when every τ-pair intersects with some other τ-pair. Since each τ-pair intersects with exactly one other τ-pair and in exactly one position, two intersecting τ-pairs form a unique triple of positions of $\boldsymbol{x}$, and these triples are mutually disjoint, and so there are at most $\frac{1}{3}|\boldsymbol{x}|$ such triples and hence there are $\frac{2}{3}|\boldsymbol{x}|$ such intersecting τ-pairs. The value of $|\tau(\boldsymbol{x})|$ must lie between these two extremes. □

Example 4.4. *Let $\boldsymbol{x}$ - bab bab bab consists of τ-pairs (underlined) that intersect:* $\underline{ba}\underline{b}\,\underline{ba}\underline{b}\,\underline{ba}\underline{b}$

$\tau(abbabab) = ABAA$, *note that* $|\boldsymbol{x}| = 7$ *and* $|\tau(\boldsymbol{x})| = 4$, *so* $\frac{2}{3}|\boldsymbol{x}| = \frac{2}{3}7 = \frac{14}{3} > \frac{12}{3} = 4 = |\tau(\boldsymbol{x})| > 3.5 = \frac{1}{2}|\boldsymbol{x}|$.

(b) Any maximal Lyndon factor of $\tau(\boldsymbol{x})$ corresponds in a unique way to a maximal Lyndon factor of $\boldsymbol{x}$.

(c) The Lyndon array of $\tau(\boldsymbol{x})$ can thus be extended to a partially filled Lyndon array of $\boldsymbol{x}$.

We are aiming at establishing that τ-reduction preserves certain maximal Lyndon substrings, so we need to establish first a relationship between Lyndon substrings and suffixes of a string and then between maximal Lyndon substrings and suffixes.

Fact 4.5. *$\boldsymbol{x}[i..j]$ is a Lyndon susbstring of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ if and only if $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ for any $i < k \leq j$.*

The relationship between maximal Lyndon susbstrings and the suffixes of a string was first introduced in [23] and it is quite natural:

Lemma 4.6. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string and let its alphabet $\mathcal{A}(\boldsymbol{x})$ be ordered by $\prec$. For any $1 \leq i \leq j \leq n$, the substring $\boldsymbol{x}[i..j]$ is a maximal Lyndon susbstring of $\boldsymbol{x}$ if and only if $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ for any $i < k \leq j$ and either $j = n$ or $\boldsymbol{x}[j+1..n] \prec \boldsymbol{x}[i..n]$.*

First we show that τ-reduction preserves relationships of certain suffixes of $\boldsymbol{x}$.

Lemma 4.7. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \leq i \leq j \leq n$. If i and j are both black positions, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$ implies $\tau(\boldsymbol{x})[t(i)..m] \lhd \tau(\boldsymbol{x})[t(j)..m]$.*

Proof. Since i and j are both blackpositions, both $t(i)$ and $t(j)$ are defined . Let $i_1 = t(i)$ and $j_1 = t(j)$.

- Case when $\boldsymbol{x}[i..n]$ is a prefix of $\boldsymbol{x}[j..n]$.

  Then $j < i$. Moreover, $\boldsymbol{x}[j..j{+}n{-}i{+}1] = \boldsymbol{x}[i..n]$ as $\boldsymbol{x}[i..n]$ is a border of $\boldsymbol{x}[j..n]$. If $j{+}n{-}i{+}1$ is a black position, then $\tau(\boldsymbol{x})[t(j)..t(j{+}n{-}i){+}1] = \tau(\boldsymbol{x})[t(i)..m{-}1]$ and $\boldsymbol{x}[j{+}n{-}i{+}1]$ is τ-paired with $\boldsymbol{x}[j{+}n{-}i{+}2]$ while $\boldsymbol{x}[n]$ is τ-paired with \$, and so $\tau(\boldsymbol{x})[t(i)..m] \lhd \tau(\boldsymbol{x})[t(j)..m]$.

  If $j{+}n{-}i{+}1$ is a white position, then $\tau(x)[t(i)..m] = \tau(\boldsymbol{x})[t(j)..t(j{+}n{-}i)]$, and so $\tau(\boldsymbol{x})[t(i)..m]$ is a prefix of $\tau(\boldsymbol{x})[t(j)..m]$, therefore $\tau(\boldsymbol{x})[t(i)..m] \lhd \tau(\boldsymbol{x})[t(j)..m]$.

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

  Then $\tau(\boldsymbol{x})[i_1] = p(\boldsymbol{x}[i], \boldsymbol{x}[i{+}1])$ and $\tau(\boldsymbol{x})[j_1] = p(\boldsymbol{x}[j], \boldsymbol{x}[j{+}1])$. Since $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$, we have $\tau(\boldsymbol{x})[i_1] \lhd \tau(\boldsymbol{x})[j_1]$ and so $\tau(\boldsymbol{x})[t(i)..m] \lhd \tau(\boldsymbol{x})[t(j)..m]$.

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

  (a) Case $i{+}\ell{-}1$ is white.

  Since $\boldsymbol{x}[i..i+\ell-1] = \boldsymbol{x}[j..j+\ell-1]$, it follows that $j+\ell-1$ is also white. Hence both $i+\ell-2$ and $j+\ell-2$ are black and $\tau(\boldsymbol{x})[i_1..t(i{+}\ell{-}2)] = \tau(\boldsymbol{x})[j_1..t(j{+}\ell{-}2)]$.

  $$\tau(\boldsymbol{x})[t(i{+}\ell{-}2){+}1] = \begin{cases} p(x[i{+}\ell{-}1], \boldsymbol{x}[i{+}\ell]) & \textit{if } i{+}\ell{-}1 \textit{ is black \& white,} \\ p(\boldsymbol{x}[i{+}\ell], \boldsymbol{x}[i{+}\ell{+}1]) & \textit{if } i{+}\ell{-}1 \textit{ is pure white} \end{cases}$$

  and

  $$\tau(\boldsymbol{x})[t(j{+}\ell{-}2){+}1] = \begin{cases} p(x[j{+}\ell{-}1], \boldsymbol{x}[i{+}\ell]) & \textit{if } j{+}\ell{-}1 \textit{ is black \& white,} \\ p(\boldsymbol{x}[j{+}\ell], \boldsymbol{x}[i{+}\ell{+}1]) & \textit{if } j{+}\ell{-}1 \textit{ is pure white.} \end{cases}$$

  (α) $\tau(\boldsymbol{x})[t(i{+}\ell{-}2){+}1] = p(\boldsymbol{x}[i{+}\ell{-}1], \boldsymbol{x}[i{+}\ell])$ and $\tau(\boldsymbol{x})[t(j{+}\ell{-}2){+}1] =$

$p(\boldsymbol{x}[j+\ell-1], \boldsymbol{x}[j+\ell])$.

Since $\boldsymbol{x}[i+\ell-1] = \boldsymbol{x}[j+\ell-1]$ and $\boldsymbol{x}[i+\ell] \prec \boldsymbol{x}[j+\ell]$, we get $\tau(\boldsymbol{x})[t(i+\ell-2)+1] \lhd \tau(\boldsymbol{x})[t(j+\ell-2)+1]$, and so $\tau(\boldsymbol{x})[i_1..n] \lhd \tau(\boldsymbol{x})[j_1..n]$.

(β) $\tau(\boldsymbol{x})[t(i+\ell-2)+1] = p(\boldsymbol{x}[i+\ell-1], \boldsymbol{x}[i+\ell])$ and $\tau(\boldsymbol{x})[t(j+\ell-2)+1] = p(\boldsymbol{x}[j+\ell], \boldsymbol{x}[j+\ell+1])$.

Since $i+\ell-1$ is black & white, we have $\boldsymbol{x}[i+\ell-2] \succ \boldsymbol{x}[i+\ell-1] \prec \boldsymbol{x}[i+\ell]$. Since $j+\ell-1$ is pure white and $\boldsymbol{x}[i+\ell-2] = \boldsymbol{x}[j+\ell-2]$ and $\boldsymbol{x}[i+\ell-1] = \boldsymbol{x}[j+\ell-1]$, we have $\boldsymbol{x}[j+\ell-2] \succ \boldsymbol{x}[j+\ell-1] \succ \boldsymbol{x}[j+\ell]$. Thus, $\tau(\boldsymbol{x})[t(i+\ell-2)+1] \lhd \tau(\boldsymbol{x})[t(j+\ell-2)+1]$ and so $\tau(\boldsymbol{x})[i_1..n] \lhd \tau(\boldsymbol{x})[j_1..n]$.

(γ) $\tau(\boldsymbol{x})[t(i+\ell-2)+1] = p(\boldsymbol{x}[i+\ell], \boldsymbol{x}[i+\ell+1])$ and $\tau(\boldsymbol{x})[t(j+\ell-2)+1] = p(\boldsymbol{x}[j+\ell+1], \boldsymbol{x}[j+\ell])$.

Since $j+\ell-1$ is black & white, we have $\boldsymbol{x}[j+\ell-2] \succ \boldsymbol{x}[j+\ell-1] \prec \boldsymbol{x}[j+\ell]$. Since $i+\ell-1$ is pure white and $\boldsymbol{x}[i+\ell-2] = \boldsymbol{x}[j+\ell-2]$ and $\boldsymbol{x}[i+\ell-1] = \boldsymbol{x}[j+\ell-1]$ and, we have $\boldsymbol{x}[i+\ell-2] \succ \boldsymbol{x}[i+\ell-1] \succ \boldsymbol{x}[i+\ell]$, and so $\boldsymbol{x}[j+\ell] \succ \boldsymbol{x}[j+\ell-1] = \boldsymbol{x}[i+\ell-1] \succ \boldsymbol{x}[i+\ell]$, a contradiction. This case cannot happen.

(δ) $\tau(\boldsymbol{x})[t(i+\ell-2)+1] = p(\boldsymbol{x}[i+\ell], \boldsymbol{x}[i+\ell+1])$ and $\tau(\boldsymbol{x})[t(j+\ell-2)+1] = p(x[j+\ell], \boldsymbol{x}[j+\ell+1])$.

Since $\boldsymbol{x}[i+\ell] \prec \boldsymbol{x}[j+\ell]$, we get $\tau(\boldsymbol{x})[t(i+\ell-2)+1] \lhd \tau(\boldsymbol{x})[t(j+\ell-2)+1]$ and so $\tau(\boldsymbol{x})[i_1..n] \lhd \tau(\boldsymbol{x})[j_1..n]$.

(b) Case $i+\ell-1$ is not white.

Since $\boldsymbol{x}[i..i+\ell-1] = \boldsymbol{x}[j..j+\ell-1]$, it follows that $j+\ell-1$ is also not white. Thus both are pure black and so $\tau(\boldsymbol{x})[i_1..t(i+\ell-2)] = \tau(\boldsymbol{x})[j_1..t(j+\ell-2)]$. $\tau(\boldsymbol{x})[t(i+\ell-2)+1] = p(\boldsymbol{x}[i+\ell-1], x[i+\ell])$ and $\tau(\boldsymbol{x})[t(j+\ell-2)+1] = p(x[j+\ell-1], \boldsymbol{x}[j+\ell])$. Therefore, $\tau(\boldsymbol{x})[t(i+\ell-2)+1] \lhd \tau(\boldsymbol{x})[t(j+\ell-2)+1]$ and so $\tau(\boldsymbol{x})[i_1..n] \lhd \tau(\boldsymbol{x})[j_1..n]$.

□

Lemma 4.8 shows that τ-reduction preserves certain maximal Lyndon substrings of $\boldsymbol{x}$.

Lemma 4.8. *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 maximal Lyndon susbtsring of $\boldsymbol{x}$, and let i be a black position. Then*

$$\begin{cases} \tau(\boldsymbol{x})[t(i)..t(j)] \text{ is Lyndon} & \text{if } j \text{ is black} \\ \tau(\boldsymbol{x})[t(i)..t(j-1)] \text{ is Lyndon} & \text{if } j \text{ is not black.} \end{cases}$$

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 4.5. Hence, $\tau(\boldsymbol{x})[t(i)..m] \lhd \tau(\boldsymbol{x})[t(k)..m]$ by Lemma 4.7. Therefore, $\tau(\boldsymbol{x})[t(i)..t(j)]$ is Lyndon.

Now assume that j is not black.

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 of the string.

Theorem 4.9. *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 of $\boldsymbol{x}$, and let i be a black position.*

$$\textit{Then} \begin{cases} \tau(\boldsymbol{x})[t(i)..t(j)] \textit{ is a maximal Lyndon substring} & \textit{if } j \textit{ is black} \\ \tau(\boldsymbol{x})[t(i)..t(j-1)] \textit{ is a maximal Lyndon substring} & \textit{if } j \textit{ is not black.} \end{cases}$$

Proof. By the Lemma 4.8, we know that in both cases the respective substring is Lyndon. Thus, we only need to show its maximality.
Since $\boldsymbol{x}[i..j]$ is maximal, $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$. We also know that $\boldsymbol{x}[i] \prec \boldsymbol{x}[j]$.

- First assume that j is a black position.

  Hence we want to show that $\tau(\boldsymbol{x})[t(i)..t(j)]$ is a maximal Lyndon substring. Since j is black, $(\boldsymbol{x}[j], \boldsymbol{x}[j{+}1])$ is a τ-pair. Since $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$, $\boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[i]$.

  (*a*) If $\boldsymbol{x}[j] \succ \boldsymbol{x}[i] \succeq \boldsymbol{x}[j{+}1] \preceq \boldsymbol{x}[j{+}2]$, then $j{+}1$ is black and $t(j){+}1 = t(j{+}1)$ and so $\tau(\boldsymbol{x})[t(j{+}1)..m] \lhd \tau(\boldsymbol{x})[t(i)..m]$ and so $\tau(\boldsymbol{x})[t(j){+}1..m] \lhd \tau(\boldsymbol{x})[t(i)..m]$, giving the maximality of $\tau(\boldsymbol{x})[t(i)..t+\tau(j)]$.

  (*b*) If $\boldsymbol{x}[j] \succ \boldsymbol{x}[i] \succeq \boldsymbol{x}[j{+}1] \succ \boldsymbol{x}[j{+}2]$, then $\tau(\boldsymbol{x})[t(i)] = p(\boldsymbol{x}[i], \boldsymbol{x}[i{+}1])$, and $\tau(\boldsymbol{x})[t(j)] = p(\boldsymbol{x}[j], \boldsymbol{x}[j+1])$, and $\tau(\boldsymbol{x})[t(j)+1] = p(\boldsymbol{x}[j+2], \boldsymbol{x}[j+3])$. Thus, $\tau(\boldsymbol{x})[t(j)+1] \lhd \tau(\boldsymbol{x})[t(i)]$, and so $\tau(\boldsymbol{x})[t(j){+}1..m] \lhd \tau(\boldsymbol{x})[t(i)..m]$, giving the maximality of $\tau(\boldsymbol{x})[t(i)..t(j)]$.

- Assume that j is not black.

  Hence we want to show that $\tau(\boldsymbol{x})[t(i)..t(j-1)]$ is a maximal Lyndon substring. $j-1$ and $j+1$ are both black and $t(j-1)+1 = t(j+1)$. Hence, $\tau(\boldsymbol{x})[t(j+1)..m] \lhd \tau(\boldsymbol{x})[t(i)..m]$ and so $\tau(\boldsymbol{x})[t(j-1)+1..m] \lhd \tau(\boldsymbol{x})[t(i)..m]$, giving the maximality of $\tau(\boldsymbol{x})[t(i)..t(j-1)]$.

□

Note that Theorem 4.9 gives a procedure for computing partially filled Lyndon array of $\boldsymbol{x}$ from Lyndon array of $\boldsymbol{y}$:

Theorem 4.10. *Let $\lambda_{\boldsymbol{y}}[1..m]$ be the Lyndon array of $\boldsymbol{y} = \tau(\boldsymbol{x})$, where $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let all values of $\lambda_{\boldsymbol{y}}$ be known. Let $\lambda_{\boldsymbol{x}}[1..n]$ be the Lyndon array of $\boldsymbol{x}$. For any black $i \in 1..n$,*

$$\lambda_{\boldsymbol{x}}[i] = \begin{cases} j-i+1 & \text{if } j \text{ is black} \\ j-i & \text{otherwise} \end{cases} \quad \text{where } j = b\big(\lambda_{\boldsymbol{y}}[t(i)]+t(i)-1\big)$$

These properties indicate the structure of our algorithm:

1. compute τ-reduction of $\boldsymbol{x}$ of size $\leq 2/3n$.
2. Make a recursive call that returns the Lyndon array of $\tau(\boldsymbol{x})$.
3. Expand the Lyndon array of $\tau(\boldsymbol{x})$ to a partial Lyndon array of $\boldsymbol{x}$.
4. Compute the missing values in the partial Lyndon array.
5. Return the fully computed Lyndon array of $\boldsymbol{x}$.

Since steps 1, 3, and 5 are linear and since the reduction of the size of $\boldsymbol{x}$ is at least 1/3, this scheme's overall complexity really depends on the complexity of step 4, which, at this point we can do at best in $\Theta(n \log(n))$.

A challenging problem is how to compute the missing values efficiently. It is due to this step of the algorithm (step 4.), that the overall complexity so far is $\Theta(n \log(n))$. When the τ-pairs to not overlap, then each missing value can be computed in constant time, and hence the overall algorithm performs in linear time. Have a look at the following example:

Example 4.11. *Let* $\boldsymbol{x}$*=abababab*

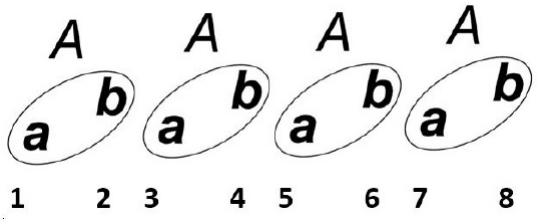

Figure 4.4: Illustration of τ-reduction of a string ***abababab***

In Fig. 4.4 the τ-reduction of ***abababab*** *is illustrated. We assign the τ-symbol A to τ-pair (a, b). So, $\tau(abababab) = AAAA$. $\lambda_{\tau(x)} = \{1, 1, 1, 1\}$. $\lambda_x = \{2, \_, 2, \_, 2, \_, 2, \_\}$ from the partially filled Lyndon array we can compute $\lambda_x = \{2, 1, 2, 1, 2, 1, 2, 1\}$ in linear time.*

Other "easy" cases where the algorithm tends to be linear include:

1. Strings with all ascending ranges.
2. Strings with all descending ranges.

4.5 How to compute the Missing Values

In the above given algorithm the most furtive step is to find the missing values in the partially filled Lyndon array for the given string $\boldsymbol{x}$ of length $\boldsymbol{n}$. As the τ-reduction gives us a string of length between $1/2n$ and $2/3n$, we end up in having between $1/2n$ and $1/3n$ of missing values in the Lyndon array.

Below is a description as how to compute the missing values which gives the worst case complexity. Missing values are always computed starting from the right hand side, i.e. the end of the partially filled Lyndon array. So, at each position i being processed, we assume to have the Lyndon array entries for all j, $i+1 \leq j \leq n$. Also note, that for an index i we are trying to compute the missing value $\lambda[i]$, we have available $\lambda[i-1]$.

4.5.1 Cases when determining $\lambda[i]$ takes a constant time

- If $\boldsymbol{x}[i] \succ \boldsymbol{x}[i+1]$, then $\lambda[i] = 1$.

  Since we are assuming that $\boldsymbol{x}[n+1] = \$$*, the latter case includes the case when* $i = n$*.*

- If $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i+1]$ and either $\mathcal{L}[i+1] = n$ or $\mathcal{L}[i+1] = \mathcal{L}[i-1]$, then $\lambda[i] = \lambda[i+1] + 1$.

  Note that $\boldsymbol{x}[i..\mathcal{L}[i+1]]$ is Lyndon because $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i+1]$, so the real question is if it is maximal. Since $\mathcal{L}[i+1] \leq \mathcal{L}[i] \leq \mathcal{L}[i-1]$, then $\mathcal{L}[i-1] = \mathcal{L}[i] = \mathcal{L}[i+1]$.

4.5.2 Cases when determining $\lambda[i]$ may take $\mathcal{O}(i)$ steps

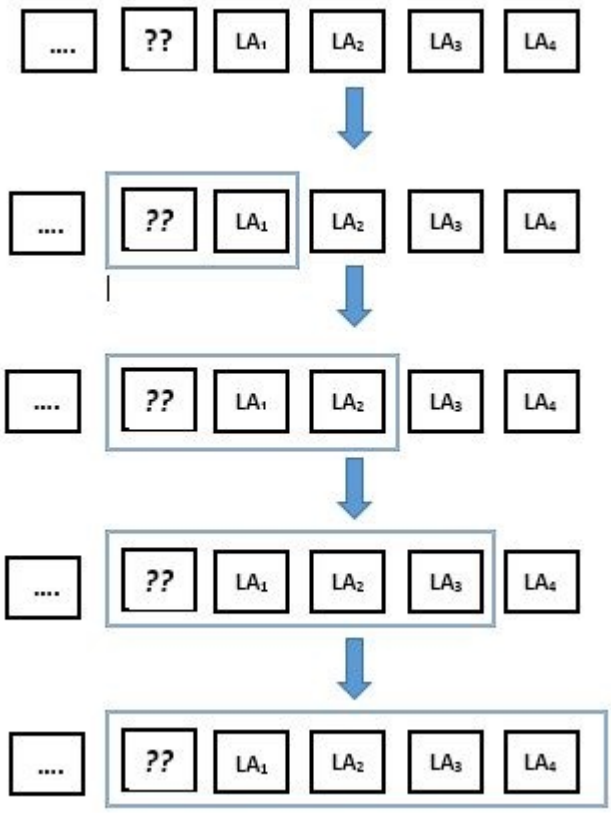

Figure 4.5: Illustration of computing the missing value at position ??

This situation arises when $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i+1]$, $\mathcal{L}[i+1] < n$ and $\mathcal{L}[i-1] > \mathcal{L}[i+1]$. Again, $\boldsymbol{x}[i..\mathcal{L}[i+1]]$ is Lyndon, but it may not be maximal. Consider a situation when the unknown value $\lambda[i]$ is followed by at least two maximal Lyndon factors denoted $LA_1 = \boldsymbol{x}[j_1..\mathcal{L}[j_1]]$, ..., $LA_m = \boldsymbol{x}[j_m..\mathcal{L}[j_m]]$. Note that $LA_1 \succeq LA_2 \succeq LA_3 \succeq \cdots \succeq LA_m$ and that $j_1 = i+1$. Then the Lyndon factor $\boldsymbol{x}[i..\mathcal{L}[i+1]]$ must be compared to LA_2 in $\min\{\lambda[j_1]+1, \mathcal{L}[j_2]\}$ steps.

If $\boldsymbol{x}[i..\mathcal{L}[j_1]] \succeq \boldsymbol{x}[j_2..\mathcal{L}[j_2]$, we can stop and set $\mathcal{L}[i] = \mathcal{L}[j_1]$. However, if $\boldsymbol{x}[i..\mathcal{L}[j_1]] \prec \boldsymbol{x}[j_2..\mathcal{L}[j_2]$, then $\boldsymbol{x}[i..\mathcal{L}[j_2]]$ is Lyndon and we have to continue. We have to compare $\boldsymbol{x}[i..\mathcal{L}[j_2]]$ to $\boldsymbol{x}[j_3..\mathcal{L}[j_3]]$ to find out whether we need to extend $\boldsymbol{x}[i..\mathcal{L}[j_2]$ or not, possibly all the way to comparison of the Lyndon factor $\boldsymbol{x}[i..\mathcal{L}[j_{m-1}]]$ to $\boldsymbol{x}[j_m..\mathcal{L}[j_m]]$.

Note, that we do not have to go past $\mathcal{L}[i-1]$, so it may limit how far we need to go. So, it is not necessarily $\mathcal{O}(i)$, but it may. Fig. 4.5 illustrates this situation with 4 maximal Lyndon factors following $\boldsymbol{x}[i]$.

The above analyzes suggests that the complexity of filling of the missing values may in fact be $\mathcal{O}(n^2)$. Before we prove that it is $\Theta(n \log(n))$, we go over a few examples how the filling of the missing values work.

4.6 Working Examples

4.6.1 Example 1: Simple Case

Consider $\boldsymbol{x} = abbabab$

τ(abbabab) = ABAA with τ- symbol $\boldsymbol{A}$ representing τ-pair (a,b)

and τ- symbol $\boldsymbol{B}$ representing τ-pair (b,a)

$\boldsymbol{\lambda_{\tau(x)}}$ = { 2,1,1,1 }

$\boldsymbol{\lambda_x}$ = { 3, ?, 1, 2, ?, 2, ? }

Now find the missing values:

- We start with $i = n$: case 4.5.1 applies here, and so $\lambda[7] = 1$

- We continue with $i = 5$: case 4.5.1 again applies here as $\boldsymbol{x}[5] = b \succ \boldsymbol{x}[6] = a$ and so $\lambda[5] = 1$
- We continue with $i = 2$: case 4.5.2 applies here as $\boldsymbol{x}[2] = \boldsymbol{x}[3]$. Since $\mathcal{L}[1] = 3$, we set $\lambda[2] = 1$

Thus, the complete Lyndon array for the given string $\boldsymbol{x}$ is { 3, 1, 1, 2, 1, 2, 1 }.

4.6.2 Example 2: Complex Case

Consider $\boldsymbol{x} = baababab$

the τ-pairing is $(b,a)(a,a)(b,b)(a,b)(a,b)$. We assign $(a,a) = A$, $(a,b) = B$, $(b,a) = C$, and $(b,b) = D$. Thus, $\tau(baababab) = CADBB$. The recursive call will return the Lyndon array of $CADBB$, i.e. { 1, 4, 1, 1, 1 } which is expanded to the partial Lyndon array of $\boldsymbol{x}$ { 1, 8, ?, 1, ?, 2, ?, 2, ? }

By the case 4.5.1 as $\boldsymbol{x}[i] \succ \boldsymbol{x}[i+1]$, the missing values $\lambda[9]$, $\lambda[7]$, $\lambda[5]$ are all 1, i.e. we have { 1, 8, ?, 1, 1, 2, 1, 2, 1 }.

The hard case 4.5.2 is for $i = 3$: $\Big[a\boldsymbol{a}[b][b][ab][ab]\Big]$ where the brackets indicate the maximal Lyndon factors. Since $\boldsymbol{x}[3] = \boldsymbol{a} \preceq \boldsymbol{x}[4] = b$, we can extend $\boldsymbol{x}(a)[b]$ to a Lyndon factor $[\boldsymbol{a}b]$, i.e. we have $\Big[a[\boldsymbol{a}b][b][ab][ab]\Big]$. Now, since $[\boldsymbol{a}b] \preceq [b]$, we can extend $[\boldsymbol{a}b][b]$ to $[\boldsymbol{a}bb]$, i.e. we have $\Big[a[\boldsymbol{a}bb][ab][ab]\Big]$. Now, since $[\boldsymbol{a}bb] \succ [ab]$ we can stop since $[\boldsymbol{a}bb]$ is maximal and hence $\lambda[3] = 3$ corresponding to $[\boldsymbol{a}bb]$.

4.6.3 Example 3: More Complex Case

Consider $\boldsymbol{x} = baaababab$

the τ-pairing is $(b,a)(a,a)(a,b)(a,b)(a,b)$. We assign $(a,a) = A$, $(a,b) = B$, $(b,a) = C$. Thus, $\tau(baaababab) = CABBB$. The recursive call will return the Lyndon array of $CABBB$, i.e. { 1, 4, 1, 1, 1 } which is expanded to the partial Lyndon array of $\boldsymbol{x}$ { 1, 8, ?, 2, ?, 2, ?, 2, ? }

By the case 4.5.1 as $\boldsymbol{x}[i] \succ \boldsymbol{x}[i+1]$, the missing values $\lambda[9]$, $\lambda[7]$, $\lambda[5]$ are all 1, i.e. we have { 1, 8, ?, 1, 1, 2, 1, 2, 1 }.

The hard case 4.5.2 is for $i = 3$: $\Big[a\boldsymbol{a}[ab][ab][ab]\Big]$ where the brackets indicate the maximal Lyndon factors. Since $\boldsymbol{x}[3] = \boldsymbol{a} \preceq \boldsymbol{x}[4] = a$, we can extend $\boldsymbol{x}(a)[ab]$ to a Lyndon factor $[\boldsymbol{a}ab]$, i.e. we have $\Big[a[\boldsymbol{a}ab][ab][ab]\Big]$. Now, since $[\boldsymbol{a}ab] \preceq [ab]$, we can extend $[\boldsymbol{a}ab][ab]$ to $[\boldsymbol{a}abab]$, i.e. we have $\Big[a[\boldsymbol{a}abab][ab]\Big]$. Now, since $[\boldsymbol{a}abab] \prec [ab]$ we can extend $[\boldsymbol{a}abab]$ to $[\boldsymbol{a}ababab]$ and we can stop. Hence $\boldsymbol{a}ababab$ is maximal and $\lambda[3] = 7$ corresponding to $[\boldsymbol{a}ababab]$, i.e. we end with $\Big[a[\boldsymbol{a}ababab]\Big]$. Note that this is a case when $\mathcal{L}[i-1] = n$ and so it does not help to limit the process.

4.7 The complexity of the algorithm

In the section 4.5 we described how the missing values are computed and went over a few examples in section 4.6. Since in the worst case computing of a missing value for i can take $\mathcal{O}(i)$, it seems that computing all $\mathcal{O}(n)$ missing values (there are between $1/2n$ to $1/3n$ missing values) may take $\mathcal{O}(n^2)$ steps.

In this section we shall establish that in fact it takes $\Theta(n \log(n))$ steps in the worst case to compute all the missing values.

When in case 4.5.2 we are comparing $\boldsymbol{x}[i..\mathcal{L}[j_k]]$ with $\boldsymbol{x}[j_{k+1}..\mathcal{L}[j_{k+1}]]$ we have to "step" on some (or may be all) of the indices of $j_{k+1}..\mathcal{L}[j_{k+1}]$. Thus, in order to "step" on an index j for the first time, we must be dealing with a missing value at an index i_1 that is at least as far from the beginning of LA_1 as i is from the beginning of LA_1, where LA_1 is the smallest maximal Lyndon factor containing i (let r be the distance): $.....i_1 \underbrace{\ldots}_{r}[\underbrace{\ldots}_{r} j.....].....$ To step on j for the second time, we must be dealing with a missing value for an index i_2 that is a far from the beginning of LA_2, which is the next smallest maximal Lyndon factor containing j, etc. Thus, $n \geq 2^r k \geq 2^r$ and so $r \leq \log(n)$, i.e. we "step" on each index at most $\mathcal{O}(\log(n))$ times giving the overall complexity as $\mathcal{O}(n \log(n))$ steps. Note, that this is a natural consequence of the fact that maximal Lyndon factors must be nested.

In order to establish that the complexity is $\Theta(n \log(n))$, we give an example of a string that will force that many steps.

Let $\boldsymbol{u}_1$ be a Lyndon string over an alphabet Σ_1. Let $a_1 \notin \Sigma_1$ and define $a_1 \prec b$ for any $b \in \Sigma_1$. Let $\boldsymbol{u}_2 = a_1 a_1 \boldsymbol{u}_1 \boldsymbol{u}_1$ and let $\Sigma_2 = \{a_1\} \cup \Sigma_1$, ..., $\boldsymbol{u}_{i+1} = a_i a_i \boldsymbol{u}_i \boldsymbol{u}_i$ and $\Sigma_{i+1} = \{a_i\} \cup \Sigma_i$, for $i \leq k$. Let $\boldsymbol{x} = a_k a_k \boldsymbol{u}_k \boldsymbol{u}_k$.
Note that $|\boldsymbol{u}_2| = 2|\boldsymbol{u}_1| + 2$, $|\boldsymbol{u}_3| = 2|\boldsymbol{u}_2| + 2 = 2^2|\boldsymbol{u}_1| + 2 * 2$, $|\boldsymbol{u}_4| = 2|\boldsymbol{u}_3| + 2 = 2^3|\boldsymbol{u}_1| + 2 * 3$, ..., $|\boldsymbol{u}_k| = 2|\boldsymbol{u}_{k-1}| + 2 = 2^{k-1}|\boldsymbol{u}_1| + 2 * (k-1)$, and so $|\boldsymbol{x}| = 2^k|\boldsymbol{u}_1| + 2 * k$. Every second occurrence of a_i is an index of a missing value and every processing of it is the case 4.5.2 forcing a comparison of

$[a_i \boldsymbol{u}_i]$ with $[\boldsymbol{u}_i]$, i.e. we are guaranteed to "step" on the indices of the second occurrence $\boldsymbol{u}_1$ k-times and $k = \log(\frac{|\boldsymbol{x}|-2*k}{|\boldsymbol{u}_1}$). Thus, the complexity of the process for filling in the missing values is $\Theta(n \log(n))$.

Chapter 5

Linear Time Suffix Sorting By Baier

Suffix arrays have been used as significant data structure in broad spectrum of applications including data compression, indexing, retrieval , sorting and text processing. Computation of suffix array for a given string is not an easy task. [52] states the following criteria to be fulfilled by every suffix array construction algorithm:

- Minimal asymptotic runtime: linear time complexity.
- Fast runtime in practice, tested on real world data.
- Minimal space requirements, space usage for the suffix array and the text itself in an optimal way.

Although the paper was published in 2007, no suffix array construction

algorithm truly meets the above mentioned criteria. In the succeeding sections we will review the recent work in SACAs (Suffix Array Construction Algorithm) by Baier.

5.1 History

Started as an open question by Weiner [45] *Can suffix trees be constructed in linear time?*, Farach [46] settled the open problem by building suffix trees for integer alphabets in linear time. After the introduction of suffix array by Manber and Myes in 1993, most of the attention had been diverted to them.

Recently the research became more focused on time and space efficient algorithms for the construction and sorting of suffix arrays due to their increasing demand in the large scale applications like web searching and genome database, where the magnitude of huge datasets is measured often in billions of characters. The fastest linear SACA among all the latest results obtained so far is the KA algorithm from Aluru and Ko, [21].

Table 5.1 below summarizes some of the significant work done in the area so far.

Year	Researcher	Achievement
1973 1996 1997	Weiner Grossi and Italiano Gusfield	Introduced and used suffix trees as a powerful data structure with numerous applications.
1973 1976 1995 1997 2000	Weiner McCreight Ukkonen Farach Farach-Colton et al.	All of them worked to prove Suffix trees can be constructed in linear time in the length of the string.
1992 1993 1994 2002	Gonnet et al. Manber and Myers Burrows and Wheeler Abouelhoda et al.	Introduced Suffix array as a space efficient substitute for suffix trees.
2003 2003 2005 2005	Karkkainen and Sanders Kim et al. Ko and Aluru Joong Chae Na	All worked towards developing linear-time algorithms for suffix array construction inspired by Farach's approach.
2009	Wai Hong Chan Ge Nong Sen Zhang	Invented two new linear time suffix sorting algorithms using the induced sorting principles.
2011	C Hohlweg, C Reutenauer	Used NSV to sort suffixes in linear time.

Table 5.1: Evolution of Linear Time Suffix Sorting Algorithms

5.2 Linear-time Suffix Sorting

I believe, that the biggest step in suffix sorting history was made by Nong et al. in 2009. They invented two new algorithms using the induced sorting principle [21]. One of those algorithms, called SA-IS [20], was able

to outperform most of the superlinear algorithms that were known to "work good in practice'', while guaranteeing asymptotic linear runtime and almost optimal space requirements. As a consequence, SA-IS is the currently fastest known linear-time SACA that is able to fulfill almost all of the requirements noted by Puglisi et al. [52], and stays the candidate to beat.

In 2015, Baier presented a linear-time algorithm that directly sorts the suffixes of a string, the first such algorithm that is not recursive. Since his algorithm as a first step groups the suffixes according to the first letter with the groups being in the lexicographic order, his algorithm is not truly linear. For a general alphabet Σ of the input string that would need to be sorted first, the complexity is $\mathcal{O}(n \log(|\Sigma|))$. But for strings over slowly growing alphabets (e.g. if $\Sigma \subset \{1, 2, 3, ...\}$, and $\max_\Sigma \leq |\boldsymbol{x}|$), they can be sorted in $\mathcal{O}(n)$ time in a simple bucket sort and so the overall complexity is $\mathcal{O}(n)$.

In fact, his approach implicitly gives quite a bit more: it includes an elementary algorithm for computing what turns out to be a partially sorted version of the Lyndon array, and then shows how this can be used to sort the suffixes, [24].

Baier introduces the concept of suffix group $\mathcal{G}$,which is the main ingredient in his algorithm.

5.2.1 Suffix Groups

A *suffix group* $\mathcal{G}$ with a group context $\boldsymbol{u}$ is defined as a set $\mathcal{G} \subseteq 1..n$ such that $\boldsymbol{u}$ is a prefix of all suffixes in $\mathcal{G}$, i.e. if $i \in \mathcal{G}$, then $\boldsymbol{u}$ is a prefix of $\boldsymbol{x}[i..n]$.

The index set of a given string $\boldsymbol{x}$ is divided into suffix groups $\mathcal{G}_1$, ..., $\mathcal{G}_m$ such that all the group prefixes of $\mathcal{G}_1$, ..., $\mathcal{G}_m$ differ. For any $i, j \in 1..m$, $\mathcal{G}_i < \mathcal{G}_j$ if the group context of $\mathcal{G}_i$ is lexicographically smaller than the group context of $\mathcal{G}_j$. Within such a partition, $group(i)$ identifies the suffix group to which the index $\boldsymbol{i}$ belongs.

5.3 Basic Sorting Principle

Baier's algorithm is divided into two phases:

5.3.1 Phase I

In this phase, suffixes are divided into groups while the group contexts are being sorted. When the process ends, all the suffixes $\boldsymbol{s}(i)$ sharing the same prefix $\boldsymbol{x}[i..\hat{i})$ must be placed in the same group, where $\hat{i}$ is the first index after i so that $\boldsymbol{s}(\hat{i}) \prec \boldsymbol{s}(i)$ and where $\boldsymbol{x}[i..j)$ denotes $\boldsymbol{x}[i..j-1]$. The groups themselves must be ordered by their group contexts at all times. A group $\mathcal{G}'$ is said to be of a lower order than a group $\mathcal{G}''$, if the context of $\mathcal{G}'$ is lexicographically smaller than that of $\mathcal{G}''$.

5.3.2 Phase II

Sorted groups from the last phase are used here to finally sort the suffixes. After the first phase, all the suffixes in a lower order group are lexicographically smaller than all the suffixes from a higher order group, so to achieve

a total sort of all suffixes, it is sufficient to have the suffixes in each group sorted properly. The groups are sorted from the lowest to the highest order, i.e. from left to right. The process starts with the lowest order group, i.e. the group that consists only of a single suffix $\boldsymbol{s}(n) = \$$ and so the process starts with having all the suffixes in the lowest order group in proper order. By the end of the i^{th} iteration, all suffixes within the i^{th} group should be sorted. The order of the lower order groups helps determine the order in the group being ordered and the algorithm uses the greedy approach for that. Baier's implemented this phase using a kind of dynamic programming approach akin to the group context doubling (i.e. prefix doubling). We will discuss this phase in more details in the next chapter.

5.4 Asymptotic Complexity Of The Algorithm

Main tasks of the algorithm based on the above mentioned sorting principle are:

- Build initial groups.
- Iterate them in descending group order.
- Compute previous smaller suffixes.
- Rearrange them.

To implement the algorithm, Baier used five arrays of size $\boldsymbol{n}$. The initial setup of those arrays can be performed in linear time using *bucket sort* (if the alphabet of the string allows it, as we discussed in 5.2, otherwise it would

require $\mathcal{O}(n \log(n))$ time) and further iterations using a character count table which requires $\mathcal{O}(n)$ time. After then, the next biggest step is group sorting which also happens in $\mathcal{O}(n)$. All remaining operations which required sorting within the group $\mathcal{G}$ takes $\mathcal{O}(\mathcal{G})$ time.

This leads to the overall complexity of both the two phases to be $\mathcal{O}(n)$. The overall space complexity is $\mathcal{O}(n)$ words.These are the proposed complexities, when implemented for a real world problem it leads to a number of downsides. The actual implementation turned out to be very complex and not too efficient in comparison to other SACAs.

Table 5.2 below shows the performance comparison of Baier's algorithm with the existing SACAs.

Algorithm	**Resource**	**Runtime**	**Extra Working Space**
divsufsort	[53]	O(nlogn)	O(1)
SA-IS	[54]	O(n)	O(logn) + max2n
KA	[55]	O(n)	O(logn) + 4.16n
DC3	[56]	O(n)	O(logn) + max24n
GSACA	[57]	O(n)	O(1) + 12n

Table 5.2: Performance Comparison of SACAs, [24]

However, Baier was the first one to come up with an algorithm not based on Farach's approach, by sorting suffixes directly via sorting maximal Lyndon factors – in the next chapter we will explain how. He opened the doors towards non-recursive linear algorithms for suffix sorting, and for finding and sorting maximal Lyndon factors in a direct fashion. He also explores a lot in terms of previous and next smaller values, he hints to a stack based approach

for lexicographically smaller suffixes.He shows that group membership of a suffix can be computed *on-the-fly* during the computation of lexicographically smaller suffixes. He also provides a cache friendly implementation of phase 2 to achieve a faster algorithm, [24].

Chapter 6

Our Modification to Phase II of Baier's Algorithm

6.1 Background Information

In the last chapter, we discussed Baier's algorithm that directly sorts the suffixes of a string, the first such algorithm that is not recursive. In fact, his approach implicitly gives quite a bit more: it includes an algorithm for computing what turns out to be a partially sorted version of the Lyndon array. For strings over alphabets that can be sorted in linear time in the length of the input string, phase I represents an elementary linear-time algorithms for computing Lyndon array. Note, that Bair does not explicitly use the term *Lyndon* in his thesis nor in the paper [24, 32]. In phase two, this partially sorted array is used to order all suffixes, as discussed in the previous chapter.

At the same time, it is known that the Lyndon array can be computed in linear time from the inverse suffix array, see Chapter 3.

In this chapter, we encompass those aspects of his work to establish the *linear equivalence* of certain ordering of maximal Lyndon factors and of fully sorted suffixes and provided an alternative algorithm for Phase II. We are here concerned with the second phase of Baier's algorithm and with the relationship between the groups of suffixes and the maximal Lyndon factors is explored in detail.

6.2 Linear Equivalence of Suffix and Partially Sorted Lyndon Arrays

In this section, we introduce data structures that are linearly equivalent to the suffix array, which means that there is a true linear-time algorithm computing the structure from the suffix array and a true linear-time algorithm computing the suffix array from the structure. The structures we are introducing are to formalize the intuitive notion of "partially ordered Lyndon array". These structures are two-dimensional arrays *Lyndon Grouping array*, *Partially Sorted Lyndon Array*, and *Sorted Lyndon Array.* A very superficial view of these arrays is summarized below:

1. Make the suffix groups $\longrightarrow$ Lyndon Grouping Array.
2. Sort the suffix groups according to their context $\longrightarrow$ Partially Sorted

Lyndon Array.

3. Sort the indices within each group $\longrightarrow$ Sorted Lyndon Array.

The definition of Lyndon array was previously given in Chapter 1, we are putting it here again for a quick reference The Lyndon array was introduced in [58] — it is closely related to the *Lyndon tree* of [23]:

Definition 6.1. *For a given string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$*, the* Lyndon array *of* $\boldsymbol{x}$ *is an integer array* $L[1..n]$ *such that* $L[i] = j$ *if and only if* j *is the length of the maximal Lyndon substring at* i*.*

6.2.1 Lyndon Grouping Array

Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string of length n. The *Lyndon grouping array* of $\boldsymbol{x}$ is a two-dimensional integer array $\mathbb{L}[1..2][1..n]$ such that

1. $\mathbb{L}_1[1..n]$ is a permutation of $1..n$;
2. if $\mathbb{L}_2[i] > 0$, then the maximal Lyndon substring starting at $\mathbb{L}_1[i]$ has length $\mathbb{L}_2[i]$;
3. if $\mathbb{L}_2[i] = 0$, then the maximal Lyndon substring starting at $\mathbb{L}_1[i]$ has length $\mathbb{L}_2[j]$ where j is the greatest integer less than i such that $\mathbb{L}_2[j] > 0$.

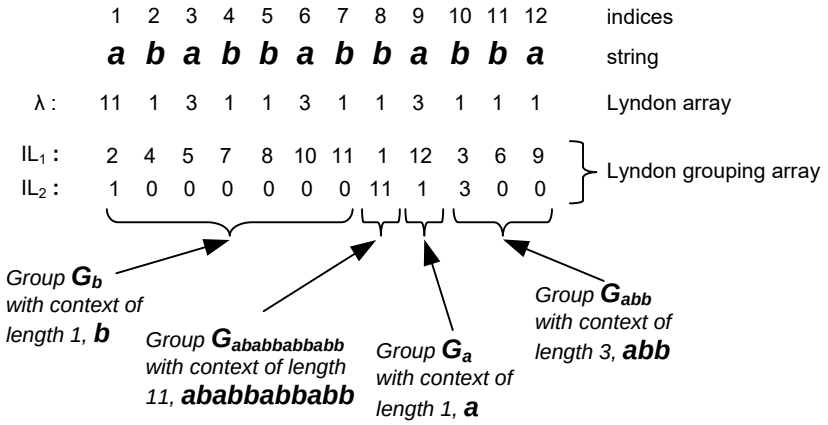

Figure 6.1: A Lyndon grouping array for ***ababbabbabba***

In the above example for the string $\boldsymbol{x} = ababbabbabba$ there are four groups: a group G_b with the context b, a group $G_{ababbabbabb}$ with the context $ababbabbabb$, a group G_a with the context a, and finally a group G_{abb} with the context abb as a, b, abb, and $ababbabbabba$ are all the maximal Lyndon factors of $\boldsymbol{x}$.

Thus a Lyndon grouping array just partitions the positions of a string into groups determined by identical *maxLyn* (an abbreviation for *maximal Lyndon* used in this chapter) substrings: all indices in the same group are starting positions of the same maxLyn substring. We denote a group with the context $\boldsymbol{u}$ as $G_{\boldsymbol{u}}$. Note that the Lyndon grouping array is not unique; that is, for a given $\boldsymbol{x}$ there may exist several such arrays with different orderings of the groups and different orderings of the indices in the groups.

The following lemma proves linear correspondance of Lyndon array from

the given Lyndon grouping array.

Lemma 6.2. *Let* $\mathbb{L}[1..2][1..n]$ *be a Lyndon grouping array of* $\boldsymbol{x} = \boldsymbol{x}[1..n]$. *Then the Lyndon array* L *of* $\boldsymbol{x}$ *can be computed from* $\mathbb{L}[1..2]$ *in* $\mathcal{O}(n)$ *steps.*

Proof. Replacing zeros in $\mathbb{L}_2$ with the value at the start of each group yields $L[\mathbb{L}_1[i]] = \mathbb{L}_2[i]$ for all $i \in 1..n$:

for $i = 1$ **to** n **do**

if $\mathbb{L}_2[i] \neq 0$ **then** $m \leftarrow \mathbb{L}_2[i]$

$L[\mathbb{L}_1[i]] \leftarrow m$ □

Note that the Lyndon array may provide a weaker information than a Lyndon grouping array in the sense that the Lyndon array can be computed from a Lyndon grouping array in linear time, but we do not know at this point how to compute a Lyndon grouping array from the Lyndon array in linear time.

6.2.2 Partially Sorted Lyndon Array

A *partially sorted Lyndon array* of $\boldsymbol{x}$ is a Lyndon grouping array whose groups are sorted in ascending lexicographic order; that is,

4. For $i < j$ such that $\mathbb{L}_2[i] > 0$, $\mathbb{L}_2[j] > 0$, $\boldsymbol{x}\big[\mathbb{L}_1[i]..\mathbb{L}_1[i]+\mathbb{L}_2[i]-1\big] \prec \boldsymbol{x}\big[\mathbb{L}_1[j]..\mathbb{L}_1[j]+\mathbb{L}_2[j]-1\big]$.

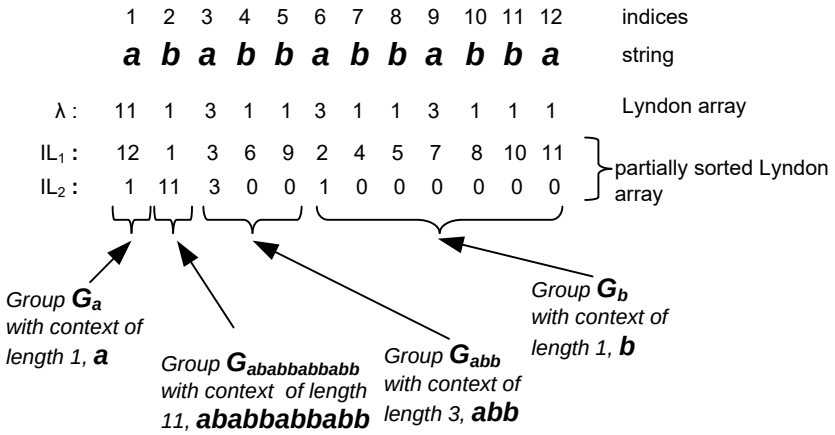

Figure 6.2: A partially sorted Lyndon array for ***ababbabbabba***

In Figure 6.2 the determinants a, $ababbabbabb$, abb, and b of the groups are sorted in ascending lexicographic order. However, the indices within the groups need not be in any particular order, though in our example they happen to fall in the ascending order of positions. Like the Lyndon grouping array, a partially sorted Lyndon array may not be unique as the indices in the groups may be in any order.

6.2.3 Sorted Lyndon Array

A *sorted Lyndon array* of $\boldsymbol{x}$ is a partially sorted Lyndon array whose indices are ordered within each group in the *perfect order*, i.e. according to the lexicographical order of the corresponding suffixes; more precisely:

5. If $\mathbb{L}_1[i]$ and $\mathbb{L}_1[j]$ belong to the same group, then
   $i < j \iff \boldsymbol{x}\big[\mathbb{L}_1[i]..n\big] \prec \boldsymbol{x}\big[\mathbb{L}_1[j]..n\big]$.

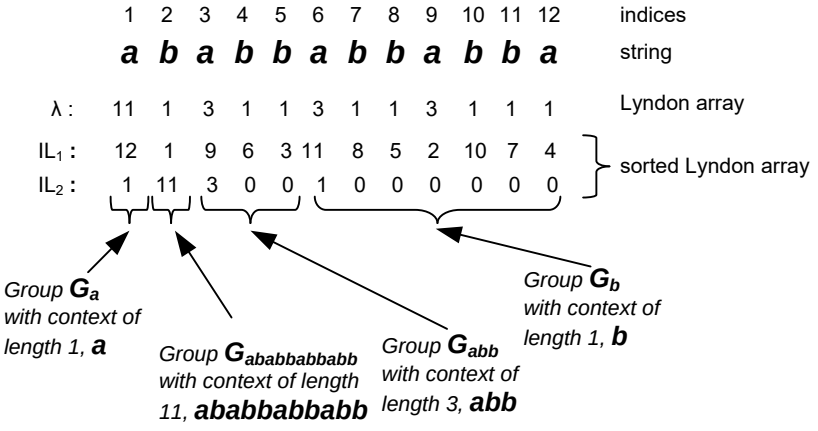

Figure 6.3: The sorted Lyndon array of ***ababbabbabba***

Along with the above mentioned arrays, we will be discussing three integer arrays defined previously, but whose definition we provide here for a quick reference:

(a) Given $\boldsymbol{x} = \boldsymbol{x}[1..n]$, the integer array $SA[1..n]$ is the *suffix array* of $\boldsymbol{x}$ iff the entries of SA form a permutation of $1..n$ and for every $1 \leq i < n$, $\boldsymbol{x}\big[SA[i]..n\big] \prec \boldsymbol{x}\big[SA[i{+}1]..n\big]$.

(b) The *lcp array* associated with SA is an integer array $lcp[1..n]$ in which $lcp[i]$ is the size of the longest common prefix of $\boldsymbol{x}\big[SA[i]..n\big]$ and $\boldsymbol{x}\big[SA[i{-}1]..n\big]$ for any $1 < i \leq n$.

(c) The *inverse suffix array* $ISA[1..n]$ is an integer array such that $SA[i] = j$ iff $ISA[j] = i$.

Note that if $\mathbb{L}[1..2][1..n]$ is a sorted Lyndon array of $\boldsymbol{x}$, then in fact $\mathbb{L}_1[1..n]$ is a suffix array of $\boldsymbol{x}$. Thus a sorted Lyndon array is unique, unlike a Lyndon grouping array and a partially sorted Lyndon array. Therefore we speak of **the** sorted Lyndon array of $\boldsymbol{x}$.

	1	2	3	4	5	6	7	8	9	10	11	12	
	a	***b***	***a***	***b***	***b***	***a***	***b***	***b***	***a***	***b***	***b***	***a***	indices / string
SA :	12	1	9	6	3	11	8	5	2	10	7	4	suffix array
lcp :	-	1	2	4	6	0	2	5	8	1	3	6	lcp array
ISA :	2	9	5	12	8	4	11	7	3	10	6	1	inverse suffix array

Figure 6.4: suffix array, inverse suffix array, and lcp array of ***ababbabbabba***

Theorem 6.3. *Let $SA[1..n]$ be the suffix array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. The sorted Lyndon array of $\boldsymbol{x}$ can be computed from $\boldsymbol{x}$ and SA in $\mathcal{O}(n)$ steps.*

Proof. As just observed, the top array $\mathbb{L}_1[1..n]$ is exactly the suffix array of $\boldsymbol{x}$. Thus we need to compute only $\mathbb{L}_2[1..n]$. First we compute the inverse suffix array ISA from SA in $\mathcal{O}(n)$ steps. Then, as noted in [59] and explained in [58], we compute the Lyndon array $L[1..n]$ of $\boldsymbol{x}$ from ISA, also in $\mathcal{O}(n)$ steps, using the next smaller value (NSV) algorithm. Thus we set $\mathbb{L}_2[i] = L\big[\mathbb{L}_1[i]\big]$ for every i.

To complete the calculation, we need only set the $\mathbb{L}_2$ values to zero except for the first entry in each group. For that we can use the $\mathcal{O}(n)$-time algorithm of Kasai *et al.* [60] to compute the lcp array. Then, for every i, if $lcp(\mathbb{L}_1[i], \mathbb{L}_1[i+1]) \geq \mathbb{L}_2[i]$ and $\mathbb{L}_2[i-1] = \mathbb{L}_2[i]$, we change the value of $\mathbb{L}_2[i]$

to 0. □

Several items need to be defined before our proof of Lemma 6.7. The definitions are not in a formal form, but the structures are explained before the proof to give a better understanding.

Definition 6.4. *The* delta operator *is defined as follows: for* $i \in G_{\boldsymbol{u}}$, $\Delta(i) = i+|\boldsymbol{u}|$. *If* $\Delta(i) \leq n$, *then consider* $\boldsymbol{v}$, *the maxLyn substring at the position* $\Delta(i)$. *If* $\boldsymbol{u}$ *were lexicographically smaller than* $\boldsymbol{v}$, *then* $\boldsymbol{uv}$ *would be Lyndon, contradicting the maximality of* $\boldsymbol{u}$. *Thus,* $\boldsymbol{v} \preceq \boldsymbol{u}$. *It follows that for*

$$i \in G_{\boldsymbol{u}} \quad \begin{cases} \Delta(i) = n+1, \text{ or} \\ \Delta(i) \in G_{\boldsymbol{v}} \text{ for some maxLyn } \boldsymbol{v} \prec \boldsymbol{u}, \text{ or} \\ \Delta(i) \in G_{\boldsymbol{u}}. \end{cases}$$

Definition 6.5. *The groups form a partition of the set of indices. Using the delta operator we define the* Δ*-refinement of this partition: let* $\boldsymbol{u}$, $\boldsymbol{v}$ *be maxLyn substrings of* $\boldsymbol{x}$ *so that* $\boldsymbol{v} \preceq \boldsymbol{u}$, *then we define the* subgroup $G_{\boldsymbol{u}}^{\boldsymbol{v}} = \{i \in G_{\boldsymbol{u}} : \Delta(i) \in G_{\boldsymbol{v}}\}$, *while we define the* subgroup $G_{\boldsymbol{u}}^{\$} = \{i \in G_{\boldsymbol{u}} : \Delta(i) = n+1\}$.

Definition 6.6. *Each group* $G_{\boldsymbol{u}}$ *is a disjoint union of non-empty subgroups* $G_{\boldsymbol{u}}^{\boldsymbol{v}}$ *for all maxLyn* $\boldsymbol{v} \preceq \boldsymbol{u}$ *and possibly* $G_{\boldsymbol{u}}^{\$}$. *If* $i \in G_{\boldsymbol{u}}^{\boldsymbol{v}_1}$ *and* $j \in G_{\boldsymbol{u}}^{\boldsymbol{v}_2}$, *and* $\boldsymbol{v}_1 \prec \boldsymbol{v}_2 \preceq \boldsymbol{u}$, *then* $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$, *as* $\boldsymbol{x}[i..n] = \boldsymbol{u}\boldsymbol{v}_1\boldsymbol{w}_1$ *for some* $\boldsymbol{w}_1$, *and* $\boldsymbol{x}[j..n] = \boldsymbol{u}\boldsymbol{v}_2\boldsymbol{w}_2$ *for some* $\boldsymbol{w}_2$. *Since* $|G_{\boldsymbol{u}}^{\$}| \leq 1$, *if* $i \in G_{\boldsymbol{u}}^{\$}$ *and* $i \neq j \in G_{\boldsymbol{u}}$, *then* $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$, *as* $\boldsymbol{x}[i..n] = \boldsymbol{u}$ *and* $\boldsymbol{x}[j..n] = \boldsymbol{u}\boldsymbol{w}$ *for some* $\boldsymbol{w}$. *Thus, if we separately perfectly order the subgroup* $G_{\boldsymbol{u}}^{\boldsymbol{v}}$ *for each maxLyn* $\boldsymbol{v} \prec \boldsymbol{u}$,

then the group $G_{\boldsymbol{u}}$ will be perfectly ordered, as an important property of each subgroup $G^{\boldsymbol{v}}_{\boldsymbol{u}}$, $\boldsymbol{v} \prec \boldsymbol{u}$, is the fact that a perfect order of the group $G_{\boldsymbol{v}}$ induces a perfect order on $G^{\boldsymbol{v}}_{\boldsymbol{u}}$: we simply let i precede j only if $\Delta(i)$ precedes $\Delta(j)$. Similarly, a perfect order of $G^{1}_{\boldsymbol{u}}$, which is defined as the disjoint union of all subgroups of $G_{\boldsymbol{u}}$ except $G^{\boldsymbol{u}}_{\boldsymbol{u}}$, induces a perfect order on $G^{\boldsymbol{u}}_{\boldsymbol{u}}$.

Example

Let $\boldsymbol{x}$ be a string , $\boldsymbol{x} = abb\,abb\,aa\,abb\,abb\,abb$ with $G_{abb} = \{1, 4, 9, 12, 15\}$. $G^{\$}_{abb} = \{15\}$, $G^{aaabbabbabb}_{abb} = \{4\}$, $G^{abb}_{abb} = \{1, 9, 12\}$ and $G_{abb} = G^{\$}_{abb} \cup G^{aaabbabbabb}_{abb} \cup G^{abb}_{abb}$. A perfect order of $G^{\$}_{abb}$ is 15, a perfect order of $G^{aaabbabbabb}_{abb}$ is 4. The elements of $G^{\$}_{abb}$ will be listed first, the elements of $G^{aaabbabbabb}_{abb}$. The perfect order of $G^{\$}_{abb} \cup G^{aaabbabbabb}_{abb} = \{15, 4\}$ determines the order of $G^{abb}_{abb} = \{1, 9, 12\}$. Now, $\Delta(1) = 1+3 = 4$, $\Delta(9) = 9+3 = 12$, and $\Delta(12) = 12+3 = 15$. Thus, 12 goes before 1, and then goes 9, i.e. the perfect order of G_{abb} is 15, 4, 12, 1, 9.

Lemma 6.7. *Let $\mathbb{L}[1..2][1..n]$ be a partially sorted Lyndon array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Then in $\mathcal{O}(n)$ steps we can order the items in the groups to obtain the sorted Lyndon array.*

Proof. We can achieve the desired ordering of $\mathbb{L}[1..2][1..n]$ by computing the suffix array SA of $\boldsymbol{x}$ and copying it into $\mathbb{L}_1[1..n]$. First, we compute triples $(I[i], G[i], SG[i])$ for $i \in 1..n$, where $I[i] = \mathbb{L}_1[i]$, $G[i]$ represent group (we are using integers $1..n$ to represent groups), and using $\Delta(i)$ we compute the

subgroups (we are using integers $0..n$ to represent the subgroups). This can be achieved in two traversals.

Then we use a radix sort to sort the triples to be ascending in G and within each group to be ascending in SG. This can be achieved in six traversals. In two traversals we can compute the inverse Δ relation, i.e. $i \in \Delta^{-1}(j)$ iff $\Delta(i) = j$.

Then we traverse the inverse Δ relation Δ^{-1} and record the indices as we encounter them. As explained above, the perfect order of the previous groups induces a perfect order on the current group via the Δ operator. □

Theorem 6.8. *Let* $\mathbb{L}[1..2][1..n]$ *be a partially sorted Lyndon array of a string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$. *The suffix array* $SA[1..n]$ *of* $\boldsymbol{x}$ *can be computed from* $\boldsymbol{x}$ *and* $\mathbb{L}$ *in* $\mathcal{O}(n)$ *steps.*

Proof. Using Lemma 6.7, we can compute the sorted Lyndon array $\mathbb{L}[1..2][1..n]$ of $\boldsymbol{x}$ by perfectly ordering $\mathbb{L}$. As previously noted, $\mathbb{L}[1][1..n]$ is then the suffix array of $\boldsymbol{x}$. □

We have implemented in C++ the algorithm that computes from the partially sorted Lyndon array the suffix array based on the proof of Lemma 6.7. The source code can be accessed at

`http://www.cas.mcmaster.ca/~franek/research/ub.cpp`

In essence, it is an alternative algorithm for phase II of Baier's algorithm. The only difference is that it expects as the input $\boldsymbol{x}$ and a partially sorted Lyndon array, while Baier's phase II algorithm expects as the input $\boldsymbol{x}$ and

the system of the groups after phase I is completed.

6.3 Comparison of the two approaches of computing Lyndon array

Since 1998, when Farach presented the linear algorithm for suffix tree construction, most of the following work was based on his approach. Baier was the first one to propose a totally different approach to sorting suffixes. Baier observed in [24, 32], that his algorithm was slower than the state-of-the-art suffix sorting algorithms. He ascribed that to the early stages of the existence of his non-recursive approach and conjectured that with time, the approach would become more refined and thus faster.

Baier's phase I algorithm is linear only for strings over alphabets that can be sorted in linear time. Our algorithm for computing Lyndon array presented in Chapter 4 works in $\Theta(n \log(n))$, so would be outperformed by Baeir's for any strings over constant alphabets (such as binary, ternary etc.). However, if we consider strings over increasing alphabets that cannot be sorted in linear time, Baier's algorithm's complexity becomes $\mathcal{O}(n \log(n))$, while our algorithm works as before, the log factors kicks in based on the structure of the string and not based on the alphabet of the string.

The proof of Theorem 6.3 actually shows how much extra work is needed to get from the suffix array to a sorted Lyndon array. Thus, it seems to us that computing a partially sorted Lyndon array is essentially a harder task

than "plain sorting" of the suffixes. So, maybe, no algorithm for computing a partially sorted Lyndon array can be as fast as sorting of suffixes, which in no way detracts from Baier's discovery of the deep connection hitherto unnoticed between the order of maximal Lyndon substrings and the order suffixes of a string.

In Fig. 6.5, the arrows represent "simple linear computation". The diagram summarizes the relationships among the various arrays we were investigating. The two arrows with ? represent open questions: *Can a Lyndon array be used in a simple linear computation to compute a Lyndon grouping array?* and *Can a Lyndon grouping array be used in a simple linear computation to compute a sorted Lyndon array?*. Note that phase I of Baier's algorithm basically says **Yes** to both of these questions. However, it is not using any Lyndon array or Lyndon grouping array, it just computes it directly from the string. Maybe, having a Lyndon array or Lyndon grouping array can simplify the computation. From our point of view, having been interested in computation of Lyndon arrays, answer to the first question is much more interesting.

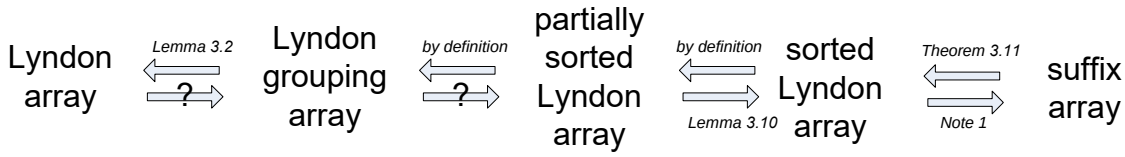

Figure 6.5: Linear computation of Lydon grouping,partially sorted Lyndon and sorted Lyndon arrays

Chapter 7

Conclusion and Future Work

This thesis deals with Lyndon factors of a string and their relationships with periodicities of the string, in particular runs. The contribution of the thesis is twofold: first to collect and present in a unified fashion all the various pieces of knowledge and results spread through many publications concerning Lyndon strings and Lyndon arrays, and second, to provide a novel way to compute the Lyndon array. The publication of Bair's work during my work on this thesis expanded the work to formalizing the intuitive notion of partially sorted Lyndon array and lead to a novel algorithm for phase II of Baier's algorithm. Since both are implemented in C++, it is imperative to continue with the research outlined in Chapter 6 and compare the two implementations. The second imperative task is to provide new insight and deal with 4.5.2 in more efficient way to bring the complexity of the algorithm presented in Chapter 5 to $\mathcal{O}(n)$. The possible avenues to explore include to

work with a partially ordered Lyndon array rather than the Lyndon array with the additional information harnessed for reducing the complexity, or computing some data structures in addition to the Lyndon array. The next step will include the implementation of the algorithm and comparison of the performance of my algorithm and phase I algorithm of Baeir's.

Bibliography

[1] M. Main, "Detecting leftmost maximal periodicities.," *Discrete Applied Maths*, vol. 25, pp. 145–153, 1989.

[2] S. Puglisi, W. Smyth, and A. Turpin, "Some restrictions on periodicity in strings," in *Proceeding of Australasian Workshop on Combinatorial Algorithms*, 2005.

[3] M. Crochemore, "An optimal algorithm for computing the repetitions in a word," *Information Processing Letters*, vol. 12, pp. 297–315, 1981.

[4] R. Kolpakov and G. Kucherov, "Finding maximal repetitions in a word in linear time," in *Proceedings of FOCS 1999, Washington, USA*, pp. 596–604, IEEE Computer Society, 1999.

[5] W. Rytter, "The number of runs in a string: Improved analysis of the linear upper bound," *Lecture Notes in Computer Science*, vol. 3884, pp. 184–195, 2006.

[6] W. Rytter, "The number of runs in a string," *Information and Computation*, vol. 205, pp. 1459–1469, 2007.

[7] M. Crochemore and L. Ilie, "Maximal repetitions in strings," *Journal of Computer and System Sciences*, vol. 74, pp. 796–807, 2008.

[8] M. Crochemore, L. Ilie, and L. Tinta, "Towards a solution to the "runs" conjecture," *Lecture Notes in Computer Science*, vol. 5029, pp. 290–302, 2008.

[9] F. Franek, J. Simpson, and W. Smyth, "The maximum number of runs in a string," in *Proceedings of AWOCA 2003, Seoul, Korea*, pp. 13–16, 2008.

[10] F. Franek and Q. Yang, "An asymptotic lower bound for the maximal number of runs in a string," *International Journal of Foundations of Computer Science*, vol. 19, pp. 195–203, 2008.

[11] M. Giraud, "Not so many runs in strings," in *Proceedings of LATA 2008, Tarragona, Spain*, pp. 232–239, Springer, 2008.

[12] W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara, "A series of run-rich strings," *Lecture Notes in Computer Science*, vol. 5457, pp. 578–587, 2009.

[13] W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara, "New lower bounds for the maximum number of runs in a string," in *Proceedings of PSC 2008, Prague, Czech Republic*, pp. 140–145, 2008.

[14] W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara, "Lower bounds for the maximum number of runs in a string." `http://www.shino.ecei.tohoku.ac.jp/runs/`.

[15] S. Puglisi, J. Simpson, and W. Smyth, "How many runs can a string contain?," *Theoretical Computer Science*, vol. 401, pp. 165–171, 2008.

[16] H. Bannai, S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, and T. I, "The runs theorem," *SIAM Journal on Computing*, vol. 46, no. 5, pp. 1501–1514, 2017.

[17] A. Deza and F. Franek, "A d-step approach to the maximum number of distinct squares and runs in strings," *Discrete Applied Mathematics*, vol. 163, pp. 268–274, 2014.

[18] J. Fischer, Š. Holub, T. I, and M. Lewenstein, "Beyond the runs theorem," in *Proceedings of SPIRE 2015, London, UK*, pp. 277–286, Springer-Verlag, 2015.

[19] A. Deza and F. Franek, "Bannai et al. method proves the d-step conjecture for strings," *to appear in Discrete Applied Mathematics*, 2016.

[20] G. Nong, S. Zhang, and W. Chan, "Linear Suffix Array Construction by Almost Pure Induced-sorting," in *Proceedings of the 2009 Data Compression Conference*, DCC '09, (Washington, DC, USA), pp. 193–202, IEEE Computer Society, 2009.

[21] G. Nong, S. Zhang, and W. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Trans. Comput.*, vol. 60, pp. 1471–1484, 2011.

[22] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," 2005.

[23] C. Hohlweg and C. Reutenauer, "Lyndon words, permutations and trees," *Theoretical computer science*, vol. 307 (1), pp. 173–178, 2003.

[24] U. Baier, "Linear-time Suffix Sorting — A New Approach for Suffix Array Construction." M.Sc. Thesis, University of Ulm, 2015.

[25] R. Lyndon, "On Burnside's problem," *Trans. Amer. Math. Soc.*, vol. 77, pp. 202–215, 1954.

[26] K. Chen, R. Fox, and R. Lyndon, "Free differential calculus IV. The quotient groups of the lower central series," *Annals of Mathematics, 2nd Ser.*, vol. 68 (1), pp. 81–95, 1958.

[27] S. Golomb, "Irreducible polynomials, synchronizing codes, primitive necklaces and cyclotomic algebra," vol. 4, pp. 358–370, 1967.

[28] P. Flajolet, X. Gourdon, and D. Panario, "The complete analysis of a polynomial factorization algorithm over finite fields," *Journal of Algorithms*, vol. 40, pp. 37–81, 2001.

[29] D. Panario and B. Richmond, "Smallest components in decomposable structures:exp-log class," *Algorithmica*, vol. 29, pp. 205–226, 2001.

[30] J. Duval, "Factorizing words over an ordered alphabet," *Journal of Algorithms*, vol. 4, pp. 363–381, 1983.

[31] H. Fredricksen and J. Maiorana, "Necklaces of beads in k colors and k-ary de Bruijn sequences," *Discrete Math.*, vol. 23 (3), pp. 207–210, 1983.

[32] U. Baier, "Linear-time suffix sorting — a new approach for suffix array construction," in *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)* (R. Grossi and M. Lewenstein, eds.), vol. 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 23:1–23:12, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[33] F. Franek, A. Paracha, and W. Smyth, "The linear equivalence of the suffix array and the partially sorted lyndon array," in *Proceedings of the Prague Stringology Conference 2017* (J. Holub and J. Žďárek, eds.), (Czech Technical University in Prague, Czech Republic), pp. 77–84, 2017.

[34] J. Berstel and M. Pocchiola, "Average cost of duval's algorithm for generating lyndon words," *Theoretical Computer Science*, vol. 132, pp. 415–425, 1994.

[35] S. Neuburger and D. Sokol, "Succinct 2d dictionary matching," *Journal of Algorithms*, vol. 65(3), pp. 662–684, 2013.

[36] A. Deza, F. Franek, and A. Thierry, "How many double squares can a string contain?," *Discrete Applied Mathematics*, vol. 180, pp. 52–69, 2015.

[37] S. Holub and D. Nowotka, "On the Relation between Periodicity and Unbordered Factors of Finite Words," *Lecture Notes in Computer Science*, vol. 5257, pp. 408–418, 2008.

[38] M. Kubica, J. Radoszewsk, W. Rytter, and T. Waleń, "On the maximum number of cubic subwords in a word," *European Journal of Combinatorics*, vol. 34, pp. 27–37, 2013.

[39] F. Franek, S. Gao, W. Lu, P. Ryan, W. Smyth, Y. Sun, and L. Yang, "Verifying a border array in linear time," *J. Combinatorial Math. & Combinatorial Computing*, vol. 42, pp. 223–236, 2002.

[40] W. Bland, G. Kucherov, and W. Smyth, "Prefix table construction and conversion," *Proc. 24th*, pp. 41–53, 2013.

[41] M. Christodoulakis, P. Ryan, W. Smyth, and S. Wang, "Indeterminate strings, prefix arrays and undirected graphs," *Theoretical Comput. Sci.*, vol. 600, pp. 34–48, 2015.

[42] F. Bassino, J. Clément, and C. Nicaud, "The standard factorization of Lyndon words: an average point of view," *Discrete Mathematics*, vol. 290, no. 1, pp. 1–25, 2005.

[43] G. Nong, S. Zhang, and W. Chan, "Linear time suffix array construction using D-critical substrings," in *20th Annual Symp. on Combinatorial Pattern Matching* (G. K. . E. Ukkonen, ed.), vol. 5577 of *Lecture Notes in Computer Science*, pp. 54–67, Springer-Verlag, 2009.

[44] S. Puglisi, W. Smyth, and A. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Comput. Surv.*, vol. 39, pp. 1–31, July 2007.

[45] P. Weiner, "Linear pattern matching algorithm," in *14th IEEE Symposium on switching and automata theory*, pp. 1–11, 1973.

[46] M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," in *The 38th Annual IEEE Symposium on Foundations of Computer Science*, pp. 137 –143, 1997.

[47] D. Gusfield, *Algorithms on strings, trees, and sequences : computer science and computational biology.* Cambridge University Press, 1997.

[48] A. Apostolico, "The Myriad Virtues of Subword Trees," in *Combinatorial Algorithms on Words*, vol. 12, pp. 85–96, 1985.

[49] R. Grossi and G. Italiano, "Suffix trees and their applications in string algorithms," in *Proceedings of the 1st south American workshop on string processing*, pp. 57–76, 1993.

[50] G. Gonnet, R. Baeza-Yates, and T. Snider, "New indices for text: PAT trees and PAT arrays," in *Information retrieval: data structures and algorithms*, pp. 66–82, 1992.

[51] U. Manber and E. Myers, "Suffix Arrays: A New Method for On-Line String Searches," in *First ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–327, 1990.

[52] S. Puglisi, W. Smyth, and A. Turpin., "A taxonomy of suffix array construction algorithms.," *ACM Computational Survey*, vol. 39(2), 2007.

[53] Y. Mori, "libdivsufsort." `http://github.com/y-256/libdivsufsort`. [Online; accessed October-2015].

[54] Y. Mori, "sais-lite-2.4.1." `http://sites.google.com/site/yuta256/sais`. [Online; accessed October-2015].

[55] P. Ko and S. Aluru, "Ko-Aluru Algorithm." `http://sites.google.com/site/yuta256/KA.tar.bz2`. [Online; accessed October-2015].

[56] P. Sanders, "Dc3 algorithm." `http://people.mpi-inf.mpg.de/~sanders/programs/suffix/.` [Online; accessed October-2015].

[57] U. Baier., "Gsaca algorithm." `http://github.com/waYne1337/gsaca`. [Online; accessed October-2015].

[58] F. Franek, A. Islam, M. Rahman, and W. Smyth, "Algorithms to compute the Lyndon array," in *Proceedings of Prague Stringology Conference 2016*, PSC'16, pp. 172–184, 2016.

[59] C. Hohlweg and C. Reutenauer, "Lyndon words, permutations and trees," *Theoretical Computer Science*, vol. 307, no. 1, p. 173 178, 2003.

[60] T. Kasaim, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings* (A. Amir, ed.), (Berlin, Heidelberg), pp. 181–192, Springer Berlin Heidelberg, 2001.