Combinatorial Properties and Algorithms on Lyndon Words

中島, 祐人

https://doi.org/10.15017/1807061

Combinatorial Properties and Algorithms on Lyndon Words

Yuto Nakashima

January, 2017

Abstract

Recently, large quantities of data is produced every day, such as web pages, sensor data, social media data, digital pictures and videos, purchase transaction records, and so on. Development of techniques for data analysis is very important because the large data may hold key to solving problems in our daily life. The difficulty in dealing with the rapidly growing data is not only because they are vast, but also because they are often "unstructured" and "non-uniform". Traditional database technologies are based on the theory of relational algebra proposed by Codd in 1970 [16], and are able to handle structured data (relations) that can be represented as tables, but have difficulty in handling unstructured data. This new type of data can be regarded as sequences of symbols (i.e., strings), and therefore efficient string processing techniques can play key role in processing them. Developing efficient string processing algorithms requires to have not only rich knowledge of algorithms and data structures, but also deep insights on algebraic and combinatorial properties on strings.

In this thesis, we focus on combinatorial properties concerning Lyndon words, discover combinatorial properties, and develop algorithms on Lyndon words. A string λ is said to be a Lyndon word if λ is lexicographically smaller than every proper suffix of λ. Since Lyndon words are used for detecting structures in strings (e.g., repetitive structures), studies concerning Lyndon words could lead to making efficient string processing algorithms. Lyndon factorizations (sequence of Lyndon words in lexicographically non-increasing order) are also studied. We consider the following four problems concerning Lyndon words. Firstly, we consider the reverse-engineering problems on Lyndon factorizations. When we are given a form of the Lyndon factorization of some string, we compute a string which have the input as the Lyndon factorization. Solving the reverse-engineering problem gives combinatorial properties and deeper insights in to the input structure. We propose some variant of the reverse-engineering problems and present efficient algorithms. Secondly, we present efficient algorithms to compute the Lyndon factorization of "compressed string". Thirdly, we study relations on the size of Lyndon

factorizations and Lempel-Ziv 77 factorizations. We give the first direct connection of these two factorizations. Finally, we consider relations between Lyndon words and runs. Runs are important repetitive structures in strings. We show a new upper bound on the maximum number of runs in a string. We also present a new algorithm to compute all runs in a string.

Acknowledgments

I would like to express my gratitude to everyone who supported my research life at Kyushu University. First of all, I am deeply grateful to Professor Masayuki Takeda who is my supervisor and the committee chair of my thesis. He taught depth of research, attitude toward research, and so on to me. I would also like to express my appreciation to Professor Masafumi Yamashita, Professor Eiji Takimoto, and Associate Professor Hideo Bannai, who are the members of the committee of my thesis. I also thank all of those in Department of Informatics, Kyushu University, for their generous support.

I would also express my appreciation to Associate Professor Hideo Bannai and Associate Professor Shunsuke Inenaga. They taught how to research and gave knowledge and many ideas to me. I am deeply grateful to Professor Eiji Takimoto and Associate Professor Kohei Hatano. They gave many fruitful comments to me in weekly seminar.

This research was partly supported by JSPS (Japan Society for the Promotion of Science). The results in the thesis were partially published in the Proc. of CPM'13, the Proc. of SPIRE'13, the Proc. of MFCS'14, and the Proc. of SODA'15. The results in the thesis will be partially published in the Proc. of STACS'17 in this March. Also, the journal version of SPIRE'13 were published in Theoretical Computer Science by Elsevier. I am thankful for all editors, committees, anonymous referees, and publishers.

I would like to thank my appreciation to Dr. Juha Kärkkäinen, Dr. Simon J. Puglisi, Dr. Dominik Kempa, and Professor Arseny Shur. I enjoyed the discussion with them in my visit to Department of Computer Science, University of Helsinki. One of the results of the thesis will be published in the Proc. of STACS'17 in this March as a joint contribution with them.

Last, but not least, I really thank my parents for their support.

Contents

Chapter 1

Introduction

Recently, large quantities of data is produced every day, such as web pages, sensor data, social media data, digital pictures and videos, purchase transaction records, and so on. Development of techniques for data analysis is very important because the large data may hold key to solving problems in our daily life. The difficulty in dealing with the rapidly growing data is not only because they are vast, but also because they are often "unstructured" and "non-uniform". Traditional database technologies are based on the theory of relational algebra proposed by Codd in 1970 [16], and are able to handle structured data (relations) that can be represented as tables, but have difficulty in handling unstructured data. This new type of data can be regarded as sequences of symbols (i.e., strings), and therefore efficient string processing techniques can play key role in processing them. Developing efficient string processing algorithms requires to have not only rich knowledge of algorithms and data structures, but also deep insights on algebraic and combinatorial properties on strings.

Let us take an example of *string pattern matching problem*, which is one of the most important problems in Computer Science, where given a pattern string and a text string, the objective is to find the occurrences of the pattern in the text. For this problem, the *Knuth-Morris-Pratt algorithm* [64] is known as the first linear time algorithm. This algorithm is based on the *Periodicity Lemma* [35] which is one of well-known combinatorial properties of periodicity on strings. On the other hand, the *Crochemore-Perrin algorithm* [25] works in linear time and constant space for the same problem. The correctness is guaranteed by *Critical Factorization Theorem* [72] which is one of combinatorial properties. From these facts, discovering new combinatorial properties on strings leads to efficient and simple string processing algorithms.

In this thesis, we focus on combinatorial properties concerning *Lyndon words* [73]. We study some problems concerning Lyndon words. Firstly, we tackle the reverse-engineering

problems. Secondly, we consider the problem of computing the *Lyndon factorizations* [14] over compressed strings. Thirdly, we study relations on the size of Lyndon factorizations and LZ77 factorizations. Finally, we consider relations between Lyndon words and runs.

1.1 Lyndon Words and Lyndon Factorizations

A string λ is said to be a *Lyndon word*, if λ is lexicographically smaller than all of proper suffixes of λ. For example, `aab` is a Lyndon word, but `aba` and `baa` are not. Lyndon words are named after Roger Lyndon, a mathematician who introduced them in [73] under the name of *standard lexicographic sequences*.

Lyndon words have various and important applications in algebra and combinatorics. For example, Lyndon words have an application to the description of free Lie algebras in constructing bases. This fact is considered as original motivation for introducing Lyndon words. Lyndon words also have many applications to musicology [12], bioinformatics [27], approximation algorithm [79], string matching [25], and word combinatorics [39, 66, 80].

The *Lyndon factorization* of a non-empty string $T \in \Sigma^+$, is the factorization $\lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ of T, such that each $\lambda_i \in \Sigma^+$ is a Lyndon word, $e_i \geq 1$, and $\lambda_i \succ \lambda_{i+1}$ for all $1 \leq i < m$. We call each element in the sequence a *factor*. If each factor in a factorization of T is a Lyndon word and all factors are arranged in the lexicographically non-increasing order, the factorization is called the *Lyndon factorization* of T. Lyndon factorizations are introduced by Chen, Fox and Lyndon in [14]. It is known that the Lyndon factorization is unique for each string. Lyndon factorizations are used in a bijective variant of Burrows-Wheeler transform [47, 70] and a digital geometry algorithm [9].

Several efficient algorithms to compute Lyndon factorizations exist: Duval [30] proposed an elegant on-line algorithm to compute the Lyndon factorization of a given string T of length N in $O(N)$ time. Efficient parallel algorithms to compute the Lyndon factorization of a given string are also known [1, 26]. Recently, algorithms to compute the Lyndon factorization of a given compressed string were proposed [60, 61].

1.2 Our Problems

In this section, we state problems we consider in this thesis and explain motivation and related works of the problems.

1.2.1 Reverse-engineering problems on Lyndon factorizations

Reverse-engineering problems on structures of strings have been studied. Solving reverse-engineering problems on structures of strings provides us with further insight concerning combinatorial properties on strings, and string data structures. There exists a linear-time algorithm that finds a string over an unbounded alphabet, such that the border array of the string coincides with a given integer array [37]. There also exists a simpler linear-time to the same problem for a bounded alphabet [32]. For the parameterized version of border arrays, linear-time and $O(N^{1.5})$-time inferring algorithms on a binary alphabet and an unbounded alphabet, respectively, exist [58]. Linear-time inferring algorithms for suffix arrays [3, 34], KMP failure tables [33, 46], prefix tables [15], cover arrays [23], palindromic structures [56], suffix trees on binary alphabets [57], directed acyclic word graphs [3] and directed acyclic subsequence graphs [3] have been proposed. On the other hand, some hardness results are also known. The reverse-engineering problem on longest previous factor tables is NP-hard [51]. Also, inferring a string from runs is NP-hard, on a finite alphabet of size at least 4 [76]. Counting and enumerating versions of some of the above-mentioned problems have also been studied in the literature [58, 78, 84].

We consider reverse-engineering problems on Lyndon factorizations. Since the Lyndon factorization of a string can be represented by the sequence $(|\lambda_1|, e_1), \ldots, (|\lambda_m|, e_m)$ of two positive integers, we consider a sequence of pairs of positive integers as an input of our problem. For example, most basic problem is to compute a string which has an input sequence as its Lyndon factorization. We solve four variants of the problem in Chapter 3.

1.2.2 Lyndon factorization algorithms for compressed strings

Compressed string processing (*CSP*) is a task of processing compressed string data without explicit decompression. As any method that first decompresses the data requires time and space dependent on the decompressed size of the data, CSP without explicit decompression has been gaining importance due to the ever increasing amount of data produced and stored. A number of efficient CSP algorithms have been proposed (e.g., see [42, 43, 44, 50, 52, 91]).

Many of problems on CSPs considered a *straight-line program* (*SLP*) as an input compressed string. An SLP is a context free grammar in the Chomsky normal form that derives a single string. Since SLPs can widely accept various text compression schemes, CSPs for an SLP are widely studied. On the other hand, CSPs for specific representations of strings (e.g.,

LZ78 [93], run-length encoded strings) are also well studied.

In this thesis, we consider Lyndon factorization algorithms for compressed strings. In Chapter 4 and 5, we present Lyndon factorization algorithms for SLPs. In Chapter 5, we also present a Lyndon factorization algorithm for an LZ78 compressed string.

1.2.3 Relations between Lyndon factorizations and LZ77 factorizations

The Lempel-Ziv factorizations (shortly LZ factorizations) are factorizations defined by using information of previous occurrences of factors. The LZ factorizations have its origins in data compression, and are still used in popular file compressors [1] and as part of larger software systems (see, e.g., [10, 54] and references therein). Originally, Ziv and Lempel were introduced LZ77 factorization [92] and LZ78 factorization [93] (shortly LZ77 and LZ78, respectively). LZ factorizations are well used for detecting structures in strings and making efficient string processing algorithms. For example, LZ77 gives a lower bound of the smallest grammar [11], and is used in algorithms to compute all runs in strings (described in section 1.2.4).

On the other hand, Lyndon factorizations also give a lower bound of the smallest grammar [61] (see also Chapter 5). Our overarching motivation is to obtain a deeper understanding of how these two fundamental factorizations — LZ77 and Lyndon — relate. Toward this aim, we ask: by how much can the number of Lempel-Ziv factors and the number of Lyndon factors for the same string differ? We give relations of these factorizations in Chapter 6.

1.2.4 Detecting runs by using Lyndon words

Repetitions in strings are one of the most basic and well studied characteristics of strings, with various theoretical and practical applications (see [19, 87, 88] for surveys). A maximal repetition, or a *run*, is a maximal periodic sub-interval of a string, that is at least as long as twice its smallest period. For example, for a string $T[1..11] = \mathtt{aababaababb}$, $[1..2] = \mathtt{a}^2$, $[6..7] = \mathtt{a}^2$, and $[10..11] = \mathtt{b}^2$ are runs with period 1, $[2..6] = (\mathtt{ab})^{5/2}$ and $[7..10] = (\mathtt{ab})^2$ are runs with period 2, $[4..9] = (\mathtt{aba})^2$ is a run with period 3, and $[1..10] = (\mathtt{aabab})^2$ is a run with period 5. Runs essentially capture all consecutive repeats of a substring in a string.

The most remarkable property of runs, first proved by Kolpakov and Kucherov [68], is that the maximum number of runs $\rho(N)$ in a string of length N, is in fact linear in N. Although their proof did not give a specific constant factor, it was conjectured that $\rho(N) < N$.

[1] For example `gzip`, `p7zip`, `lz4`, and `snappy` all have the LZ factorization at their core.

In order to further understand the combinatorial structure of runs in strings, this "runs conjecture" has, since then, become the focus of many investigations. The first explicit constant was given by Rytter [83], where he showed $\rho(N) < 5N$. This was subsequently improved to $\rho(N) < 3.48N$ by Puglisi et al. [81] with a more detailed analysis using the same approach. Crochemore and Ilie [18] further reduced the bound to $\rho(N) < 1.6N$, and showed how better bounds could be obtained by computer verification. Based on this approach, Giraud proved $\rho(N) < 1.52N$ [48] and later $\rho(N) < 1.29N$ [49], but only for binary strings. The best known upper bound is $\rho(N) < 1.029N$ obtained by intense computer verification (almost 3 CPU years) [20], based on [18]. On the other hand, a lower bound of $\rho(N) \geq 0.927N$ was shown by Franek et al. [38]. Although this bound was first conjectured to be optimal, the bound was later improved by Matsubara et al. [77] to $\rho(N) \geq 0.944565N$. The best known lower bound is $\rho(N) \geq 0.944575712N$ by Simpson [86]. While the conjecture was very close to being proved, all of the previous linear upper bound proofs are based on heavy application of the periodicity lemma by Fine and Wilf [35], and are known to be very technical, which seems to indicate that we still do not yet have a good understanding of how runs can be contained in strings. For example, the proof for $\rho(N) < 1.6N$ by Crochemore and Ilie [18] required consideration of at least 61 cases (Table 2 of [18]) in order to bound the number of runs with period at most 9 by N.

Algorithms to compute all runs in the string are also well-studied. The first linear-time algorithm for computing all runs, proposed by Kolpakov and Kucherov [68], relies on the computation of the LZ77 factorization [92] of the string. All other existing linear-time algorithms basically follow their algorithm, but focus on more efficient computation of the parsing, which is the bottleneck [13, 17].

1.3 Our Contributions

1.3.1 Reverse-engineering problems on Lyndon factorizations

We solve four variants of reverse-engineering problems on Lyndon factorizations. Firstly, we present a simple $O(N)$-time algorithm to compute a string T of length N such that a given sequence of pairs of positive integers corresponds to the Lyndon factorization of T. Secondly, we propose an $O(N)$-time algorithm to compute a string T such that an input is the Lyndon factorization of T and the existing character in T is the smallest possible. This algorithm

has a generating algorithm to compute the lexicographically next smaller Lyndon word of a given length as a subroutine. Some generating algorithm to compute the lexicographically next smaller or larger Lyndon word of the same length exist. Since our generating algorithm can generate the next Lyndon word of "any length", our algorithm is somewhat general algorithm. We also present an algorithm to solve the same problem in compact representation. Thirdly, we show an $O(m)$-time algorithm to compute only the smallest size of alphabet where m is the size of the input (i.e., the size of the Lyndon factorization). Finally, we consider an enumerating problem, we propose a compact representation of all valid strings and an efficient algorithm to compute the representation.

1.3.2 Lyndon factorization algorithms for compressed strings

We propose Lyndon factorization algorithms for compressed strings. For an uncompressed string of length N, an elegant $O(N)$-time algorithm to compute the Lyndon factorization is well-known [30]. We consider an SLP and an LZ78 compressed strings as inputs. In Chapter 4, we present an efficient Lyndon factorization algorithm for an SLP. This algorithm compute Lyndon factors from right to left by computing the smallest suffix of a string. The rightmost decomposed Lyndon factor of a string T is the lexicographically smallest suffix of T. In Chapter 5, we present a faster Lyndon factorization algorithm for an SLP. The algorithm is based on key lemmas in parallel algorithms for Lyndon factorizations [1, 26], but unfortunately their proof (and the algorithm therein) appears to be incorrect. We give a corrected proof and algorithm. As yet another byproduct, we show that the size m of the Lyndon factorization of string T is bounded by the size n of *any* SLP representing T, i.e. $m \leq n$, which may be of independent interest. In Chapter 5, we also propose an efficient algorithm for an LZ78 factorization. All of our algorithms are faster than an $O(N)$-time algorithm if the input string is highly compressed.

1.3.3 Relations between Lyndon factorizations and LZ77 factorizations

We study relations between Lyndon factorizations and non-overlapping LZ77 factorizations. For most strings, the number of Lyndon factors is much smaller. Indeed, any string has a rotation with a Lyndon factorization of size one. However, we showed that there are strings with $t + \Theta(\sqrt{t})$ Lyndon factors, where t is the number of LZ77 factors. Our main result is to show that number of Lyndon factors cannot be more than $2t$. This result improves significantly a previous, indirect bound given in Chapter 5 that the number of Lyndon factors cannot be more

than the size of the smallest SLP. Since the smallest SLP is at most a logarithmic factor bigger than the LZ77 factorization [11, 82], this established an indirect, logarithmic factor bound, which we improve to a constant factor two.

1.3.4 Detecting runs by using Lyndon words

We give new insights into the runs conjecture, significantly improving our understanding of the structure of runs in strings. Our study of runs is based on combinatorics of Lyndon words. Lyndon words have recently been considered in the context of runs [21, 22], since any run with period p must contain a length-p substring that is a Lyndon word, called an L-root of the run. Concerning the number of cubic runs (runs with exponent at least 3), Crochemore et al. [21] gave a very simple proof that it can be no more than $0.5N$. The key observation is that, for any given lexicographic order, a cubic run must contain at least two consecutive occurrences of its L-root, and that the boundary position cannot be shared by consecutive L-roots of a different cubic run. However, this idea does not work for general runs, since, unlike cubic runs, only one occurrence of an L-root for a given lexicographic order is guaranteed, and the question of how to effectively apply Lyndon arguments to the analysis of the number of general runs has so far not been answered.

The contributions in Chapter 7 are summarized below:

Proof of $\rho(N) < N$ and $\sigma(N) < 3N$ We discover and establish a connection between the L-roots of runs and the longest Lyndon word starting at each position of the string. Based on this novel observation, we give an affirmative answer to the runs conjecture. The proof is remarkably simple.

Based on the same observation, we obtained a bound of $3N$ for the maximum sum of exponents $\sigma(N)$ of runs in a string of length N. The best known bound was $4.1N$ by Crochemore et al. [24], whose arguments were based on the bound of $\rho(N) < 1.029N$. We note that plugging-in $\rho(N) < N$ into their proof still only gives a bound of $4N$.

Let $\rho_k(N)$ be the maximum number of runs with exponent at least k in a string of length N, and let $\sigma_k(N)$ be the maximum sum of exponents of runs with exponent at least k in a string of length N. For any integer $k \geq 2$, we prove a bound of $\rho_k(N) < N/(k-1)$ and $\sigma_k(N) < N(k+1)/(k-1)$. For $k = 3$, this yields $\sigma_3(N) < 2N$ which improves on the bound of $2.5N$ by Crochemore et al. [24]. We also proved conjectured bounds

of $\rho(N,d) \leq N-d$ and if $N > 2d$, $\rho(N,d) \leq N-d-1$ [28], where $\rho(N,d)$ is the maximum number of runs in a string of length N that contains exactly d distinct symbols[2].

Linear-time computation of all runs without Lempel-Ziv parsing We give a novel, conceptually simple linear-time algorithm for computing all runs contained in a string, based on the proof of $\rho(N) < N$. Our algorithm is the first linear-time algorithm which does *not* rely on the Lempel-Ziv factorization of the string, and thus may help pave the way to more efficient algorithms for computing all runs in the string [89].

Runs and Lyndon trees We also establish a relationship between L-roots of runs in a string and nodes of what is called the Lyndon tree of the string [4], which is a full binary tree defined by recursive standard factorization. We showed a simple optimal solution to the 2-Period Query problem that was recently solved by Kociumaka et al. [67], i.e., given any interval $[i..j]$ of a string T of length N, return the smallest period p of $T[i..j]$ with $p \leq (j-i+1)/2$, if such exists, in constant time with $O(N)$ preprocessing.

1.4 Organization

The rest of this thesis is organized as follows: In Chapter 3, we consider the reverse-engineering problems for Lyndon factorizations. In Chapter 4, we propose Lyndon factorization algorithms for compressed strings. In Chapter 5, we propose faster Lyndon factorization algorithms for compressed strings. In Chapter 6, we study the relation between Lyndon factorizations and non-overlapping LZ77 factorizations. In Chapter 7, we show the runs theorem and propose a linear time algorithm to compute all runs in a given string.

[2]We note that Deza and Franek have independently and simultaneously proved similar bounds [29], based on our proof of the runs conjecture in an earlier version of this result.

Chapter 2

Preliminaries

In this chapter, we give the notations to be used in this thesis.

2.1 Notation

Let Σ be an *ordered finite alphabet*, and let $\sigma = |\Sigma|$. An element of Σ^* is called a *string*. The *length* of a string w is denoted by $|w|$. The *empty string* ε is a string of length 0. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix (resp. suffix) x of w is called a *proper prefix* (resp. *proper suffix*) of w if $x \neq w$. The set of suffixes of w is denoted by $\mathit{Suffix}(w)$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$. For any string w let $w^0 = \varepsilon$, and for any integer $k \geq 1$ let $w^k = ww^{k-1}$, i.e., w^k is a k-time repetition of w. Let w^∞ denote an infinite repetition of w. For any non-empty string w and integer $2 \leq i \leq |w|$, let $\mathbf{cs}_i(w)$ denote the i-th cyclic shift of w, namely, $\mathbf{cs}_i(w) = w[i..|w|]w[1..i-1]$, and let $\mathbf{cs}_1(w) = w$.

A sequence of non-empty strings $w_1, \ldots, w_j$ is said to be a *factorization* of a string w if $w_1 \cdots w_j = w$.

An integer $p \geq 1$ is said to be a *period* of a string w if $w[i] = w[i+p]$ for all $1 \leq i \leq |w|-p$. If p is a period of a string w with $p < |w|$, then $|w| - p$ is said to be a *border* of w. If w has no borders, then w is said to be *border-free*.

If character c is lexicographically smaller than another character c', then we write $c \prec c'$. For any non-empty strings $x, y \in \Sigma^+$, let $\mathit{lcp}(x, y)$ be the length of the longest common prefix of x and y, namely, $\mathit{lcp}(x, y) = \max(\{j \mid x[i] = y[i] \text{ for all } 1 \leq i \leq j\} \cup \{0\})$. For any

non-empty strings $x, y \in \Sigma^+$, we write $x \prec y$ iff either $lcp(x, y) + 1 \leq \min\{|x|, |y|\}$ and $x[lcp(x, y) + 1] \prec y[lcp(x, y) + 1]$, or x is a proper prefix of y. For any non-empty set $A \subseteq \Sigma^*$ of strings, let $\min_{\prec} A$ denote the lexicographically smallest string in A.

2.2 Lyndon Words

Here, we introduce two equivalent definitions of Lyndon words [73].

Definition 1. *A string λ is said to be a* Lyndon word*, if $\lambda \prec cs_i(\lambda)$ for all $2 \leq i \leq |\lambda|$.*

Definition 2. *A string λ is said to be a* Lyndon word*, if $\lambda \prec x$ for any non-empty proper suffix x of λ.*

Notice that any Lyndon word is border-free. The following lemma is also useful.

Lemma 1 (Proposition 1.3 [30])**.** *For any Lyndon words λ_1 and λ_2, $\lambda_1\lambda_2$ is a Lyndon word iff $\lambda_1 \prec \lambda_2$.*

2.3 Lyndon Factorizations

Definition 3 (Lyndon factorization [14])**.** *The* Lyndon factorization *of a non-empty string $T \in \Sigma^+$, denoted LF_T, is the factorization $\lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ of T, such that each $\lambda_i \in \Sigma^+$ is a Lyndon word, $e_i \geq 1$, and $\lambda_i \succ \lambda_{i+1}$ for all $1 \leq i < m$.*

Each $\lambda_i^{e_i}$ is called a *Lyndon factor*, and λ_i is called a *decomposed Lyndon factor*. The Lyndon factorization is unique for each string T, and can be represented by the sequence $(|\lambda_1|, e_1), \ldots, (|\lambda_m|, e_m)$ of two positive integers. For string $T = \texttt{abaaabaaabaa}$, $LF_T = (\texttt{ab})^1(\texttt{aaab})^2(\texttt{a})^2$. Note that all strings `ab`, `aaab`, and `a` in LF_T are Lyndon words, aligned in lexicographically decreasing order. LF_T can be represented by the sequence $(2, 1), (4, 2), (1, 2)$. The follwing lemma can be obtaied by the definition.

Lemma 2 ([30])**.** *It holds that λ_1 is the longest prefix of T which is a Lyndon word and p_1 is the largest integer k such that λ_1^k is a prefix of T.*

Also, the Lyndon factorization of any string T of length N can be computed in $O(N)$ time [30]. Algorithm 1 shows a pseudo-code of the main part of the algorithm that computes the leftmost Lyndon factor $(|\lambda|, e)$ of a given string T. The next Lyndon factor can be computed by running this algorithm on the suffix $T[|\lambda|e + 1..|T|]$ of T. Throughout this thesis, we refer to the algorithm as *Duval's algorithm*.

Algorithm 1: Algorithm to compute the leftmost Lyndon factor of T.

Input: String T.
Output: Leftmost Lyndon factor $(|\lambda|, e)$ of T.

```
1 i ← 1, j ← 2;
2 while j ≤ |T| and T[i] ⪯ T[j] do
3     if T[i] ≺ T[j] then i ← 1; else i ← i + 1;
4     j ← j + 1;
5 (|λ|, e) ← (j − i, ⌊(j − 1)/(j − i)⌋);
6 output (|λ|, e);
```

2.4 Lempel-Ziv Factorizations

In this section, we introduce LZ77 factorizations [92] and LZ78 factorizations [93].

2.4.1 LZ77 factorizations

The *non-overlapping LZ77 factorization* of a string T is its factorization $g_1, \ldots, g_t$ built left to right in a greedy way by the following rule: each g_i is either the leftmost occurrence of a letter in T or the longest prefix of $g_i \cdots g_t$ which occurs in $g_1 \cdots g_{i-1}$. We refer to each g_i as *LZ77 phrase*.

2.4.2 LZ78 factorizations

The *LZ78 factorization* of a string T is a factorization $f_1, \ldots, f_s$ of T, where each $f_i \in \Sigma^+$ for each $1 \leq i \leq s$ is defined as follows: For convenience, let $f_0 = \varepsilon$. Then, $f_i = T[p..p + |f_j|]$ where $p = |f_0 \cdots f_{i-1}| + 1$ and $f_j (0 \leq j < i)$ is the longest previous factor which is a prefix of $T[p..|T|]$. The LZ78 encoding of T is a sequence $(k_1, a_1), \ldots, (k_s, a_s)$ of pairs s.t. each pair (k_i, a_i) represents the i-th LZ78 factor f_i, where k_i is the identifier of the previous factor f_{k_i}, and a_i is the new character $T[|f_1 \cdots f_i|]$. The LZ78 encoding requires $O(s)$ space. Regarding this pair as a parent and edge label, the factors can also be represented as a trie of size $O(s)$, see Figure 2.1.

2.5 Straight Line Programs

A *straight line program* (*SLP*) is a set of productions $\mathcal{S} = \{X_1 \to expr_1, \ldots, X_n \to expr_n\}$, where each X_i is a variable and each $expr_i$ is an expression, where $expr_i = a$ ($a \in \Sigma$), or

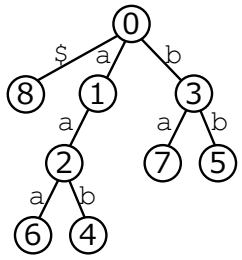

Figure 2.1: LZ78 trie for aaabaabbbaaaba$. Node i represents f_i, e.g., $f_4 =$ aab.

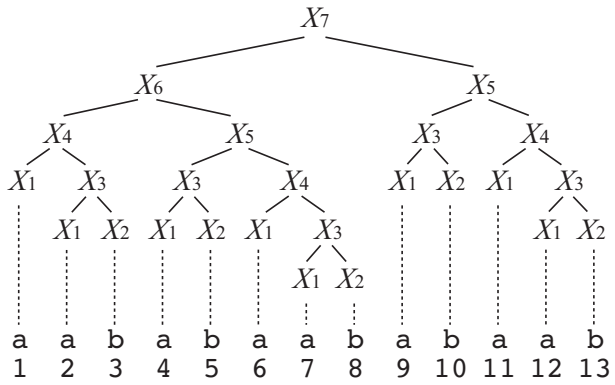

Figure 2.2: The derivation tree of SLP $\mathcal{S} = \{X_1 \to \mathtt{a}, X_2 \to \mathtt{b}, X_3 \to X_1X_2, X_4 \to X_1X_3, X_5 \to X_3X_4, X_6 \to X_4X_5, X_7 \to X_6X_5\}$, representing string $S = val(X_7) =$ aababaababaab.

$expr_i = X_{l(i)}X_{r(i)}$ $(i > l(i), r(i))$ for any $1 \leq i \leq n$. It is essentially a context free grammar in Chomsky normal form, that derives a single string. Let $val(X_i)$ represent the string derived from variable X_i. To ease notation, we sometimes associate $val(X_i)$ with X_i and denote $|val(X_i)|$ as $|X_i|$, and $val(X_i)[u..v]$ as $X_i[u..v]$ for $1 \leq u \leq v \leq |X_i|$. An SLP $\mathcal{S}$ *represents* the string $T = val(X_n)$. The *size* of the program $\mathcal{S}$ is the number n of productions in $\mathcal{S}$. Let N be the length of the string represented by SLP $\mathcal{S}$, i.e., $N = |T|$. Then N can be as large as 2^{n-1}.

The derivation tree of SLP $\mathcal{S}$ is a labeled ordered binary tree where each internal node is labeled with a non-terminal variable in $\{X_1, \ldots, X_n\}$, and each leaf is labeled with a terminal character in Σ. The root node has label X_n. An example of the derivation tree of an SLP is shown in Figure 2.2.

2.5.1 Longest common extension problem on SLPs

The longest common extension (LCE) problem on SLPs is to preprocess an SLP so that we can efficiently answer LCE queries that ask to compute $lcp(val(X_i)[k_1..|X_i|], val(X_i)[k_2..|X_i|])$ for any variable X_i and $1 \leq k_1, k_2 \leq |X_i|$. The currently best data structures to solve this problem

deterministically are the following.

Lemma 3 ([55]). *Given an SLP of size n representing a string of length N, we can preprocess in $O(n \log(N/n))$ time and $O(n + t \log(N/t))$ space to support LCE queries in $O(\log N)$ time where t is the size of non-overlapping LZ77 factorization.*

With the data structure, we are also able to answer the lexicographical order of $val(X_i)[k_1..|X_i|]$ and $val(X_i)[k_2..|X_i|]$ in the same complexities.

In order to describe our complexity independent from the choice of LCE data structures, we denote by $P(n, N)$, $S(n, N)$ and $Q(n, N)$, the preprocessing time, space and query time of an LCE data structure, respectively.

2.6 Computation Model

Our model of computation is the word RAM: We assume the computer word size is at least $\lceil \log_2 |T| \rceil$, and hence, standard operations on values representing lengths and positions of string T can be manipulated in $O(1)$ time. Space complexities will be determined by the number of computer words (not bits).

Chapter 3

Inferring Strings from Lyndon Factorization

In this chapter, we tackle some reverse-engineering problems on Lyndon factorizations. This is the first study for the problems. In Section 3.1, we present an algorithm to compute a string on an alphabet of arbitrary size s.t. its Lyndon factorization corresponds to an input. We also propose an efficient algorithm to compute a string on an alphabet of smallest size. In Section 3.2, we show an algorithm to compute only the smallest alphabet size. In Section 3.3, we solve an enumerating problem.

3.1 Computing a String with Given Lyndon Factorization

In this section, we consider two problems. Firstly, we explain an input of our problems. Since the Lyndon factorization of a string can be represented by ordered pairs of integers (see Section 2.3), an input of each problem considered in this chapter is given as a sequence I of ordered pairs of integers. We naturally assume that each integer is strictly greater than zero.

Throughout this chapter, we assume $\Sigma = \{c_1, \ldots, c_\sigma\}$, where c_i is the i-th "largest" element of Σ for any $1 \leq i \leq \sigma$. Namely, $c_i \succ c_{i+1}$ for any $1 \leq i < \sigma$.

3.1.1 Computing a string on an alphabet of arbitrary size

The simplest variant of our reverse-engineering problem is the following:

Problem 1. *Given a sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ of ordered pairs of positive integers, compute a string $T \in \Sigma^+$ such that $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ and $|\lambda_i| = \ell_i$.*

The length N of T is clearly $N = \sum_{i=1}^{m} \ell_i e_i$. In Problem 1, there is no restriction on the size of the alphabet from which the output string T is drawn. We get the following Theorem 1 by, basically just assigning decreasingly smaller characters to the first character of each factor.

Theorem 1. *Problem 1 can be solved in $O(N)$ time, where N is the length of an output string.*

Proof. We process the input sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ from left to right. Assume we have processed the first j elements of I, and have computed the prefix $T[1..p_j]$ of output string T, where $p_j = \sum_{i=1}^{j}(\ell_i e_i)$. Let $q(0) = 0$, and for $1 \leq j \leq m$ let $q(j)$ be the number of distinct characters occurring in the prefix $T[1..p_j]$. For the j-th element (ℓ_j, e_j) of I, let $\lambda_j = c_{q(j-1)+1}$ if $\ell_j = 1$, and $\lambda_j = c_{q(j-1)+2}(c_{q(j-1)+1})^{\ell_j - 1}$ otherwise. Also, let $\Lambda_j = (\lambda_j)^{e_j}$. It is easy to see λ_j is a Lyndon word. Since $c_{q(j-1)} \succ c_{q(j)}$, $\lambda_{j-1} \succ \lambda_j$ holds. Therefore, string $T = \Lambda_1 \cdots \Lambda_m$ is a solution to the problem. Since we can compute each Λ_j in $O(|\Lambda_j|) = O(\ell_j e_j)$ time, the overall time complexity is $O(\sum_{j=1}^{m} \ell_j e_j) = O(N)$. □

Theorem 1 also implies that for any input sequence I of ordered pairs of positive integers, there exists a string T which is a solution to Problem 1. We show a supplementary example of the simple solution in the following.

Example 1. *Let $\Sigma = \{\mathtt{h}, \mathtt{g}, \mathtt{f}, \mathtt{e}, \mathtt{d}, \mathtt{c}, \mathtt{b}, \mathtt{a}\}$. For an input sequence $I = ((3,1), (2,2), (2,1), (4,1))$, a solution to Problem 1 is $T = \mathtt{ghhefefcdabbb}$, since $LF_T = (\mathtt{ghh})^1, (\mathtt{ef})^2, (\mathtt{cd})^1, (\mathtt{abbb})^1$.*

3.1.2 Computing a string on an alphabet of the smallest size

Now, we consider a more interesting variant of our reverse-engineering problem, where a string on an alphabet of the smallest size is to be computed.

For any $1 \leq j \leq \sigma$, let $\Sigma_j = \{c_1, \ldots, c_j\}$ denote the set of the j largest characters of Σ. The problem is formalized as follows:

Problem 2. *Given a sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ of ordered pairs of positive integers, compute a string $T \in \Sigma_r^+$ such that $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$, $|\lambda_i| = \ell_i$, and r is the smallest possible.*

An example of Problem 2 is shown below. Let $\Sigma = \{\mathtt{d}, \mathtt{c}, \mathtt{b}, \mathtt{a}\}$. For an input sequence $I = ((3,1), (2,2), (2,1), (4,1))$ of positive integers, a solution to Problem 2 is string $T =$

cddcdcdbdbcdd where $LF_T = (\texttt{cdd})^1, (\texttt{cd})^2, (\texttt{bd})^1, (\texttt{bcdd})^1$. Remark that the output string of the example contains only 3 distinct characters, while that of example above contains 8 distinct characters, for the same input sequence I.

In what follows, we present an $O(N)$-time algorithm to solve Problem 2. This algorithm computes Lyndon factors from left to right, and is based on the lemma below.

Lemma 4. *Let $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ be a sequence of ordered pairs of positive integers. If for some string $w \in \Sigma^+$, $LF_w = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$, where λ_1 is the lexicographically largest Lyndon word of length ℓ_1 and for all $2 \leq i \leq m$, λ_i is the lexicographically largest Lyndon word of length ℓ_i which is lexicographically smaller than λ_{i-1}, then, $T \in \Sigma_r^+$ where $\Sigma_r = \{c_1, \ldots, c_r\} \subseteq \Sigma$ and r is the smallest possible.*

Proof. Let $L_1 \succ \cdots \succ L_\alpha$ be the decreasing sequence of Lyndon words on Σ of length at most $\max\{\ell_i \mid 1 \leq i \leq m\}$. Then the sequence $\lambda_1, \ldots, \lambda_m$ is a subsequence of $L_1, \ldots, L_\alpha$, i.e., there exist $1 \leq i_1 < \cdots < i_m \leq \alpha$ s.t. $\lambda_1 = L_{i_1}, \ldots, \lambda_m = L_{i_m}$. If $L_i \in \Sigma_r^+ - \Sigma_{r-1}^+$ for some i, then it must be that $L_i[1] = c_r$ or else L_i cannot be a Lyndon word. Thus, for any Lyndon words $L_i \in \Sigma_{r-1}^+$ and $L_j \in \Sigma_r^+ - \Sigma_{r-1}^+$, $L_i \succ L_j$, and thus $i < j$ holds. As the condition on T indicates that $i_1 = \min\{i \mid |L_i| = \ell_1\}$ and $i_j = \min\{i \mid L_{i_{j-1}} \succ L_i, |L_i| = \ell_j\}$ for any $1 < j \leq m$, i_m is the smallest possible, and thus r is the smallest possible. □

See also Figure 3.1 for an example of Lemma 4. The string T of Lemma 4 is the lexicographically largest string whose Lyndon factorization corresponds to input I. We compute T as defined in Lemma 4. In the rest of this chapter, for any string x and integer k, the lexicographically largest Lyndon word of length k which is lexicographically smaller than x by $PredLyn(x, k)$.

Duval [31] proposed a linear time algorithm which, given a Lyndon word, computes the next Lyndon word (i.e., the lexicographical successor) of the same length. Although our algorithm to be shown in this section is somewhat similar to his algorithm, ours is more general in that it can compute the previous Lyndon word (i.e., the lexicographical predecessor) of a "given length", in linear time.

If $\ell_1 = 1$, then $\lambda_1 = c_1$. If $\ell_1 \geq 2$, then $\lambda_1 = c_2 c_1^{\ell_1 - 1}$. Assume that we have already computed $\lambda_1, \ldots, \lambda_{i-1}$ for $1 < i \leq m$, and we are computing λ_i. In so doing, we will need Lemma 5 and Lemma 6.

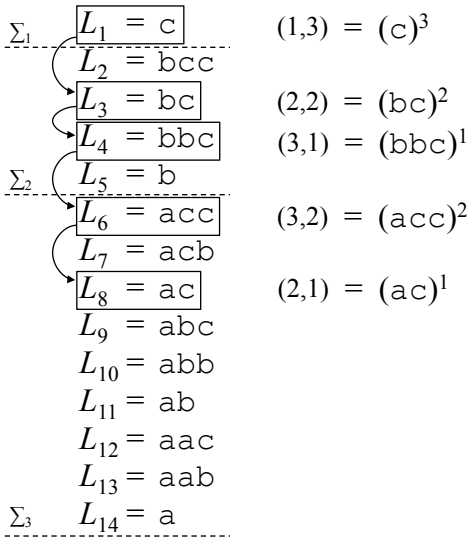

Figure 3.1: An example of Lemma 4. Let $\Sigma = \{\mathtt{c}, \mathtt{b}, \mathtt{a}\}$ and $I = ((1,3),(2,2),(3,1),(3,2),(2,1))$. For example, the first factor c is the lexicographically largest Lyndon word of length 1, and the last factor ac is the lexicographically largest Lyndon word of length 2 which is lexicographically smaller than previous factor acc.

Lemma 5. *Let x be any Lyndon word such that $|x| \geq 2$ and $x[i..|x|] = c_q c_1^{|x|-i}$ for some $1 < i \leq |x|$ and some $1 < q \leq \sigma$. Then, for any $1 \leq p < q$, $y = x[1..i-1]c_p c_1^{|x|-i}$ is a Lyndon word.*

Proof. Firstly, we show that $y[1..i]$ is a Lyndon word. Assume on the contrary that $y[1..i]$ is not a Lyndon word. Then, there exists $2 \leq j \leq i$ satisfying $y[1..i] \succ y[j..i]$. Since $x[1..i-1] = y[1..i-1]$ and $x[i] = c_q \prec c_p = y[i]$, $y[1..i] \succ x[1..i]$ and $y[j..i] \succ x[j..i]$. Since $2 \leq j$, $|y[j..i]| = i - j + 1 \leq i - 1$. Since $y[1..i-1] = x[1..i-1]$, we get $x[1..i] \succ y[j..i]$, which implies that $x[1..i] \succ x[j..i]$. However, this contradicts that x is a Lyndon word. Hence $y[1..i]$ is a Lyndon word.

Now we show y is a Lyndon word by induction on k, where $i \leq k \leq |y|$. The case where $k = i$ has already been shown. Assume $y[1..k]$ is a Lyndon word for $i \leq k < |y|$. As $2 \leq i \leq k$, $y[1..k] \prec y[k+1] = c_1$. Since $y[k+1] = c_1$ is a Lyndon word, by Lemma 1, $y[1..k+1]$ is a Lyndon word. This completes the proof. □

Lemma 6. *For any Lyndon word x with $|x| \geq 2$ and any $1 \leq i \leq |x|$, $y = x[1..i]c_1^{|x|-i}$ is a Lyndon word.*

Proof. Let $k = |x| - i$. We prove the lemma by induction on k. If $k = 0$, i.e. $i = |x|$, then $y = x$ and hence the lemma trivially holds. Assume the lemma holds for some $0 \leq k < |x| - 1$, i.e.,

$x[1..|x|-k]c_1^k = x[1..i]c_1^{|x|-i}$ is a Lyndon word. Then, by Lemma 5, $x[1..|x|-(k+1)]c_1^{k+1} = x[1..i-1]c_1^{|x|-i+1}$ is also a Lyndon word. Hence the lemma holds. □

We show examples of the above lemmas. Let $x = \texttt{abcadd}$ be a Lyndon word and $\Sigma = \{\texttt{d}, \texttt{c}, \texttt{b}, \texttt{a}\}$. By Lemma 5, `abcbdd`, `abccdd`, and `abcddd` are also Lyndon words. By Lemma 6, `abcddd`, `abdddd`, `addddd` are also Lyndon words.

Computing λ_i from λ_{i-1} when $\ell_i = \ell_{i-1}$.

Here, we describe how to compute λ_i from λ_{i-1} when $\ell_i = \ell_{i-1}$, namely, $|\lambda_i| = |\lambda_{i-1}|$. The following is a key lemma:

Lemma 7. *For any non-empty string x, $PredLyn(x, |x|) = x[1..i-1]c_{j+1}c_1^{|x|-i}$ where $x[i] = c_j$ and i is the largest position s.t. $x[1..i-1]c_{j+1}c_1^{|x|-i}$ is a Lyndon word.*

Proof. Let $y = PredLyn(x, |x|)$. Assume on the contrary that there is a Lyndon word λ of length $|x|$ s.t. $y \prec \lambda \prec x$. As $x[1..i-1] = y[1..i-1]$ and $c_j = x[i] \succ y[i] = c_{j+1}$, there is a position $i' > i$ s.t. $\lambda[1..i'-1] = x[1..i'-1]$ and $\lambda[i'] \prec x[i']$. By Lemma 6, $\lambda[1..i']c_1^{|\lambda|-i'} = x[1..i'-1]\lambda[i']c_1^{|x|-i'}$ is a Lyndon word. By Lemma 5, $x[1..i'-1]c_{j'+1}c_1^{|x|-i'}$ is a Lyndon word, where $x[i'] = c_{j'} \succ c_{j'+1} \succeq \lambda[i']$. This contradicts that i is the largest position in x s.t. $x[1..i-1]c_{j+1}c_1^{|x|-i}$ is a Lyndon word. □

Algorithm 2 shows a pseudo-code of our linear-time algorithm to find $PredLyn(x, |x|)$. To efficiently compute i of Lemma 7, we use, as a sub-routine, Duval's algorithm [30] which computes the Lyndon factorization of a string. In the next lemma, we show how Algorithm 2 works and its time complexity.

Lemma 8. *For any non-empty string x, Algorithm 2 computes $PredLyn(x, |x|)$ in $O(|x|)$ time.*

Proof. Let C_x be an array of length $|x|$ such that, for any $1 \le i \le |x|$, $C_x[i] = \max\{q \mid x[i..i+q-1] = c_1^q\}$. Namely, $C_x[i]$ represents the number of consecutive c_1's starting at position i in x. Algorithm 2 firstly computes C_x.

For $1 \le k \le |x|$, let $x_k = x[1..k-1]c_{\hat{j}+1}c_1^{|x|-k}$, where $c_{\hat{j}} = x[k]$. Namely, x_k is the concatenation of the prefix of x of length $k-1$, the lexicographically next character $c_{\hat{j}+1}$ to the character $c_{\hat{j}} = x[k]$, and the repetition of c_1 of length $|x|-k$ (see also Figure 3.2 for illustration of x_k). The algorithm checks whether x_k is a Lyndon word for all $1 \le k \le |x|$ in increasing order of k, based on Duval's algorithm [30].

Algorithm 2: Compute next smaller Lyndon word of same length.

Input: String x.
Output: $PredLyn(x, |x|)$.

```
1  compute C_x;
2  k' ← 1, k ← 2, i ← 0;
3  if x[1] ≠ c_σ then i ← 1;
4  while k ≤ |x| do
5      if x[k] ≠ c_σ then
6          if x[k'] ≺ c_{ĵ+1} then i ← k ;                    // c_ĵ = x[k]
7          else if x[k'] = c_{ĵ+1} then
8              len ← min{C_x[k'+1], k−k'−1};
9              if len < |x| − k then i ← k;
       // Operation of Duval's algorithm.
10     if x[k'] ≺ x[k] then k' ← 1, k ← k+1;
11     else if x[k'] = x[k] then k' ← k'+1, k ← k+1;
12     else break;
13 output x[1..i−1] c_{j+1} c_1^{|x|−i};                      // c_j = x[i]
```

For each k, our algorithm maintains a variable k' to be the largest integer satisfying $x_k[1..k'-1] = x_k[k-k'+1..k-1]$ and $x_k[1..k-1]$ is a Lyndon word (see also Figure 3.2). To check if x_k is a Lyndon word, we compare $x_k[k'] = x[k']$ and $x_k[k] = c_{\hat{j}+1}$ (lines 6 and 7). There are the three following cases:

- If $x[k'] \prec c_{\hat{j}+1}$, then we know that $x[1..k-1]c_{\hat{j}+1}$ is a Lyndon word, due to Duval's algorithm [30]. It follows from Lemma 1 that $x[1..k-1]c_{\hat{j}+1}c_1^{|x|-k}$ is a Lyndon word. The value of i is replaced by k (line 6).

- If $x[k'] \succ c_{\hat{j}+1}$, then we know that $x[1..k-1]c_{\hat{j}+1}c_1^{|x|-k}$ is not a Lyndon word, due to Duval's algorithm [30].

- If $x[k'] = c_{\hat{j}+1}$, then $x[1..k'] = x[k-k'+1..k-1]c_{\hat{j}+1}$. In this case, we compare the substrings immediately following $x[1..k']$ and $x[k-k'+1..k-1]c_{\hat{j}+1}$, respectively. Since $x_k[k+1..|x|] = c_1^{|x|-k}$, if we know the number of consecutive c_1's from position $k'+1$ in x, then we can efficiently check' whether or not x_k is a Lyndon word. Let len be the number of consecutive c_1's from position $k'+1$ in x_k which can be calculated by $\min\{C_x[k'+1], k-k'-1\}$. We compare len with the number of consecutive c_1's from position $k+1$ in x_k, which is clearly $|x|-k$. If $len < |x|-k$, then x_k is a Lyndon word. Otherwise, x_k is not a Lyndon word since it has a border.

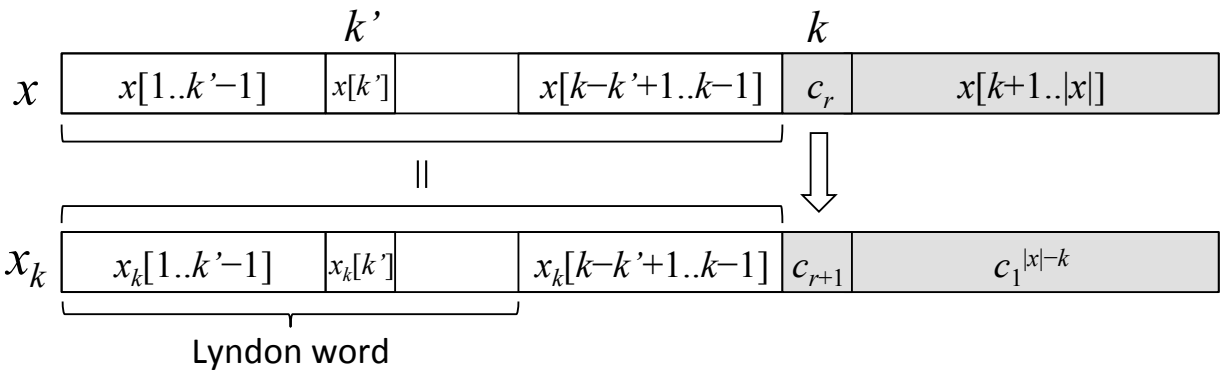

Figure 3.2: Illustration of string x_k in proof of Lemma 8.

In lines 10-12 we update the values of k' and k using Duval's algorithm [30].

After the **while** loop, the variable i stores the largest integer s.t. $x[1..i-1]c_{j+1}c_1^{|x|-i}$ is a Lyndon word, which is the output of the algorithm (line 13).

It is easy to see that C_x can be computed in $O(|x|)$ time. The **while** loop repeats at most $|x|$ times, and each operation in the **while** loop takes constant time. Therefore the overall time complexity is $O(|x|)$. □

Computing λ_i from λ_{i-1} when $\ell_{i-1} \neq \ell_i$.

Here, we show how to compute λ_i from λ_{i-1} when their lengths ℓ_i and ℓ_{i-1} are different. Firstly, we consider the case where $\ell_{i-1} > \ell_i$, namely $|\lambda_{i-1}| > |\lambda_i|$:

Lemma 9. *For any non-empty string x and positive integer $k < |x|$, if $x[1..k]$ is a Lyndon word, then $PredLyn(x,k) = x[1..k]$. Otherwise, $PredLyn(x,k) = x[1..i-1]c_{j+1}c_1^{k-i}$, where $x[i] = c_j$ and i is the largest position s.t. $x[1..i-1]c_{j+1}c_1^{k-i}$ is a Lyndon word.*

Proof. Let $x' = x[1..k]$. No string of length k which is lexicographically smaller than x and larger than x' exists. Thus, if x' is a Lyndon word, $PredLyn(x,k) = x'$. Otherwise, $PredLyn(x,k) = PredLyn(x',k)$. Since $|x'| = k$, the statement follows from Lemma 7. □

Secondly, we consider the case where $\ell_{i-1} < \ell_i$, namely $|\lambda_{i-1}| > |\lambda_i|$.

Lemma 10. *For any non-empty string x and positive integer $k > |x|$, $PredLyn(x,k) = x[1..i-1]c_{j+1}c_1^{k-i}$, where $x[i] = c_j$ and i is the largest position s.t. $x[1..i-1]c_{j+1}c_1^{k-i}$ is a Lyndon word and $1 \leq i \leq |x|$.*

Proof. Let $y = PredLyn(x, k)$. Assume on the contrary that there exists a Lyndon word λ of length k s.t. $y \prec \lambda \prec x$. This implies that $\lambda[1..i'-1] = x[1..i'-1]$ and $\lambda[i'] \prec x[i']$ with $i < i' < |x|$. The rest of proof is in a similar way of Lemma 7. □

Due to the two above lemmas, λ_i can be computed from λ_{i-1} in a similar way to the case where $|\lambda_i| = |\lambda_{i-1}|$, using a slightly modified version of Algorithm 2, as described in the following theorem.

Theorem 2. *Problem 2 can be solved in $O(N)$ time, where N is the length of an output string.*

Proof. Assume we have already computed $\lambda_1, \ldots, \lambda_{i-1}$ and are computing λ_i.

- If $|\lambda_{i-1}| > |\lambda_i|$, then let x be the prefix of λ_{i-1} of length ℓ_i, namely, $x = \lambda_{i-1}[1..\ell_i]$. By Lemma 9, if x is a Lyndon word, then $\lambda_i = x$. We can check whether x is a Lyndon word or not in $O(|x|)$ time, by using Duval's algorithm [30]. Otherwise, λ_i can be computed from x by Algorithm 2. This takes $O(|\lambda_i|) = O(\ell_i)$ time by Lemma 8.
- If $|\lambda_{i-1}| = |\lambda_i|$, then λ_i can be computed from λ_{i-1} in $O(\ell_i)$ time, by Lemma 8.
- If $|\lambda_{i-1}| < |\lambda_i|$, then let $x = \lambda_{i-1} c_1^{|\lambda_i| - |\lambda_{i-1}|}$. We take x as input to Algorithm 2, with a slight modification to the algorithm. Since $\lambda_{i-1} \succ x \succ \lambda_i$ must hold, we are only interested in positions from 1 to $|\lambda_{i-1}|$ in x. Hence, as soon as the value of k in Algorithm 2 exceeds λ_{i-1}, we exit from the **while** loop, and the resulting string is λ_{i-1}. The above modification clearly does not affect the time complexity of the algorithm, and hence it takes $O(\ell_i)$ time.

Thus we can compute the output string in $O(\sum_{i=1}^{m} \ell_i e_i) = O(N)$ time. □

We can remark that for computing $\lambda_1, \ldots, \lambda_m$ we do not use $e_1, \ldots, e_m$, and hence, the following corollaries are immediate from Theorem 2.

Corollary 1. *We can compute the Lyndon factorization $\lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ of a string which is a solution to Problem 2 in $O(\sum_{i=1}^{m} \ell_i)$ time.*

Corollary 2. *Given a sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ of ordered pairs of integers and an integer $r \geq 1$, we can determine in $O(\sum_{i=1}^{m} \ell_i)$ time if there exists a string T over an alphabet of size at most r s.t. $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ and $|\lambda_i| = \ell_i$.*

3.2 Computing the Smallest Alphabet Size

In this section we consider the following problem.

Problem 3. *Given a sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ of ordered pairs of positive integers, compute the smallest integer r for which there exists a string $T \in \Sigma_r^+$ such that $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ and $|\lambda_i| = \ell_i$.*

An example of Problem 3 is shown below.

Example 2. *For input sequence $I = ((3,1),(2,2),(2,1),(4,1))$, the solution is $r = 3$. This is because for string $T =$ `bccbcbcacabcc` over alphabet $\{\mathtt{c}, \mathtt{b}, \mathtt{a}\}$, $LF_T = (\mathtt{bcc})^1$, $(\mathtt{bc})^2$, $(\mathtt{ac})^1$, $(\mathtt{abcc})^1$, and there is no string over an alphabet of size two or one whose Lyndon factorization coincides with I.*

Clearly, this problem can be solved in $O(N)$ time by Theorem 2. However, since only the smallest alphabet size is of interest, a string does not have to be computed in this problem. To this end, we present an optimal $O(m)$-time algorithm to solve Problem 3, where $m \leq n$ is the size of the input sequence I. The basic strategy is the same as the previous algorithm, i.e., we simulate the algorithm of computing λ_i from λ_{i-1}, for all $1 < i \leq m$. The difficulty is that, in order to achieve an $O(m)$-time algorithm, we cannot afford to store λ_i's explicitly. Hence, we simulate the previous algorithm on a compact representation of λ_i's.

We introduce *the largest character block encoding ($LCBE$)* Y of a non-empty string x. Consider factorizing x into blocks according to the following rules; the b-th block Y_b is the longest prefix of $x[pos(Y_b)..|x|]$ s.t. $Y_b = c_j c_1^q$ for some $j \geq 1$ and $q \geq 0$, where $pos(Y_b) = 1$ if $b = 1$, and $pos(Y_b) = pos(Y_{b-1}) + |Y_{b-1}|$ otherwise. Let $\|Y\|$ denote the number of blocks of Y, i.e., $x = Y = Y_1 Y_2 \ldots Y_{\|Y\|}$. For any $1 < b \leq \|Y\|$, $Y_b[1] \neq c_1$. Notice that Y can be encoded in $O(\|Y\|)$ space by storing $Y_b[1]$ and $pos(Y_b)$ for each block.

Let Y and Y' be the $LCBE$ of λ_{i-1} and λ_i, respectively. It holds that $\|Y'\| \leq \|Y\| + 1$ and $\|Y\| \leq m$ because of the algorithm described in the previous subsection. To compute Y' from Y efficiently, each block Y_b stores the information about the position k' s.t. $Y[k']$ and $Y[pos(Y_b)]$ are supposed to be compared in Algorithm 2. Since Y is a Lyndon word, $Y[k'] \preceq Y[pos(Y_b)] \prec c_1$ and there exists a block starting at k'. Also, $Y[1..k'-1]$ is the longest prefix which is a suffix of $Y[1..pos(Y_b)-1]$. Thus we let Y_b have the value $pbi(Y_b) = \max\{b' \mid 1 \leq b' < b, Y[1..pos(Y_{b'})-1] = Y[pos(Y_b) - pos(Y_{b'}) + 1..pos(Y_b)-1]\}$, that is, $b' = pbi(Y_b)$

is the block index s.t. $k' = pos(Y_{b'})$. We can let $pbi(Y_1)$ remain undefined since we will never use it. An example of $LCBE$ follows.

Example 3. *Let* $\Sigma = \{\mathtt{c}, \mathtt{b}, \mathtt{a}\}$ *and* $\lambda_{i-1} = \mathtt{acaccacbacaccbcc}$. *Then the* $LCBE$ *of* λ_{i-1} *is* $Y = \mathtt{ac}, \mathtt{acc}, \mathtt{ac}, \mathtt{b}, \mathtt{ac}, \mathtt{acc}, \mathtt{bcc}$ *and* $pbi(Y_2) = 1, pbi(Y_3) = 1, pbi(Y_4) = 2, pbi(Y_5) = 1, pbi(Y_6) = 2, pbi(Y_7) = 3$.

Lemma 11 shows how to efficiently compute Y' from Y using pbi. Since Y and Y' share at least the first $\|Y'\| - 2$ blocks, we do not build Y' from scratch.

Lemma 11. *We can compute* $LCBE$ Y' *of* λ_i *in* $O(\max\{1, \|Y\| - \|Y'\|\})$ *time from the given* $LCBE$ Y *of* λ_{i-1} *with* pbi *(see also Algorithm 3).*

Proof. Since it is trivial when $Y = c_1$, we consider the case where $Y \neq c_1$ and $Y[1] \prec c_1$. We simulate the task described in Theorem 2. First, we adjust the length of Y to ℓ_i, i.e., add $c_1^{\ell_i - \ell_{i-1}}$ if $\ell_{i-1} < \ell_i$, or truncate Y to represent $Y[1..\ell_i]$ if $\ell_{i-1} > \ell_i$. A major difference from the algorithm of Theorem 2 is that we process the blocks from right to left, checking whether each block contains the position k s.t. $Y[1..k-1]c_{j+1}c_1^{|Y|-k}$ is a Lyndon word, where $c_j = Y[k]$. We show each block Y_b can be investigated in $O(1)$ time by using $LCBE$ and pbi.

For any $1 < k \leq |Y|$, let $p(k)$ be the position k' s.t. $Y[1..k'-1]$ is the longest prefix of Y which is a suffix of $Y[1..k-1]$. In Algorithm 2, $Y[k]$ is compared with $Y[p(k)]$. As described in Lemma 8, for any $1 < k \leq |Y|$ with $Y[p(k)] \prec Y[k] = c_j$, $Y[1..k-1]c_{j+1}c_1^{|Y|-k}$ is a Lyndon word iff $Y[p(k)] \prec c_{j+1}$ or $|Y| - k > d$, where d is the maximum repeat of c_1's as a prefix of $Y[p(k)+1..k-1]$.

Consider the case where $b \neq 1$. Let $b' = pbi(Y_b)$. Since we know $p(pos(Y_b)) = pos(Y_{b'})$, position $\mu = pos(Y_b) + lcp(Y_b, Y_{b'})$ is the leftmost position inside Y_b s.t. $Y[p(\mu)] \prec Y[\mu]$ if $|Y_b| > lcp(Y_b, Y_{b'})$. By the definition of $LCBE$ and that Y is a prefix of a Lyndon word, $lcp(Y_b, Y_{b'}) = 0$ if $Y_b[1] \neq Y_{b'}[1]$, and $lcp(Y_b, Y_{b'}) = |Y_{b'}|$ otherwise. For any k with $\mu < k \leq pos(Y_b) + |Y_b| - 1$, $Y[p(k)] \prec Y[k] = c_1$ holds since $Y[1] \prec c_1$ and $p(k) = 1$. Since we can compute d from the information of $LCBE$ in $O(1)$ time, we can check if $Y[1..\mu-1]c_{j+1}c_1^{|Y|-\mu}$ is a Lyndon word or not in $O(1)$ time. Since $p(\mu'') = 1$ and $Y[\mu''] = c_1$ for any μ'' with $\mu < \mu'' \leq pos(Y_b) + |Y_b| - 1$, if $Y[1] \prec c_2$ or $|Y| - \mu'' > d'$, then $Y[1..\mu''-1]c_{j+1}c_1^{|Y|-\mu''}$ is a Lyndon word, where d' is the maximum integer s.t. $c_1^{d'}$ is a prefix of $Y[2..\mu''-1]$. Hence by a simple arithmetic operation we can compute in $O(1)$ time the largest position μ' with

$\mu < \mu' \leq pos(Y_b) + |Y_b| - 1$ s.t. $Y[1..\mu' - 1]c_2 c_1^{|Y|-\mu'}$ is a Lyndon word. A minor remark is that we add a constraint for μ' not to exceed ℓ_{i-1} when $\ell_{i-1} < \ell_i$ and $b = \|Y\|$.

The case where $b = 1$ can be managed in a similar way to the case where $\mu < \mu'' \leq pos(Y_b) + |Y_b| - 1$ described above, and hence it takes $O(1)$ time.

We check each block from right to left until we find the largest position k s.t. $Y[1..k-1]c_{j+1}c_1^{|Y|-k}$ is a Lyndon word, where $c_j = Y[k]$. Since each block can be checked in $O(1)$ time, the whole computational time is $O(\max\{1, \|Y\| - \|Y'\|\})$.

Since pbi for the blocks in Y' other than the last block remain unchanged from pbi for Y, it suffices to calculate pbi for the last block of Y'. Let $b = \|Y'\|$. Assume $b > 1$ since no updates are needed when $b = 1$. Then $pbi(Y'_b) = pbi(Y'_{b-1}) + 1$ if $b - 1 \geq 2$ and $Y'_{pbi(Y'_{b-1})} = Y'_{b-1}$, and $pbi(Y'_b) = 1$ otherwise. □

Theorem 3. *Problem 3 can be solved in $O(m)$ time and $O(m)$ space.*

Proof. We begin with $LCBE$ of λ_1 and transform it to $LCBE$ of $\lambda_2, \lambda_3, \ldots, \lambda_m$ in increasing order, using Lemma 11. Finally we get $LCBE$ of λ_m and we can obtain the alphabet size by looking into the first character of λ_m.

Let $B_1, B_2, \ldots, B_m$ denote the number of blocks in $LCBE$s of $\lambda_1, \lambda_2, \ldots, \lambda_m$, respectively. Clearly $B_1 = 1$. By Lemma 11, the total time complexity to get $LCBE$ of λ_m is $O(\sum_{i=2}^{m} \max\{1, B_{i-1} - B_i\}) = O(m + B_1 - B_m) = O(m)$. □

3.2.1 Computing a string on an alphabet of the smallest size in a compact representation

We also remark that an $O(m)$-size compact representation of the lexicographically largest solution for Problem 2 can be computed in $O(m)$ time through the algorithm described in the above. To do so, we store all $LCBE$'s of $\lambda_1, \lambda_2, \ldots, \lambda_m$ as a tree where the common prefix blocks are shared. Using this representation, we can obtain the desired string in $O(N)$ time.

Our compact representation is defined as follows:

- each node is labeled with pair (c, q) s.t. $c \in \Sigma$, q is an integer,
- a pair (c, q) represents a block cc_1^q,
- a path from the root to a node represents a string that is the concatenation of strings represented by each node on the path,

Algorithm 3: Algorithm to convert $LCBE\ Y$ of λ_i into that of λ_{i-1}.

Input: $LCBE\ Y$ of λ_{i-1} with pbi.
Result: Make Y to be the $LCBE$ of λ_i with pbi.

if $Y = c_1$ **then** return $Y \leftarrow c_2 c_1^{\ell_i - 1}$;
if $\ell_{i-1} < \ell_i$ **then** $Y \leftarrow Y c_1^{\ell_i - \ell_{i-1}}$;
else if $\ell_{i-1} > \ell_i$ **then**
    $Y \leftarrow Y[1..\ell_i]$;
    if Y *is a Lyndon word* **then** return Y;
$b \leftarrow ||Y||$;
while true do
    if $\ell_{i-1} < \ell_i$ *and* $b = ||Y||$ **then** $\eta \leftarrow \ell_{i-1}$;
    else $\eta \leftarrow pos(Y_b) + |Y_b| - 1$;
    if $b = 1$ **then**
        if $Y_1[1] \prec c_2$ **then** return $Y \leftarrow Y[1..\eta - 1] c_{j+1} c_1^{\ell_i - \eta}$; // $c_j = Y[\eta]$
        $\mu' \leftarrow \max\{\mu'' \leq \eta \mid \ell_i - \mu'' \geq \mu'' - 1\}$;
        return $Y \leftarrow Y[1..\mu' - 1] c_{j+1} c_1^{\ell_i - \mu'}$; // $c_j = Y[\mu']$
    // In what follows, $b \neq 1$.
    $b' \leftarrow pbi(Y_b)$;
    if $Y_b[1] = Y_{b'}[1]$ **then**
        if $|Y_b| = |Y_{b'}|$ **then** $b \leftarrow b - 1$, continue;
        $\mu \leftarrow pos(Y_b) + |Y_{b'}|$, $b' \leftarrow b' + 1$;
    else $\mu \leftarrow pos(Y_b)$;
    if $\mu < \eta$ **then**
        if $Y_1[1] \prec c_2$ **then** return $Y \leftarrow Y[1..\eta - 1] c_{j+1} c_1^{\ell_i - \eta}$; // $c_j = Y[\eta]$
        $\mu' \leftarrow \max\{\mu'' \leq \eta \mid \ell_i - \mu'' \geq |Y_1|\}$;
        if $\mu < \mu'$ **then** return $Y \leftarrow Y[1..\mu' - 1] c_{j+1} c_1^{\ell_i - \mu'}$; // $c_j = Y[\mu']$
    if $Y_{b'}[1] = c_{j+1}$ **then** // $c_j = Y[\mu]$
        if $b = b'$ **then** $d \leftarrow \mu - pos(Y_b)$;
        else $d \leftarrow |Y_{b'}|$;
        if $\ell_i - \mu < d$ **then** $b \leftarrow b - 1$, continue;
    return $Y \leftarrow Y[1..\mu - 1] c_{j+1} c_1^{\ell_i - \mu}$; // $c_j = Y[\mu]$

- a path from the root to a leaf represents a Lyndon factor of T,
- leaves are sorted in the order of Lyndon factors.

We show an example in Figure 3.3 for $T =$ `bcbbcccbbcbcbbcbbccac`.

By the definition, each node corresponds to a block of the $LCBE$. Thus the number of nodes is $O(m)$, and the number of edges is also $O(m)$. Therefore, this representation takes $O(m)$ space. We can also compute this representation by using the algorithm described in the

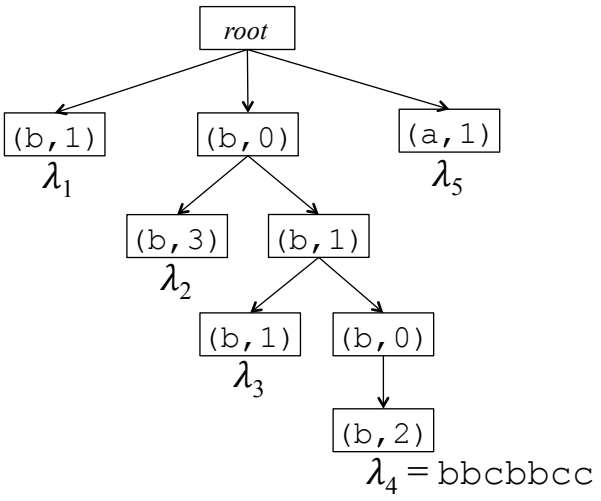

Figure 3.3: Let $I = ((2,1),(5,1),(5,1),(7,1),(2,1))$. Then the solution of Problem 3 is $T = \texttt{bcbbcccbbcbcbbcbbccac}$, since $LF_T = (\texttt{bc})^1, (\texttt{bbccc})^1, (\texttt{bbcbc})^1, (\texttt{bbcbbcc})^1, (\texttt{ac})^1$. This trie represents LF_T. For example, $\lambda_4 = \texttt{bbcbbcc}$.

above.

Theorem 4. *Given a sequence of integer pairs $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$, we can compute $O(m)$-representation of a string T over an alphabet of the smallest size in $O(m)$ time s.t. $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$, $|\lambda_i| = \ell_i$.*

3.3 Enumerate Strings with Given Lyndon Factorization

In this section, we consider a problem of enumerating all strings whose Lyndon factorizations correspond to a given sequence of integer pairs:

Problem 4. *Given a sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ of ordered pairs of positive integers and $\Sigma_r = \{c_1, \ldots, c_r\}$, compute all strings $T \in \Sigma_r^+$ such that $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ and $|\lambda_i| = \ell_i$.*

We show an example of this problem.

Example 4. *Let $\Sigma_r = \{\texttt{c}, \texttt{b}, \texttt{a}\}$. For an input sequence $I = ((3,1),(2,2),(2,1))$ of positive integers, a solution to Problem 4 is $T =$* bccbcbcac, bccbcbcab, bccacacab, bbcacacab, accacacab, acbacacab, *since $LF_T = ((\texttt{bcc})^1, (\texttt{bc})^2, (\texttt{ac})^1)$, $((\texttt{bcc})^1, (\texttt{bc})^2, (\texttt{ab})^1)$, $((\texttt{bcc})^1, (\texttt{ac})^2, (\texttt{ab})^1)$, $((\texttt{bbc})^1, (\texttt{ac})^2, (\texttt{ab})^1)$, $((\texttt{acc})^1, (\texttt{ac})^2, (\texttt{ab})^1)$, $((\texttt{acb})^1, (\texttt{ac})^2, (\texttt{ab})^1)$.*

Let K be the set of output strings for Problem 4. We consider a tree *lfTree* defined as follows. Let `root` be the root of *lfTree*.

- `root` is $T_{\texttt{root}}$, which is the lexicographically largest string in K, computed by the algorithm of Section 3.1.2;
- Each child v of any node u is a pair (T_v, j) of a string and integer j, such that T_v is the string obtained by replacing the j-th factor $\lambda_j^{e_j}$ of the Lyndon factorization of T_u with $(\lambda'_j)^{e_j}$, where λ'_j is a Lyndon word of length ℓ_j satisfying $\lambda_j \succ \lambda'_j \succ \lambda_{j+1}$ (λ_{m+1} denotes ε for convenience);
- For any non-root node $u = (T_u, i)$ and any child $v = (T_v, j)$ of u, $i > j$.

The next lemma shows that *lfTree* represents all and only the strings in K.

Lemma 12. *Let $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ be a sequence of ordered pairs of positive integers. Then lfTree contains all and only strings $T \in \Sigma_r^+$ s.t. $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ with $|\lambda_i| = \ell_i$ for all $1 \leq i \leq m$.*

Proof. If T is the lexicographically largest string in K, then it is represented by `root`, i.e., $T = T_{\texttt{root}}$. Otherwise, let $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$ and $LF_{T_{\texttt{root}}} = (\hat{\lambda}_1)^{e_1}, \ldots, (\hat{\lambda}_m)^{e_m}$. Let $J = \{j \mid \hat{\lambda}_j \succ \lambda_j\}$ and $\mu = |J|$. For any $1 \leq i \leq \mu$, let j_i denote the i-th smallest element of J. The node u that corresponds to T can be located from `root`, as follows. By the definition of *lfTree*, (λ_{j_μ}, j_μ) is a child of `root`. Assume we have arrived at a non-root node $v_{j_i} = (\lambda_{j_i}, j_i)$ with $1 < i \leq \mu$. Let $LF_{T_v} = x_1^{e_1}, \ldots, x_m^{e_m}$. Then, for any $k < j_i$, $x_k = \hat{\lambda}_k$. Thus we have that $\hat{\lambda}_{j_{i-1}} = x_{j_{i-1}} \succ \lambda_{j_{i-1}} \succ \lambda_{j_{i-1}+1} = x_{j_{i-1}+1}$. This implies that $v_{j_{i-1}} = (\lambda_{j_{i-1}}, j_{i-1})$ is a child of v_{j_i}. Note that for any $k' > j_{i-1}$, $x_{k'} = \lambda_{k'}$. Hence, $v_{j_1} = u$, the desired node which corresponds to T.

By the definition of *lfTree*, any string corresponding to a node of *lfTree* is in K. □

A naïve representation of *lfTree* requires $O(|K|N)$ space. To reduce the output size of Problem 4, we introduce the following compact representation of *lfTree*:

- `root` is $LF_{T_{\texttt{root}}} = (\hat{\lambda}_1)^{e_1}, \ldots, (\hat{\lambda}_m)^{e_m}$, where $T_{\texttt{root}}$ is the lexicographically largest string in K, computed by the algorithm of Section 3.1.2;

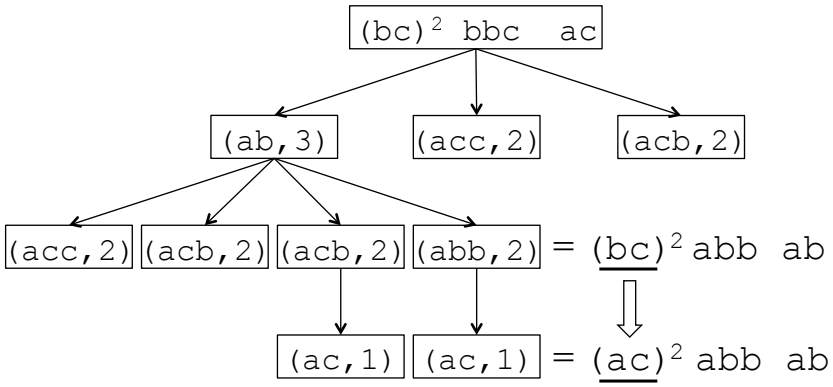

Figure 3.4: This figure shows the tree *lfTree* when $I = ((2,2),(3,1),(2,1))$ and $\Sigma = \{\mathtt{c},\mathtt{b},\mathtt{a}\}$.

- Each child v of any node u is a pair (λ'_j, j) of a Lyndon word λ'_j of length ℓ_j and integer j, such that LF_{T_v} is obtained by replacing the j-th factor $\lambda_j^{e_j}$ of the Lyndon factorization LF_{T_u} of T_u with $(\lambda'_j)^{e_j}$, where λ'_j satisfies $\lambda_j \succ \lambda'_j \succ \lambda_{j+1}$ (λ_{m+1} denotes ε for convenience);

- For any non-root node $u = (\lambda'_i, i)$ and any child $v = (\lambda''_j, j)$ of u, $i > j$.

Let $\ell_{\max} = \max\{\ell_i \mid 1 \le i \le m\}$ and $N' = \sum_{i=1}^{m} \ell_i$. Then, this compact representation of T requires only $O(|K|\ell_{\max} + N')$ space. In the sequel, we mean by *lfTree* the compact representation of *lfTree*. An example of *lfTree* is given in Figure 3.4.

The number of nodes in *lfTree* is $|K|$. Since the root has $LF_{T_{root}}$ and other nodes have a pair of a factor and an integer, *lfTree* takes $O(|K|\ell_{\max} + N')$ space.

We show how to construct *lfTree* in linear time. Let $LF_{T_{\mathtt{root}}} = (\hat{\lambda}_1)^{e_1}, \ldots, (\hat{\lambda}_m)^{e_m}$. Let H be the set of integers η ($1 \le \eta \le m$) s.t. there exists a Lyndon word $\tilde{\lambda}_\eta$ satisfying $\hat{\lambda}_\eta \succ \tilde{\lambda}_\eta \succ \hat{\lambda}_{\eta+1}$, where $\hat{\lambda}_{m+1}$ is the empty string ε for convenience.

Lemma 13. *Given a sequence $I = ((\ell_1, e_1), \ldots, (\ell_m, e_m))$ of ordered pairs of integers, H can be computed in $O(N')$ time, where $N' = \Sigma_{i=1}^{m} \ell_i$.*

Proof. We compute $LF_{T_{\mathtt{root}}} = (\hat{\lambda}_1)^{e_1}, \ldots, (\hat{\lambda}_m)^{e_m}$ in $O(N')$ time by Corollary 1. For each $1 \le \eta \le m$, we apply Algorithm 2 to $\hat{\lambda}_\eta$ and compute the lexicographically largest Lyndon word $\tilde{\lambda}_\eta$ of length $|\tilde{\lambda}_\eta| = \ell_\eta$ that is lexicographically smaller than $\hat{\lambda}_\eta$. If $\eta < m$, then we lexicographically compare $\tilde{\lambda}_\eta$ with $\hat{\lambda}_{\eta+1}$, and $\eta \in H$ only if $\tilde{\lambda}_\eta \succ \hat{\lambda}_{\eta+1}$. This takes $O(\ell_\eta)$ time for each η. If $\eta = m$, then $m \in H$ only if $\tilde{\lambda}_m$ contains at most $r = |\Sigma_r|$ distinct characters. This can be easily checked in $O(\ell_m)$ time. Hence, it takes a total of $O(N')$ time for all $1 \le \eta \le m$. □

Theorem 5. *An $O(|K|\ell_{\max} + N')$-size representation of the solution to Problem 4 can be computed in $O(|K|\ell_{\max} + N')$ time with $O(N')$ extra working space, where $|K|$ is the number of strings corresponding to a given I, $\ell_{\max} = \max\{\ell_i \mid 1 \leq i \leq m\}$ and $N' = \sum_{i=1}^{m} \ell_i$.*

Proof. We first check if there is a string over a given alphabet Σ_k whose Lyndon factorization corresponds to the input sequence I, in $O(N')$ time by Corollary 2. If there exist such strings, then $K = \emptyset$ and thus the theorem holds.

Assume $K \neq \emptyset$. The root node `root` of T is the Lyndon factorization $LF_{w_{\texttt{root}}} = (\hat{\lambda}_1)^{e_1}, \ldots, (\hat{\lambda}_m)^{e_m}$, which can be computed in $O(N')$ by Corollary 1. Then, we compute the children of `root` as follows. For all $\eta \in H$, we compute all the Lyndon words $\tilde{\lambda}_\eta$ of length $|\hat{\lambda}_\eta| = \ell_\eta$ that satisfy $\hat{\lambda}_\eta \succ \tilde{\lambda}_\eta \succ \hat{\lambda}_{\eta+1}$, over alphabet Σ_r. Each of these Lyndon words can be computed in $O(\ell_\eta) = O(\ell_{\max})$ time by Lemma 8.

Given a non-root node $u = (\lambda_j, j)$, we compute the children of u as follows. If $j = 1$, then u is a leaf and has no children. Otherwise, we first compute all the Lyndon words λ'_{j-1} of length $|\hat{\lambda}_{j-1}| = \ell_{j-1}$ that satisfy $\hat{\lambda}_{j-1} \succ \lambda'_{j-1} \succ \lambda_j$, over alphabet Σ_r. Then, for all $\eta \in H \cap \{1, .., j-2\}$, we compute all the Lyndon words λ'_η of length $|\hat{\lambda}_\eta| = \ell_\eta$ that satisfy $\hat{\lambda}_\eta \succ \lambda'_\eta \succ \hat{\lambda}_{\eta+1}$, over alphabet Σ_r. Each of these Lyndon words can be computed in $O(\ell_{\max})$ time as well.

We can compute H in $O(N')$ time by Lemma 13. Since each node can be computed in $O(\ell_{\max})$ time as above, the total running time for constructing *lfTree* is $O(|K|\ell_{\max} + N')$. We need extra $O(N')$ working space to store H.

We show the correctness of the algorithm. Clearly, all the children of `root` are computed by the above algorithm. Consider any non-root node $u = (\lambda_j, j)$ of *lfTree*. Let T_u be the string that corresponds to node u. Since $\hat{\lambda}_j \succ \lambda_j$, if $j \geq 2$, then there may exist some Lyndon words λ'_{j-1} of length $|\hat{\lambda}_{j-1}|$ with $\hat{\lambda}_{j-1} \succ \lambda'_{j-1} \succ \lambda_j$. All such Lyndon words over Σ_r are computed by the above algorithm. Consider the other children of u. Since the first $j-1$ factors of LF_{T_u} are $(\hat{\lambda}_1)^{e_1}, \ldots, (\hat{\lambda}_{j-1})^{e_{j-1}}$, all the Lyndon words λ'_η of length $|\hat{\lambda}_\eta|$ satisfying $\hat{\lambda}_\eta \succ \lambda'_\eta \succ \hat{\lambda}_{\eta+1}$ with $\eta \in H \cap \{1, ..., j-2\}$ correspond to the children of u. All these Lyndon words over Σ_r are also computed by the above algorithm. This completes the proof.

□

3.4 Conclusions

In this chapter, we considered the reverse-engineering problems on Lyndon factorizations. This work is the first contribution of these problems. In Section 3.1, we presented a linear time algorithm to compute a string T on an alphabet of the smallest size such that the Lyndon factorization of T can be represented by an input sequence I. In Section 3.2, we also presented an efficient algorithm to compute only the smallest alphabet size (we do not compute a string T explicitly). Due to this technique, we can represent a string T which is obtained by our algorithm in $O(m)$ space where m is the number of the Lyndon factorization. In Section 3.3, we considered an enumerating version of our problem. For this problem, we proposed a compact representation of all valid strings and an efficient algorithm to compute the representation.

One of the most important points of our algorithms is how to generate the next Lyndon word of a given length. Duval [31] proposed a linear time algorithm which, given a Lyndon word, computes the next Lyndon word (i.e., the lexicographical successor) of the same length. Ours is more general in that it can compute the previous Lyndon word (i.e., the lexicographical predecessor) of a "given length", in linear time.

A remaining our interest for the reverse-engineering problem on Lyndon factorization is a counting version of the problem. Can we compute only the number of strings such that the Lyndon factorizations can be represented by I without computing all strings explicitly?

Chapter 4

Lyndon Factorization Algorithm for Compressed Text

In this chapter, we propose a Lyndon factorization algorithm for a compressed string. We focus on a grammar compressed string called an SLP as an input string.

4.1 Computing Lyndon Factorization from SLP

The problem we tackle in this chapter is the following.

Problem 5. *Given an SLP $\mathcal{S}$ representing a string T, compute the Lyndon factorization LF_T.*

In this section, we show how, given an SLP $\mathcal{S}$ of n productions representing string T, we can compute LF_T of size m in $O(nh(P(n,N)+Q(n,N)\log N)+n^3h)$ time. We will make use of the following known results:

Lemma 14 ([30])**.** *For any string T, let $LF_T = \lambda_1^{e_1}, \ldots, \lambda_m^{e_m}$. Then, $\lambda_m = \min_{\prec} Suffix(w)$, i.e., λ_m is the lexicographically smallest suffix of T.*

Lemma 15 ([71])**.** *Given an SLP $\mathcal{S}$ of size n representing a string T of length N, and two integers $1 \leq i \leq j \leq N$, we can compute in $O(n)$ time another SLP of size $O(n)$ representing the substring $T[i..j]$.*

Lemma 16 ([59])**.** *Given an SLP $\mathcal{S}$ of size n representing a string T of length N, we can compute the shortest period of T in $O(n^2h)$ time and $O(n^2)$ space.*

By Lemma 14, Problem 5 is reduced to the following sub-problem:

Problem 6. *Given an SLP $\mathcal{S}$ representing a string T, compute the length of the lexicographically smallest suffix of T.*

For any non-empty string $w \in \Sigma^+$, let $LFCand(w) = \{x \mid x \in \mathit{Suffix}(w), \exists y \in \Sigma^+ \text{ s.t. } xy = \min_{\prec} \mathit{Suffix}(wy)\}$. Intuitively, $LFCand(w)$ is the set of suffixes of w which are a prefix of the lexicographically smallest suffix of string wy, for some non-empty string $y \in \Sigma^+$.

The following lemma may be almost trivial, but will play a central role in our algorithm.

Lemma 17. *For any two strings $u, v \in LFCand(w)$ with $|u| < |v|$, u is a prefix of v.*

Proof. If $v[1..|u|] \prec u$, then for any non-empty string y, $vy \prec uy$. However, this contradicts that $u \in LFCand(w)$. If $v[1..|u|] \succ u$, then for any non-empty string y, $vy \succ uy$. However, this contradicts that $v \in LFCand(w)$. Hence we have $v[1..|u|] = u$. □

Lemma 18. *For any string w, let $\lambda = \min_{\prec} \mathit{Suffix}(w)$. Then, the shortest string of $LFCand(w)$ is λ^p, where $p \geq 1$ is the maximum integer such that λ^p is a suffix of w.*

Proof. For any string $x \in LFCand(w)$, and any non-empty string y, $xy = \min_{\prec} \mathit{Suffix}(wy)$ holds only if $y \in \{y' \mid \lambda \prec \min_{\prec} \mathit{Suffix}(y')\}$. Let $M = \{y' \mid \lambda \prec \min_{\prec} \mathit{Suffix}(y')\}$.

Firstly, we compare λ^p with the suffixes x of w shorter than λ^p, and show that $\lambda^p y \prec xy$ holds for *any* $y \in M$. Such suffixes x are divided into two groups: (1) If x is of form λ^k for any integer $1 \leq k < p$, then $\lambda^p y \prec \lambda^k y = xy \prec y$ holds for any $y \succ \lambda$; (2) If x is not of form λ^k, then since λ is border-free, λ is not a prefix of x, and x is not a prefix of λ, either. Thus $\lambda \lhd x$ holds, implying that $\lambda^p y \lhd xy$ for any y.

Secondly, we compare λ^p with the suffixes x of w longer than λ^p, and show that $\lambda^p y \prec xy$ holds for *some* $y \in M$. Since $\lambda = \min_{\prec} \mathit{Suffix}(w)$ and $|x| > |\lambda^p|$, $x \succ \lambda^p$ holds. If $x \rhd \lambda^p$, then $\lambda^p y \prec xy$. Let $x = \lambda^k u$ s.t. $q \geq p$ is the maximum integer such that λ^k is a prefix of x, and $u \in \Sigma^+$. By definition, $\lambda \prec u$ and λ is not a prefix of u. Choosing $y = \lambda^{k-p} u'$ with $u' \prec u$, we have $\lambda^p y = \lambda^k u' \prec \lambda^k u = x \prec xy$.

Hence, $\lambda^p \in LFCand(w)$ and no shorter strings exist in $LFCand(w)$. □

By Lemma 14 and Lemma 18, computing the last Lyndon factor $\lambda_m^{e_m}$ of $T = val(X_n)$ reduces to computing $LFCand(X_n)$ for the last variable X_n. In what follows, we propose a dynamic programming algorithm to compute $LFCand(X_i)$ for each variable. Firstly we show the number of strings in $LFCand(X_i)$ is $O(\log N)$, where $N = |val(X_n)| = |T|$.

Lemma 19. *For any string w, let x_j be the jth shortest string of $LFCand(w)$. Then, $|x_{j+1}| > 2|x_j|$ for any $1 \leq j < |LFCand(w)|$.*

Proof. Assume on the contrary that $|x_{j+1}| \leq 2|x_j|$. If $|x_{j+1}| = 2|x_j|$, then $x_{j+1} = x_j x_j$. There are two cases to consider: (1) If $x_j y \prec y$, then $x_{j+1} y \prec x_j y$. (2) Otherwise, $y \prec x_j y$. It means that $\min_{\prec}\{x_{j+1} y, y\} \prec x_j y$ for any y, however, this contradicts that $x_j \in LFCand(w)$. If $|x_{j+1}| < 2|x_j|$, by Lemma 17, x_j is a prefix of x_{j+1}, and therefore x_j has a period q such that $x_{j+1} = u^k v$ and $x_j = u^{k-1} v$, where $u = x_j[1..q]$, $k \geq 1$ is an integer, and v is a proper prefix of u. There are two cases to consider: (1) If $uvy \prec vy$, then $u^k vy \prec u^{k-1} vy = x_j y$. (2) If $vy \prec uvy$, then $vy \prec uvy \prec u^2 vy \prec \cdots \prec u^{k-1} vy = x_j y$. It means that $\min_{\prec}\{u^k vy, vy\} \prec x_j y$ for any $y \succ \lambda$, however, this contradicts that $x_j \in LFCand(w)$. Hence $|x_{j+1}| > 2|x_j|$ holds. □

Since x_j is a suffix of x_{j+1}, it follows from Lemma 17 and Lemma 19 that $x_{j+1} = x_j t x_j$ with some non-empty string $t \in \Sigma^+$. This also implies that the number of strings in $LFCand(w)$ is $O(\log N)$, where N is the length of T. By identifying each suffix of $LFCand(X_i)$ with its length, and using Lemma 19, $LFCand(X_i)$ for all variables can be stored in a total of $O(n \log N)$ space.

The following lemma shows a dynamic programming approach to compute $LFCand(X_i)$ for each variable X_i. We will mean by a sorted list of $LFCand(X_i)$ the list of the elements of $LFCand(X_i)$ sorted in increasing order of length.

Lemma 20. *Let $X_i = X_l X_r$ be any production of a given SLP $\mathcal{S}$ of size n. Provided that sorted lists for $LFCand(X_l)$ and $LFCand(X_r)$ are already computed, a sorted list for $LFCand(X_i)$ can be computed in $O(P(n, N) + Q(n, N) \log N)$ time and $O(S(n, N) + \log N)$ space.*

Proof. Let D_i be a sorted list of the suffixes of X_i that are candidates for $LFCand(X_i)$. We initially set $D_i \leftarrow LFCand(X_r)$.

We process the elements of $LFCand(X_l)$ in increasing order of length. Let x be any string in $LFCand(X_l)$, and d be the longest string in D_i. Since any string of $LFCand(X_r)$ is a prefix of d by Lemma 17, in order to compute $LFCand(X_i)$ it suffices to lexicographically compare $x \cdot val(X_r)$ and d. Let $L = lcp(x \cdot val(X_r), d)$. See also Figure 4.1.

- If $(x \cdot val(X_r))[L+1] \prec d[L+1]$, then $x \cdot val(X_r) \prec d$. Since any string in D_i is a prefix of d by Lemma 17, we observe that any element in D_i that is longer than L cannot be an element of $LFCand(X_i)$. Hence we delete any element of D_i that is longer than L from D_i, then add $x \cdot val(X_r)$ to D_i, and update $d \leftarrow x \cdot val(X_r)$. See also Figure 4.2.

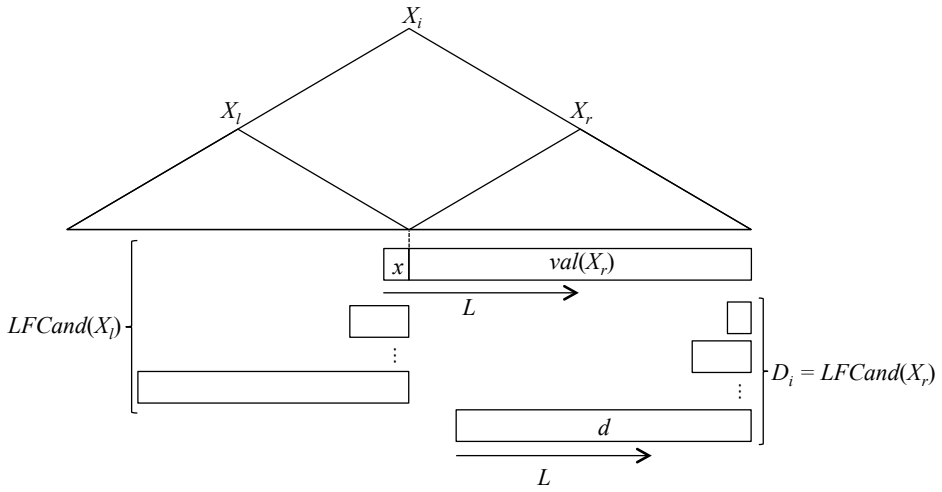

Figure 4.1: Lemma 20: Initially $D_i = LFCand(X_r)$ and $L = lcp(x \cdot val(X_r), d)$ with x being the shortest string of $LFCand(X_l)$.

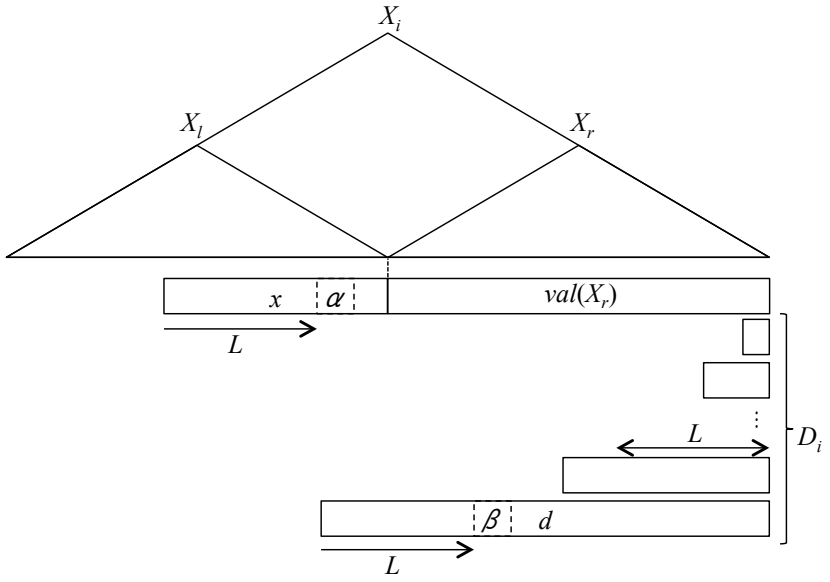

Figure 4.2: Lemma 20: Case where $(x \cdot val(X_r))[L+1] = \alpha \prec d[L+1] = \beta$. d and any string in D_i that is longer than L are deleted from D_i. Then $x \cdot val(X_r)$ becomes the longest candidate in D_i.

- If $(x \cdot val(X_r))[L+1] \succ d[L+1]$, then $x \cdot val(X_r) \succ d$. Since $x \cdot val(X_r)$ cannot be an element of $LFCand(X_i)$, in this case neither D_i nor d is updated. See also Figure 4.3.

- If $L = |d|$, i.e., d is a prefix of $x \cdot val(X_r)$, then there are two sub-cases:

  - If $|x \cdot val(X_r)| \leq 2|d|$, d has a period q such that $x \cdot val(X_r) = u^k v$ and $d = u^{k-1} v$,

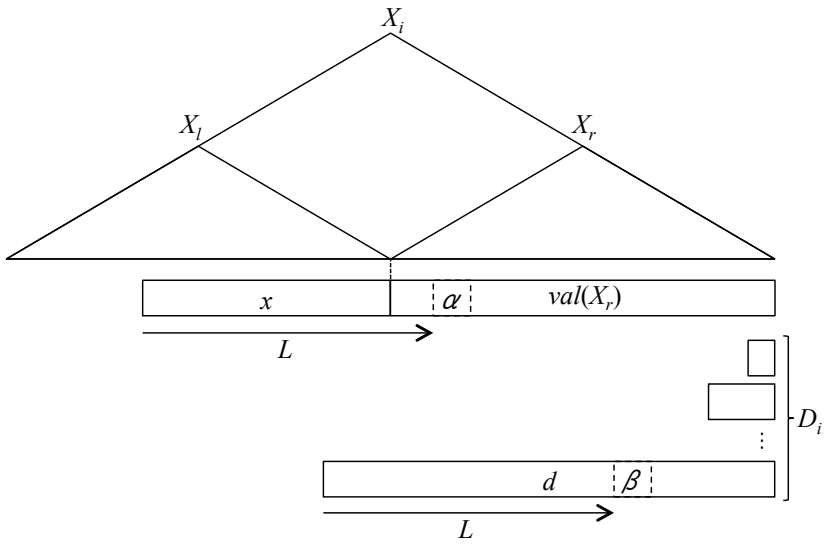

Figure 4.3: Lemma 20: Case where $(x \cdot val(X_r))[L+1] = \alpha \succ d[L+1] = \beta$. There are no updates on D_i.

where $u = d[1..q]$, $k \geq 1$ is an integer, and v is a proper prefix of u. By similar arguments to Lemma 19, we observe that d cannot be a member of $LFCand(X_i)$ while $x \cdot val(X_r)$ may be a member of $LFCand(X_i)$. Thus we add $x \cdot val(X_r)$ to D_i, delete d from D_i, and update $d \leftarrow x \cdot val(X_r)$. See also Figure 4.4.

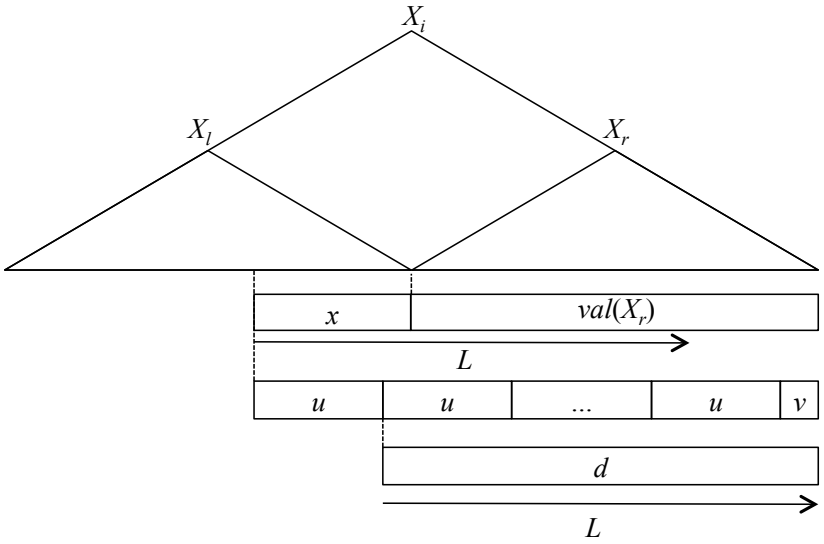

Figure 4.4: Lemma 20: Case where $L = |d|$ and $|x \cdot val(X_r)| \leq 2|d|$. Since $x \cdot val(X_r) = u^k v$ and $d = u^{k-1}v$, d is deleted from D_i and $x \cdot val(X_r)$ is added to D_i.

– If $|x \cdot val(X_r)| > 2|d|$, then both d and $x \cdot val(X_r)$ may be a member of $LFCand(X_i)$.

Thus we add $x \cdot val(X_r)$ to D_i, and update $d \leftarrow x \cdot val(X_r)$. See also Figure 4.5.

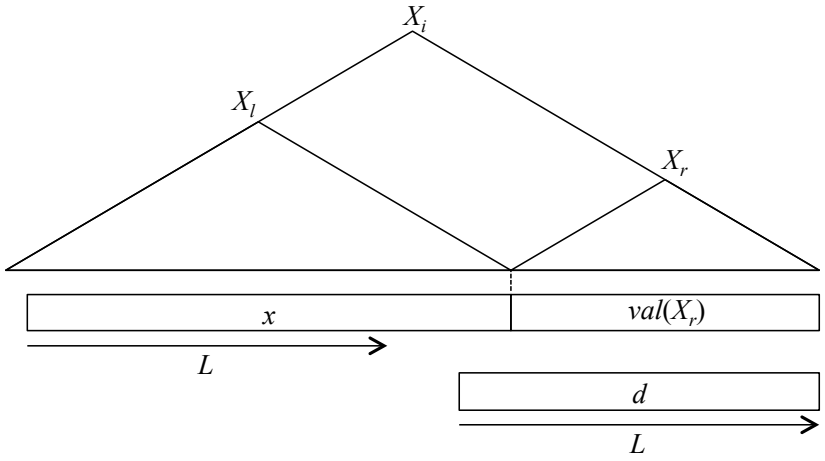

Figure 4.5: Lemma 20: Case where $L = |d|$ and $|x \cdot val(X_r)| > 2|d|$. We add $x \cdot val(X_r)$ to D_i, and $x \cdot val(X_r)$ becomes the longest member of D_i.

We represent the strings in $LFCand(X_l)$, $LFCand(X_r)$, $LFCand(X_i)$, and D_i by their lengths. Given sorted lists of $LFCand(X_l)$ and $LFCand(X_r)$, the above algorithm computes a sorted list for D_i, and it follows from Lemma 19 that the number of elements in D_i is always $O(\log N)$. Thus all the above operations on D_i can be conducted in $O(\log N)$ time in each step.

By using LCE queries on SLPs, $lcp(x \cdot val(X_r), d)$ can be computed in $O(Q(n, N))$ time for each $x \in LFCand(X_l)$. Since there exist $O(\log N)$ elements in $LFCand(X_l)$, we can compute $LFCand(X_i)$ in $O(P(n, N) + Q(n, N) \log N)$ time. The total space complexity is $O(S(n, N) + \log N)$. □

Since there are n productions in a given SLP, using Lemma 20 we can compute $LFCand(X_n)$ for the last variable X_n in a total of $O(n(P(n, N) + Q(n, N) \log N))$ time. The main result of this chapter follows.

Theorem 6. *Given an SLP $\mathcal{S}$ of size n representing a string T, we can compute LF_T in $O(nh(P(n, N) + Q(n, N) \log N) + n^3 h)$ time and $O(n^2 + S(n, N))$ space, where h is the height of the derivation tree of $\mathcal{S}$.*

Proof. Let $LF_T = \lambda_1^{e_1} \cdots \lambda_m^{e_m}$. First, using Lemma 20 we compute $LFCand$ for all variables in $\mathcal{S}$ in $O(n(P(n, N) + Q(n, N) \log N))$ time. Next we will compute the Lyndon factors from right to left. Suppose that we have already computed $\lambda_{j+1}^{e_{j+1}} \cdots \lambda_m^{e_m}$, and we are computing the

jth Lyndon factor $\lambda_j^{e_j}$. Using Lemma 15, we construct in $O(n)$ time a new SLP of size $O(n)$ describing $T[1..|T| - \sum_{k=j+1}^{m} e_k|\lambda_k|]$, which is the prefix of T obtained by removing the suffix $\lambda_{j+1}^{e_{j+1}} \cdots \lambda_m^{e_m}$ from T. Here we note that the new SLP actually has $O(h)$ new variables since $T[1..|T|-\sum_{k=j+1}^{m} e_k|\lambda_k|]$ can be represented by a sequence of $O(h)$ variables in $\mathcal{S}$. Let Y be the last variable of the new SLP. Since $LFCand$ for all variables in $\mathcal{S}$ have already been computed, it is enough to compute $LFCand$ for $O(h)$ new variables. Hence using Lemma 20, we compute a sorted list of $LFCand(Y) = LFCand(T[1..|T|-\sum_{k=j+1}^{m} e_k|\lambda_k|])$ in a total of $O(h(P(n,N)+Q(n,N)\log N))$ time. It follows from Lemma 18 that the shortest element of $LFCand(Y)$ is $\lambda_j^{e_j}$, the jth Lyndon factor of T. Note that each string in $LFCand(Y)$ is represented by its length, and so far we only know the total length $e_j|\lambda_j|$ of the jth Lyndon factor. Since λ_j is border free, $|\lambda_j|$ is the shortest period of $\lambda_j^{e_j}$. We construct a new SLP of size $O(n)$ describing $\lambda_j^{e_j}$, and compute $|\lambda_j|$ in $O(n^2h)$ time using Lemma 16. We repeat the above procedure m times and use an inequality $m \leq n$ by [61] (see also Chapter 5), and hence LF_T can be computed in a total of $O(n(P(n,N) + Q(n,N)\log N) + m(h(P(n,N) + Q(n,N)\log N) + n^2h)) = O(nh(P(n,N) + Q(n,N)\log N) + n^3h)$ time. To compute each Lyndon factor of LF_T, we need $O(n^2 + S(n,N))$ space for Lemma 16 and Lemma 20. Since $LFCand(X_i)$ for each variable X_i requires $O(\log N)$ space, the total space complexity is $O(n^2+S(n,N)+n\log N) = O(n^2 + S(n,N))$. □

Using Lemma 3 we can obtain the following corollary:

Corollary 3. *Given an SLP of size n representing string T of length N, we can compute LF_T in $O(n^3h)$ time and $O(n^2)$ space.*

4.2 Conclusions

In this chapter, we developed a Lyndon factorization algorithm for a grammar compressed string. We can compute the Lyndon factorization of a string which is represented by an SLP in $O(nh(P(n,N) + Q(n,N)\log N) + n^3h)$ time and $O(n^2 + S(n,N))$ space, where n is the size of an input SLP $\mathcal{S}$, N is the length of the string derived by $\mathcal{S}$, h is the height of the derivation tree of $\mathcal{S}$, and $P(n,N)$, $S(n,N)$ and $Q(n,N)$ are the preprocessing time, space and query time of an LCE data structure, respectively. Our algorithm computes the smallest suffix of a string. If we want the Lyndon factorization of a string, we only have to use the algorithm recursively. Our

algorithm is faster than Duval's algorithm (for an uncompressed string) when the input string is highly compressed.

We consider the same problem in the next chapter. Our future works for the problem will state in the next chapter.

Chapter 5

Faster Lyndon Factorization Algorithms for Compressed Texts

In this chapter, we propose a faster Lyndon factorization algorithms for an SLP. We also propose a Lyndon factorization algorithm for an LZ78 compressed strings. This chapter is organized as follows: In Section 5.1, we give additional notation for this chapter. In Section 5.2, we show useful properties on strings and Lyndon words. In Section 5.3 and 5.4, we present Lyndon factorization algorithms for an SLP compressed string and an LZ78 compressed string, respectively.

5.1 Notation

For any string T, let $LF_T = \lambda_1^{p_1} \cdots \lambda_m^{p_m}$. Let $lfb_T(i)$ denote the position where the i-th Lyndon factor begins in T, i.e., $lfb_T(1) = 1$ and $lfb_T(i) = lfb_T(i-1) + |\lambda_{i-1}^{p_{i-1}}|$ for any $2 \leq i \leq m$. For any $1 \leq i \leq m$, let $lfs_T(i) = \lambda_i^{p_i} \lambda_{i+1}^{p_{i+1}} \cdots \lambda_m^{p_m}$ and $lfp_T(i) = \lambda_1^{p_1} \lambda_2^{p_2} \cdots \lambda_i^{p_i}$. For convenience, let $lfs_T(m+1) = lfp_T(0) = \varepsilon$.

5.2 Properties of Strings and Lyndon Words

In this section, we introduce some fundamental properties of strings and Lyndon words which will be used in our algorithms.

Lemma 21. *Let $u \in \Sigma^+$ and $v \in \Sigma^*$. If $v \prec u^\infty$, then $v \prec u^1 v \prec u^2 v \prec \ldots$ holds. If $v \succ u^\infty$, then $v \succ u^1 v \succ u^2 v \succ \ldots$ holds.*

Proof. We only show the former statement, as the latter can be shown similarly. It suffices to show $v \prec uv$, since then for any positive integer k, $u^k v \prec u^{k+1}v$ holds. If $v \preceq u$, then clearly $v \prec uv$. If $v \succ u$, then let $x = lcp(v, u^\infty)$. Since $u \prec v \prec u^\infty$, $|x| \geq |u|$ holds. Let j be the maximum integer such that u^j is a prefix of x, and let $v = u^j w$. Since $v \prec u^\infty$, we have $w \prec u$, and therefore $v = u^j w \prec u^{j+1} \prec u^{j+1}w = uv$. □

Lemma 22 ([30]). *For any* $1 \leq i < m$, $LF_T = LF_{lfp_T(i)} LF_{lfs_T(i+1)}$.

The following lemmas are essentially the same as what Duval's algorithm is founded on but are tailored for explaining our algorithm.

Lemma 23. *Let* $j > 1$ *be any position of a string* T *such that* $T \prec T[i..|T|]$ *for any* $1 < i \leq j$. *Then,* $T \prec T[k..|T|]$ *also holds for any* $j < k \leq j + lcp(T, T[j..|T|])$.

Proof. Let $h = lcp(T, T[j..|T|])$. There are two cases:

- When $j > h{+}1$: Since $T \prec T[j..|T|]$, $T[h{+}1] \prec T[j{+}h]$. Hence, $T \prec T[k{-}j{+}1..|T|] \prec T[k..|T|]$ holds for any $j < k \leq j + lcp(T, T[j..|T|])$.
- When $j \leq h{+}1$: Since $T = T[1..h]T[h{+}1..|T|] \prec T[j..|T|] = T[j..j{+}h{-}1]T[j{+}h..|T|]$, we have that $T[h+1] \prec T[j+h]$. Noting that $T[1..j+h-1]$ has period $q = j - 1$, we have that $T[k..j+h-1] = T[k-q..j+h-q-1] = \cdots = T[i..i+j+h-k-1]$ and thus $T[k..j+h] \succ T[k-q..j+h-q] = \cdots = T[i..i+j+h-k]$ where $i = k - pq$, $p \geq 1$, $1 < i \leq j$. The lemma follows since $T \prec T[i..|T|]$.

□

Lemma 24. *It holds that* $|\lambda_1| = \hat{j} - 1$ *and* $p_1 = 1 + \lfloor \hat{h}/|\lambda_1| \rfloor$, *where* $\hat{j} = \min\{j \mid w \succ T[j..|T|]\}$ *and* $\hat{h} = lcp(T, T[\hat{j}..|T|])$.

Proof. For any $1 < k \leq \hat{j} - 1$, let $h_k = lcp(T, T[k..|T|])$. By definition of $\hat{j}$ we have $T \prec T[k..|T|]$, and also $k{+}h_k \leq \hat{j}{-}1$ due to Lemma 23. Thus, $T[h_k{+}1..\hat{j}{-}1] \prec T[k{+}h_k..\hat{j}{-}1]$ and therefore $T[1..\hat{j}{-}1] = T[1..h_k]T[h_k{+}1..\hat{j}{-}1] \prec T[k..k{+}h_k{-}1]T[k{+}h_k..\hat{j}{-}1] = T[k..\hat{j}{-}1]$, for all $1 < k \leq \hat{j} - 1$, indicating that $\hat{\lambda}_1 = T[1..\hat{j} - 1]$ is a Lyndon word. Next, suppose that there exists a Lyndon word $\lambda = T[1..|\lambda|]$ such that $|\lambda| > |\hat{\lambda}_1|$. By definition of $\hat{j}$ and $\hat{h}$, either $\hat{j} + \hat{h} = n + 1$ or $T[1 + \hat{h}] \succ T[\hat{j} + \hat{h}]$, thus, λ must be a prefix of $T[1..\hat{j} + \hat{h} - 1]$. However, since $T[1..\hat{h}] = T[\hat{j}..\hat{j} + \hat{h} - 1]$, λ has period $|\hat{\lambda}_1|$ and thus is not border-free, contradicting that λ is a Lyndon word. Hence, from Lemma 2, $\hat{\lambda}_1 = \lambda_1$. It is easy to see that $p_1 = 1 + \lfloor \hat{h}/|\lambda_1| \rfloor$, and the lemma follows. □

Thus, computing the first Lyndon factor reduces to computing $\hat{j}$ and $\hat{h}$. From a definition of Lyndon words and Lemma 24, the following lemma holds.

Lemma 25. *For any* $1 \leq i \leq m$ *and* $1 \leq j < lfb_T(i)$, $T[j..|T|] \succ lfs_T(i)$.

5.3 Computing Lyndon Factorization from SLP

Here, we present a faster algorithm to compute Lyndon factorization of a string represented by an SLP. Our algorithm employs the following lemma which is used in parallel algorithms to compute Lyndon factorization of an uncompressed string. Below, let $LF_u = u_1^{p_1} \cdots u_m^{p_m}$ and $LF_v = v_1^{q_1} \cdots v_{m'}^{q_{m'}}$ for $u, v \in \Sigma^+$.

Lemma 26 ([1, 26])**.** $LF_{uv} = u_1^{p_1} \ldots u_c^{p_c} z^k v_{c'}^{q_{c'}} \ldots v_{m'}^{q_{m'}}$ *for some* $0 \leq c \leq m$, $1 \leq c' \leq m' + 1$ *and* $LF_{lfs_u(c+1)lfp_v(c'-1)} = z^k$.

This lemma implies that we can obtain LF_{uv} from LF_u and LF_v by computing the medial Lyndon factor z^k since the other Lyndon factors remain unchanged in uv, see Figure 5.1.

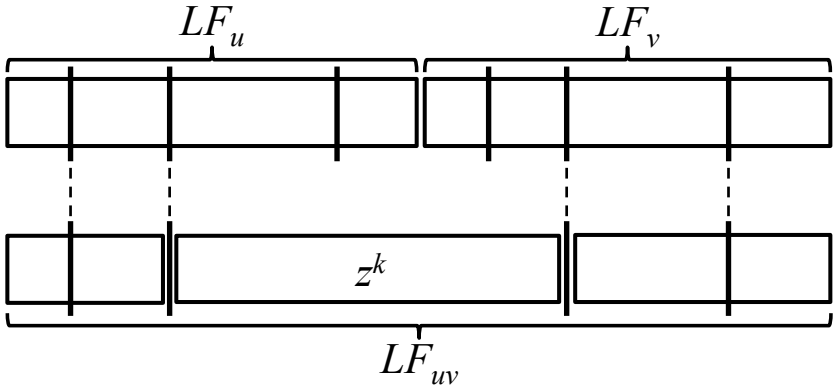

Figure 5.1: This is a conceptual diagram of Lemma 26.

5.3.1 How to compute the medial Lyndon factor z^k

Unfortunately, the algorithm for computing z^k given in the proof of Theorem 2.2 of [26], appears to be incorrect.

Here is a counter example: Consider two strings $u =$ `abc|abbb|ab` and $v =$ `bcc|bc|b|abc|abb|ab|a`, where | denotes the boundary of Lyndon factors. To compute the Lyndon factorization of uv, the right-extension procedure of [26] extends the last Lyndon factor `ab` of u to the right,

and obtains string abbccbcbabcabb. Then, the left-extension procedure extends this string to the left, and obtains string $y =$ abbbabbccbcbabcabb. Theorem 2.2 of [26] claims that this string y must be the medial Lyndon factor of uv that crosses the boundary of u and v, but clearly y is not a Lyndon word.

In the sequel, we present several combinatorial properties from which a correct and efficient algorithm to compute the medial Lyndon factor z^k follows.

Let γ_u be the minimum integer such that $lfs_u(i+1)$ is a prefix of u_i for any $\gamma_u \leq i \leq m$ (see also Example 5).

Lemma 27. *For any* $1 \leq j < \gamma_u$, $u_j \triangleright lfs_u(j+1)$.

Proof. Since we have $lfs_u(j) \succ lfs_u(j+1)$ from Lemma 25, we only have to show that $lcp(u_j, lfs_u(j+1)) < \min\{|u_j|, |lfs_u(j+1)|\}$. Note that u_j is not a prefix of $lfs_u(j+1)$ since otherwise the j-th Lyndon factor should extend to the right with at least another occurrence of u_j. It follows from the definition of γ_u that $lfs_u(\gamma_u)$ is not a prefix of u_{γ_u-1}, and hence $u_{\gamma_u-1} \triangleright lfs_u(\gamma_u)$ holds. If we assume on the contrary that $lfs_u(j+1)$ is a prefix of u_j for some $1 \leq j < \gamma_u - 1$, since $lfs_u(\gamma_u)$ appears before the $(\gamma_u - 1)$-th Lyndon factor, there exists a suffix $u[i..|u|] = lfs_u(\gamma_u)z$ of u with $i < lfb_u(\gamma_u - 1)$. It follows from $u_{\gamma_u-1} \triangleright lfs_u(\gamma_u)$ that $u[i..|u|] = lfs_u(\gamma_u)z \triangleleft u_{\gamma_u-1} \prec lfs_u(\gamma_u - 1)$, which contradicts Lemma 25. □

We define the set of *significant suffixes* Γ_u of u as $\Gamma_u = \{lfs_u(i) \mid \gamma_u \leq i \leq m\}$. It is clear from the definition of Lyndon factorization and γ_u that for any $\gamma_u \leq i \leq m$, $u_i = lfs_u(i+1)y_i$ for some non-empty string y_i. Let x_i denote the suffix of u of length $|u_i|$. Note that $x_i = y_i lfs_u(i+1)$ and $lfs_u(i) = u_i^{p_i} lfs_u(i+1) = (lfs_u(i+1)y_i)^{p_i} lfs_u(i+1) = lfs_u(i+1)x_i^{p_i}$.

Example 5. *If* $u =$ ababbababababbababa*, then* $LF_u =$ ababb|abababb|abab|a*. Thus* $\gamma_u = 2$ *and* $\Gamma_u = \{$abababbababa, ababa, a$\}$*. Also,* $y_2 =$ bb*,* $y_3 =$ bab*,* $y_4 =$ a*,* $x_2 =$ bbababa*,* $x_3 =$ baba *and* $x_4 =$ a*.*

Lemma 28. *For any string* u, $|\Gamma_u| = O(\log |u|)$.

Proof. Straightforward, since $|lfs_u(i)| > 2|lfs_u(i+1)|$ for any $\gamma_u \leq i \leq m$. □

Lemma 29. *For any* $\gamma_u \leq i < m$, $y_i \triangleright x_{i+1}^{\infty}$.

Proof. Since $u_i = lfs_u(i+1)y_i = (lfs_u(i+2)y_{i+1})^{p_{i+1}} lfs_u(i+2)y_i$, if we assume that $y_i = x_{i+1}^{\infty}[1..|y_i|] = (y_{i+1}lfs_u(i+2))^{\infty}[1..|y_i|]$ we get a contradiction that u_i has period $|lfs_u(i+2)y_{i+1}|$. Also, if $y_i \prec x_{i+1}^{\infty}[1..|y_i|]$, $u_{i+1}[1+|lfs_u(i+2)y_{i+1}|..|u_{i+1}|] \prec u_{i+1}$ holds, a contradiction. □

From Lemma 29, we get $x_{\gamma_u-1}^{\infty} \succ y_{\gamma_u} \succ x_{\gamma_u+1}^{\infty} \succ y_{\gamma_u+1} \succ \cdots \succ x_{m-1}^{\infty} \succ y_{m-1} \succ x_m^{\infty} = u_m^{\infty}$, where we assume for convenience that $x_{\gamma_u-1}^{\infty} = \omega^{\infty}$ (ω is the lexicographically largest character in Σ).

Now let us see what happens to the lexicographical order of suffixes of u when extending them by appending v.

Lemma 30. *If* $sv = \min_{\prec}\{s'v \mid s' \in Suffix(u)\}$, *then* $s \in \Gamma_u$.

Proof. We show for any $s \in (Suffix(u) - \Gamma_u)$, $sv \neq \min_{\prec}\{s'v \mid s' \in Suffix(u)\}$.

- If $s \neq u_i^k lfs_u(i+1)$ with $1 \leq k \leq p_i$, namely, $s = tu_i^r lfs_u(i+1)$ with $1 \leq i \leq m$ and $0 \leq r < p_i$, where t is a proper suffix of u_i. It follows from the definition of Lyndon word $t \triangleright u_i$, and hence, $sv \neq \min_{\prec}\{s'v \mid s' \in Suffix(u)\}$.
- If $s = u_i^k lfs_u(i+1)$ with $1 \leq k < p_i$. From Lemma 21, $sv \succ \min_{\prec}\{u_i^0 lfs_u(i+1)v, u_i^{p_i} lfs_u(i+1)v\}$. Therefore, $sv \neq \min_{\prec}\{s'v \mid s' \in Suffix(u)\}$.
- If $s = lfs_u(i) \notin \Gamma_u$. From Lemma 27, $s \triangleright lfs_u(i+1)$. Therefore, $sv \notin \min_{\prec}\{s'v \mid s' \in Suffix(u)\}$.

□

Lemma 31. *If* $x_{i-1}^{\infty} \succ v \succ x_i^{\infty}$ *with* $\gamma_u \leq i \leq m$, $lfs_u(i)v = \min_{\prec}\{sv \mid s \in Suffix(u)\}$, *and* $lfs_u(1)v \succ \cdots \succ lfs_u(i-1)v \succ lfs_u(i)v \prec lfs_u(i+1)v \prec \cdots \prec lfs_u(m+1)v$ *holds.*

Proof. By Lemma 27, $lfs_u(1)v \triangleright \ldots \triangleright lfs_u(\gamma_w - 1)v \triangleright u_{\gamma_u}v$. By Lemma 29, $x_j^{\infty} \succ v$ and also $lfs_u(j+1)x_j^{\infty} = u_j^{\infty} \succ lfs_u(j+1)v$ hold for any $\gamma_u \leq j < i$, and hence, it follows from Lemma 21 that $lfs_u(j)v = u_j^{p_j} lfs_u(j+1)v \succ lfs_u(j+1)v$. Also, since $v \succ x_{j'}^{\infty}$ for any $i \leq j' \leq m$, $lfs_u(j')v \prec lfs_u(j'+1)v$ holds. Therefore, we get $lfs_u(1)v \succ \cdots \succ lfs_u(i-1)v \succ lfs_u(i)v \prec lfs_u(i+1)v \prec \cdots \prec lfs_u(m)v$ holds. It is clear from Lemma 30 that $lfs_u(i)v$ is the lexicographically smallest string in $\{sv \mid s \in Suffix(u)\}$. □

We can compute the medial Lyndon factor as follows:

Lemma 32. *Given $LF_u = u_1^{p_1} \cdots u_m^{p_m}$ and $LF_v = v_1^{q_1} \cdots v_{m'}^{q_{m'}}$ for $u, v \in \Sigma^+$, we can compute $LF_{uv} = u_1^{p_1} \cdots u_c^{p_c} z^k v_{c'}^{q_{c'}} \cdots v_{m'}^{q_{m'}}$ by $O(\log m + \log m')$ lexicographical string comparisons.*

Proof. Clearly, it holds that $LF_{uv} = LF_u LF_v$ if $u_m \succ v_1$, and that $LF_{uv} = u_1^{p_1} \cdots u_{m-1}^{p_{m-1}} u_m^{p_m+q_1} v_2^{q_2} \cdots v_{m'}^{q_{m'}}$ if $u_m = v_1$. In what follows we consider the case when $u_m \prec v_1$. Note that $v \succ u_m^\infty$ holds in this situation.

First, we compute integer j such that $1 \leq j \leq m+1$ and $lfs_u(j)v = \min_\prec\{sv \mid s \in Suffix(u)\}$. From Lemma 31, for any $1 \leq i \leq m$, $j \leq i$ iff $lfs_u(i)v \prec lfs_u(i+1)v$. Hence we can find j by binary search which requires $O(\log m)$ lexicographical string comparisons. Next, we compute $j' = \min\{j'' \mid 1 \leq j'' \leq m', lfs_u(j)v \succ lfs_v(j''+1)\}$. Since $lfs_v(1) \succ lfs_v(2) \succ \ldots \succ lfs_v(m'+1)$, j' can be found by $O(\log m')$ lexicographical string comparisons with binary search.

We show $z = lfs_u(j)lfp_v(j')$ is the first decomposed Lyndon factor of $lfs_u(j)v$. By definition of j, for any position i with $lfb_u(j) < i \leq |u|$, $lfs_u(j)v \prec (uv)[i..|uv|]$. Since $lfs_u(j)v \prec lfs_v(j')$, it follows from Lemma 25 that for any $|u| < i < |u| + lfb_v(j')$, $lfs_u(j)v \prec lfs_v(j') \prec (uv)[i..|uv|]$. Next we show $v_{j'}$ is not a prefix of $lfs_u(j)v$. Assume on the contrary that $lfs_u(j)v = v_{j'}t$. The beginning position of t in uv is at most $|u| + lfb_v(j')$ since the occurrences of $v_{j'}$ cannot overlap, and hence $lfs_u(j)v \prec t$. Since $lfs_u(j)v = v_{j'}t \prec lfs_v(j')$, $lfs_u(j)v \prec t \prec v_{j'}^{q_{j'}-1} lfs_v(j'+1)$. Applying this deduction $q_{j'}$ times, we get $lfs_u(j)v \prec t \prec lfs_v(j'+1)$, a contradiction. Thus, $lfs_u(j)v \lhd v_{j'} \preceq (uv)[i..|uv|]$ for any $|u| + lfb_v(j') \leq i < |u| + lfb_v(j'+1)$. Since $|u| + lfb_v(j'+1)$ is the first position where the suffix becomes lexicographically smaller than $lfs_u(j)v$, the claim follows from Lemma 24.

Finally, we show $u_{j-1} \succeq z$. Assume on the contrary that $u_{j-1} \prec z$. By Lemma 1 $u_{j-1}z$ is a Lyndon word, which implies $u_{j-1}z \lhd z$. This contradicts $lfs_u(j)v = \min_\prec\{s'v \mid s' \in Suffix(u)\}$ due to $u_{j-1} lfs_u(j)v \lhd lfs_u(j)v$.

The above procedure correctly computes the decomposed Lyndon factorization of uv. The exponent of z can be computed by checking if $u_j = z$ and/or $u_{j'+1} = z$ and packing them together if needed. Hence the total number of lexicographical string comparisons is $O(\log m + \log m')$. □

5.3.2 Algorithm

Given an SLP $\mathcal{S}$ of size n, we process each production $X_i \to X_l X_r$ in increasing order of i, and compute LF_{X_i} from LF_{X_l} and LF_{X_r} using dynamic programming. We use Lemma 32 to com-

pute the medial Lyndon factor for each variable X_i. In the final stage where $i = n$, we obtain the Lyndon factorization $LF_{X_n} = LF_T$ for the uncompressed string T. Using Lemma 32, LF_{X_i} can be computed by $O(\log m_l + \log m_r)$ string comparisons, where m_l and m_r are respectively the number of Lyndon factors in LF_{X_l} and LF_{X_r}.

In order to bound the size of the Lyndon factorization for SLPs we can show the following lemma.

Lemma 33. *Let n be the size of any SLP representing a string T. The size m of the Lyndon factorization of T is at most n.*

Proof. It follows from Lemma 26 that every production introduces at most one new Lyndon factor, that is, the medial Lyndon factor. Hence LF_T consists of at most n distinct Lyndon words. Since all Lyndon factors in LF_T are distinct, the statement holds. □

We are ready to show the main result of this section.

Theorem 7. *Given an SLP of size n representing string T of length N, we can compute LF_T in $O(n^2 + P(n, N) + Q(n, N)n \log n)$ time and $O(n^2 + S(n, N))$ space.*

Proof. By Lemmas 32, 33 and an LCE data structure, for each production $X_i \to X_l X_r$ we can compute LF_{X_i} in $O(Q(n, N) \log n)$ time, provided that LF_{X_l} and LF_{X_r} are already computed. Using a dynamic programming method, this takes a total of $O(Q(n, N)n \log n)$ time. The space complexity for this dynamic programming is $O(n^2)$ since for each variable X_i we have to store at most n beginning positions of the Lyndon factors of X_i. Putting these and the pre-processing costs of an LCE data structure together, we conclude that our algorithm takes a total of $O(n^2 + P(n, N) + Q(n, N)n \log n)$ time and $O(n^2 + S(n, N))$ space. □

Using Lemma 3 we can obtain the following corollary:

Corollary 4. *Given an SLP of size n representing string T of length N, we can compute LF_T in $O(n^2 + n \log n \log N)$ time and $O(n^2)$ space.*

Lemma 34. *We can pre-process, in $O(n^2 + P(n, N) + Q(n, N)n \log n)$ time and $O(n^2 + S(n, N))$ space, an SLP of size n and height h describing string T of length N so that the following query can be answered in $O(h(n + Q(n, N) \log n))$ time: given an interval $[b, e]$ with $1 \leq b \leq e \leq N$, compute $LF_{T[b..e]}$.*

By Lemma 34, we can compute the Lyndon factorization of a query substring of T, without decompression. Lemma 34 is more efficient than applying Theorem 7 to an SLP describing substring $T[b..e]$.

5.4 Computing Lyndon Factorization from LZ78

In this section, we show how, given an LZ78 encoding of string T, we compute the Lyndon factorization $LF_T = \lambda_1^{p_1} \cdots \lambda_m^{p_m}$ of T. At a high level, our algorithm is based on Duval's algorithm [30] which computes the Lyndon factorization of a given string T of length N in $O(N)$ time by scanning T from left to right.

From Lemma 23 and Lemma 24, we can compute the first Lyndon factor by initializing $j \leftarrow 2$ and executing the following: 1) compute $h \leftarrow lcp(T, T[j..|T|])$. 2) if $T[1+h] \prec T[j+h]$, set $j \leftarrow j+h+1$ and go back to Step 1); otherwise, output $|\lambda_1| \leftarrow j-1$ and $p_1 \leftarrow 1+\lfloor h/|\lambda_1| \rfloor$.

Let $\hat{j}$ and $\hat{h}$ denote the last values of j and h, respectively. Duval's algorithm computes $h \leftarrow lcp(T, T[j..|T|])$ by character comparisons, and it takes a total of $O(\hat{j} + \hat{h})$ time. Note $O(\hat{j} + \hat{h}) = O(|\lambda_1|p_1)$ since $\hat{j} + \hat{h} < |\lambda_1|p_1 + |\lambda_1|$. By Lemma 22, we can compute the second Lyndon factor by executing the above procedure with the remaining string $T[1+|\lambda_1|p_1..|T|]$. By applying this recursively, the Lyndon factorization of T can be computed in $O(\sum_{i=1}^{m} |\lambda_i|p_i) = O(|T|)$ time.

In Section 5.4.3, we show how to simulate, in $O(s \log s)$ time and space, the above algorithm on the LZ78 encoding of size s. The key ideas to achieve this are summarized as follows:

- Let us call a substring of an LZ78 factor *LZ-block*. After pre-processing LZ78 factors in $O(s)$ time, we can check, given two equivalent-length LZ-blocks, if they are the same string or not in constant time (see Section 5.4.1).

- A major difference between our algorithm and Duval's lies in how we reset j when $T[1+h] \prec T[j+h]$ happens. While Duval's algorithm sets $j \leftarrow j+h+1$, our algorithm skips some positions by utilizing the fact that the LZ78 factor f_k appears before, where $T[j+h]$ is in the i-th LZ78 factor with the form $f_i = f_k a$ (see Section 5.4.2).

- We can represent T by a sequence of LZ-blocks (trivially by LZ78 factorization itself). During the computation, our algorithm occasionally finds consecutive LZ-blocks that can be replaced by a single LZ-block, and greedily restructure it. The restructuring shrinks the number of LZ-blocks involved in the future LZ-block wise matching, and makes our analysis possible. This kind of technique has been employed in efficient algorithms on LZ78-compressed strings [41, 45, 2].

5.4.1 LZ-block wise matching

Given the LZ78 encoding of size s corresponding to a string T, we can build the LZ78 trie $LZtrie_T$ in $O(s)$ time. For any LZ78 factor v, let $\overline{v}$ denote the corresponding node of $LZtrie_T$. We use the following data structures LAQ and LCS:

Lemma 35 (Level Ancestor Query (LAQ) [7])**.** *We can pre-process a given rooted tree in linear time and space so that the i-th node in the path from any node to the root can be found in $O(1)$ time for any $i \geq 0$, if such exists.*

The suffix tree of a reversed trie can be constructed in linear time [85]. Combined with the constant-time LCA data structure [7], we obtain the following:

Lemma 36 (Longest Common Suffix (LCS))**.** *We can pre-process a given trie in linear time and space so that the length of the longest common suffix of any two strings in the trie can be answered in $O(1)$ time.*

Using LAQ, given a node $\overline{v}$ of the LZ78 trie $LZtrie_T$, we can access any position of the corresponding LZ78 factor v in $O(1)$ time.

Let u be an LZ-block, i.e., a substring of some LZ78 factor of T. Since any node of $LZtrie_T$ corresponds to an LZ78 factor, there exists at least one node $\overline{v}$ of the trie s.t. u is a suffix of v. Such node $\overline{v}$ is called a *handler* of u. Then u can be represented by a pair $(\overline{v}, |u|)$, in constant space. Let $\overline{\rho}(u)$ and $\rho(u)$ denote a handler of u and its corresponding LZ78 factor, respectively. For any LZ-block u and $1 \leq i \leq j \leq |u|$, $u[i..j]$ is also an LZ-block and its handler can be computed from $\overline{\rho}(u)$ in $O(1)$ time by using LAQ, i.e., when we write $u' \leftarrow u[i..j]$, it means we compute $\overline{\rho}(u')$ as the $(|u| - j)$-th ancestor of $\overline{\rho}(u)$. Using LCS, we can check the equality of two given LZ-blocks u and u' of the same length in $O(1)$ time. $lcp(u, u')$ can be computed in $O(\log |u|)$ time by a binary search and finding the position where the first mismatch occurs.

5.4.2 Skipping insignificant suffixes that appear before

We use the following Lemma.

Lemma 37. *Let T be non-empty string such that $T = xvyvz$ with $v \in \Sigma^+$ and $x, y, z \in \Sigma^*$. If $|xvy| < lfb_T(k) \leq |xvyv|$ for some k, then $lfb_T(k) \in \{|xvy| + lfb_v(j) \mid \gamma_v \leq j \leq m'\}$, where $LF_v = v_1^{q_1} \cdots v_{m'}^{q_{m'}}$.*

Proof. By Lemma 26, $lfb_T(k) \in \{|xvy| + lfb_v(j) \mid 1 \leq j \leq m'\}$. On the contrary, assume $lfb_T(k) = |xvy| + lfb_v(j)$ with $j < \gamma_v$. By Lemma 27, $lfs_v(j) \triangleright lfs_v(\gamma_v)$ and $T[|xvy| + lfb_v(j)..|T|] \triangleright T[|x| + lfb_v(\gamma_v)..|T|]$, a contradiction due to Lemma 25. □

Thanks to Lemma 37, when a string u appears multiple times without overlapping, we can utilize Γ_u to skip some suffix comparisons of Duval's algorithm.

Also, we can compute Γ_u for all LZ78 factors u efficiently.

Lemma 38. *Given the LZ78 encoding of size s of a string T, we can compute Γ_{f_k} for all LZ78 factors f_k, $1 \leq k \leq s$, in a total of $O(s \log s)$ time and space.*

Proof. It follows from $|f_k| \leq s$ and Lemma 28 that $|\Gamma_{f_k}| = O(\log s)$ for any $1 \leq k \leq s$. We consider an LZ78 factor $f_k = f_h a$, where $1 \leq h < k \leq s$ and $a \in \Sigma$, and show how to compute $LF_{lfs_{f_k}(\gamma_{f_k})}$ from $LF_{lfs_{f_h}(\gamma_{f_h})}$ in $O(\log s)$ time.

Note that $LF_{f_k} = LF_{lfp_{f_h}(\gamma_{f_h}-1)} LF_{lfs_{f_h}(\gamma_{f_h})a}$ holds from Lemmas 32 and 27. Moreover $\gamma_{f_h} \geq \gamma_{f_k}$, and hence, we do not need the information of $LF_{lfp_{f_h}(\gamma_{f_h}-1)}$ to compute $LF_{lfs_{f_k}(\gamma_{f_k})}$. We can use a simplified version of the algorithm of Lemma 32 to compute Lyndon factorization of $lfs_{f_h}(\gamma_{f_h})a$ by $O(\log \log s)$ lexicographical string comparisons. Since $lfs_{f_h}(j)$ is a proper prefix of $lfs_{f_h}(i)$ for any $\gamma_{f_h} \leq i < i' \leq m$, it suffices to compare $lfs_{f_h}(i)[|lfs_{f_h}(i')| + 1]$ and a in order to compare $lfs_{f_h}(i)a$ and $lfs_{f_h}(i')a$, which can be done in $O(1)$ time by using a data structure of LAQ (see Lemma 35). Let $LF_{lfs_{f_h}(\gamma_{f_h})a} = x_1^{p_1} \cdots x_m^{p_m}$. Since $m = O(\log s)$, we use $O(\log s)$ time and space to store $LF_{lfs_{f_h}(\gamma_{f_h})a}$. Next we get $LF_{lfs_{f_k}(\gamma_{f_k})}$ from $LF_{lfs_{f_h}(\gamma_{f_h})a}$ by discarding $x_1^{p_1} \ldots x_j^{p_j}$ of $LF_{lfs_{f_h}(\gamma_{f_h})a}$, where j is the largest integer such that x_{j+1}^{j+1} is not a prefix of x_j. Such j can be found by binary search, requiring $O(\log \log s)$ string comparisons. Again each comparison can be done in $O(1)$ time by using the fact that $lfs_{f_h}(j)$ is a proper prefix of $lfs_{f_h}(i)$ for any $\gamma_{f_h} \leq i < i' \leq m$.

Hence we can compute Γ_{f_k} for all LZ78 factors f_k in a total of $O(s \log s)$ time and space. □

5.4.3 Algorithm

Any substring of T can be represented by a sequence of LZ-blocks. Our algorithm to compute the first Lyndon factor of T maintains a sequence of LZ-blocks representing T by a dynamic linked list K, which is initially set to the LZ78 factorization of T itself but is restructured during the computation. After computing the leftmost Lyndon factor, we will also modify K

Algorithm 4: Algorithm to compute the first Lyndon factor.

Input: The linked list of LZ-blocks K initialized to the sequence of the LZ78 factors of T.

Output: The first Lyndon factor $\lambda_1^{p_1}$ of str_K.

`// Variables` u, u', v, v', x `and` y `are LZ-blocks and manipulated via handlers.`

$u \leftarrow K$.first; $v \leftarrow u[2..|u|]$;
$c_u \leftarrow 0$; $c_v \leftarrow 0$;
$j \leftarrow 2$; $h \leftarrow 0$;
while true do
  if $u = \varepsilon$ **then** $u \leftarrow \texttt{next}(u)$;
  if $v = \varepsilon$ **then** $v \leftarrow \texttt{next}(v)$;
  if v *is an LZ78 factor which is used for the first time* **then**
    $x \leftarrow K$.first; $y \leftarrow K$.second;
    $v' \leftarrow$ the longest member in $\Gamma_v \cup \{\varepsilon\}$ s.t. $(xy)[1..|v'|] = v'$;
    if $|x| + 1 = |v'|$ **then**
      restructure the first LZ-block to be v', and reset u and/or v if needed;
  $d \leftarrow \min\{|u|, |v|\}$;
  $u' \leftarrow u[1..d]$; $u \leftarrow u[d+1..|u|]$;
  $v' \leftarrow v[1..d]$; $v \leftarrow v[d+1..|v|]$;
  if $u' = v'$ **then**
    $h \leftarrow h + d$;
    if $u = \varepsilon$ & $c_u \geq 2$ **then**
      restructure the last two LZ-blocks before u to be a single LZ-block;
    if $v = \varepsilon$ & $c_v \geq 2$ **then**
      restructure the last two LZ-blocks before v to be a single LZ-block;
    if $u = \varepsilon$ & $v = \varepsilon$ **then** $c_u \leftarrow 1$; $c_v \leftarrow 1$;
    else if $u = \varepsilon$ **then** $c_u \leftarrow c_u + 1$; $c_v \leftarrow 0$;
    else if $v = \varepsilon$ **then** $c_v \leftarrow c_v + 1$; $c_u \leftarrow 0$;
  else
    $h' \leftarrow lcp(u', v')$;
    if $u'[1 + h'] \succ v'[1 + h']$ **then** $h \leftarrow h + h'$; break;
    `// Below` $u'[1 + h'] \prec v'[1 + h']$`. Reset` j `by Lemma 37.`
    $c_u \leftarrow 0$; $c_v \leftarrow 0$;
    $x \leftarrow K$.first;
    $v' \leftarrow$ the longest member in $\Gamma_{v[1..|v|-1]} \cup \{\varepsilon\}$ s.t. $x[1..|v'|] = v'$;
    $j \leftarrow$ the position in str_K where the v' begins if $v' \neq \varepsilon$, the position where the v ends otherwise;
    $h \leftarrow |v'|$; $u \leftarrow x[1 + h..|x|]$; $v \leftarrow v[|v|]$;
output $|\lambda_1| \leftarrow j - 1$ and $p_1 \leftarrow 1 + \lfloor h/|\lambda_1| \rfloor$;

to represent the remaining suffix of T in order to compute the remaining Lyndon factors. Let str_K denote the string represented by K. For any positions $i \leq j$ of str_K, let $\#_K[i,j]$ denote the number of LZ-blocks used to represent $str_K[i..j]$.

A pseudo-code of our algorithm is shown in Algorithm 4, which simulates Duval's algorithm to compute the first Lyndon factor of T on K.

The algorithm initializes $j \leftarrow 2$ and $h \leftarrow 0$, then starts with computing $lcp(T, T[j..|T|])$. Here, variables u and v are used for showing LZ-blocks which describe prefixes of $T[1+h..|T|]$ and $T[j+h..|T|]$, respectively, where variable h shows that currently $T[1..h]$ and $T[j..j+h-1]$ match. We can see at Lines 12-16 that the algorithm computes $lcp(T, T[j..|T|])$ by block-to-block comparisons, namely, the prefixes of length $d = \min\{|u|, |v|\}$ of u and v are cut out to LZ-blocks u' and v', and compared at Line 15. If $u' = v'$, we set $h \leftarrow h + d$ and continue matching the following LZ-blocks.

When we face LZ-blocks u' and v' that have a mismatch, we compute $h' \leftarrow lcp(u', v')$ by binary search at Line 25. At this moment $h \leftarrow h + h'$ is equal to $lcp(T, T[j..|T|])$, and T and $T[j..|T|]$ mismatch with $u'[1 + h']$ and $v'[1 + h']$. If $u'[1 + h'] \succ v'[1 + h']$, we have done the computation as Duval's algorithm does.

When we reset j after $u'[1 + h'] \prec v'[1 + h']$ happens, we skip some positions by utilizing $\Gamma_{v[1..|v|-1]}$ in light of Lemma 37. Let $f_i = f_k v[|v|] = \rho(v)$, i.e., we are processing the i-th LZ78 factor. Since f_k is an LZ78 factor appearing before f_i, we can use Lemma 37, i.e., we only have to consider the positions where significant suffixes of f_k begin. Moreover, at Lines 8-11 we have maintained the first LZ-block x of K to be the longest member in $\bigcup_{i'=1}^{i} \Gamma_{f_{i'}}$ that is also a prefix of T. Since $x[1..|v'|] \preceq v'$ for any $v' \in \Gamma_{f_k}$, we can notice that $x[1..|v'|] \prec v'$ if $x[1..|v'|] \neq v'$, and hence we are able to skip such positions. Then we set j to be the beginning position of the longest member $v' \in \Gamma_{f_k}$ with $x[1..|v'|] = v'$ if such exists, otherwise the ending position of v, and restart suffix competition.

As for the maintenance of the first LZ-block of K at Lines 8-11, since any LZ78 factor has form $f_i = f_k a$ with $1 \leq k < i \leq s$ and $a \in \Sigma$, the length of the longest member in $\bigcup_{i'=1}^{i} \Gamma_{f_{i'}}$ that is also a prefix of T increases at most one when processing the new LZ78 factor. Hence the procedures at Lines 8-11 works fine as far as the first LZ-block has maintained properly.

The following is the main theorem of this section.

Theorem 8. *Given the LZ78 encoding of size s for string T, we can compute LF_T in $O(s \log s)$ time and space.*

Proof. We compute the Lyndon factorization of T from left to right using Algorithm 4 recursively. We pre-process in $O(s)$ time and space for data structures LAQ and LCS on $LZtrie_T$. We also compute Γ_{f_i} for all LZ78 factors f_i in $O(s \log s)$ time and space by Lemma 38.

During the whole computation, for any LZ78 factor f_i we execute Lines 8-11 just once. Since $|\Gamma_{f_i}| = O(\log s)$ it takes in total of $O(s \log s)$ time. In what follows, we consider the cost other than that comes from Lines 7-11.

Let K_1 denote the linked list of the sequence of the LZ78 factors of T. We show that Algorithm 4 computes, given K_1, the first Lyndon factor of T in $O(\mu_1 \log s + \eta_1)$ time, where $\hat{j}_1$ and $\hat{h}_1$ are respectively the last values of j and h when algorithm halts, and $\mu_1 = \#_{K_1}[1, \hat{j}_1]$ and $\eta_1 = \#_{K_1}[1, \hat{j}_1 + \hat{h}_1]$.

Firstly, let us estimate the total cost for the if-control of Line 15. Let t, t' and t'' be the numbers we execute Lines 21, 22 and 23, respectively. When we enter the if-control, any one of them must be executed. Here note that $\texttt{next}(v)$ is executed at most η_1. Since $\texttt{next}(v)$ must be executed just after either Line 21 or Line 23 is executed, $t + t'' \leq \eta_1$. In addition, if we execute Line 22 more than three consecutive times Line 18 reduces the number of LZ-blocks in K_1, and hence $t' \leq 3\eta_1$. Since the unit cost of the if-control is $O(1)$, the total cost for the if-control is $O(t + t' + t'') = O(\eta_1)$. Next, the else-control of Line 24 is executed $O(\mu_1)$ times since we either halt the computation at Line 26, or reset j to be in the last LZ-block we are processing at Line 30 and j will get over that LZ-block when Line 30 is executed next time. Since Line 25 and Line 29 take $O(\log s)$ time, the cost for the else-control is $O(\mu_1 \log s)$ in total. Hence the first Lyndon factor of T can be computed in $O(\mu_1 \log s + \eta_1)$ time.

After computing the first Lyndon factor, we modify K_1 to K_2 which represents the remaining suffix of T, i.e., we discard the LZ-blocks representing $T[1..|\lambda_1|p_1]$. Also we maintain its first block to be the longest member in $\bigcup_{i'=1}^{i} \Gamma_{f_{i'}}$, where f_i is the last LZ78 factor we have processed. The modification takes $O(\eta_1)$ time.

Then we use Algorithm 4 to compute the second Lyndon factor of T, i.e., the first Lyndon factor of str_{K_2}. The computation takes $O(\mu_2 \log s + \eta_2)$ time, where $\hat{j}_2$ and $\hat{h}_2$ are respectively the last values of j and h when algorithm halts, and $\mu_2 = \#_{K_2}[1, \hat{j}_2]$ and $\eta_2 = \#_{K_2}[1, \hat{j}_2 + \hat{h}_2]$. We iterate this procedure until we get the last Lyndon factor $\lambda_m^{p_m}$ of T. The sum of the cost is $O(\sum_{i=1}^{m} \mu_i \log s + \sum_{i=1}^{m} \eta_i)$. Since $lfb_T(i) + \hat{j}_i \leq lfb_T(i+1)$ for any $1 \leq i < m$, $\sum_{i=1}^{m} \mu_i = O(\sum_{i=1}^{m} \#_{K_i}[1, \hat{j}_i]) = O(\#_{K_1}[1, |T|]) = O(s)$, and hence $O(\sum_{i=1}^{m} \mu_i \log s) = O(s \log s)$.

The final concern is how we can analyze $\sum_{i=1}^{m} \eta_i = O(s)$. Since the substrings of T considered in each iteration are overlapped, e.g., $T[1..\hat{j}_1 + \hat{h}_1]$ and $T[lfb_T(2)..lfb_T(2) + \hat{j}_2 + \hat{h}_2 - 1]$

are overlapped at most $|\lambda_1|$, we cannot conclude immediately that $\sum_{i=1}^{m} \eta_i = O(\#_{K_1}[1, |T|]) = O(s)$. However, we can charge the cost from the overlapped LZ-blocks to the previous LZ-blocks thanks to the restructuring at Line 20. For example, when $T[1..\hat{j}_1+\hat{h}_1]$ and $T[lfb_T(2)..|T|]$ are overlapped (i.e., $lfb_T(2) = |\lambda_1|p_1+1 \leq \hat{j}_1+\hat{h}_1$), $\#_{K_2}[1, \hat{j}_1+\hat{h}_1-|\lambda_1|p_1] = O(\#_{K_1}[lfb_T(2)-|\lambda_1|, \hat{j}_1 + \hat{h}_1 - |\lambda_1|])$ holds (see also Figure 5.2). Hence, $\sum_{i=1}^{m} \eta_i = O(2\sum_{i=1}^{m} \#_{K_i}[1, |\lambda_i|p_i]) = O(2\#_{K_1}[1, |T|]) = O(s)$.

Therefore the statement holds. □

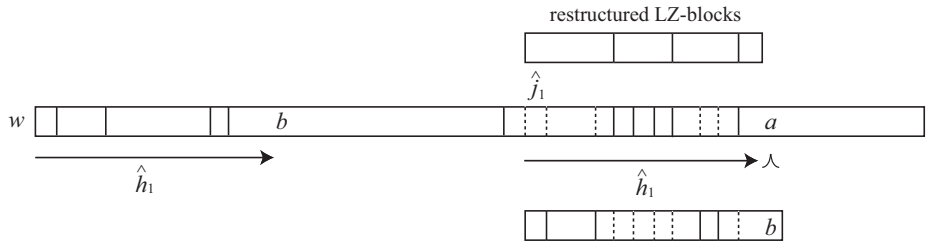

Figure 5.2: Illustration for Theorem 8. We compute $\hat{j}_1$ and $\hat{h}_1$ by LZ-block wise comparisons. In this picture, $p_1 = 1$ and $lfb_T(2) = |\lambda_1|p_1 + 1 = \hat{j}_1$. Then $T[1..\hat{j}_1 + \hat{h}_1]$ and $T[lfb_T(2)..|T|]$ overlap by $\hat{h}_1$ characters. As a result of restructuring at Line 20, the number of the restructured LZ-blocks representing $T[\hat{j}_1..\hat{j}_1 + \hat{h}_1]$ is upper bounded by $O(\#_{K_1}[\hat{j}_1 - |\lambda_1|, \hat{j}_1 + \hat{h}_1 - |\lambda_1|])$, and hence $\#_{K_2}[1, \hat{j}_1 + \hat{h}_1 - |\lambda_1|p_1] = O(\#_{K_1}[\hat{j}_1 - |\lambda_1|, \hat{j}_1 + \hat{h}_1 - |\lambda_1|])$.

5.5 Conclusions

In this chapter, we developed Lyndon factorization algorithms for compressed strings again. Firstly, we presented an algorithm to compute the Lyndon factorization of a grammar compressed string. Our algorithm runs in $O(n^2 + P(n, N) + Q(n, N)n \log n)$ time and $O(n^2 + S(n, N))$ space. This algorithm is faster than the algorithm presented in Chapter 4. Secondly, we presented an algorithm to compute the Lyndon factorization of an LZ78 compressed string. This algorithm works in $O(s \log s)$ time and space where s is the size of the LZ78 factorization. In this chapter, we showed an independent result that the size of the Lyndon factorization, for any string, is a lower bound of the size of the smallest grammar. This result led to the problem which is considered in Chapter 6.

Our question for this problem are the following:

- Can we compute the Lyndon factorization of an SLP compressed string more efficiently?

- Can we compute the Lyndon factorization of an LZ78 compressed string in $O(s)$ time with $O(s \log s)$ space?
- Can we make an efficient algorithm for other compressed strings?

Chapter 6

On the Size of Lempel-Ziv and Lyndon Factorizations

In this chapter, we study relations between Lyndon factorizations and non-overlapping LZ77 factorizations. We present an upper bound and a lower bound of the size of Lyndon factorizations by using the size of LZ77 factorizations.

6.1 Notation

We consider finite strings over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$, which is linearly ordered: $c_1 \prec c_2 \prec \cdots \prec c_\sigma$. A factor u may be equal to several substrings of T, referred to as *occurrences* of u in T.

A string u over Σ is lexicographically smaller than a string v (denoted by $u \preceq v$) if either u is a prefix of v or $u = xaw_1$, $v = xbw_2$ for some strings x, w_1, w_2 and some letters $a \prec b$. In the latter case, we refer to this occurrence of a (resp., of b) as the *mismatch* of u with v (resp., of v with u).

6.2 Upper Bound

The aim of this section is to prove the following theorem.

Theorem 9. *Let $LF_T = \lambda_1^{e_1} \cdots \lambda_m^{e_m}$ and $LZ77_T = g_1 \cdots g_t$. For any string T, $m < 2t$ holds.*

Let us fix an arbitrary string T and relate all notation $(\lambda_i, e_i, \Lambda_i, g_i, m, t)$ to T. The main line of the proof is as follows. We identify occurrences of some substrings in T that must

contain a boundary between two LZ77 phrases. Non-overlapping occurrences contain different boundaries, so our aim is to prove the existence of more than $m/2$ such occurrences. We start with two basic facts; the first one is obvious.

Lemma 39. *For any strings u, v, w_1, w_2, the relation $uw_1 \prec v \prec uw_2$ implies that u is a prefix of v.*

Lemma 40. *The inequality $j < i$ implies $\lambda_j \succ \Lambda_i$.*

Proof. We prove that $\lambda_j \succ \lambda_i^k$ for any k, arguing by induction on k. The base case $k = 1$ follows from the definitions. Let $\lambda_j \succ \lambda_i^{k-1}$. In the case of mismatch, $\lambda_j \succ \lambda_i^k$ holds trivially. Otherwise, $\lambda_j = \lambda_i^{k-1}x$ for some $x \neq \varepsilon$. If $x = \lambda_i$ or $x \prec \lambda_i$, then $x \prec \lambda_j$, and so λ_j is not a Lyndon word. Hence $x \succ \lambda_i$ and thus $\lambda_j = \lambda_i^{k-1}x \succ \lambda_i^k$. Thus, the inductive step holds. □

The next lemma locates the leftmost occurrences of the Lyndon factors and their products.

Lemma 41. *Let $d \geq 1$ and $1 \leq i \leq m - d + 1$, and assume that $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ has an occurrence to the left of the trivial one in T. Then:*

1. *The leftmost occurrence of $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ is a prefix of λ_j for some $j < i$;*
2. *$\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ is a prefix of every λ_k with $j < k < i$.*

Proof. (1) Let j be the smallest integer such that the leftmost occurrence of $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ in T overlaps Λ_j. Suppose first that the leftmost occurrence of $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ is not entirely contained inside a single occurrence of λ_j. Then there exists a non-empty suffix u of λ_j that is equal to some prefix of one of the decomposed Lyndon factors $\lambda_i, \ldots, \lambda_{i+d-1}$, say $\lambda_{i'}$. We cannot have $u = \lambda_j$ because then $\lambda_j \preceq \lambda_{i'}$ which is impossible since $j < i'$. Thus u must be a proper suffix of λ_j. But then $u \preceq \lambda_{i'} \prec \lambda_j$, which contradicts λ_j being a Lyndon word.

Suppose then that the leftmost occurrence of $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ in T is entirely contained inside λ_j but is not its prefix, i.e., $\lambda_j = v\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}v'$ for some strings $v \neq \varepsilon$ and v'. Since λ_j is a Lyndon word we have $\lambda_j \prec \Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}v'$. Consider the position of the mismatch of $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}v'$ with λ_j. If the mismatch occurs inside $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$, we can write $\lambda_j = \Lambda_i\cdots\Lambda_{i'-1}\lambda_{i'}^e x$ where $i \leq i' < i + d$, $0 \leq e < e_i$, and x is a suffix of λ_j that satisfies $x \prec \lambda_{i'} \prec \lambda_j$, which contradicts λ_j being a Lyndon word. On the other hand, the mismatch inside v' implies that λ_j begins with $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$, contradicting the assumption that the inspected occurrence of $\Lambda_i\Lambda_{i+1}\cdots\Lambda_{i+d-1}$ is the leftmost in T.

(2) We prove this part by induction on d. Let $d = 1$. By Lemma 40 we have $\lambda_j \succ \lambda_k \succ \Lambda_i$. Since λ_j begins with Λ_i by statement 1, so does λ_k by Lemma 39. Assume now that the claim holds for all $d' < d$. From the inductive assumption Λ_i and $\Lambda_{i+1} \cdots \Lambda_{i+d-1}$ are both prefixes of λ_k. Let y, y', and z be such that $\lambda_j = \Lambda_i\Lambda_{i+1} \cdots \Lambda_{i+d-1}y$, $\lambda_k = \Lambda_{i+1} \cdots \Lambda_{i+d-1}y' = \Lambda_i z$. We have $j < k$ and thus $\lambda_k \prec \lambda_j$ must hold which, since Λ_i is a prefix of both λ_j and λ_k, implies $z \prec \Lambda_{i+1} \cdots \Lambda_{i+d-1}y$. On the other hand, since λ_k is a Lyndon word, we have $\lambda_k = \Lambda_{i+1} \cdots \Lambda_{i+d-1}y' \prec z$. By Lemma 39, $\Lambda_{i+1} \cdots \Lambda_{i+d-1}y' \prec z \prec \Lambda_{i+1} \cdots \Lambda_{i+d-1}y$ implies that $\Lambda_{i+1} \cdots \Lambda_{i+d-1}$ is a prefix of z or equivalently that $\Lambda_i\Lambda_{i+1} \cdots \Lambda_{i+d-1}$ is a prefix of λ_k. □

6.2.1 Domains

Lemma 41 motivates the following definition.

Definition 4. *Let $d \geq 1$ and $1 \leq i \leq m - d + 1$. We define the* d-domain *of Lyndon factor Λ_i as the substring $\mathrm{dom}_d(\Lambda_i) = \Lambda_j\Lambda_{j+1} \cdots \Lambda_{i-1}$, $j \leq i$ of T, where Λ_j is the Lyndon factor (which exists by Lemma 41) starting at the same position as the leftmost occurrence of $\Lambda_i\Lambda_{i+1} \cdots \Lambda_{i+d-1}$ in T. Note that if $\Lambda_i\Lambda_{i+1} \cdots \Lambda_{i+d-1}$ does not have any occurrence to the left of the trivial one then $\mathrm{dom}_d(\Lambda_i) = \varepsilon$. The integers d and $i - j$ are called the* order *and* size *of the domain, respectively.*

The extended d-domain *of Λ_i is the substring $\mathrm{extdom}_d(\Lambda_i) = \mathrm{dom}_d(\Lambda_i)\Lambda_i \cdots \Lambda_{i+d-1}$ of T.*

Lemma 41 implies two easy properties of domains presented below as Lemma 42. These properties lead to a convenient graphical notation to illustrate domains (see Figure 6.1).

Lemma 42. *Let $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}$, $j \leq i$. Then:*

- *For any $d' > d$, $\mathrm{dom}_{d'}(\Lambda_i)$ is a suffix of $\mathrm{dom}_d(\Lambda_i)$;*
- *For any $d' \geq 1$, $\mathrm{dom}_{d'}(\Lambda_k)$ is a substring of $\mathrm{dom}_d(\Lambda_i)$ if $j \leq k < i$.*

Definition 5. *Consider $\mathrm{dom}_d(\Lambda_i)$ for some $d \geq 1$, $1 \leq i \leq m-d+1$, and let $\alpha = \Lambda_i \cdots \Lambda_{i+d-1}$. We say that the leftmost occurrence of α in T is* associated *with $\mathrm{dom}_d(\Lambda_i)$.*

For example, in Figure 6.1, $T[7..9]$ is associated with $\mathrm{dom}_2(T[23..24])$; $T[7..17]$ is associated with $\mathrm{dom}_1(T[7..17])$ even though it is not shown, since $\mathrm{dom}_1(T[7..17]) = \varepsilon$. Observe that due to Lemma 42 this implies that $\mathrm{dom}_d(T[7..17]) = \varepsilon$ for any $d > 1$, and hence for example the substring of T associated with $\mathrm{dom}_2(T[7..17])$ is $T[7..22]$.

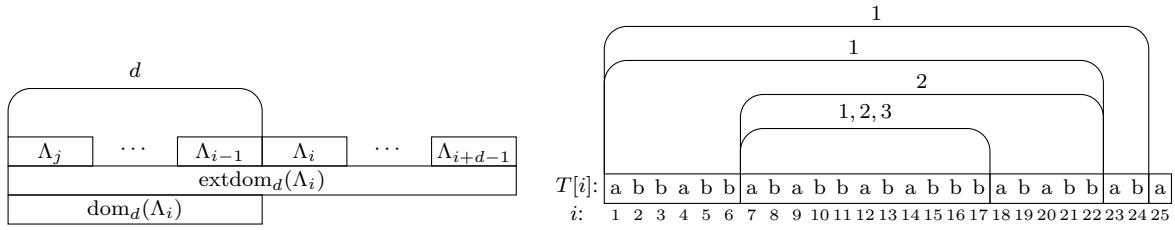

Figure 6.1: Left: Graphical notation used to illustrate $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}$. Also shown is $\mathrm{extdom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i+d-1}$. Right: all non-empty domains for the example string with the Lyndon factorization of size 5. Note that due to Lemma 42 there are no non-trivial intersections between domains.

A substring $T[i..i+k]$, $k \geq 0$ is said to *contain an LZ77 phrase boundary* if some phrase of the LZ-factorization of T begins in one of the positions $i, \ldots, i+k$. Clearly, non-overlapping substrings contain different phrase boundaries. Furthermore, if the substring of T does not have any occurrence to the left (in particular, if it is the leftmost occurrence of a single symbol), it contains an LZ77 phrase boundary, thus we obtain the following easy observation.

Lemma 43. *Each substring associated with a domain contains an LZ77 phrase boundary.*

6.2.2 Tandem Domains

Definition 6. *Let $d \geq 1$ and $1 \leq i \leq m - d$. A pair of domains $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$ is called a* tandem domain *if $\mathrm{dom}_{d+1}(\Lambda_i) \cdot \Lambda_i = \mathrm{dom}_d(\Lambda_{i+1})$ or, equivalently, if $\mathrm{extdom}_{d+1}(\Lambda_i) = \mathrm{extdom}_d(\Lambda_{i+1})$. Note that we permit $\mathrm{dom}_{d+1}(\Lambda_i) = \varepsilon$.*

For example, $\mathrm{dom}_3(T[18..22]), \mathrm{dom}_2(T[23..24])$ is a tandem domain in Figure 6.1, because we have $\mathrm{extdom}_3(T[18..22]) = \mathrm{extdom}_2(T[23..24]) = T[7..25]$.

Definition 7. *Let $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$ be a tandem domain. Since $\Lambda_{i+1} \cdots \Lambda_{i+d}$ is a prefix of Λ_i by Lemma 41, we let $\Lambda_i = \Lambda_{i+1} \cdots \Lambda_{i+d} x$. The leftmost occurrence of $\Lambda_i \cdots \Lambda_{i+d}$ in T can thus be written as $\Lambda_{i+1} \cdots \Lambda_{i+d} x \Lambda_{i+1} \cdots \Lambda_{i+d}$. We say that this particular occurrence of the substring $x\Lambda_{i+1} \cdots \Lambda_{i+d}$ is* associated *with the tandem domain $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$.*

Remark 1. *Note that the above definition permits $\mathrm{dom}_{d+1}(\Lambda_i) = \varepsilon$. If $\mathrm{dom}_{d+1}(\Lambda_i) \neq \varepsilon$, then α, the substring of T associated with $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$, is (by Lemma 41) a substring of Λ_j, where Λ_j, $j < i$ is the leftmost Lyndon factor inside $\mathrm{dom}_{d+1}(\Lambda_i)$. Otherwise, α overlaps at least two Lyndon factors. In both cases, however, α is a substring of $\mathrm{extdom}_{d+1}(\Lambda_i)$.*

Lemma 44. *Each substring associated with a tandem domain contains an LZ77 phrase boundary.*

Proof. Let $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$ be a tandem domain and let $u = x\Lambda_{i+1}\cdots\Lambda_{i+d}$ be the associated substring of T. Suppose to the contrary that u contains no LZ77 phrase boundaries. Then some LZ77-phrase g_τ contains u and the letter preceding u. Since we consider a non-overlapping LZ77 variant, the previous occurrence of g_τ in T must be a substring of $g_1\cdots g_{\tau-1}$. Note, however, that u is preceded in T by the leftmost occurrence of $\Lambda_{i+1}\cdots\Lambda_{i+d}$, which is the prefix of Λ_j (see Definition 7). Thus, the leftmost occurrence of u in T either immediately precedes the associated substring, or overlaps it, or coincides with it. This, however, rules out the possibility that the previous occurrence of g_τ occurs in $g_1\cdots g_{\tau-1}$, a contradiction. □

We say that a tandem domain $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$ is *disjoint* from a tandem domain $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)$, $\mathrm{dom}_{\hat{d}}(\Lambda_{k+1})$ if all $i, i+1, k, k+1$ are different, i.e., $i+1<k$ or $k+1<i$.

Lemma 45. *Substrings associated with disjoint tandem domains do not overlap each other.*

Proof. Let $\mathrm{dom}_{d+1}(\Lambda_i)$, $\mathrm{dom}_d(\Lambda_{i+1})$ and $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)$, $\mathrm{dom}_{\hat{d}}(\Lambda_{k+1})$ be tandem domains called the d-tandem and $\hat{d}$-tandem, respectively. Without the loss of generality let $i+1<k$.

Case 1: $\mathrm{dom}_{d+1}(\Lambda_i)\neq\varepsilon$ and $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)\neq\varepsilon$. First observe that if the d-tandem and $\hat{d}$-tandem begin with different Lyndon factors, then the associated substrings trivially do not overlap by the above Remark. Assume then that all considered domains start with Λ_j, $j<i$. By Definition 7 we can write Λ_j as $\Lambda_j = \Lambda_{i+1}\cdots\Lambda_{i+d}x\Lambda_{i+1}\cdots\Lambda_{i+d}y$, where $|\Lambda_{i+1}\cdots\Lambda_{i+d}x| = |\Lambda_i|$ and $x\Lambda_{i+1}\cdots\Lambda_{i+d}$ is the substring of T associated with the d-tandem. Similarly we have $\Lambda_j = \Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}x'\Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}y'$ where $|\Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}x'| = |\Lambda_k|$ and $x'\Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}$ is the substring of T associated with the $\hat{d}$-tandem. However, by Lemma 41, $\Lambda_k\cdots\Lambda_{k+\hat{d}}$ is a prefix of Λ_{i+1} and thus $|\Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}x'\Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}| \leq |\Lambda_{i+1}|$, i.e., the substring of T associated with the $\hat{d}$-tandem is inside the prefix Λ_{i+1} of Λ_j and thus is on the left of the substring associated with the d-tandem.

Case 2: $\mathrm{dom}_{d+1}(\Lambda_i) = \Lambda_j\cdots\Lambda_{i-1}$, $j<i$, and $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)=\varepsilon$. In this case the substring associated with the $\hat{d}$-tandem begins in Λ_k by the above Remark and thus is on the right of the substring associated with the d-tandem.

Case 3: $\mathrm{dom}_{d+1}(\Lambda_i)=\varepsilon$ and $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)=\varepsilon$. This is only possible if $i+d<k$ since otherwise Λ_k (and thus also $\Lambda_{k+1}\cdots\Lambda_{k+\hat{d}}$) occurs in Λ_i, contradicting $\mathrm{dom}_{\hat{d}}(\Lambda_{k+1}) = \Lambda_k$. Then, $\mathrm{extdom}_{d+1}(\Lambda_i)$ does not overlap $\mathrm{extdom}_{\hat{d}+1}(\Lambda_k)$, and the claim holds by above Remark.

Case 4: $\mathrm{dom}_{d+1}(\Lambda_i) = \varepsilon$ and $\mathrm{dom}_{\hat{d}+1}(\Lambda_k) = \Lambda_j \cdots \Lambda_{k-1}$, $j < k$. Then, the substring of T associated with $\hat{d}$-tandem is a substring of Λ_j. If $i > j$, then clearly $\mathrm{extdom}_{d+1}(\Lambda_i)$ does not overlap Λ_j. On the other hand, if $i < j$, it must also hold $i + d < j$ since otherwise Λ_j (and thus also $\Lambda_k \cdots \Lambda_{k+\hat{d}}$) occurs in Λ_i, contradicting $\mathrm{dom}_{\hat{d}+1}(\Lambda_k) = \Lambda_j \cdots \Lambda_{k-1}$, and thus again, $\mathrm{extdom}_{d+1}(\Lambda_i)$ does not overlap Λ_j. In both cases the Remark above implies the claim. Finally, if $i = j$, we must also have $i + 1 < k$ from the assumption about the disjointness of d- and $\hat{d}$-tandem. By Lemma 41 we can write $\Lambda_i = \Lambda_{i+1} \cdots \Lambda_{i+d} x$, $\Lambda_{i+1} = \Lambda_k \cdots \Lambda_{k+\hat{d}} x'$ and hence also $\Lambda_i \cdots \Lambda_{i+d} = \Lambda_k \cdots \Lambda_{k+\hat{d}} x' \Lambda_{i+2} \cdots \Lambda_{i+d} x \Lambda_{i+1} \cdots \Lambda_{i+d}$. In this decomposition, the substring associated with the $\hat{d}$-tandem occurs inside the prefix $\Lambda_k \cdots \Lambda_{k+\hat{d}}$, and the substring associated with the d-tandem is the suffix $x\Lambda_{i+1} \cdots \Lambda_{i+d}$, which proves the claim. □

6.2.3 Groups

We now generalize the concept of tandem domain.

Definition 8. *Let* $d \geq 1$, $2 \leq p \leq m$, *and* $1 \leq i \leq m - d - p + 2$. *A set of* p *domains* $\mathrm{dom}_{d+p-1}(\Lambda_i)$, $\mathrm{dom}_{d+p-2}(\Lambda_{i+1})$, …, $\mathrm{dom}_d(\Lambda_{i+p-1})$ *is called a* p-group *if for all* $q = 0, \ldots, p-2$ *the equality* $\mathrm{dom}_{d+p-1-q}(\Lambda_{i+q}) \cdot \Lambda_{i+q} = \mathrm{dom}_{d+p-2-q}(\Lambda_{i+q+1})$ *holds or, equivalently,* $\mathrm{extdom}_{d+p-1}(\Lambda_i) = \ldots = \mathrm{extdom}_d(\Lambda_{i+p-1})$. *Note that we permit* $\mathrm{dom}_{d+p-1}(\Lambda_i) = \varepsilon$.

Lemma 46. *Substrings associated with tandem domains from the same group do not overlap each other.*

Proof. Consider a p-group, $p \geq 3$ and assume first that $p = 3$. By Lemma 41 we have $\Lambda_i = \Lambda_{i+1} \cdots \Lambda_{i+d+1} x'$ and $\Lambda_{i+1} = \Lambda_{i+2} \cdots \Lambda_{i+d+1} x$ for some words x' and x. We can thus write the leftmost occurrence of $\Lambda_i \cdots \Lambda_{i+d+1}$ in T as $\Lambda_{i+2} \cdots \Lambda_{i+d+1} x \Lambda_{i+2} \cdots \Lambda_{i+d+1} x' \Lambda_{i+1} \cdots \Lambda_{i+d+1}$. It is easy to see that those occurrences of $x\Lambda_{i+2} \cdots \Lambda_{i+d+1}$ and $x'\Lambda_{i+1} \cdots \Lambda_{i+d+1}$ are associated with (resp.) tandem domains $\mathrm{dom}_{d+1}(\Lambda_{i+1})$, $\mathrm{dom}_d(\Lambda_{i+2})$ and $\mathrm{dom}_{d+2}(\Lambda_i)$, $\mathrm{dom}_{d+1}(\Lambda_{i+1})$, and thus the claim holds.

For $p > 3$ it suffices to consider all subgroups of size three, in left-to-right order, to verify that the substrings associated with all tandem domains occur in reversed order as a contiguous substring and thus no two substrings overlap each other. □

The above Lemma is illustrated in Figure 6.2. It also motivates the following definition which generalizes the concept of associated substring from tandem domains to groups.

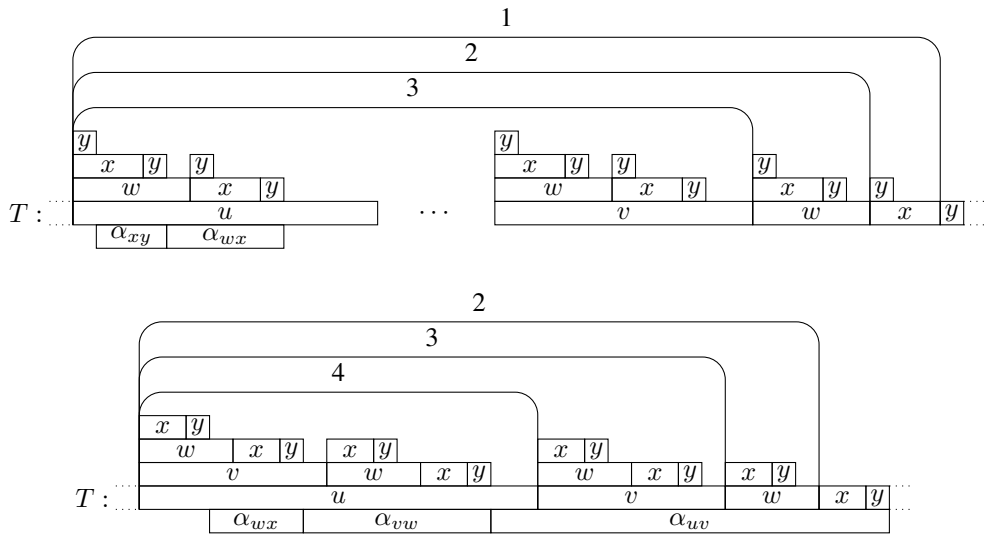

Figure 6.2: Illustration of Lemma 46. In the examples u, v, w, x, y are Lyndon factors from the Lyndon factorization of T. The top figure shows a 3-group: $\mathrm{dom}_3(w) = u \cdots v$, $\mathrm{dom}_2(x) = u \cdots vw$, $\mathrm{dom}_1(y) = u \cdots vwx$. α_{wx} is a substring associated with the tandem domain $\mathrm{dom}_3(w)$, $\mathrm{dom}_2(x)$, and α_{xy} is a substring associated with the tandem domain $\mathrm{dom}_2(x)$, $\mathrm{dom}_1(y)$. Observe that the substrings associated with tandem domains occur as a contiguous substring and in reverse order (compared to the order of the corresponding tandem domains in T). The bottom figure shows a 4-group: $\mathrm{dom}_5(u) = \varepsilon$, $\mathrm{dom}_4(v) = u$, $\mathrm{dom}_3(w) = uv$, $\mathrm{dom}_2(x) = uvw$ and demonstrates the case when the leftmost domain in a group is empty.

Definition 9. *Consider a p-group $\mathrm{dom}_{d+p-1}(\Lambda_i)$, $\mathrm{dom}_{d+p-2}(\Lambda_{i+1})$, …, $\mathrm{dom}_d(\Lambda_{i+p-1})$ for some $p \geq 2$. From Lemma 41, $\Lambda_{i+p-1} \cdots \Lambda_{i+p+d-2}$ is a prefix of Λ_i. Thus, the leftmost occurrence of $\Lambda_i \cdots \Lambda_{i+p+d-2}$ in T can be written as $\Lambda_{i+p-1} \cdots \Lambda_{i+p+d-2} x \Lambda_{i+1} \cdots \Lambda_{i+p+d-2}$. We say that this particular occurrence of the substring $x\Lambda_{i+1} \cdots \Lambda_{i+p+d-2}$ is* associated *with the p-group $\mathrm{dom}_{d+p-1}(\Lambda_i)$, $\mathrm{dom}_{d+p-2}(\Lambda_{i+1})$, …, $\mathrm{dom}_d(\Lambda_{i+p-1})$.*

It is easy to derive a formal proof of the following Lemma from the proof of Lemma 46.

Lemma 47. *The substring associated with a p-group is the concatenation, in reverse order, of the $p-1$ substrings associated with the tandem domains belonging to the p-group.*

Our consideration of groups culminates in the next two results.

Corollary 5. *The substring associated with a p-group contains at least $p-1$ different LZ77 phrase boundaries.*

We say that a p-group $\mathrm{dom}_{d+p-1}(\Lambda_i)$, …, $\mathrm{dom}_d(\Lambda_{i+p-1})$ is *disjoint* from a p'-group $\mathrm{dom}_{d'+p'-1}(\Lambda_k)$, …, $\mathrm{dom}_{d'}(\Lambda_{k+p'-1})$ if $i + p - 1 < k$ or $k + p' - 1 < i$. By combining Lemma 45 and Lemma 47 we obtain the following fact.

Lemma 48. *Substrings associated with disjoint groups do not overlap.*

6.2.4 Subdomains

The concept of p-group does not easily extend to $p = 1$. If we simply define the 1-group as a single domain and extend the notion of groups to include 1-groups then Lemma 48 no longer holds (e.g. in Figure 6.1 the substring associated with tandem domain $\mathrm{dom}_3(T[18..22])$, $\mathrm{dom}_3(T[23..24])$ is $T[10..14]$ and the substring associated with domain $\mathrm{dom}_1(T[7..17])$ is $T[7..17]$). Instead, we introduce a weaker lemma (Lemma 49) that also includes single domains.

Definition 10. *We say that a domain* $\mathrm{dom}_{\hat{d}}(\Lambda_k)$ *is a* subdomain *of a domain* $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}$, $j \leq i$ *if* $k = i$ *and* $\hat{d} = d$ *(i.e., the domain is its own subdomain), or* $j \leq k < i$ *and* $\mathrm{extdom}_{\hat{d}}(\Lambda_k)$ *is a substring of* $\mathrm{extdom}_d(\Lambda_i)$ *(or equivalently, if* $k + \hat{d} \leq i + d$*). In other words,* Λ_k *has to be one of the Lyndon factors among* $\Lambda_j, \ldots, \Lambda_{i-1}$ *and the extended domain of* Λ_k *cannot extend (to the right) beyond the extended domain of* Λ_i*.*

Lemma 49. *Consider a tandem domain* $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)$, $\mathrm{dom}_{\hat{d}}(\Lambda_{k+1})$ *such that* $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)$ *and* $\mathrm{dom}_{\hat{d}}(\Lambda_{k+1})$ *are subdomains of* $\mathrm{dom}_d(\Lambda_i)$*. Then, the substring associated with the tandem domain* $\mathrm{dom}_{\hat{d}+1}(\Lambda_k)$, $\mathrm{dom}_{\hat{d}}(\Lambda_{k+1})$ *does not overlap the substring associated with* $\mathrm{dom}_d(\Lambda_i)$*.*

Proof. First, observe that in order for a tandem domain consisting of two subdomains to exist, $\mathrm{dom}_d(\Lambda_i)$ has to be non-empty. Thus, let $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}$ for some $j < i$. This implies (Lemma 41) that the substring associated with $\mathrm{dom}_d(\Lambda_i)$ is a prefix of Λ_j.

Assume first that $\mathrm{dom}_{\hat{d}+1}(\Lambda_k) = \Lambda_{j'} \cdots \Lambda_{k-1}$, $j < j' \leq k$. The substring associated with the tandem domain is a substring of $\mathrm{extdom}_{\hat{d}+1}(\Lambda_k)$ thus it trivially does not overlap Λ_j.

Assume then that $\mathrm{dom}_{\hat{d}+1}(\Lambda_k) = \Lambda_j \cdots \Lambda_{k-1}$. If $k+1 < i$ then by Lemma 41, $\Lambda_i \cdots \Lambda_{i+d-1}$ is a prefix of Λ_{k+1}. By Definition 7 the leftmost occurrence of $\Lambda_k \cdots \Lambda_{k+\hat{d}}$ in T can be written as $\Lambda_{k+1} \cdots \Lambda_{k+\hat{d}} x \Lambda_{k+1} \cdots \Lambda_{k+\hat{d}}$. Thus clearly the leftmost occurrence of $\Lambda_i \cdots \Lambda_{i+d-1}$ (associated with $\mathrm{dom}_k(\Lambda_i)$) occurs in a prefix Λ_{k+1} not overlapped by $x\Lambda_{k+1} \cdots \Lambda_{k+\hat{d}}$ (which is a substring associated with the tandem domain).

The remaining case is when $k + 1 = i$. Then by Definition 10 we must have $\hat{d} = d$ and again the claim holds easily from Definition 7. □

For any domain $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}, j < i$ we define the set of *canonical subdomains* as follows. Consider the following procedure. Initialize the set of canonical subdomains to

contain $\mathrm{dom}_d(\Lambda_i)$. Then initialize $\delta = d$ and start scanning the Lyndon factors $\Lambda_j, \ldots, \Lambda_{i-1}$ right-to-left. When scanning Λ_τ we check if $\mathrm{dom}_{\delta+1}(\Lambda_\tau) = \Lambda_j \cdots \Lambda_{t-1}$.

- If yes, we include $\mathrm{dom}_{\delta+1}(\Lambda_\tau)$ into the set, increment δ and continue scanning from $\Lambda_{\tau-1}$.
- Otherwise, i.e., if $\mathrm{dom}_{\delta+1}(\Lambda_\tau) = \Lambda_{j'} \cdots \Lambda_{\tau-1}$ for some $j' > j$, we include the domain $\mathrm{dom}_{\delta+1}(\Lambda_\tau)$ into the set. Then we set $\delta = 0$ and continue scanning from $\Lambda_{j'-1}$. All domains that were included into the set of canonical subdomains in this case are called *loose* subdomains.

See Figure 6.3 for an example. The above procedure simply greedily constructs groups of domains, and whenever the candidate for the next domain in the current group does not have a domain that starts with Λ_j, we terminate the current group, add the loose subdomain into the set and continue building groups starting with the next Lyndon factor outside the (just included) loose subdomain.

Note that the current group can be terminated when containing just one domain, so it is not a group in this case. Hence we call the resulting sequences of non-loose domains *clusters*, i.e., a cluster is either a single domain, or a p-group ($p \geq 2$). Note also that during the construction we may encounter more than one loose subdomain in a row, so clusters and loose subdomains do not necessarily alternate, but no two clusters occur consecutively.

Finally, observe that the sequence of clusters and loose subdomains always ends with a cluster (possibly of size one) containing $\mathrm{dom}_{d'}(\Lambda_j)$ for some d' ($d' = 3$ for the example in Figure 6.3), since $\mathrm{dom}_{d'}(\Lambda_j) = \varepsilon$ for all d'.

6.2.5 Proof of the Main Theorem

We are now ready to prove the key Lemma of the proof. Recall that the size of $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}$, $j \leq i$ is defined as $i - j$.

Lemma 50. *Let* $\mathrm{dom}_d(\Lambda_i)$ *be a domain of size* $k \geq 0$. *Then* $\mathrm{extdom}_d(\Lambda_i)$ *contains at least* $\lceil k/2 \rceil + 1$ *different LZ77 phrase boundaries.*

Proof. Let $\mathrm{dom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{i-1}$, $j \leq i$ and $k = i - j$. The proof is by induction on k. For $k = 0$, $\mathrm{extdom}_d(\Lambda_i)$ is the substring of T associated with $\mathrm{dom}_d(\Lambda_i)$ (see Definition 5) and thus by Lemma 43, $\mathrm{extdom}_d(\Lambda_i)$ contains at least one LZ77 phrase boundary.

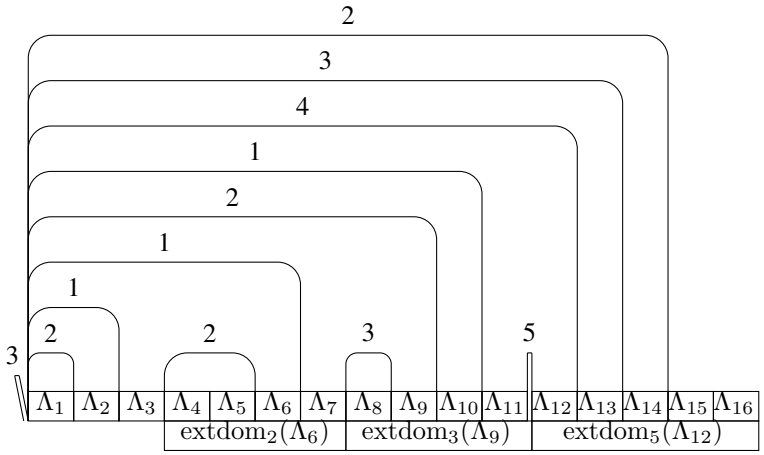

Figure 6.3: An example showing the set of canonical subdomains of $\mathrm{dom}_2(\Lambda_{15})$. Using notation from Lemma 50, the set has $p = 4$ clusters of size (left-to-right): $q_1 = 3, q_2 = 1, q_3 = 2, q_4 = 3$, and $t = 3$ loose subdomains: $\mathrm{dom}_2(\Lambda_6) = \Lambda_4\Lambda_5$, $\mathrm{dom}_3(\Lambda_9) = \Lambda_8$, $\mathrm{dom}_5(\Lambda_{12}) = \varepsilon$ of size $k_1 = 2$, $k_2 = 1$, $k_3 = 0$. Note how the extended domains of loose subdomains do not overlap each other. Furthermore, note that $\mathrm{extdom}_2(\Lambda_{15}) = \Lambda_1 \cdots \Lambda_{16}$ can be factorized as $\Lambda_1 \cdots \Lambda_{q_1}$ concatenated with the extended domains. By Corollary 5 and Lemmas 48 and 49, $\Lambda_1 \cdots \Lambda_{q_1}$ contains $1 + \sum_{h=1}^{p}(q_h - 1) = 6$ LZ77 phrase boundaries, while the extended domains $\mathrm{extdom}_2(\Lambda_6)$, $\mathrm{extdom}_3(\Lambda_9)$, $\mathrm{extdom}_5(\Lambda_{12})$ contain $\sum_{h=1}^{t}(\lceil k_h/2 \rceil + 1) = 5$ LZ77 phrase boundaries by Lemma 50.

Let $k > 0$ and assume now that the claim holds for all smaller k. Consider the set $\mathcal{C}_{i,d}$ of canonical subdomains of $\mathrm{dom}_d(\Lambda_i)$. If $\mathcal{C}_{i,d}$ contains no loose subdomain, it consists of a single cluster which is a $(k+1)$-group. By Corollary 5, the substring associated with this group contains k phrase boundaries; by Lemma 49, one more boundary is provided by the domain $\mathrm{dom}_d(\Lambda_i)$ itself. We have $1 + k \geq 1 + \lceil k/2 \rceil$, which concludes the proof of this case.

For the rest of the proof assume that $\mathcal{C}_{i,d}$ contains $\tau \geq 1$ loose subdomains, denoted, left to right, by $\mathrm{dom}_{d_1}(\Lambda_{i_1}), \ldots, \mathrm{dom}_{d_\tau}(\Lambda_{i_\tau})$. Note that $d_\tau > d$. Let k_h be the size of $\mathrm{dom}_{d_h}(\Lambda_{i_h})$, $h = 1, \ldots, \tau$. Further, let $q \geq 1$ be the size of the leftmost cluster (the one that contains some domain of Λ_j). By the construction of the canonical set we have

$$\mathrm{extdom}_d(\Lambda_i) = \Lambda_j \cdots \Lambda_{j+q-1}\mathrm{extdom}_{d_1}(\Lambda_{i_1})\mathrm{extdom}_{d_2}(\Lambda_{i_2}) \cdots \mathrm{extdom}_{d_\tau}(\Lambda_{i_\tau}). \tag{6.1}$$

Both clusters and loose subdomains contribute some number of LZ77 phrase boundaries into their total. The boundaries contributed by clusters are all different by Lemma 48; let S be their number. These boundaries are also different from the boundary inside the substring associated with $\mathrm{dom}_d(\Lambda_i)$ by Lemma 49. Furthermore, it is easy to see from the proof of Lemma 49 that all these phrase boundaries are located inside $\Lambda_j \cdots \Lambda_{j+q-1}$. The number of phrase boundaries

inside the extended domains of loose subdomains can be estimated by the inductive assumption (by Eq. 6.1, these external domains do not overlap each other or $\Lambda_j \cdots \Lambda_{j+q-1}$). So we obtain that $\mathrm{extdom}_d(\Lambda_i)$ contains at least

$$1 + \sum_{h=1}^{\tau} \left(\left\lceil \frac{k_h}{2} \right\rceil + 1 \right) + S \tag{6.2}$$

different LZ77 phrase boundaries. Let us evaluate $\sum_{h=1}^{t} k_h$. By the construction, a loose d_h-subdomain is followed by exactly d_h Lyndon factors which are outside loose subdomains; then another loose subdomain follows (cf. Figure 6.3). The only exception is the rightmost loose subdomain, which is followed by $d_\tau - d$ Lyndon factors outside loose subdomains (note that we only count Lyndon factors inside $\mathrm{dom}_d(\Lambda_i)$). Then

$$\sum_{h=1}^{\tau} k_h = k - q - \sum_{h=1}^{\tau} d_h + d. \tag{6.3}$$

Next we evaluate S. By Corollary 5, a cluster of size r contributes $r - 1$ phrase boundaries. Then the leftmost (resp., rightmost) cluster contributes $q-1$ (resp., $d_\tau - d - 1$) boundaries. Each of the remaining clusters is preceded by a loose d_h-subdomain, where $d_h > 1$, and contributes $d_h - 2$ boundaries. Using Knuth's notation [predicate] for the numerical value (0 or 1) of the predicate in brackets, we can write

$$S = q - 1 + \sum_{h=1}^{\tau} d_h - \tau - d - \sum_{h=1}^{\tau-1} [d_h > 1]. \tag{6.4}$$

Finally, we estimate the number in Eq. 6.2 using Eq. 6.3 and Eq. 6.4:

$$1+\sum_{h=1}^{\tau} \left(\left\lceil \frac{k_h}{2} \right\rceil + 1 \right) + S \geq 1+\tau+\frac{k-q-\sum_{h=1}^{\tau} d_h + d}{2}+q-1+\sum_{h=1}^{\tau} d_h-\tau-d-\sum_{h=1}^{\tau-1}[d_h > 1]$$

$$= \frac{k}{2} + \frac{q}{2} + \sum_{h=1}^{\tau} \frac{d_h}{2} - \frac{d}{2} - \sum_{h=1}^{\tau-1} [d_h > 1] = \frac{q + d_\tau - d}{2} + \frac{k}{2} + \sum_{h=1}^{\tau-1} \left(\frac{d_h}{2} - [d_h > 1] \right) \geq 1 + \frac{k}{2}.$$

The obtained lower bound for an integer can be rounded up to $1 + \lceil k/2 \rceil$, as required.

□

Using the above Lemma we can finally prove the main Theorem.

Proof.[Proof of Theorem 9] Partition the string T into extended domains as follows: take the

string T' such that $T = T' \cdot \mathrm{extdom}_1(\Lambda_m)$ and partition T' recursively to get

$$T = \mathrm{extdom}_1(\Lambda_{i_1}) \cdots \mathrm{extdom}_1(\Lambda_{i_\tau}), \text{ where } i_\tau = m.$$

By Lemma 50, each extended domain $\mathrm{extdom}_1(\Lambda_{i_h})$ contains at least $\lceil k_h/2 \rceil + 1$ phrase boundaries, where k_h is the size of the domain $\mathrm{dom}_1(\Lambda_{i_h})$. Clearly, $\sum_{h=1}^{\tau} k_h = m - \tau$; hence the total number t of the boundaries satisfies

$$t \geq \sum_{h=1}^{\tau} \left(\left\lceil \frac{k_h}{2} \right\rceil + 1 \right) \geq \left\lceil \frac{m-\tau}{2} \right\rceil + \tau = \left\lceil \frac{m+\tau}{2} \right\rceil > \frac{m}{2},$$

as required. □

6.3 Lower Bound

The upper bound on the number of factors in the Lyndon factorization of a string, given in of Theorem 9, is supported by the following lower bound. Consider a string $T_k = B_0 \cdots B_k a$, $k \geq 0$, where:

$$\begin{aligned} B_0 &= b, \\ B_1 &= ab, \\ B_2 &= a^2baba^2b, \\ &\cdots \\ B_k &= (a^k ba^1 b) \cdots (a^k ba^{k-1} b) a^k b. \end{aligned}$$

For example, $T_3 = (b)(ab)(a^2baba^2b)(a^3baba^3ba^2ba^3b)(a)$.

Theorem 10. *Let $\lambda_1 \cdots \lambda_{m_k}$ and $g_1 \cdots g_{t_k}$ be the Lyndon factorization and the non-overlapping LZ77 factorization of the string T_k, $k \geq 2$. Then $m_k = k^2/2 + k/2 + 2$, $t_k = k^2/2 - k/2 + 4$, and thus $m_k = t_k + \Theta(\sqrt{t_k})$.*

Proof. First we count Lyndon factors. All factors will be different, so their number coincides with the number of Lyndon factors. By the definition of Lyndon factorization, the block B_i $(0 < i \leq k)$ is factorized into i Lyndon factors:

$$B_i = a^i ba^1 b \cdot a^i ba^2 b \cdots a^i ba^{i-1} b \cdot a^i b. \tag{6.5}$$

For any suffix u of $B_0 \cdots B_{i-1}$ and any prefix v of B_i, $u \succ v$ holds since a^i is a prefix of B_i and this is the leftmost occurrence of a^i. Thus there is no Lyndon word that begins in $B_0 \cdots B_{i-1}$ and ends in B_i. This implies that the factorization of T_k is the concatenation of the first b, then k factorizations Eq. 6.5, and the final a, $k^2/2 + k/2 + 2$ factors in total.

Let $LZ77_{T_k}$ denote the LZ factorization of T_k. The size of $LZ77_{T_2} = b \cdot a \cdot ba \cdot aba \cdot baaba$ is 5. For $k \geq 3$, we prove by induction that

$$LZ77_{T_k} = LZ77_{T_{k-1}} \cdot a^{k-1}baba^{k-1} \cdot aba^2ba^{k-1} \cdots aba^{k-2}ba^{k-1} \cdot aba^{k-1}ba^kba. \tag{6.6}$$

For $k = 3$ we have $LZ77_{T_3} = LZ77_{T_2} \cdot aababaa \cdot abaabaaaba$ and thus the claim holds. If $k > 3$, by the inductive hypothesis the last phrase in $LZ77_{T_{k-1}}$ is $g = aba^{k-2}ba^{k-1}ba$. The substring g has only one previous occurrence: it occurs at the boundary between B_{k-2} and B_{k-1}, followed by b. So, g remains a phrase in $LZ77_{T_k}$. Each of subsequent $k-2$ phrases of Eq. 6.6 also has a single previous occurrence (inside B_{k-1}), and this occurrence is followed by b because B_{k-1} has no substring a^k. Thus, Eq. 6.6 correctly represents $LZ77_{T_k}$. Direct computation now gives $t_k = k^2/2 - k/2 + 4$. □

6.4 Conclusions

In Chapter 6, we studied relations between Lyndon factorizations and non-overlapping LZ77 factorizations. We showed that the number of Lyndon factors cannot be more than $2t$ where t is the number of LZ77 phrases. This result is the first direct connection of these two factorizations. The result improves significantly a previous, indirect bound given in Chapter 5. We also showed that there are strings with $t + \Theta(\sqrt{t})$ Lyndon factors.

Our question for this problem are the following:

- What are tighter upper bound and lower bound?
- Can we show some bounds for overlapping LZ77 factorizations?

Chapter 7

The Runs Theorem

In this chapter, we give new insights into relations between runs and Lyndon words. Firstly, we give additional notation for this chapter in Section 7.1. In Section 7.2, we show a new upper bound of the maximum number of runs in a string. In Section 7.3, we propose a new algorithm to compute all runs in a string. This algorithm is the first algorithm without Lempel-Ziv factorization. Finally, we show how to characterize runs in a string using Lyndon trees in Section 7.4.

7.1 Notation

For any set I of intervals, let $Beg(I)$ denote the set of beginning positions of intervals in I.

Definition 11 (Runs)**.** *A triple $r = (i, j, p)$ is a run of string T, if the smallest period p of $T[i..j]$ satisfies $|T[i..j]| \geq 2p$, and the periodicity cannot be extended to the left or right, i.e., $i = 1$ or $T[i-1] \neq T[i+p-1]$, and, $j = N$ or $T[j+1] \neq T[j-p+1]$. The rational number $\frac{j-i+1}{p}$ is called the exponent of r, and is denoted by e_r.*

Let $Runs(T)$ denote the set of runs of string T. Denote by $\rho(N)$, the maximum number of runs that are contained in a string of length N, and by $\sigma(N)$, the maximum sum of exponents of runs that are contained in a string of length N.

Example 6. *Consider string $T =$* `babbabbababbabbabc` *which contains nine runs. $Runs(T) = \{(1, 9, 3), (1, 17, 8), (3, 4, 1), (4, 14, 5), (6, 7, 1), (7, 11, 2), (9, 17, 3), (11, 12, 1), (14, 15, 1)\}$. (See Figure 7.1.)*

The following is an important lemma that is central to two of our main observations (Lemmas 52 and 53) used to prove the runs conjecture in Section 7.2.

Lemma 51 (Lemma 1.6 of [30]). *Let $T = u^q u' a$ be a string for some Lyndon word u, a possibly empty proper prefix u' of u, a positive integer q, and $a \in \Sigma$ with $T[|u'|+1] \neq a$. If $u[|u'|+1] \prec a$, T is a Lyndon word. If $a \prec u[|u'|+1]$, u is the longest prefix Lyndon word of any string having a prefix $u^q u' a$.*

Definition 12 (L-root [22]). *Let $r = (i, j, p)$ be a run in string $T \in \Sigma^*$. An interval $\lambda = [i_\lambda..j_\lambda]$ of length p is an L-root of r with respect to $\prec$ if $i \leq i_\lambda \leq j_\lambda \leq j$ and $T[i_\lambda..j_\lambda]$ is a Lyndon word with respect to $\prec$.*

It is easy to see that for any run and lexicographic order $\prec$, there exists at least one L-root with respect to $\prec$.

7.2 The Runs Theorem

Since any string over a unary alphabet can only have at most one run, we assume a non-unary alphabet Σ. Furthermore, we consider lexicographic orders on strings over Σ, induced by an arbitrary pair of total orders $\prec_0$, $\prec_1$ on Σ such that for any pair of characters $a, b \in \Sigma, a \prec_0 b \Leftrightarrow b \prec_1 a$. For $\ell \in \{0, 1\}$, let $\overline{\ell} = 1 - \ell$. For any string $T \in \Sigma^*$, let $\hat{T} = T\$$, where $\$ \notin \Sigma$ is a special character that satisfies $\$ \prec_0 a$ (and thus $a \prec_1 \$$) for any $a \in \Sigma$. We first define a notation for representing the interval corresponding to the longest Lyndon word that starts at a given position.

Definition 13 (Longest Lyndon word that starts at given position). *For any string T, position i ($1 \leq i \leq |T|$), and $\ell \in \{0, 1\}$, let $l_\ell(i) = [i..j]$ where $j = \max\{j' \mid \hat{T}[i..j']$ is a Lyndon word with respect to $\prec_\ell\}$.*

Our first main observation is that, if we consider the longest Lyndon words with respect to $\prec_0$ and $\prec_1$ that starts at a given position i, one of them will be of length 1, while the other will be of length greater than 1.

Lemma 52. *For any string T and position i ($1 \leq i \leq |T|$), let $\ell \in \{0, 1\}$ be such that $\hat{T}[k] \prec_\ell \hat{T}[i]$ for $k = \min\{k' \mid \hat{T}[k'] \neq \hat{T}[i], k' > i\}$. Then, $l_\ell(i) = [i..i]$ and $l_{\overline{\ell}}(i) = [i..j]$ for some $j > i$.*

Proof. The lemma follows from the definition of $\prec_\ell$ and Lemma 51, with $u = \hat{T}[i]$, $u' = \varepsilon$, $q = k - i$, and $a = \hat{T}[k]$. □

Example 7. *Consider string* $T =$ `abbabc` *and assume* `a` $\prec_0$ `b` $\prec_0$ `c`, `c` $\prec_1$ `b` $\prec_1$ `a`. *Then* $(l_0(1), \ldots, l_0(6)) = ((1,6),(2,2),(3,3),(4,6),(5,6),(6,6))$ *and* $(l_1(1), \ldots, l_1(6)) = ((1,1),(2,4),(3,4),(4,4),(5,5),(6,7))$.

Our next main observation is that, given a run r, there is an order $\prec_{\ell_r} \in \{\prec_0, \prec_1\}$ such that the L-roots of the run with respect to $\prec_{\ell_r}$ coincides with the longest Lyndon word with respect to $\prec_{\ell_r}$ that starts at that position. Note that $\prec_{\ell_r}$ is defined for each run, and depends on the order between the two characters which break the periodicity of the run, i.e., the character that immediately follows the run, and the character p positions earlier, where p is the period of r.

Lemma 53. *Let* $r = (i, j, p)$ *be an arbitrary run in string* T*, and let* $\ell_r \in \{0, 1\}$ *be such that* $\hat{T}[j+1] \prec_{\ell_r} \hat{T}[j+1-p]$. *Then, any L-root* $\lambda = [i_\lambda..j_\lambda]$ *of* r *with respect to* $\prec_{\ell_r}$ *is equal to* $l_{\ell_r}(i_\lambda)$.

Proof. Let $[i_\lambda..j_\lambda]$ be an L-root of r with respect to $\prec_{\ell_r}$. Then, the lemma follows from the definition of $\prec_{\ell_r}$ and Lemma 51 with $u = [i_\lambda..j_\lambda]$, $u^q u' = [i_\lambda..j]$, and $a = \hat{T}[j+1]$. □

Example 8. *Consider string* $T =$ `babbabbababbabbabc` *and assume* `a` $\prec_0$ `b` $\prec_0$ `c` *and* `c` $\prec_1$ `b` $\prec_1$ `a`. *For run* $r_4 = (4, 14, 5)$, $\prec_{r_4}$ *is* $\prec_1$ *because* $T[15] =$ `b` $\prec_1$ `a` $= T[10]$, *and its L-root with respect to* $\prec_1$ *is* `bbaba`, *which is also the longest Lyndon word with respect to* $\prec_1$ *that starts at position* 6. *Note that although runs* $r_1 = (1, 9, 3)$ *and* $r_7 = (9, 17, 3)$ *correspond to the same string* `babbabbab`, $\prec_{\ell_{r_1}}$ *is* $\prec_0$ *while* $\prec_{\ell_{r_7}}$ *is* $\prec_1$, *since the runs are followed by different characters (see Figure 7.1).*

From Lemmas 52 and 53, we can show that for any two runs, their L-roots with respect to the orders specified in Lemma 53, i.e., L-roots which correspond to the longest Lyndon word starting at that position, cannot start at the same position except possibly at the beginning of the run. More precisely, for any run $r = (i, j, p)$ of T, let $B_r = \{\lambda = [i_\lambda..j_\lambda] \mid \lambda$ is an L-root of r with respect to $\prec_{\ell_r}$, $i_\lambda \neq i\}$, where $\ell_r \in \{0, 1\}$ is such that $\hat{T}[j+1] \prec_{\ell_r} \hat{T}[j+1-p]$, i.e., B_r is the set of all L-roots $[i_\lambda..j_\lambda]$ of r with respect to $\prec_{\ell_r}$ such that $[i_\lambda..j_\lambda] = l_{\ell_r}(i_\lambda)$, except for the one that starts from i if it exists. The following lemma holds.

Lemma 54. *For any two distinct runs* r *and* r' *of string* T, $Beg(B_r) \cap Beg(B_{r'})$ *is empty.*

Proof. Suppose that there exist $i \in Beg(B_r) \cap Beg(B_{r'})$, and $\lambda = [i..j_\lambda] \in B_r$ and $\lambda' = [i..j_{\lambda'}] \in B_{r'}$. Let $\ell_r \in \{0, 1\}$ be such that $\lambda = l_{\ell_r}(i)$ from the definition of B_r and Lemma 53.

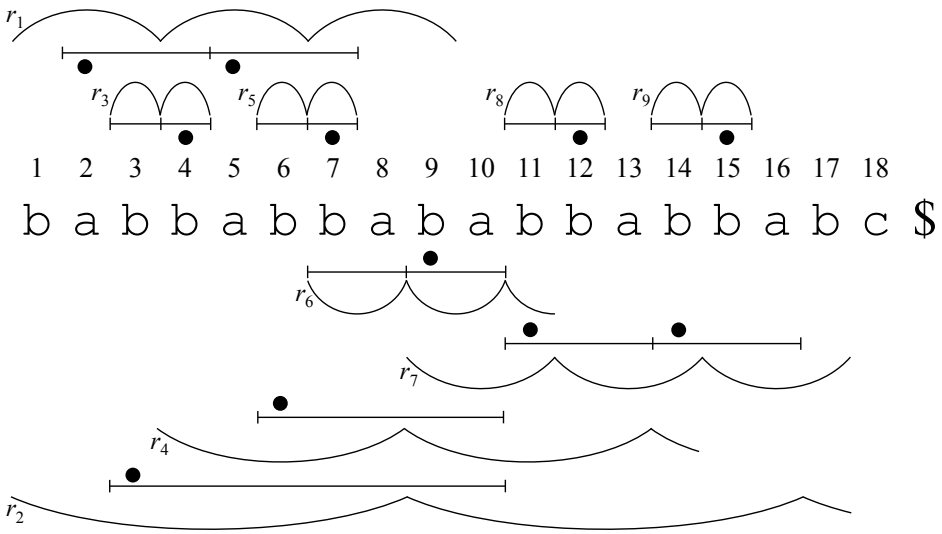

Figure 7.1: Figure showing all runs r_i $(1 \leq i \leq 9)$ in the string $T =$ babbabbababbabbabc, where the curved lines indicate the period, as well as the start and end positions of each run. Assuming a $\prec_0$ b $\prec_0$ c and c $\prec_1$ b $\prec_1$ a, the intervals indicate the L-roots of each run with respect to the lexicographic order $\prec_{\ell_{r_i}}$ as defined in the statement of Lemma 53, and thus each of them correspond to the longest Lyndon word with respect to $\prec_{\ell_{r_i}}$ that starts at that position. Here, $\ell_{r_i} = 0$ for the runs depicted above the string, and $\ell_{r_i} = 1$ for the runs depicted below the string. The black circles show the positions which are contained in $Beg(B_{r_i})$, which can be seen to be disjoint for all runs, as claimed by Lemma 54. Notice that although runs r_1 and r_7 correspond to the same string babbabbab, the lexicographic orders $\prec_{\ell_{r_1}}$ and $\prec_{\ell_{r_7}}$ which define B_{r_1} and B_{r_7} are different, since they are followed by different characters.

Since $\lambda \neq \lambda'$, $\lambda' = l_{\overline{\ell_r}}(i)$. By Lemma 52, either λ or λ' is $[i..i]$. Assume w.l.o.g. that $\lambda = [i..i]$ and $j_{\lambda'} > i$. Since $T[i..j_{\lambda'}]$ is a Lyndon word, $T[i] \neq T[j_{\lambda'}]$. By the definition of B_r and $B_{r'}$, the beginning positions of runs r and r' are both less than i, which implies $T[i-1] = T[i]$ (due to r) and $T[i-1] = T[j_{\lambda'}]$ (due to r'). Hence we get $T[i] = T[i-1] = T[j_{\lambda'}]$, a contradiction. □

Lemma 54 shows that each run r can be associated with a disjoint set of positions $Beg(B_r)$ (see Figure 7.1). Note that $|Beg(B_r)| = |B_r| \geq \lfloor e_r - 1 \rfloor \geq 1$, since the exponent e_r of any run is at least 2. Also, since $1 \notin Beg(B_r)$ for any run r, $\sum_{r \in Runs(T)} |B_r| = \sum_{r \in Runs(T)} |Beg(B_r)| \leq |T| - 1$ holds. Therefore, we obtain the following results.

Theorem 11. $\rho(N) < N$.

Proof. Consider string T of length N. Since $|B_r| \geq 1$ for any $r \in Runs(T)$, it follows from Lemma 54 that $|Runs(T)| \leq \sum_{r \in Runs(T)} |B_r| \leq N - 1$. □

Theorem 12. $\sigma(N) < 3N - 3$.

Proof. Consider string T of length N. Since $|B_r| \geq \lfloor e_r - 1 \rfloor > e_r - 2$ for any $r \in Runs(T)$, it follows from Lemma 54 that $\sum_{r \in Runs(T)}(e_r - 2) < \sum_{r \in Runs(T)} \lfloor e_r - 1 \rfloor \leq \sum_{r \in Runs(T)} |B_r| \leq N - 1$. Using $|Runs(T)| \leq N - 1$ from Theorem 11, we get $\sigma(N) = \sum_{r \in Runs(T)} e_r < N + 2|Runs(T)| - 1 \leq 3N - 3$. □

7.2.1 Higher exponent runs

Let $Runs_k(T)$ denote the set of runs of string T with exponent at least k, $\rho_k(N)$ the maximum number of runs with exponent at least k in a string of length N, and $\sigma_k(N)$ the maximum sum of exponents of runs with exponent at least k in a string of length N. Crochemore et al. [24] have shown a bound of $2.5N$ for $\sigma_3(N)$. Below, we prove a tighter bound, and show bounds for general integer k as well.

Theorem 13. *For any integer $k \geq 2$, $\rho_k(N) < N/(k-1)$, $\sigma_k(N) < N(k+1)/(k-1)$.*

Proof. Notice that for any run r with exponent at least k, $|B_r| \geq \lfloor e_r - 1 \rfloor \geq k - 1$, since k is an integer and $e_r \geq k$. Therefore, $|Runs_k(T)| \leq \sum_{r \in Runs_k(T)} |B_r|/(k-1) < N/(k-1)$. Also, $\sum_{r \in Runs_k(T)} e_r = \sum_{r \in Runs_k(T)}(e_r - 2) + 2|Runs_k(T)| < \sum_{r \in Runs_k(T)} |B_r| + 2N/(k-1) < N + 2N/(k-1) = N(k+1)/(k-1)$. □

7.2.2 Runs with d distinct symbols

Let $\rho(N, d)$ denote the maximum number of runs in a string of length N that contains exactly d distinct symbols. We prove the following bounds conjectured in [28].

Theorem 14. $\rho(N, d) \leq N - d$. *Furthermore, if $N > 2d$, then $\rho(N, d) \leq N - d - 1$.*

Proof. Let $\Sigma = \{c_1, \ldots, c_d\}$. First, we show $\rho(N, d) \leq N - d$. For any character $c_k \in \Sigma$, let i_k denote its last occurrence, i.e. $i_k = \max\{i \mid T[i] = c_k, 1 \leq i \leq N\}$. Choose the pair of total orders $\prec_0, \prec_1$ on Σ, so that for any $1 \leq k, k' \leq d$, $c_{k'} \prec_0 c_k \Leftrightarrow c_k \prec_1 c_{k'} \Leftrightarrow i_k < i_{k'}$. Also, let $i'_k = \min\{i \leq i_k \mid T[i..i_k] = c_k^{i_k - i + 1}\}$. Then, for any $1 \leq k \leq d$, since $c_k = T[i'_k] = \cdots = T[i_k]$ is smaller than any character in $\hat{T}[i_k + 1..N + 1]$ with respect to $\prec_1$, we have that $l_1(i'_k) = [i'_k..N + 1]$, and from Lemma 52, $l_0(i'_k) = [i'_k..i'_k]$. Since $\hat{T}[i'_k..N + 1]$ includes the symbol \$ which does not occur elsewhere in $\hat{T}$, $[i'_k..N + 1]$ cannot be an L-root

of a run. On the other hand, if $[i'_k..i'_k]$ is an L-root of some run, then by definition of i'_k, the run must start at i'_k. Therefore, neither $l_0(i'_k)$ nor $l_1(i'_k)$ can be included in $\cup_{r\in Runs(T)}B_r$ and thus, $i'_k \notin \cup_{r\in Runs(T)}Beg(B_r)$. Noticing that $T[i'_k] = c_k$, we have that i'_k is different for each $1 \leq k \leq d$, and therefore, $\rho(N, d) \leq N - d$.

Next, we prove $\rho(N, d) \leq N - d - 1$ for $N > 2d$. Since $1 \notin \cup_{r\in Runs(T)}Beg(B_r)$, if $i'_k > 1$ for all k, then $Runs(T) \leq N - d - 1$. Therefore, we can assume $i'_1 = 1$, which means that $T[1..i_1] = c_1^{i_1}$, and $T[i_1 + 1..N]$ does not contain an occurrence of c_1. Thus, any position in $T[1..i_1]$ can only be part of a single run $(1, i_1, 1)$ if $i_1 > 1$, or of none if $i_1 = 1$. If $i_1 > 1$, we have from the first statement that $Runs(T) \leq 1 + \rho(N - i_1, d - 1) \leq 1 + (N - i_1) - (d - 1) = N - d - (i_1 - 2)$. Since $Runs(T) \leq N - d - 1$ for $i_1 \geq 3$, we assume that $i_1 \leq 2$. We prove the statement by induction on d. For $d = 1$, we have that $\rho(N, 1) \leq 1$, and thus $\rho(N, 1) \leq N-d-1$ for any $N > 2$, and the statement holds. Suppose the statement holds for any $d' < d$, i.e., for any $d' < d$, if $N > 2d'$ then $\rho(N, d') \leq N-d'-1$. If $i_1 = 1$, then, since $(N-1) > 2(d-1)$, we have $Runs(T) \leq \rho(N-1, d-1) \leq (N-1)-(d-1)-1 \leq N-d-1$. If $i_1 = 2$, then, again since $(N-2) > 2(d-1)$, we have $Runs(T) \leq 1+\rho(N-2, d-1) \leq (N-2)-(d-1) \leq N-d-1$. Thus, the statement holds. □

This leads to a slightly better bound of $\rho(N)$ compared to Theorem 11, i.e., $\rho(N) \leq N - 3$ for $N > 4$, since $\rho(N, 1) \leq 1$.

7.3 New Linear-Time Algorithm for Computing All Runs

In this section, we describe our new linear-time algorithm for computing all runs in a given string T of length N. Note that an $\Omega(N \log N)$ time lower bound exists for general unordered alphabets, i.e., any algorithm that is based on character equality comparisons [74]. It is an open problem whether all runs can be computed in linear time for general ordered alphabets [8, 69]. Here, we assume an integer alphabet, i.e. $\Sigma = \{1, ..., N^c\}$ for some constant c. Let $L = \{l_\ell(i) \mid \ell \in \{0, 1\}, 1 \leq i \leq N\}$. From Lemma 53, we know that for any run r, L contains an L-root of r. Our new algorithm (1) computes the set L in linear time, and (2) for each element $l_\ell(i) \in L$, checks if it is equal to $\arg_{[i..j]\in B_r} \min i$ for some run, and if so determine the run, in constant time, therefore achieving linear time. Below are the algorithmic tools used in our algorithm.

Definition 14 (Suffix Array/Inverse Suffix Array [75])**.** *The suffix array $SA_T[1..N]$ of a string T of length N, is an array of integers such that $SA_T[i] = j$ indicates that $T[j..N]$ is the*

lexicographically ith smallest suffix of T. The inverse suffix array $ISA_T[1..N]$ is an array of integers such that $ISA_T[SA_T[i]] = i$.

Theorem 15 (Suffix Array/Inverse Suffix Array [63, 65, 62])**.** *The suffix array and inverse suffix array of a string over an integer alphabet can be computed in linear time.*

Theorem 16 (Longest Common Extension (LCE) Query (e.g., [36]))**.** *A string T over an integer alphabet can be preprocessed in linear time, so that for any $1 \leq i \leq j \leq |T|$, $|lcp(T[i..|T|], T[j..|T|])|$ can be answered in constant time.*

7.3.1 Linear-time computation of $l_\ell(i)$

For a string T of length N and $\ell \in \{0, 1\}$, Algorithm 5 shows a pseudo-code of a linear-time algorithm that computes $l_\ell(i)$ for all $1 \leq i \leq N$ in decreasing order of i by a right-to-left scan on T.

For each i, the algorithm computes the value $E_\ell[i]$. From Lemma 1, it is easy to see that at the end of each loop for i in the algorithm, $E_\ell[i..N]$ encodes a lexicographically non-increasing list of Lyndon words that decomposes $\hat{T}[i..N+1]$, i.e., the Lyndon factorization of $\hat{T}[i..N+1]$; the first element is $\hat{T}[i..E_\ell[i]]$, and the remaining elements are recursively defined for the suffix $\hat{T}[E_\ell[i]+1..N+1]$. From Lemma 2, it is clear that for any i, $\hat{T}[i..E_\ell[i]]$ is $l_\ell(i)$, i.e., the longest Lyndon word that starts at position i.

The lexicographic comparison of Line 5 can be performed in constant time by an LCE query and a single character comparison. We also note that it can be performed in constant time by utilizing $ISA_{\hat{T}}$, i.e., the lexicographic order of the suffix of $\hat{T}$ starting at the same position. Consider a Lyndon word λ_0 starting at position i, and the decomposed Lyndon factorization $\lambda_1 \cdots \lambda_m$ of $\hat{T}[i_v..N+1]$ s.t. $\lambda_i \succeq \lambda_{i+1}$ for any i, where $i_v = i + |\lambda_0|$. If $\lambda_0 \prec \lambda_1$, then, $\lambda = \lambda_0\lambda_1$ is a Lyndon word from Lemma 1. Therefore, $\lambda[1..|\lambda_1|] \prec \lambda_1$ and thus $\lambda_0 \cdots \lambda_m \prec \lambda_1 \cdots \lambda_m$ ($ISA_{\hat{T}}[i] < ISA_{\hat{T}}[i_v]$). If $\lambda_1 \preceq \lambda_0$, then $\lambda_0 \cdots \lambda_m$ is a Lyndon factorization of $\hat{T}[i..N+1]$. It follows from Lemma 51 that $\lambda_1 \cdots \lambda_m \prec \lambda_0 \cdots \lambda_m$ ($ISA_{\hat{T}}[i_v] < ISA_{\hat{T}}[i]$), since λ_0 must be the longest Lyndon prefix of $\hat{T}[i..N+1]$. Therefore $\lambda_0 \prec \lambda_1 \iff ISA[i] < ISA[i_v]$.

We note that the intervals $[i..E_\ell[i]]$ with each update of $E_\ell[i]$ on line 6 during the algorithm correspond to internal nodes of what is called the Lyndon tree [4], described in Section 7.4. Hohlweg and Reutenauer [53] showed that the Lyndon tree can be constructed in linear time given ISA, by showing that the Cartesian tree [90, 40] of the subarray $ISA[2..N]$ coincides with

Algorithm 5: Computing $l_\ell(i) = [i..E_\ell[i]]$ in linear time for all $1 \leq i \leq N$.

Input: String T of length N

```
1 for i = 1 to N + 1 do E_ℓ[i] ← i;                          // End positions of l_ℓ(i)
2 for i = N downto 1 do
3     while E_ℓ[i] < N + 1 do
4         j ← E_ℓ[i] + 1;
5         if not T̂[i..E_ℓ[i]] ≺_ℓ T̂[j..E_ℓ[j]] then exit while loop ;   // O(1) from
          ISA_T̂[i], ISA_T̂[j]
6         E_ℓ[i] ← E_ℓ[j] ;          // T̂[i..E_ℓ[i]]T̂[j..E_ℓ[j]] = T̂[i..E_ℓ[j]] is Lyndon
          w.r.t.≺_ℓ
7     l_ℓ(i) ← [i..E_ℓ[i]];
```

the internal nodes of the Lyndon tree. Algorithm 5 is, in essence, an implementation of the same idea.

7.3.2 Computing all runs of T from $l_\ell(i)$

Consider a candidate interval $l_\ell(i) = [i..j] \in L$. Let $T[i'..i-1]$ be the longest common suffix of $T[1..i-1]$ and $T[1..j]$, and let $T[j+1..j']$ be the longest common prefix of $T[i..N]$ and $T[j+1..N]$. It is easy to see that $[i..j] = \arg_{[i''..j'']\in B_r} \min i''$ of run $r = (i', j', p)$, if and only if $p = j - i + 1$, $|T[i'..j']| \geq 2p$, and $i' < i \leq i' + p$. Using Theorem 16, we can compute j' in constant time per query and linear-time preprocessing. If we consider LCE queries on the reverse string, we can query the length of the longest common suffix between two prefixes of T. Thus, i' can also be computed in constant time per query and linear-time preprocessing.

7.4 Runs and Lyndon Trees

In this section, we characterize runs in strings using Lyndon trees.

Definition 15 (Standard Factorization [14, 72])**.** *The* standard factorization *of a Lyndon word T with $|T| \geq 2$ is an ordered pair (u, v) of Lyndon words u, v such that $T = uv$ and v is the lexicographically smallest proper suffix of T.*

It can be shown that for any Lyndon word T longer than 1, the standard factorization (u, v) of T always exists. The Lyndon tree of a Lyndon word T, defined below, is the full binary tree defined by recursive standard factorization of T.

Definition 16 (Lyndon Tree [4])**.** *The* Lyndon tree *of a Lyndon word T, denoted $LTree(T)$, is an ordered full binary tree defined recursively as follows:*

- *if $|T| = 1$, then $LTree(T)$ consists of a single node labeled by T;*
- *if $|T| \geq 2$, then the root of $LTree(T)$, labeled by T, has left child $LTree(u)$ and right child $LTree(v)$, where (u, v) is the standard factorization of T.*

Each node α in $LTree(T)$ can be represented by an interval $[i..j]$ $(1 \leq i \leq j \leq |T|)$ of T, and we say that the interval $[i..j]$ corresponds to a node in $LTree(T)$. Let $lca([i..j])$ denote the lowest node in $LTree(T)$ containing all leaves corresponding to positions in $[i..j]$ in its subtree, or equivalently, the lowest common ancestor of leaves at position i and j. Note that an interval $[i..j]$ corresponds to a node in the Lyndon tree, iff $lca([i..j]) = [i..j]$. Figure 7.2 shows an example of a Lyndon tree for the Lyndon word `aababaababb`.

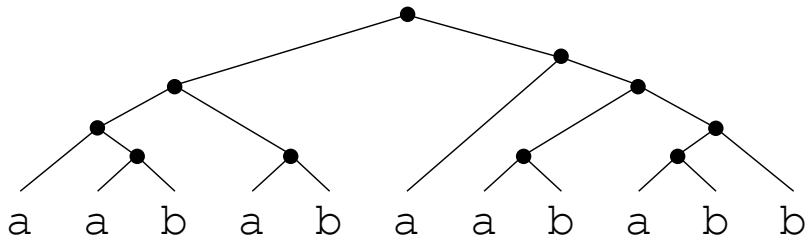

Figure 7.2: The Lyndon tree for the Lyndon word `aababaababb` with respect to order `a` $\prec$ `b`.

We first show a simple yet powerful lemma characterizing Lyndon substrings of a Lyndon word, in terms of the Lyndon tree.

Lemma 55. *Let T be a Lyndon word. For any interval $[i..j]$, if $T[i..j]$ is a Lyndon word, then the node $\alpha = lca([i..j]) = [i_\alpha..j_\alpha]$ in $LTree(T)$ satisfies $i_\alpha = i \leq j \leq j_\alpha$.*

Proof. If $i = j$, then α is a leaf node and corresponds to $[i..i]$. If $i < j$, α is an internal node. Let $\beta = [i_\alpha...j']$ and $\gamma = [j' + 1...j_\alpha]$ respectively be the left and right children of α. By definition of lca, we have that $i_\alpha \leq i \leq j' \leq j' + 1 \leq j \leq j_\alpha$, and for some strings $u, v \in \Sigma^*$ and $x, y \in \Sigma^+$, we have that $T[i..j] = xy$, $T[i_\alpha..j'] = ux$, $T[j' + 1..j_\alpha] = yv$, and (ux, yv) is the standard factorization of $T[i_\alpha..j_\alpha] = uxyv$. Since xy is a Lyndon word, $xy \prec y$, and therefore $xyv \prec yv$. However, if $u \neq \varepsilon$, this contradicts that yv is the lexicographically smallest proper suffix of $uxyv$. Thus, u must be empty, and $i_\alpha = i$. □

A node is called a *left node* (resp. *right node*) if it is the left (resp. right) child of its parent. The next lemma is a simple consequence of Lemma 55 yet also gives an important characterization.

Lemma 56. *Let T be a Lyndon word. For any interval $[i..j]$ except for $[1..|T|]$, $[i..j]$ corresponds to a right node of the Lyndon tree iff $T[i..j]$ is the longest Lyndon word that starts from i.*

Proof. Suppose $T[i..j]$ is the longest Lyndon word that starts from i. For any $j' > j$, $T[i..j']$ is not a Lyndon word and thus $[i..j']$ cannot be a node in $LTree(T)$. Hence, it is clear from Lemma 55 that $lca([i..j]) = [i..j]$ and it is a right node. On the other hand, suppose $T[i..j]$ is not the longest Lyndon word that starts from i. Then, there exists a Lyndon word $T[i..j']$ for some $j' > j$. Since there is a node $lca([i..j']) = [i..j'']$ with $j'' \geq j' > j$ due to Lemma 55, it is easy to see that $[i..j]$ cannot be a right node. Note that $[i..j]$ may not correspond to a node, but if it does, it must be a left node. □

Now consider again the two total orders $\prec_0$, $\prec_1$ on Σ. Let T be an arbitrary string of length N and let $T_0 = \#_0 T\$$ and $T_1 = \#_1 T\$$ where $\#_0, \#_1 \notin \Sigma \cup \{\$\}$ are special characters that are respectively lexicographically smaller than any other character in $\Sigma \cup \{\$\}$, with respect to $\prec_0$ and $\prec_1$. Thus, $\#_0 \prec_0 \$ \prec_0 a$ and $\#_1 \prec_1 a \prec_1 \$$ for any $a \in \Sigma$. For technical reasons, we assume that positions in T_0 and T_1 will start from 0 rather than 1, in order to keep in sync with positions in T, i.e., so that for any $1 \leq i \leq |T|$, $T[i] = T_0[i] = T_1[i]$. Note that T_ℓ ($\ell \in \{0, 1\}$) is a Lyndon word with respect to $\prec_\ell$, and let $LTree_\ell(T)$ denote the Lyndon tree of T_ℓ, with respect to $\prec_\ell$. Also, $lca_\ell([i..j])$ will denote $lca([i..j])$ in $LTree_\ell(T)$.

Lemma 57. *Given a string T of length N, $LTree_0(T)$ and $LTree_1(T)$ can be constructed in $O(N)$ time and space.*

The next lemma immediately follows from Lemmas 53 and 56.

Lemma 58. *Let $r = (i, j, p)$ be an arbitrary run in string T of length N, and let $\ell_r \in \{0, 1\}$ be such that $T_{\ell_r}[j+1] \prec_{\ell_r} T_{\ell_r}[j+1-p]$. Then, any L-root of r with respect to $\prec_{\ell_r}$ is a right node of $LTree_{\ell_r}(T)$.*

In light of Lemma 58, we have a structural view of the runs in a string T by two trees $LTree_0(T)$ and $LTree_1(T)$. This can be a powerful tool for algorithms and data structures employing subrepetitions in a string. In the next subsection, we exhibit an application to a data structure for 2-Period Queries.

7.4.1 Application to 2-period queries

The 2-Period Query problem is to preprocess a string T to support the following queries efficiently: Given any interval $[i..j]$ of T, return the smallest period p of $T[i..j]$ with $p \leq (j-i+1)/2$, if such exists. The 2-Period Query problem is tightly related to the runs in T: Let $exrun([i..j])$ denote a run (i', j', p') such that $i' \leq i, j \leq j'$ and $p' \leq (j-i+1)/2$ if such exists. Note that due to the periodicity lemma [35], such a run uniquely exists iff $p \leq (j-i+1)/2$ and $p' = p$. Therefore, a 2-Period Query with interval $[i..j]$ reduces to searching for $exrun(i, j)$.

An optimal solution to the 2-Period Query problem was recently proposed in [67] as a by-product of their algorithm for internal pattern matching. Their solution introduces a notion of k-runs in which a run is distributed to one or more sets of runs satisfying some conditions. We propose another optimal yet simpler solution using Lyndon trees.

Theorem 17. *For any string T of length N, there is a data structure of $O(N)$ space that supports 2-Period Queries in $O(1)$ time. The data structure can be built in $O(N)$ time.*

Proof. We construct $LTree_0(T)$ and $LTree_1(T)$ in $O(N)$ time and space using Lemma 57. At the same time, we compute the runs in T and label every right node that corresponds to an L-root of a run with the information of the run. This can be done in $O(N)$ total time by using a similar procedure as described in Section 7.3.1, by simply omitting the condition $i' < i \leq i' + p$. We also augment these trees with data structures in $O(N)$ time and space so that lowest common ancestor (LCA) queries can be answered in $O(1)$ time [6]. Given a query with interval $[i..j]$, our algorithm computes $\alpha_0 = lca_0([i..\lceil (i+j)/2 \rceil])$ and $\alpha_1 = lca_1([i..\lceil (i+j)/2 \rceil])$, and check their right children. See Algorithm 6 for a pseudo-code.

We show the correctness of our solution below. Suppose that $r = exrun([i..j]) = (i', j', p)$ exists. Let $\ell \in \{0, 1\}$ with $T_\ell[j'+1] \prec_\ell T_\ell[j'+1-p]$. Since the period p of r is at most $\lfloor (j-i+1)/2 \rfloor$, we have that $i \leq \lceil (i+j)/2 \rceil - p < \lceil (i+j)/2 \rceil + p - 1 \leq j$. Thus, there exists an L-root λ of r with respect to $\prec_\ell$ that contains position $\lceil (i+j)/2 \rceil$. By Lemma 58, λ is a right node. Moreover, α_ℓ is an ancestor of λ since λ does not contain position i while both contain position $\lceil (i+j)/2 \rceil$. We claim that the right child of α_ℓ is λ. Assume to the contrary that the right child $\beta = [i_\beta..j_\beta]$ of α_ℓ is not $\lambda = [i_\lambda..j_\lambda]$. By definition of α_ℓ, β and λ, we have that β must be an ancestor of λ and $i' \leq i < i_\beta < i_\lambda$ since λ is a right node. Also, it must be that $j \leq j' < j_\beta$ since otherwise, $T[i_\beta..j_\beta]$ would have period $p < |[i_\beta..j_\beta]|$ due to run r, contradicting that it is a Lyndon word. However, by the definition of ℓ, this implies that $T[i_\lambda..i_\beta] \prec_\ell T[i_\beta..j_\beta]$, still contradicting that $T[i_\beta..j_\beta]$ is a Lyndon word. Therefore, if

Algorithm 6: $O(N)$ time preprocessing and $O(1)$ time query for 2-Period Queries

Preprocessing Input: String T of length N

foreach $\ell \in \{0, 1\}$ **do**
  Compute $LTree_\ell(T)$ and preprocess for lca_ℓ queries;
  foreach *right node* α *in* $LTree_\ell(T)$ **do**
    If α is an L-root of some run $r = (i', j', p)$, label α with r;

Query Input: Interval $[i..j]$

foreach $\ell \in \{0, 1\}$ **do**
  $\alpha_\ell \leftarrow lca_\ell([i..\lceil (i + j)/2 \rceil])$; $\beta_\ell \leftarrow$ right child of α_ℓ;
  if β_ℓ *is labeled with run* $r = (i', j', p)$ **and** $i' \leq i \leq j \leq j'$ **and** $p \leq (j - i + 1)/2$ **then**
    return p

return nil

$exrun([i..j])$ exists, at least one of the right children of α_0 and α_1 is an L-root of $exrun([i..j])$, and can be found in constant time. □

7.5 Conclusion

We showed a remarkably simple proof to the 15 year-old runs conjecture, by discovering a beautiful connection between the L-roots of runs and the longest Lyndon word starting at each position of the string. We also showed a bound of $\sigma(N) < 3N$ for the maximum sum of exponents of runs in a string of length N, improving on the previous best bound of $4.1N$ [24], as well as improved analyses on related problems. We also proposed a simple linear-time algorithm for computing all the runs in a string. Furthermore, realizing that the longest Lyndon word starting at each position of the string corresponds to a right node in the Lyndon tree, we showed a simple optimal solution to the 2-Period Query problem.

The characterizations of runs in terms of Lyndon words as shown in this paper significantly improves our understanding of how runs can occur in strings. A remaining question is the exact value of $\lim_{N\to\infty} \rho(N)/N$ for general strings, which is known to exist but is never reached [49]. We note that this has been solved exactly only for a specific class of strings, namely, it has been shown that the maximum number of runs in standard Sturmian words is at most $0.8N$, and that there exists an infinite sequence of standard Sturmian words with strictly increasing lengths such that the limit of the ratio between the number of runs in it and its length equals 0.8 [5].

Chapter 8

Conclusions

In this thesis, we studied combinatorial properties on Lyndon words and developed algorithms on Lyndon words.

In Chapter 3, we considered the reverse-engineering problems on Lyndon factorizations. We then presented a linear time algorithm to compute a string T such that a given sequence of pairs of positive integers corresponds to the Lyndon factorization of T. We also presented an algorithm to solve the same problem in compact representation. Moreover, we showed a linear time algorithm to compute only the smallest size of alphabet. For an enumeration problem, we proposed a compact representation of all valid strings and an efficient algorithm to compute the representation.

In Chapter 4, we developed a Lyndon factorization algorithm for a grammar compressed string. The algorithm presented in the chapter computes the smallest suffix of a string. If we want the Lyndon factorization of a string, we only have to use the algorithm recursively. Our algorithm is faster than Duval's algorithm (for an uncompressed string) when the input string is highly compressed.

In Chapter 5, we developed Lyndon factorization algorithms for compressed strings again. Firstly, we presented an algorithm to compute the Lyndon factorization of a grammar compressed string. This algorithm is faster than the algorithm presented in Chapter 4. Secondly, we presented an algorithm to compute the Lyndon factorization of an LZ78 compressed string. This algorithm works in $O(s \log s)$ time where s is the size of the LZ78 factorization. In this chapter, we showed an independent result that the size of the Lyndon factorization, for any string, is a lower bound of the size of the smallest grammar. This result led to the problem which was considered in Chapter 6.

In Chapter 6, we studied relations between Lyndon factorizations and non-overlapping LZ77

factorizations. We showed that the number of Lyndon factors cannot be more than $2t$ where t is the number of LZ77 phrases. This result is the first direct connection of these two factorizations. The result improves significantly a previous, indirect bound given in Chapter 5. We also showed that there are strings with $t + \Theta(\sqrt{t})$ Lyndon factors.

In Chapter 7, we showed the runs theorem and proposed a new linear time algorithm to compute all runs in a string. We discovered and established a connection between the L-roots of runs and the longest Lyndon word starting at each position of the string. Based on this novel observation, we gave an affirmative answer to the runs conjecture. We also proved some new bounds for related problems about runs. We gave a novel, conceptually simple linear-time algorithm for computing all runs contained in a string, based on the proof of $\rho(N) < N$. Our algorithm is the first linear-time algorithm which does *not* rely on the Lempel-Ziv parsing of the string.

Bibliography

[1] A. Apostolico and M. Crochemore. Fast parallel Lyndon factorization with applications. *Mathematical Systems Theory*, 28(2):89–108, 1995.

[2] H. Bannai, P. Gawrychowski, S. Inenaga, and M. Takeda. Converting SLP to LZ78 in almost linear time. In *CPM*, pages 38–49, 2013.

[3] H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda. Inferring strings from graphs and arrays. In *Proc. MFCS 2003*, volume 2747 of *LNCS*, pages 208–217, 2003.

[4] H. Barcelo. On the action of the symmetric group on the free Lie algebra and the partition lattice. *Journal of Combinatorial Theory, Series A*, 55(1):93–129, 1990.

[5] P. Baturo, M. Piatkowski, and W. Rytter. The maximal number of runs in standard Sturmian words. *Electr. J. Comb.*, page P13, 2013.

[6] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proc. LATIN 2000*, pages 88–94, 2000.

[7] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[8] D. Breslauer. *Efficient String Algorithmics*. PhD thesis, Columbia University, 1992.

[9] S. Brlek, J.-O. Lachaud, X. Provençal, and C. Reutenauer. Lyndon + Christoffel = digitally convex. *Pattern Recognition*, 42(10):2239–2246, 2009.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[11] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[12] M. Chemillier. Periodic musical sequences and Lyndon words. *Soft Comput.*, 8(9):611–616, 2004.

[13] G. Chen, S. J. Puglisi, and W. F. Smyth. Fast and practical algorithms for computing all the runs in a string. In *Proc. CPM*, pages 307–315, 2007.

[14] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus. IV. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[15] J. Clément, M. Crochemore, and G. Rindone. Reverse engineering prefix tables. In *Proc. STACS 2009*, pages 289–300, 2009.

[16] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[17] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.

[18] M. Crochemore and L. Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, pages 796–807, 2008.

[19] M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227–5235, 2009.

[20] M. Crochemore, L. Ilie, and L. Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011.

[21] M. Crochemore, C. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. The maximal number of cubic runs in a word. *Journal of Computer and System Sciences*, 78(6):1828–1836, 2012.

[22] M. Crochemore, C. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521(13):29–41, 2014.

[23] M. Crochemore, C. Iliopoulos, S. Pissis, and G. Tischler. Cover array string reconstruction. In *Proc. CPM 2010*, volume 6129 of *LNCS*, pages 251–259, 2010.

[24] M. Crochemore, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. On the maximal sum of exponents of runs in a string. *Journal of Discrete Algorithms*, 14:29–36, 2012.

[25] M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.

[26] J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Parallel RAM algorithms for factorizing words. *Theor. Comput. Sci.*, 127(1):53–67, 1994.

[27] O. Delgrange and E. Rivals. STAR: an algorithm to search for tandem approximate repeats. *Bioinformatics*, 20(16):2812–2820, 2004.

[28] A. Deza and F. Franek. A d-step approach to the maximum number of distinct squares and runs in strings. *Discrete Applied Mathematics*, 163(3):268–274, 2014.

[29] A. Deza and F. Franek. d-step method and the number of runs. Technical Report AdvOL2015/01, Advanced Optimization Laboratory, Department of Computing and Software, McMaster University, 2015.

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

[31] J.-P. Duval. Génération d'une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornée. *Theor. Comput. Sci.*, 60:255–283, 1988.

[32] J.-P. Duval, T. Lecroq, and A. Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005.

[33] J.-P. Duval, T. Lecroq, and A. Lefebvre. Efficient validation and construction of border arrays and validation of string matching automata. *RAIRO - Theoretical Informatics and Applications*, 43(2):281–297, 2009.

[34] J.-P. Duval and A. Lefebvre. Words over an ordered alphabet and suffix permutations. *Theoretical Informatics and Applications*, 36:249–259, 2002.

[35] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[36] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. CPM*, pages 36–48, 2006.

[37] F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a border array in linear time. *J. Comb. Math. and Comb. Comp.*, 42:223–236, 2002.

[38] 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*, 1(195):195–203, 2008.

[39] H. Fredricksen and J. Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978.

[40] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143, 1984.

[41] L. Gasieniec and W. Rytter. Almost optimal fully LZW-compressed pattern matching. In *Data Compression Conference, DCC 1999, Snowbird, Utah, USA, March 29-31, 1999.*, pages 316–325, 1999.

[42] P. Gawrychowski. Optimal pattern matching in LZW compressed strings. In *Proc. SODA 2011*, pages 362–372, 2011.

[43] P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In *Proc. ESA 2011*, pages 421–432, 2011.

[44] P. Gawrychowski. Faster algorithm for computing the edit distance between SLP-compressed strings. In *Proc. SPIRE 2012*, pages 229–236, 2012.

[45] P. Gawrychowski. Tying up the loose ends in fully LZW-compressed pattern matching. In *STACS*, pages 624–635, 2012.

[46] P. Gawrychowski, A. Jeż, and L. Jez. Validating the Knuth-Morris-Pratt failure function, fast and online. In *Proc. CSR 2010*, volume 6072 of *LNCS*, pages 132–143, 2010.

[47] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.

[48] M. Giraud. Not so many runs in strings. In *Proc. LATA*, pages 232–239, 2008.

[49] M. Giraud. Asymptotic behavior of the numbers of runs and microruns. *Information and Computation*, 207(11):1221–1228, 2009.

[50] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Fast q-gram mining on SLP compressed strings. *Journal of Discrete Algorithms*, 18:89–99, 2013.

[51] J. He, H. Liang, and G. Yang. Reversing longest previous factor tables is hard. In *Proc. WADS 2011*, pages 488–499, 2011.

[52] D. Hermelin, G. M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *Proc. STACS 2009*, pages 529–540, 2009.

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

[54] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

[55] T. I. Longest common extensions with recompression. *CoRR*, abs/1611.05359, 2016.

[56] T. I, S. Inenaga, H. Bannai, and M. Takeda. Counting and verifying maximal palindromes. In *Proc. SPIRE 2010*, volume 6393 of *LNCS*, pages 135–146, 2010.

[57] T. I, S. Inenaga, H. Bannai, and M. Takeda. Inferring strings from suffix trees and links on a binary alphabet. In *Proc. PSC 2011*, pages 121–130, 2011.

[58] T. I, S. Inenaga, H. Bannai, and M. Takeda. Verifying and enumerating parameterized border arrays. *Theor. Comput. Sci.*, 412(50), 2011.

[59] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015.

[60] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Efficient lyndon factorization of grammar compressed text. In *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, pages 153–164, 2013.

[61] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theoretical Computer Science*, 656:215–224, 2016.

[62] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[63] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. CPM'03*, volume 2676 of *LNCS*, pages 186–199, 2003.

[64] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[65] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2–4):143–156, 2005.

[66] T. Kociumaka, J. Radoszewski, and W. Rytter. Computing k-th Lyndon word and decoding lexicographically minimal de bruijn sequence. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, pages 202–211, 2014.

[67] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551, 2015.

[68] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS*, pages 596–604, 1999.

[69] D. Kosolobov. Computing runs on a general alphabet. *Information Processing Letters*, 116(3):241–244, 2016.

[70] M. Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC 2009*, pages 65–79, 2009.

[71] Y. Lifshits. Solving classical string problems an compressed texts. In *Combinatorial and Algorithmic Foundations of Pattern and Association Discovery, 14.05. - 19.05.2006*, 2006.

[72] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983.

[73] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77:202–215, 1954.

[74] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.

[75] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[76] W. Matsubara, A. Ishino, and A. Shinohara. Inferring strings from runs. In *Proc. PSC 2010*, pages 150–160, 2010.

[77] W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara. New lower bounds for the maximum number of runs in a string. In *Proc. PSC'08*, pages 140–145, 2008.

[78] D. Moore, W. F. Smyth, and D. Miller. Counting distinct strings. *Algorithmica*, 23(1):1–13, 1999.

[79] M. Mucha. Lyndon words and short superstrings. In *Proc. SODA'13*, pages 958–972, 2013.

[80] X. Provençal. Minimal non-convex words. *Theor. Comput. Sci.*, 412(27):3002–3009, 2011.

[81] S. J. Puglisi, J. Simpson, and W. F. Smyth. How many runs can a string contain? *Theoretical Computer Science*, 401:165–171, 2006.

[82] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[83] W. Rytter. The number of runs in a string: Improved analysis of the linear upper bound. In *Proc. STACS 2006*, volume 3884 of *LNCS*, pages 184–195, 2006.

[84] K.-B. Schürmann and J. Stoye. Counting suffix arrays and strings. *Theoretical Computer Science*, 395(2-3):220–234, 2008.

[85] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1061–1066, 2003.

[86] J. Simpson. Modified padovan words and the maximum number of. *Australasian Journal of Combinatorics*, 46:129–145, 2010.

[87] W. F. Smyth. Repetitive perhaps, but certainly not boring. *Theoretical Computer Science*, 249(2):343–355, 2000.

[88] W. F. Smyth. Computing regularities in strings: A survey. *European Journal of Combinatorics*, 34(1):3–14, 2013.

[89] W. F. Smyth. Large-scale detection of repetitions. *Phil. Trans. R. Soc. A*, 372(2016), 2014.

[90] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.

[91] T. Yamamoto, H. Bannai, S. Inenaga, and M. Takeda. Faster subsequence and don't-care pattern matching on compressed texts. In *Proc. CPM 2011*, volume 6661 of *LNCS*, pages 309–322, 2011.

[92] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–349, 1977.

[93] J. Ziv and A. Lempel. Compression of individual sequences via variable-length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.