Efficient String Algorithmics Across Alphabet Realms

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Jonas Ellert

Dortmund

2024

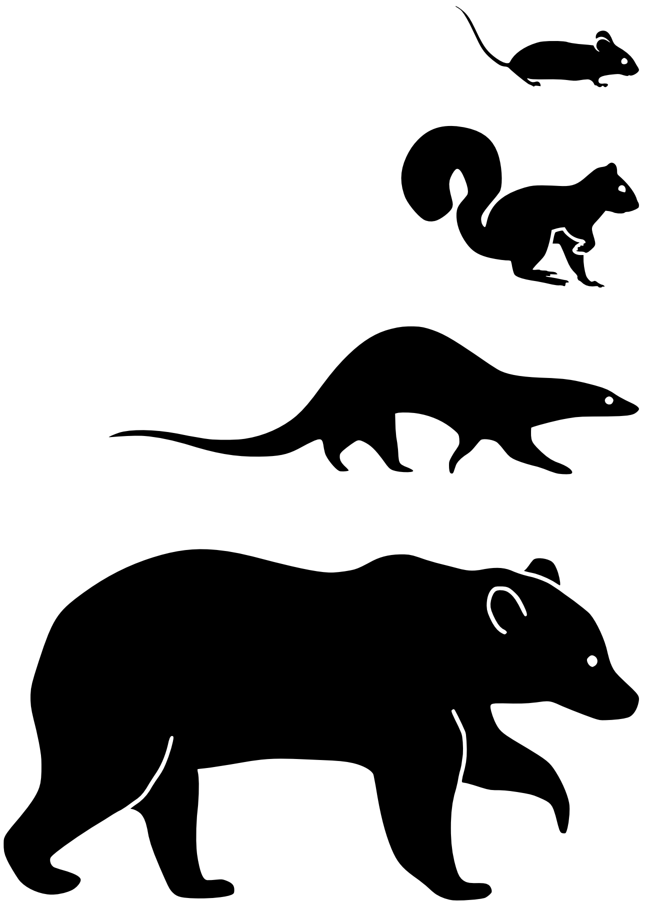
Tag der mündlichen Prüfung: 19.02.2024

Dekan: *Prof. Dr. Gernot Fink*
Fakultät für Informatik
Technische Universität Dortmund
Deutschland

Erster Gutachter: *Prof. Dr. Johannes Fischer*
Technische Universität Dortmund
Deutschland

Zweiter Gutachter: *Prof. Dr. Paweł Gawrychowski*
Uniwersytet Wrocławski
Polen

To my wife.
To my family.
To all beings, great and small.



Abstract

Stringology is a subfield of computer science dedicated to analyzing and processing sequences of symbols. It plays a crucial role in various applications, including lossless compression, information retrieval, natural language processing, and bioinformatics. Recent algorithms often assume that the strings to be processed are over polynomial integer alphabet, i.e., the symbols are from an integer range $\{0, \ldots, \sigma - 1\}$, where σ is at most polynomial in the lengths of the strings. In contrast to that, the earlier days of stringology were shaped by the weaker comparison model, in which strings can only be accessed by mere equality comparisons of symbols, or (if the symbols are totally ordered) order comparisons of symbols. Nowadays, these flavors of the comparison model are respectively referred to as *general unordered alphabet* and *general ordered alphabet.* In this dissertation, we dive into the realm of these general alphabets and show that, even though they may seem outdated and somewhat esoteric at first glance, they are certainly far from obsolete.

The main contribution are new time-wise optimal algorithms for detecting whether a given string contains a square, i.e., a substring of the form uu for some string u. We solve this problem in $\mathcal{O}(n)$ time over general ordered alphabet, and $\mathcal{O}(n \log \sigma)$ time (with a matching lower bound) over general unordered alphabet, where n is the length of the string and σ is the number of distinct symbols present in the string. This resolves two open questions that date back multiple decades. The algorithms not only detect whether the string contains a square, but actually output a complete list of the maximal periodic substrings (also known as runs), which have applications, e.g., in bioinformatics. As a stepping stone towards the solution over general ordered alphabet, we also provide the first $\mathcal{O}(n)$ time algorithm that computes the Lyndon array in this model, which is a crucial ingredient for the computation of runs. The algorithm computes the succinct $2n$ bit representation of the Lyndon array, and (apart from the space needed for input and output) it uses only $\mathcal{O}(n \log \log n / \log n)$ bits of working space. This makes it the most space efficient algorithm to date, regardless of the alphabet. For polynomial integer alphabets, we introduce the first algorithm that computes the Lyndon array in sublinear time (assuming that the alphabet is sufficiently small). On our way to the solution, we carefully dissect and analyze combinatorial properties of Lyndon words and periodic substrings, and show how to exploit them algorithmically.

The complexity analysis of the new results is quite intricate and involved. However, at least in the case of general ordered alphabet, the algorithms themselves are surprisingly simple. The simplicity is a consequence of (and hence an argument for) the general ordered alphabet. Since we have to avoid data structures and other heavy machinery that is designed for integer alphabets, we are forced to develop algorithms that are inherently more self-contained. While the main contribution lies

in the improvement of theoretical bounds, some of the new algorithms are also very easy to implement and quite fast in practice, which is a result of their simplicity.

Another strength of general alphabets is the ability to separate problems based on their time complexity, which we demonstrate with the problem of computing the Lempel-Ziv factorization (one of the most common compression schemes). Over polynomial integer alphabet, this problem is equally hard as computing runs (or detecting squares). In contrast to that, we show that computing the Lempel-Ziv factorization requires $\Omega(n \log \sigma)$ time over general ordered alphabet (a result that was previously known) and $\Omega(n\sigma)$ time over general unordered alphabet. Hence the problem is indeed harder than detecting squares. Optimal $\mathcal{O}(n \log \sigma)$ and respectively $\mathcal{O}(n\sigma)$ time algorithms can be obtained with a simple alphabet reduction, and thus we do not need dedicated algorithms for Lempel-Ziv over general alphabets.

Instead, we consider the nowadays more common setting in which the string is over integer alphabet $\{0, \ldots, \sigma - 1\}$. Under the standard assumption of a word RAM with words of width $w = \Theta(\log n)$, a length-n string can be packed in only $\mathcal{O}(n/\log_\sigma n)$ words of memory, which is sublinear in n if the alphabet is sufficiently small. Each word contains (parts of) multiple symbols, and thus word-level parallelism allows the simultaneous processing of multiple symbols. A recent trend in stringology is the design of algorithms that run in (close to) $\mathcal{O}(n/\log_\sigma n)$ time, and thus depend on the number of words occupied by the string (rather than its length). We follow this trend and show that the Lempel-Ziv factorization can be computed in $\mathcal{O}(n/\log_\sigma n + z \log^{3+\epsilon} z)$ time and $\mathcal{O}(n/\log_\sigma n)$ words of working space, where $\epsilon \in \mathbb{R}^+$ is an arbitrarily small constant, and z is the number of phrases of the factorization. This significantly improves the best previously known bounds. As part of the solution, we introduce an algorithm that computes a 3-approximate factorization (consisting of at most $3z$ phrases) in $\mathcal{O}(n/\log_\sigma n)$ time and words of working space, which is a result of independent interest. Finally, we also show new advances in the computation of the rightmost Lempel-Ziv parsing, which optimizes the encoding and hence the compression rate achieved by the factorization.

From a broader perspective, the presented results underline the importance of considering string algorithmic problems in both weaker and stronger models than the standard one (in which each symbol is stored in a separate word of a word RAM). The current trend of word-packed string algorithmics undeniably yields faster solutions, and even though it requires a stronger model of computation, it certainly reflects the nature of real-world computer architectures. However, we should not discard the older and weaker comparison-based models too quickly, as they are not only powerful theoretical tools, but also lead to fast and elegant practical solutions, even by today's standards.

Table of Contents

Chapter 1

Introduction 1

Stringology is a subfield of computer science dedicated to analyzing and processing sequences of symbols. It plays a crucial role in various applications, including lossless compression, information retrieval, natural language processing, and bioinformatics. In this dissertation, we provide lower bounds and efficient algorithms for various string algorithmic problems. Before describing the main contributions, we contextualize the content within the broader scope of stringology.

A widely accepted assumption in stringology is that strings are processed on a word RAM that operates on binary words of width w (see, e.g., [Hag98]), and each symbol of a string is represented as a bitstring of fixed length, typically no greater than w. This model is reasonably similar to real-world computer architectures, which is in alignment with the practical relevance of many string algorithmic problems. In the earlier days of stringology, algorithms often assumed the more abstract comparison model of computation, in which the algorithm can only interact with the string by comparing two of its symbols. If the algorithm merely uses equality comparisons of symbols, then we say that it works over *general unordered alphabet*. This is the case, e.g., for the famous Knuth-Morris-Pratt pattern matching algorithm [KMP77]. If, however, the symbols are totally ordered, and the algorithm accesses the string by performing order comparisons (i.e., it tests if one symbol precedes another with respect to the total order), then we say that it works over *general ordered alphabet*. This is the case, e.g., for Apostolico's parallel algorithm that tests whether a given string contains a square substring (where a square is the two-times concatenation of a shorter string, like `atat` in the string `catattack`) [Apo92].

In this dissertation, we dive into the realm of these general (ordered and unordered) alphabets. We provide new lower and upper bounds for solving classic problems in this setting. As a prime example, we present two time-wise optimal algorithms that test whether a given string contains a square, respectively over general ordered and general unordered alphabet. This resolves two open questions that date back multiple decades [ML84, Bre92].

The presented results are primarily of theoretical interest. They advance our understanding of structural elements in strings by revealing their combinatorial nature. However, the considered problems also have strong practical applications, and we complement a selection of the theoretical results with an efficient implementation. Peculiarly, some of the algorithms designed for general ordered alphabet can be implemented more efficiently than previous solutions that assume an integer alphabet. One reason for this is that algorithms for integer alphabets often rely on precomputed data structures and other heavy algorithmic machinery (e.g., the suffix array [MM93]). Such tools must be avoided by algorithms for general alphabets because in this setting they cannot be constructed efficiently. As a result, algorithms for general alphabets are often inherently simpler, more self-contained, and (in some cases) arguably more elegant. This is only one of the reasons why general alphabets, although they may seem esoteric and outdated at first glance, are certainly far from obsolete.

We also consider the more common setting of strings over integer alphabet, in which it is often assumed that a string of length n is stored in n consecutive memory words. Consequently, $\Omega(n)$ is a trivial lower bound on the time needed by any algorithm that has to access all symbols of the string. However, faster algorithms can be achieved if the string is given in *packed representation*. If the symbols are from integer alphabet $\{0, \ldots, \sigma - 1\}$ with $\sigma \ll n$, then multiple symbols can be stored in a single word, and the entire string can be packed in $\mathcal{O}(n/\log_\sigma n)$ words (which we describe in detail later). By exploiting word-level parallelism, multiple symbols can then also be processed simultaneously. This way, we obtain new algorithms that run in $\mathcal{O}(n/\log_\sigma n)$ time, matching the time needed to merely read the packed string. For example, we show how to compute an approximation of the Lempel-Ziv factorization (a popular compression scheme) in this complexity. Such word-packing results are a major recent trend in stringology, and they are motivated by real-world computer architectures. For example, most commodity processors work on 64 bit words, while real-world string data is usually stored using one byte per symbol. Hence eight symbols could be packed in one word.

One might argue that, in the last few decades, the string algorithmic landscape has been dominated by algorithms that (often implicitly) assume that each symbol is stored in a separate word of a word RAM. While the recent trend of word-packed string algorithmics offers a slightly stronger model, the older assumption of general alphabets offers a weaker model. The results presented in the dissertation underline the importance of considering not only the standard model but also the stronger and weaker ones. On one hand, the stronger models yield faster algorithms. On the other hand, the weaker comparison-based models advance our understanding of combinatorial structures in strings and sometimes lead to simple and fast algorithms.

1.1 Main Results and Overview of the Dissertation

In this section, we give a brief overview of the main contributions. This is (primarily) intended for readers that are already familiar with the field, the considered problems, and the commonly used notation. We only state the main results without providing further background information. A precise description of the problems and a brief discussion of the prevalent literature can be found in the respective chapter and part introductions.

General Alphabets In Chapter 2, we introduce basic definitions and key concepts used throughout the dissertation. We also describe and motivate the present models of computation in more detail, provide background information and literature, and discuss how they are related to the complexity of integer and comparison-based sorting. Then, we provide a trivial reduction from general alphabets to integer alphabets. Consider a length-n string that contains σ distinct symbols.

- If the string is over general ordered alphabet, then an order-isomorphic string over the integers $\{0, \ldots, \sigma - 1\}$ can be obtained in $\mathcal{O}(n \log \sigma)$ order comparisons.
- If the string is over general unordered alphabet, then an isomorphic string over the integers $\{0, \ldots, \sigma - 1\}$ can be obtained in $\mathcal{O}(n\sigma)$ equality comparisons.

Hence we do not always need dedicated algorithms for general alphabets; for some problems, we already achieve optimal time by first applying the reduction, and then solving the problem with a fast algorithm designed for integer alphabet. This is the case even for very basic problems. For example, by using the adversary method of showing lower bounds, we prove that distinguishing length-n strings that contain σ distinct symbols from length-n strings that contain $\frac{n}{2}$ distinct symbols requires $\Omega(n \log \sigma)$ order comparisons, or $\Omega(n\sigma)$ equality comparisons.

Lempel-Ziv Factorization (Part I) In Part I, we turn our attention to the Lempel-Ziv (LZ) factorization [LZ76], one of the main tools in lossless data compression. In Chapter 3, we extend the adversarial lower bounds from Chapter 2 and show that computing the LZ factorization requires $\Omega(n \log \sigma)$ order comparisons, or $\Omega(n\sigma)$ equality comparisons, even if the literal phrases of the factorization are known in advance and given as part of the input.

Theorem 3.2. *Let $\sigma, n \in \mathbb{N}^+$ with $8 \leq \sigma \leq \frac{n}{5}$ be fixed. There is no algorithm that computes the Lempel-Ziv Factorization with Known Literals (Problem 3.1) of a string of length n that contains σ distinct symbols over*

(i) *general ordered alphabet in fewer than $\frac{n \log_2 \sigma}{256}$ symbol order comparisons, and*

(ii) *general unordered alphabet in fewer than $\frac{n\sigma}{256}$ symbol equality comparisons*

in the worst case.

After reducing the alphabet to an integer alphabet, we can compute the LZ factorization in $\mathcal{O}(n)$ time (e.g., using [CI08a]). Hence we do not need to design algorithms that compute the LZ factorization over general alphabets. Instead, in Chapter 4, we focus on the practical setting in which the string is packed over integer alphabet $\{0, \ldots, \sigma - 1\}$, and stored in $\mathcal{O}(n/\log_\sigma n)$ words on a word RAM. By exploiting word-level parallelism and combining multiple known techniques from LZ compression, we obtain an algorithm that computes a constant factor approximation of the LZ factorization in optimal time. We then use the approximate factorization as a tool for computing the exact one in almost optimal time, significantly improving the best previously known bounds.

Theorem 4.1. *Let $x[1..n]$ be packed over $[0, \sigma)$. If the LZ factorization of x consists of z phrases, then an LZ-like factorization of x that consists of at most $3z$ phrases can be computed in $\mathcal{O}(n/\log_\sigma n)$ time and $\mathcal{O}(n \log \sigma)$ bits of space.*

Theorem 4.2. *Let $x[1..n]$ be packed over $[0, \sigma)$, and let $\epsilon \in \mathbb{R}^+$ be an arbitrarily small positive constant. If the LZ factorization of x consists of z phrases, then it can be computed in $\mathcal{O}(n/\log_\sigma n + z \log^{3+\epsilon} z)$ time and $\mathcal{O}(n \log \sigma)$ bits of space.*

Finally, in Chapter 5, we consider the rightmost LZ parsing, which aims to improve the compression rate by computing the rightmost previous occurrence of each referencing phrase of the factorization. It is unknown whether this computation can be performed in $\mathcal{O}(n)$ time. While we do not conclusively resolve this question, we do achieve linear (and in one case even sublinear) time for several non-trivial subsets of the referencing phrases. We further introduce the first algorithm that computes the rightmost parsing of the LZ-End factorization, which is a variation of LZ that has special properties beneficial for indexing.

Computing the Lyndon Array (Part II) The Lyndon array of a string is a data structure that plays a central role when detecting squares, or more generally when computing length-wise maximal periodic substrings (see, e.g., [Ban+17]). Hence, before discussing the computation of periodic substrings in Part III, we provide efficient algorithms for computing the Lyndon array in Part II. While the Lyndon array has a simple linear time construction algorithm for integer alphabets (using the suffix array), it was previously unknown whether it can be computed in linear time over general ordered alphabet. We positively answer this question with an algorithm that directly computes the Lyndon array without depending on the suffix array. It relies on combinatorial properties of Lyndon words and their close relation to the lexicographical order of suffixes, which we describe in Chapter 6. We algorithmically exploit these properties in Chapter 7, resulting in a simple linear time algorithm with a fast practical implementation.

Theorem 7.1. *The Lyndon array of a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of space.*

While the time is optimal, using $\mathcal{O}(n)$ words of working space is unsatisfactory. The Lyndon array itself, when stored naively, already requires n words of memory. However, it can be encoded more efficiently as a balanced parentheses sequence that requires only around $2n$ bits of memory. In Chapter 8, we extend the ideas from Chapter 7 and obtain an algorithm that directly computes this succinct version of the Lyndon array, using only a sublinear amount of additional working space. This is the most space efficient algorithm for the Lyndon array, even over polynomial integer alphabet, and we complement it with a fast practical implementation.

Theorem 8.1. *The succinct $2n + 2$ bit representation of the Lyndon array of a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and $\mathcal{O}(n \log \log n / \log n)$ bits of additional working space (apart from the space needed for input and output).*

Finally, in Chapter 9 we also provide a new algorithm for computing the succinct Lyndon array of a word-packed string over integer alphabet $\{0, \ldots, \sigma - 1\}$. While previous algorithms achieve $\mathcal{O}(n)$ time, we introduce novel lookup tables that allow the computation in $\mathcal{O}(n / \log_\sigma n)$ time. The tables are constructed by analyzing and algorithmically exploiting properties of Lyndon words and periodic substrings.

Theorem 9.1. *The succinct $2n+2$ bit representation of the Lyndon array of a length-n string packed over $[0, \sigma)$ can be computed in $\mathcal{O}(n / \log_\sigma n)$ time and $\mathcal{O}(n \log \sigma)$ bits of working space.*

Computing Maximal Periodic Substrings (Part III) In Part III, we consider the problem of computing all maximal periodic substrings of a string, which are also called runs. The question whether computing all runs in a length-n string over general ordered alphabet is possible in $\mathcal{O}(n)$ time dates back at least around 30 years, when Breslauer asked the same question for the easier problem of detecting squares [Bre92]. It was more recently conjectured that the answer is indeed positive [Kos16a]. In Chapter 11, we confirm the conjecture by providing a linear time algorithm. It first constructs the Lyndon array with the algorithm from Chapter 7, and then once more exploits properties of Lyndon words and periodic substrings to obtain the runs. While the details of the proof are intricate and quite technical, the resulting algorithm is surprisingly simple and can be implemented efficiently.

Theorem 10.1. *All the runs contained in a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of working space.*

In Chapter 10, we resolve a similar open problem for general unordered alphabet. In this setting, a simple divide and conquer algorithm computes (a representation of) all runs in $\mathcal{O}(n \log n)$ time, and the bound is tight if the string contains $\Omega(n)$ distinct symbols. This was shown by Main & Lorentz around 40 years ago [ML84], who also explicitly asked whether the bound can be improved if the number of distinct symbols is smaller. We positively answer this question with an algorithm that runs in $\mathcal{O}(n \log \sigma)$ time for a length-n string that contains σ distinct symbols. It is inspired by algorithms that compute runs from the LZ factorization. However, as shown in Chapter 3, the LZ factorization cannot be computed efficiently for unordered alphabets. Instead, we introduce a novel relaxed version of the factorization, and combine it with a wide range of other techniques to obtain the final solution. We complement the algorithm with a matching lower bound that is obtained by using the adversary method. The lower bound holds for the easier problem of testing square-freeness, which can trivially be solved once all runs have been computed.

Theorem 11.1. *Let $n, \sigma \in \mathbb{N}^+$ with $8 \leq \sigma \leq n$ be fixed. There is no deterministic algorithm that performs at most $n \ln \sigma - 3.6n = \mathcal{O}(n \log \sigma)$ equality comparisons in the worst case, and determines whether a length-n string that contains at most σ distinct symbols over general unordered alphabet is square-free.*

Theorem 11.2. *All the runs contained in a length-n string over general unordered alphabet can be computed in $\mathcal{O}(n \log \sigma)$ time, where σ is the number of distinct symbols in the string, which is not known in advance.*

Hence we fully resolve the time complexity of computing runs over general alphabets, which is the most important result presented in the dissertation.

1.2 Corresponding Publications and Contributions of the Author

Most contents of this dissertation have already been published at international conferences and are the result of collaborations with other researchers. The chapters are, for the most part, unchanged or only slightly modified copies of the conference papers. Whenever possible, the introductions and literature reviews at the beginning of the papers have been extended and merged into a single part introduction. The notation was adjusted such that it is consistent throughout the entire dissertation. Now we describe the contributions of the author of the dissertation (henceforth referred to by their name) to each of the presented results. The authors of each paper are arranged in lexicographical order (rather than by level of contribution), which is customary in the field. Please note the copyright statement provided for each of the publications.

Chapter 2 and Part I The lower bounds in Chapters 2 and 3 have been developed by Jonas Ellert specifically for the dissertation, and are inspired by their previous work on general unordered alphabets [EGG23a, EGG23b]. The sublinear time word RAM algorithms for the LZ factorization in Chapter 4 were developed and described by Jonas Ellert, and were published as a single-author paper at SPIRE 2023.

[Ell23] **Jonas Ellert**. „Sublinear time Lempel-Ziv (LZ77) factorization.“ In: *Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)*. Pisa, Italy, 2023, pages 171–187. DOI: 10.1007/978-3-031-43980-3_14

The new advances in rightmost LZ, presented in Chapter 5, are based on joint ideas by Jonas Ellert, Johannes Fischer, and Max Rishøj Pedersen. The algorithmic details as well as the description were, for the most part, developed by Jonas Ellert and Max Rishøj Pedersen, with support of Johannes Fischer. The paper was published at SPIRE 2023 and won the best paper award.

[EFP23] **Jonas Ellert**, Johannes Fischer, and Max Rishøj Pedersen. „New advances in rightmost Lempel-Ziv.“ In: *Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)*. **Winner of the SPIRE 2023 Best Paper Award**. Pisa, Italy, 2023, pages 188–202. DOI: 10.1007/978-3-031-43980-3_15

Part II The introduction to Lyndon words and arrays in Chapter 6 was created by Jonas Ellert specifically for the dissertation, and unites the combinatorial insights and basic ideas from the papers that are presented in Chapters 7 to 9. The simple Lyndon array algorithm in Chapter 7 was developed and described by Jonas Ellert (inspired by earlier work [Bil+20, Bad+22]), and published at ESA 2022 (track S).

[Ell22] **Jonas Ellert**. „Lyndon arrays simplified.“ In: *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*. Potsdam, Germany, 2022, 48:1–48:14. DOI: 10.4230/LIPICS.ESA.2022.48

The algorithm that computes the succinct Lyndon array over general ordered alphabet, presented in Chapter 8, is based on general ideas by Philip Bille, Jonas Ellert, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. The algorithmic details and most of the description were developed by Jonas Ellert, with support of Johannes Fischer and Florian Kurpicz. Note that this contribution should not be fully attributed to the dissertation, as the algorithm was partially developed as part of Jonas Ellert's Master's thesis, which was supervised by Johannes Fischer and Florian Kurpicz. The improvement over the Master's thesis is the reduction of additional working space from $\mathcal{O}(n)$ bits to $\mathcal{O}(n \log\log n / \log n)$ bits. The results were published at ICALP 2020.

[Bil+20] Philip Bille, **Jonas Ellert**, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. „Space efficient construction of Lyndon arrays in linear time.“ In: *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Saarbrücken, Germany (Virtual Conference), 2020, 14:1–14:18. DOI: 10.4230/LIPIcs.ICALP.2020.14

The sublinear time algorithm that computes the succinct Lyndon array of a string over packed integer alphabet, presented in Chapter 9, was developed and described by Jonas Ellert, with Hideo Bannai acting in a supporting role. The results were published at ESA 2023 (track A).

[BE23] Hideo Bannai and **Jonas Ellert**. „Lyndon arrays in sublinear time.“ In: *Proceedings of the 31st Annual European Symposium on Algorithms (ESA 2023)*. Amsterdam, The Netherlands, 2023, 14:1–14:16. DOI: 10.4230/LIPICS.ESA.2023.14

Part III The linear time algorithm that computes runs over general ordered alphabet, presented in Chapter 10, is based on ideas by Jonas Ellert, and the algorithmic details and description were developed by Jonas Ellert and Johannes Fischer. The paper was published at ICALP 2021.

[EF21] **Jonas Ellert** and Johannes Fischer. „Linear time runs over general ordered alphabets.“ In: *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021).* Glasgow, Scotland (Virtual Conference), 2021, 63:1–63:16. DOI: 10.4230/LIPIcs.ICALP.2021.63

The runs algorithm for general unordered alphabet is based on ideas by Paweł Gawrychowski, and the algorithmic details and description were developed by Jonas Ellert, Paweł Gawrychowski, and Garance Gourdel. A preliminary version of the algorithm tests square-freeness rather than computing all runs. The description of this algorithm was published at SODA 2023. The extended version of the paper (including the computation of runs) is available on arXiv [EGG23b].

[EGG23a] **Jonas Ellert**, Paweł Gawrychowski, and Garance Gourdel. „Optimal square detection over general alphabets.“ In: *Proceedings of the 34th Annual Symposium on Discrete Algorithms (SODA 2023).* Florence, Italy, 2023, pages 5220–5242. DOI: 10.1137/1.9781611977554.ch189

Further Contributions

The author's doctoral studies lead to three additional publications that are loosely related to the contents of the dissertation. For the sake of completeness, they are listed below; however, they will not be discussed in the dissertation.

The first publication [Bad+22] introduces an alternative to the Lyndon array algorithm from Chapter 7. The algorithm in [Bad+22] is more complex and computes the Lyndon array from right to left. This results in a back-to-front online algorithm (accessing the symbols of the string in right-to-left order, and at all times maintaining the Lyndon array of the already inspected suffix). The algorithm is quite similar to the one in Chapter 7, overcoming mostly technical challenges resulting from the online setting. This is why it is not explicitly described in the dissertation. It is based on ideas by Jonas Ellert, and was developed and described in a joint effort of Golnaz Badkobeh, Maxime Crochemore, and Jonas Ellert. It was published at CPM 2022, together with results on the average time complexity of computing the Lyndon array by Cyril Nicaud.

[Bad+22] Golnaz Badkobeh, Maxime Crochemore, **Jonas Ellert**, and Cyril Nicaud. „Back-to-front online Lyndon forest construction.“ In: *Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).* Prague, Czech Republic, 2022, 13:1–13:23. DOI: 10.4230/LIPIcs.CPM.2022.13

The second publication [BEF21] introduces a new practical implementation of Baier's suffix sorter [Bai15, Bai16]. It is based on ideas by Johannes Fischer and (to a lesser extent) Jonas Ellert. The development of the implementation began as part of Nico Bertram's master's thesis, supervised by Johannes Fischer and Jonas Ellert.

The final sequential implementation and its description were mostly created by Jonas Ellert. The parallel implementation and its description, as well as the practical evaluation, were done by Nico Bertram. Johannes Fischer provided algorithmic ideas for both the sequential and parallel implementation, and also helped with their description. The results were published at ESA 2021 (track B).

[BEF21] Nico Bertram, **Jonas Ellert**, and Johannes Fischer. „Lyndon words accelerate suffix sorting.“ In: *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021).* Lisbon, Portugal (Virtual Conference), 2021, 15:1–15:13. DOI: 10.4230/LIPIcs.ESA.2021.15

The third publication [EFS20] considers the problem of lexicographically sorting a collection of strings in parallel. A newly introduced theoretical framework serves the purpose of making existing parallel string sorters distinguishing prefix aware. This means that the complexity of the new sorters does not depend on the total size of the string collection, but on the possibly much smaller number of symbols that actually need to be inspected to establish the lexicographical ordering. The framework is based on ideas by Johannes Fischer and Nodari Sitchinava. The algorithmic details and description were developed by Jonas Ellert, with the help of Johannes Fischer and Nodari Sitchinava. The results were published at Euro-Par 2020.

[EFS20] **Jonas Ellert**, Johannes Fischer, and Nodari Sitchinava. „LCP-aware parallel string sorting.“ In: *Proceedings of the 26th International Conference on Parallel and Distributed Computing (Euro-Par 2020).* Warsaw, Poland (Virtual Conference), 2020, pages 329–342. DOI: 10.1007/978-3-030-57675-2_21

Chapter 2

Strings and Alphabets

2

In this chapter, we introduce basic concepts used throughout the dissertation. We start by formally defining strings (Section 2.1) and their representation on a word RAM (Section 2.2). In particular, we consider the word-packed representation of strings, and discuss the recent trend of word-packed string algorithmics. Then, we introduce general ordered and unordered alphabets, for which we first provide some background information and motivation, and then discuss their relation to the complexity of sorting (Section 2.2.2). We show how to reduce strings over general alphabets to strings over integer alphabets (Section 2.2.3), and finally also provide lower bounds on the time needed to solve simple problems over general alphabets (Section 2.3).

2.1 Basic Definitions and Notation

Intervals, Alphabets, and Strings We write $\mathbb{N}^0$ to denote the set of non-negative integers, and $\mathbb{N}^+$ (respectively $\mathbb{R}^+$) to denote the set of positive integers (respectively positive real numbers). For $i, j \in \mathbb{Z}$, we use the interval notation $[i, j] = [i, j+1) = (i-1, j] = (i-1, j+1)$ to denote the set $\{k \in \mathbb{Z} \mid i \leq k \leq j\}$. For $n \in \mathbb{N}^+$, we sometimes write $\log n$ to denote the base-two logarithm $\log_2 n$.

An *alphabet* Σ is a finite set of *symbols*. For $n \in \mathbb{N}^+$, a string x of length $|x| = n$ over the alphabet Σ is a sequence of n symbols from Σ. The set Σ^n contains all the $|\Sigma|^n$ distinct strings of length n over Σ. The unique and alphabet-independent empty string of length 0 is denoted by ε, and it is the only element of the set Σ^0. Finally, the set $\Sigma^* = \bigcup_{i \in \mathbb{N}^0} \Sigma^i$ contains all strings over Σ, while $\Sigma^+ = \bigcup_{i \in \mathbb{N}^+} \Sigma^i$ contains only the non-empty strings over Σ. In the context of the dissertation, the only distinction between arrays and strings is the nomenclature. Therefore, we use the notation introduced in the following paragraphs for both strings and arrays.

For some string $x \in \Sigma^n$, an integer $i \in [1, n]$ is called index or position of x, and we write $x[i]$ to denote the ith symbol of the sequence x. Given another position $j \in [1, n]$, we use the equivalent notations $x[i..j]$, $x(i-1..j]$, $x[i..j+1)$, and $x(i-1..j+1)$ to denote a *substring* of x. If $i \geq j$, then the substring $x[i..j]$ is the sequence $x[i]x[i+1]\ldots x[j] \in \Sigma^{j-i+1}$. Otherwise, it is the empty string ε. A *proper* substring of x is of length less than $|x|$. A *non-trivial* substring of x is both proper and non-empty. Substring $x[i..n]$ is a *suffix* of x, while $x[1..i]$ is a *prefix*. Additionally, x has the empty suffix $x(n..n] = \varepsilon$ and empty prefix $x[1..1) = \varepsilon$. For $i \in [1, n+1]$, we use the simplified notation $x_i = x[i..n]$. However, in some contexts (but always without ambiguity), we also use subscript indices to enumerate a collection of strings, e.g., a list of strings $y_1, y_2, \ldots, y_k$, or to identify individual (sub-)strings. Whenever we want to declare a string x of length n without specifying the alphabet, we simply write $x[1..n]$ or $x = x[1..n]$. The *reversal* of x is the string $\mathsf{rev}(x)[1..n]$ with $\forall i \in [1, n] : \mathsf{rev}(x)[i] = x[n - i + 1]$. Given two strings $x[1..n]$ and $y[1..m]$ (each possibly empty), we write $x \cdot y$ (or simply xy if there is no ambiguity) to denote the concatenation of the sequences x and y, which is a string of length $n + m$. If $x = y_1 y_2 \ldots y_k$ for some strings $y_1, y_2, \ldots, y_k$, then we say that $y_1 y_2 \ldots y_k$ is a *factorization* of x, and each substring y_h with $h \in [1, k]$ is called factor or phrase. Note that the terms factor, phrase, and substring are synonymous. For non-empty strings $x[1..n]$ and y, an *occurrence* of y in x is a position $i \in [1, n]$ such that y is a prefix of $x[i..n]$. For the occurrence i of substring $x[i..i + \ell)$ in x, a *previous occurrence* is an occurrence $j \in [1, i)$ of $x[i..i + \ell)$ in x.

Repetitions For *exponent* $k \in \mathbb{N}^0$, the kth *power* of x is $x^k = \varepsilon$ if $k = 0$, and otherwise the k-times concatenation of x, which is recursively defined as $x^k = x \cdot x^{k-1}$. A *k-power* is a string of the form x^k. A *square* is a 2-power. A string is *primitive* if and only if it is not a k-power for any integer $k \geq 2$. Let $b \in [0, n]$, then $x[1..n]$ has *border* $x[1..b]$ of length b if and only if $x[1..b] = x(n - b..n]$. A *period* $p \in [1, n]$ of x satisfies $\forall i \in [1, n - p] : x[i] = x[i + p]$, or equivalently $x[1..n - p] = x(p..n]$. Note that a length-n string has period p if and only if it has a border of length $n - p$. For a given string, we refer to *its* period or *the* period of the string (rather than *a* period) whenever we mean the minimal period. A string is *periodic* if it is non-empty and its period is at most half its length. A *repetition* in some string $x[1..n]$ is a triple $\langle i, j, p\rangle$ with $i, j, p \in [1, n]$ such that substring $x[i..j]$ is periodic with minimal period $p \leq (j - i + 1)/2$. A *run* (also called *maximal periodic substring*) is a repetition $\langle i, j, p\rangle$ that cannot be extended to either side with the same period, i.e., $i = 1$ or $x[i - 1] \neq x[i + p - 1]$ and $j = n$ or $x[j + 1] \neq x[j - p + 1]$.

Lexicographical Order and Longest Common Extensions If there is some total order $<$ on Σ, then it implies a lexicographical order $\prec$ on Σ^* as follows. Given strings $x, y \in \Sigma^*$, we say that x is lexicographically smaller than y and write $x \prec y$ if and only if either x is a proper prefix of y or there is some $\ell \in [1, \min(|x|, |y|)]$ such that $x[1..\ell) = y[1..\ell)$ and $x[\ell] < y[\ell]$. We write $x \preceq y$ to denote that y is not lexicographically smaller than x. We say that x is *co-lexicographically* smaller than y if and only if $\mathsf{rev}(x) \prec \mathsf{rev}(y)$.

We are often interested in the lexicographical order of suffixes of a single string $x[1..n]$; given positions $i, j \in [1, n]$, we want to determine if $x[i..n] \prec x[j..n]$. This can be achieved with the *longest common extension (LCE)* function, which is defined as

$$\textsc{lce}(i, j) = \max(\{\ell \in [0, n - \max(i, j) + 1] \mid x[i..i + \ell) = x[j..j + \ell)\}),$$

i.e., $\textsc{lce}(i, j)$ is the length of the longest shared prefix between $x[i..n]$ and $x[j..n]$. It is easy to see that $x[i..n] \prec x[j..n]$ if and only if $j + \textsc{lce}(i, j) \leq n$ and either $i + \textsc{lce}(i, j) = n + 1$ (in which case $x[i..n]$ is a proper prefix of $x[j..n]$) or otherwise $x[i + \textsc{lce}(i, j)] < x[j + \textsc{lce}(i, j)]$. Strings $x[1..n]$ and $y[1..n]$ are *isomorphic* if and only if $\forall i, j \in [1, n] : x[i] = x[j] \iff y[i] = y[j]$. Strings $x[1..n]$ and $y[1..n]$ over (possibly different) totally ordered alphabets are *order-isomorphic* if and only if $\forall i, j \in [1, n] : x[i] < x[j] \iff y[i] < y[j]$.

2.2 Strings on a Word RAM

The input to all considered problems is a string of length n, which we also call *the text*. The computation is performed on a word RAM that operates on binary words of width $w \geq \lceil \log_2 n \rceil$ bits. This assumption is justified by the fact that $\lceil \log_2 n \rceil$ bits are needed to address the string. For a description and discussion of the word RAM model, see, e.g., [Hag98] and the references therein.

Depending on the permitted word RAM instructions, there are crucial differences in the computational power of the model. Hence we briefly describe the instruction set. A word is interpreted as an integer in the range $[0, 2^w)$, and the word RAM supports the following operations on two such integers a and b in constant time: arithmetic operations ($\lfloor a \oplus b \rfloor \bmod 2^w$ for $\oplus \in \{+, -, \cdot, /\}$), bit-shifts (left shift $(a \cdot 2^b) \bmod 2^w$ and right shift $\lfloor a \cdot 2^{-b} \rfloor$), and bit-wise logical operations (NOT, AND, OR, XOR). Constant time modulo operations are possible due to $(a \bmod b) = a - b \cdot \lfloor a/b \rfloor$. The time required by an algorithm is measured in the number of word RAM operations performed, and we express the time (and space) complexity using big-$\mathcal{O}$ notation (see, e.g., [Cor+22, Chapter 3.2]). A polynomial time algorithm takes $\mathcal{O}(n^c)$ time for arbitrarily large constant $c \in \mathbb{N}^+$, written as $\mathcal{O}(\text{poly}(n))$. A linear time algorithm takes $\mathcal{O}(n)$ time. A polylogarithmic time algorithm takes $\mathcal{O}(\log^c n)$ time for an arbitrarily large constant $c \in \mathbb{N}^+$, written as $\mathcal{O}(\text{polylog}(n))$. The presented algorithms are deterministic.

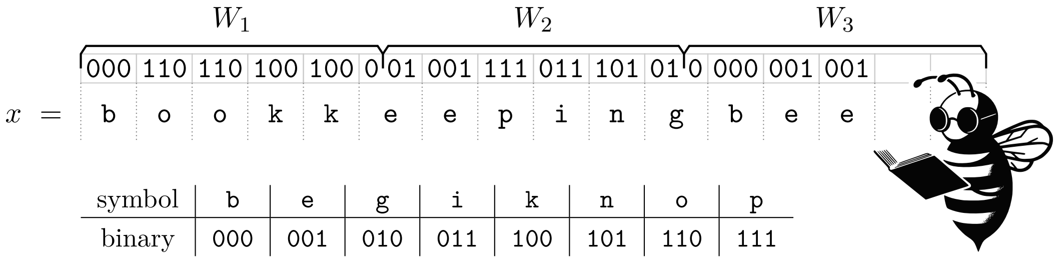
The word RAM operates on a memory of $2^{\mathcal{O}(w)}$ cells with addresses $0, 1, 2 \ldots$ (not initialized in any particular way). Each cell contains a word, i.e., an integer from $[0, 2^w)$, and given the address of a cell (stored in a constant number of memory words), the value of a cell can be read or written in constant time. Memory does not need to be allocated in any way. The space complexity (or memory usage) of an algorithm, measured in words, is $1 + a$, where a is the maximum address of a cell that has been either read or written by the algorithm. The presented algorithms generally only use a constant number of variables and few additional data structures that can be laid out sequentially without gaps in the memory. Hence the space complexity can also be expressed as the sum of the words occupied by all the data structures.

By using bit-shifts and bit-wise operations, the word RAM can simulate smaller memory cells of arbitrary width without significant time overhead. For example, an array of a entries from $[0, b)$ can be stored in $\lceil a \lceil \log_2 b \rceil / w \rceil$ (full size) words by simulating a words of width $\lceil \log_2 b \rceil$. We then express the space complexity in terms of the number of used bits, e.g., the described array uses $\mathcal{O}(a \log b)$ bits plus a constant number of additional words.

2.2.1 Integer Alphabets

A natural assumption on a word RAM is that the text is over *integer alphabet*, i.e., $x \in [0, 2^w)^n$. For $x[1..n]$ over integer alphabet, let $\sigma_{\min} \in [1, 2^w]$ be the minimal value such that $x \in [0, \sigma_{\min})^n$ (equivalently, $\sigma_{\min} - 1$ is the maximal symbol present in the string). Then x is over *polynomial integer alphabet* if $\sigma_{\min} = \mathcal{O}(\text{poly}(n))$, and over *effective integer alphabet* if x contains $\sigma_{\min}$ distinct symbols. Since symbols are integers in $[0, 2^w)$, the alphabet is totally ordered. Also, we can store the binary representation of each symbol in a memory word, and the entire string in n words. This is the *word-aligned* representation of x.

Alternatively, x might be given *packed over* $[0, \sigma)$ for some $\sigma \in [\sigma_{\min}, 2^w)$. This means that the binary representation of each symbol is stored in only $\lceil \log_2 \sigma \rceil$ bits,



symbol	b	e	g	i	k	n	o	p
binary	000	001	010	011	100	101	110	111

Figure 2.1: The string $x[1..14] =$ `bookkeepingbee` contains eight distinct symbols and can be stored in $\lceil \log_2 8 \rceil = 3$ bits per symbol. The width of a word is $w = 16$ bits, and thus $\left\lceil \frac{14 \cdot 3}{16} \right\rceil = 3$ words W_1, W_2, and W_3 suffice for storing x. In the drawing, the leftmost bit of each word is the most significant one, while the rightmost bit is the least significant one. Substrings of length at most $\lfloor 16/3 \rfloor = 5$ fit into a single word and can be extracted in constant time by using bit-shifts and bit-wise logical operations. For example, $x[4..7] =$ `kkee` can be extracted as a word $M = (0\,000\,100\,100\,001\,001)_2$ using $M = \lfloor ((W_1 \cdot 2^9) \bmod 2^{16}) \cdot 2^{-4} \rfloor$ `OR` $\lfloor W_2 \cdot 2^{-11} \rfloor$. The exponents used for the shifts can be computed with simple arithmetic. Bee drawing by DALL·E [Bet+23].

and the entire text requires $n \lceil \log_2 \sigma \rceil$ bits. Hence it can be stored in $\lceil n \lceil \log_2 \sigma \rceil / w \rceil = \mathcal{O}(n/\log_\sigma n)$ consecutive memory words. In this representation, a single memory word may contain (parts of) many consecutive symbols, as visualized in Figure 2.1. However, by using bit-shifts and bit-wise logical operations, it is easy to extract a substring of length $m \leq \lfloor w / \lceil \log_2 \sigma \rceil \rfloor$ (e.g., a single symbol) and store it in the least significant $m \lceil \log_2 \sigma \rceil$ bits of a word in constant time. In the same way, we can assign short substrings in constant time. An example of the extraction is provided in the description of Figure 2.1. Hence there is no computational disadvantage when working on a text in packed representation. On the contrary, one can benefit from word-level parallelism, as multiple symbols are stored in a word and can thus be processed simultaneously (which we exploit in Chapters 4, 5 and 9). From an algorithmic perspective, it is most beneficial if the text is packed over effective integer alphabet, which maximizes the number of symbols that are packed in a word.

Background of Packed String Algorithmics Since a length-n string packed over $[0, \sigma)$ occupies $\lceil n \lceil \log_2 \sigma \rceil / w \rceil$ words of memory, merely reading the string requires $\Theta(n \log \sigma / w)$ time, or $\Theta(n/\log_\sigma n)$ time if $w = \Theta(\log n)$ (a common assumption). Hence this time is also a natural lower bound for solving any problem that requires inspecting all symbols of the string. In contrast to that, designers of string algorithms traditionally strive for $\mathcal{O}(n)$ time solutions, i.e., they aim for time linear in the length of the string. More recently, there has been a trend towards algorithms that run in (or at least close to) $\Theta(n/\log_\sigma n)$ time, i.e., in time linear in the number of words occupied by the string, which is sublinear in n for sufficiently small σ. The general idea is to use word-level parallelism, which means that multiple symbols stored in a single word are processed simultaneously, for example by using specialized instructions that exist in real-world computers (e.g., SIMD instruction sets like AVX2 or SSE) or by simulating such instructions with precomputed universal lookup tables.

Word-level parallelism has, for example, been applied to the classic pattern matching problem, where one has to find the occurrences of a length-m pattern string in a length-n text. There are multiple algorithms [Bil11, Bel12, Ben+14] that achieve

$\mathcal{O}(n/\log_\sigma n)$ time, plus some additional time depending on m and the number of occurrences. Packed strings have also been considered [GGF13] for the approximate pattern matching problem, where each occurrence of the pattern is allowed to have a fixed number of mismatches under the Hamming distance. More recently, several full-text indices [BGS17, Tak+17, MNN20a, MNN20b] have been proposed in the packed setting. Such indices preprocess the text once, and then allow fast pattern matching for arbitrary patterns.

Another line of research aims to exploit word-level parallelism for the efficient implementation of fundamental tools and data structures that are commonly used in string algorithmics. The perhaps most important result of this kind is a novel LCE data structure by Kempa and Kociumaka [KK19] that can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time and answers LCE queries in constant time (which we will use repeatedly throughout the dissertation). This data structure has already been used to compute the longest palindromic substring (i.e., a substring that equals its reversal) [CPR22] and the longest common substring of two strings [Cha+21] in sublinear time. In the same paper [KK19], Kempa and Kociumaka also show how to compute the Burrows-Wheeler transform (a major data structure in text indexing) in $\mathcal{O}(n\log\sigma/\sqrt{\log n})$ time, which is another breakthrough result. In another work [KK23], they show how to compute compressed versions of the suffix array and suffix tree (also important data structures in text indexing) in $\mathcal{O}(n\log\sigma/\sqrt{\log n})$ time. For their solution, they use (among other things) the wavelet tree, a fundamental data structure that has only recently been shown to be constructible in $\mathcal{O}(n\log\sigma/\sqrt{\log n})$ time [Bab+15, MNV16].

2.2.2 Extending the Word RAM with General Alphabets

A more abstract and weaker model of computation extends the word RAM by assuming that the text $x[1..n]$ is over a totally ordered alphabet, but the text has no accessible representation in word RAM memory. Instead, the text can be seen as an oracle that, given some query positions $i, j \in [1, n]$, outputs whether or not $x[i] < x[j]$ in constant time. In this case, we say that x is over *general ordered alphabet*. An algorithm for general ordered alphabet performs a mix of word RAM operations and symbol order comparisons (i.e., oracle queries). The time complexity of the algorithm is the combined number of word RAM operations and symbol comparisons.

For an even weaker version of this model, we reduce the power of the oracle. Given two query positions $i, j \in [1, n]$, the oracle can now only output whether or not $x[i] = x[j]$ in constant time. In this case, we say that x is over *general unordered alphabet*. An algorithm for general unordered alphabet performs a mix of word RAM operations and symbol equality comparisons (i.e., oracle queries). The time complexity of the algorithm is the combined number of word RAM operations and symbol comparisons.

Note that an equality comparison can be simulated by two order comparisons, and an order comparison of two integers takes constant time on a word RAM. Hence the order of computational strength among the considered models is (from weakest to strongest): general unordered alphabet, general ordered alphabet, integer alphabet on a word RAM. Sometimes comparison-based algorithms are analyzed in the decision tree model, i.e., by merely counting the number of symbol comparisons and ignoring the additional computation needed to decide *which* comparisons to perform. In contrast to that, we choose to always consider the additional word RAM operations

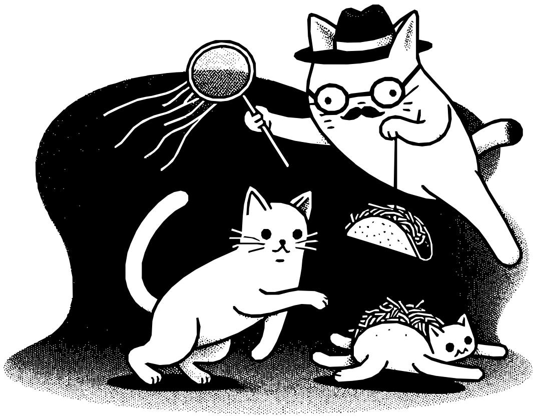


Figure 2.2: The `tacocat` is under `catattack`. The `educated-cat-catcher` tries to prevent the `catattack` by using a decoy taco. The string `tacocat` is a palindrome; the string `catattack` contains the square `atat`; the string `educated-cat-catcher` contains three occurrences of the pattern `cat`. Envisioned by DALL·E [Bet+23].

as well. This has the advantage that algorithms for the weaker models are compatible with the stronger ones, and it also reflects the fact that some of the presented algorithms indeed have fast practical implementations. However, we still use the decision tree complexity for lower bounds.

Background of General Alphabets Stringology significantly predates the word RAM model, which only gained widespread popularity in the 1990s. In contrast to that, the foundations of stringology and combinatorics on words were laid, e.g., by Axel Thue in the early 1900s (see [Thu06, Ber94]), and by Lyndon and Shirshov in the 1950s (see [Lyn54, Shi58]). The perhaps most famous string algorithm, the Knuth-Morris-Pratt pattern matching algorithm [KMP77], was introduced in the 1970s. Its input is a text of length n and a pattern string of length m. The output is a list of occurrences of the pattern in the text. For example, the text `educated-cat-catcher` contains three occurrences of the pattern `cat`. The algorithm works over general unordered alphabet, i.e., it interacts with text and pattern by testing the equality of symbols. The number of symbol equality comparisons is also the main complexity measure of the algorithm, and it is bounded by $\mathcal{O}(n+m)$. It is known that $n+\Omega(\frac{n}{m})$ equality comparisons are required to solve the pattern matching problem [Col+95], and there is an online algorithm that matches this bound [CH97]. Another algorithm solves pattern matching in $\mathcal{O}(n+m)$ equality comparisons and uses only a constant number of additional memory words [GS83]. A two-dimensional pattern with dimensions $m \times m$ can be matched in a two-dimensional text with dimensions $n \times n$ in $\mathcal{O}(n^2+m^2)$ equality comparisons [GP92].

Pattern matching is not the only classic string problem that has an efficient solution for general unordered alphabet. For example, for a text of length n, it takes $\mathcal{O}(n)$ symbol equality comparisons to compute all the length-wise maximal palindromic substrings (i.e., substrings that equal their own reversal, like `tacocat`) [Man75, ABG95]. Testing whether the text contains a square (i.e., the two-times repetition of a shorter string, like `atat` in the text `catattack`) takes $\mathcal{O}(n \log n)$ symbol equality comparisons, and there is a matching $\Omega(n \log n)$ lower bound if the text contains $\Omega(n)$ distinct symbols [ML84]. Interestingly, if the string is over general *ordered* alphabet, then testing square-freeness requires only $\mathcal{O}(n)$ symbol order comparisons (which is the topic of Chapter 10). Hence the presence of order seems to be beneficial when detecting squares, even though order is not needed for defining a square. Figure 2.2 visualizes the cat-based examples.

While there are still new results targeted at general alphabets, like the work of Duval, Lecroq, and Lefebvre [DLL14] on computing an unbordered cyclic shift, or Kosolobov's work [Kos16b] on finding the leftmost critical point, it is evident that general alphabets are becoming increasingly uncommon. However, there is a point to be made about their usefulness when analyzing the complexity of problems. General alphabets offer a weaker model of computation than word RAMs with polynomial integer alphabets. Hence general alphabets can be used to separate the hardness of problems that are easy to solve over integer alphabet. An example of this is provided in Table 2.1, which shows the time complexity of three classic problems: computing maximal palindromic substrings, testing square-freeness, and computing the Lempel-Ziv [LZ76] (LZ) factorization. If we only consider polynomial integer alphabets, then all of the problems appear to be equally hard, with matching $\Theta(n)$ time lower and upper bounds. We can only separate the problems by assuming a weaker model of computation. For general ordered alphabet, computing the LZ factorization is harder than the other two problems. If we assume a general unordered alphabet, then we also observe that testing square-freeness is harder than computing maximal palindromic substrings. This demonstrates that general alphabets are a powerful tool for the theoretical analysis of problems.

General Alphabets and the Complexity of Sorting The distinction between integer alphabet and general ordered alphabet is the same as the one between integer sorting and comparison-based sorting. If solving a problem requires sorting the symbols of the text, then there may be an $\mathcal{O}(n)$ time algorithm for polynomial integer alphabet (due to, e.g., radix sorting [Cor+22, Section 8.3]), but such an algorithm cannot exist for general ordered alphabet due to the well-known information

Table 2.1: Time complexities for three classic problems, given a length-n string that contains σ distinct symbols. All problems have a trivial $\Omega(n)$ time lower bound because they require inspecting all symbols of the string. The provided references are examples, and not always the earliest publication with the claimed bounds.

problem	word-aligned over polynomial integer alphabet	general ordered alphabet	general unordered alphabet
max. palindromic substrings	$\Theta(n)$ [Man75]	$\Theta(n)$ [Man75]	$\Theta(n)$ [Man75]
square-freeness	$\Theta(n)$ Chap.10 or [EF21]	$\Theta(n)$ Chap.10 or [EF21]	$\Theta(n \log \sigma)$ Chap.11 or [EGG23a]
LZ factorization	$\Theta(n)$ [CI08a]	$\Theta(n \log \sigma)$ Lem.2.1+[CI08a], Chap.3 or [Kos15b]	$\Theta(n\sigma)$ Lem.2.1+[CI08a], Chap.3 or [EGG23a]

theoretical $\Omega(n \log n)$ lower bound for comparison-based sorting [Cor+22, Section 8.1]. Hence, by using general alphabets, we limit the algorithm's capability to sort the symbols of the text.

It is a major open question if linear time sorting is possible for arbitrary integer alphabets (of super-polynomial size). Particularly, it is unknown whether n integers from $[0, 2^w)$ can be sorted in linear time if $w \in \omega(\log n) \cap \mathcal{O}(\log^{2+\epsilon} n)$ (for any constant $\epsilon \in \mathbb{R}^+$). This motivates a type of alphabet that characterizes exactly the texts that can be sorted in linear time. A string $x[1..n]$ is over *linearly-sortable alphabet* if and only if all of the following three conditions are satisfied: the string is over totally ordered alphabet; the equality of two symbols at any given positions can be tested in constant time; it takes $\mathcal{O}(n)$ time and words of space to sort the text, i.e., to compute a permutation π of $[1, n]$ such that $x[\pi(1)] \leq x[\pi(2)] \leq \cdots \leq x[\pi(n)]$ (where the permutation is stored as an array). Examples of linearly-sortable alphabets are polynomial integer alphabets, general ordered and unordered alphabets of constant size (where we enforce an arbitrary order of symbols for general unordered alphabet), or integer alphabets on a word RAM with $w = \Omega(\log^{2+\epsilon} n)$ for any constant $\epsilon \in \mathbb{R}^+$ (if we allow randomization and expected time bounds, see [And+98]).

2.2.3 Alphabet Reduction by Sorting

In this section, we show how to reduce a text over a linearly-sortable or general (ordered or unordered) alphabet to an order-isomorphic text packed over its effective integer alphabet. The reduction is achieved in the obvious way by sorting the symbols of the text and replacing each symbol with its rank, and the proof is provided only for the sake of completeness.

Lemma 2.1. *Let $x[1..n]$ be a string that contains σ distinct symbols.*

(a) *An order-isomorphic string packed over $[0, \sigma)$ can be computed in $\mathcal{O}(n)$ time and words of space if x is over linearly-sortable alphabet.*

(b) *An order-isomorphic string packed over $[0, \sigma)$ can be computed in $\mathcal{O}(n \log \sigma)$ time and $\mathcal{O}(\sigma \log n)$ bits of space if x is over general ordered alphabet.*

(c) *An equality-isomorphic string packed over $[0, \sigma)$ can be computed in $\mathcal{O}(n\sigma)$ time and $\mathcal{O}(\sigma \log n)$ bits of space if x is over general unordered alphabet.*

The stated space complexities ignore a constant number of words and the space occupied by x, as well as the $n \lceil \log_2 \sigma \rceil$ bits needed to store the isomorphic string.

Proof. For all results, we use a two stage reduction. In the first stage, we compute the number σ of distinct symbols in x. Then, it takes $\mathcal{O}(\log n)$ time to compute the number $\lceil \log_2 \sigma \rceil$ of bits used for each symbol of the isomorphic string in packed representation. In the second stage, we actually compute the string, which is denoted by y in the remainder of the proof.

For (a), we compute a permutation π of $[1, n]$ such that $x[\pi(1)] \leq \cdots \leq x[\pi(n)]$, which takes linear time and words of space by the definition of linearly-sortable alphabets. Now it holds $\sigma = 1 + |\{i \in [2, n] \mid x[\pi(i-1)] \neq x[\pi(i)]\}|$, which can easily be computed in $\mathcal{O}(n)$ time. Finally, we construct y. We start by assigning $y[\pi(1)] \leftarrow 0$, and then we consider $i \in [2, n]$ in ascending order. Whenever $x[\pi(i)] = x[\pi(i-1)]$, we assign $y[\pi(i)] = y[\pi(i-1)]$. Otherwise, we assign $x[\pi(i)] \leftarrow x[\pi(i-1)] + 1$. This clearly results in an order-isomorphic string and takes linear time and words of space.

For (b), we scan the text from left to right and maintain a balanced binary search tree (e.g., a red-black tree [Cor+22, Chapter 13]) of all the distinct symbols that we have seen so far. Since the symbols have no accessible representation, we use the position of the leftmost occurrence of each symbol as its representative. However, the order used by the tree is the order of symbols, not the order of positions. Whenever we process some symbol $x[i]$, we try to find it in the tree, issuing $\mathcal{O}(\log \sigma)$ symbol comparisons. If the symbol is already present, then we do nothing. Otherwise, we insert $x[i]$ (represented by i) into the tree. Afterwards, the tree contains exactly the σ distinct symbols that are present in the text, each represented by its leftmost occurrence. For each node, we compute its rank, i.e., the number of nodes that represent smaller symbols, which can be done by traversing the tree in $\mathcal{O}(\sigma)$ time. Now we can create an order-isomorphic text y packed over its effective alphabet $[0, \sigma)$, where each $y[i]$ is the rank of $x[i]$ among all symbols in x. This only requires another scan of x, during which we once more find each symbol in the tree and look up its rank in overall $\mathcal{O}(n \log \sigma)$ time. The memory usage is dominated by the $\mathcal{O}(\sigma \log n)$ bits needed for the tree.

The reduction for (c) works similarly. However, this time we maintain a simple list of all the distinct symbols seen so far (where again each symbol is represented by its leftmost occurrence). We process the positions $i \in [1, n]$ in ascending order. Whenever we process some symbol $x[i]$, we naively try to find it in the list, issuing at most σ' symbol equality comparisons, where σ' is the current length of the list. If symbol $x[i]$ has no representative in the list, then we append i to the end of the list. Otherwise, we do nothing. Afterwards, the list contains exactly the σ distinct symbols that are present in the text, each represented by its leftmost occurrence. Now we construct the text y packed over effective integer alphabet $[0, \sigma)$. For each $i \in [1, n]$, we find the unique $k \in [0, \sigma)$ such that the kth entry of the list (counting from 0) is a position j representing the symbol $x[j] = x[i]$. This takes at most σ symbol equality comparisons and allows us to assign $y[i] \leftarrow k$. Clearly, the strings y and x are isomorphic. The memory usage is dominated by the $\mathcal{O}(\sigma \log n)$ bits needed for the list. □

2.3 Lower Bounds for Basic Problems Over General Alphabets

As shown in Lemma 2.1, a general ordered alphabet can be reduced to effective integer alphabet in $\mathcal{O}(n \log \sigma)$ time, and a general unordered alphabet can be reduced to effective integer alphabet in $\mathcal{O}(n\sigma)$ time. For some problems, we cannot perform better than these reductions. That is, given a string over general alphabet, the fastest way to solve some problems is to reduce the alphabet with Lemma 2.1, and then use an algorithm that is designed for effective integer alphabet. In this section, we show that merely deciding whether a given string contains few or many distinct symbols (Problem 2.2) is one of these problems, i.e., it has an $\Omega(n \log \sigma)$ and respectively $\Omega(n\sigma)$ time lower bound for general ordered and unordered alphabet (Theorem 2.3).

Problem 2.2 (Alphabet Size Testing)**.** Let $\sigma, n \in \mathbb{N}^+$ with $\sigma \leq \frac{n}{2}$. Given a string $x[1..n]$ that contains either at most σ or at least $\frac{n}{2}$ distinct symbols, decide which of the two cases applies.

Theorem 2.3. *For any fixed $\sigma, n \in \mathbb{N}^+$ with $\sigma \leq \frac{n}{2}$, there is no deterministic algorithm that solves Alphabet Size Testing (Problem 2.2) over*

(*i*) *general ordered alphabet in fewer than $\frac{n\lceil \log_2 \sigma \rceil}{4}$ symbol order comparisons, and*

(*ii*) *general unordered alphabet in fewer than $\frac{n\sigma}{4}$ symbol equality comparisons,*

respectively in the worst case.

If the solution of some string algorithmic problem inherently reveals the number of distinct symbols in the text, then Alphabet Size Testing can be reduced to this problem, and the lower bounds from Theorem 2.3 apply. For example, the number of literal phrases in the Lempel-Ziv factorization (as defined in Part I) equals the number of distinct symbols in the string, and thus computing the Lempel-Ziv factorization takes $\Omega(n \log \sigma)$ and respectively $\Omega(n\sigma)$ time over general ordered and unordered alphabet. However, showing lower bounds in this way has the disadvantage that it only works for problems that do not expect the number σ of distinct symbols as part of the input. Thus, we also show lower bounds for the related problem of Alphabet Set Testing (Problem 2.4 and Theorem 2.5 below). This shows that making a simple statement about the alphabet of a string is hard, even if an explicit list of the distinct symbols is given in advance. In Chapter 3, we show that Alphabet Set Testing can be reduced to computing the Lempel-Ziv factorization, even if we formulate Lempel-Ziv such that the literal phrases are given as part of the input. The definition of Alphabet Set Testing is inspired by Kosolobov's lower bound for Lempel-Ziv over general ordered alphabet [Kos15b, Theorem 1].

Problem 2.4 (Alphabet Set Testing)**.** Let $\sigma, n \in \mathbb{N}^+$, let $\Sigma = \{a_1, \ldots, a_\sigma\}$, and let $\Sigma' = \{b_1, \ldots, b_\sigma\}$ with $|\Sigma \cup \Sigma'| = 2\sigma$. Given σ and $x[1..n + 2\sigma] = uvy$ with $u = a_1 a_2 \ldots a_\sigma$, $v = b_1 b_2 \ldots b_\sigma$ and $y \in (\Sigma \cup \Sigma')^n$, decide if $y \in \Sigma^n$.

Theorem 2.5. *For any fixed $\sigma, n \in \mathbb{N}^+$, there is no deterministic algorithm that solves Alphabet Set Testing (Problem 2.4) over*

(*i*) *general ordered alphabet in fewer than $\frac{n\lceil \log_2 \sigma \rceil}{2}$ symbol order comparisons, and*

(*ii*) *general unordered alphabet in fewer than $\frac{n\sigma}{2}$ symbol equality comparisons,*

respectively in the worst case.

2.3.1 Adversary Method for Lower Bounds

All lower bounds will be shown with the adversary method. The general idea is as follows. Suppose that there is some algorithm that solves a problem over general (ordered or unordered) alphabet. Then the algorithm has no direct access to the symbols of the text. Instead, the text can be seen as an oracle. The algorithm provides a pair $i, j \in [1, n]$ of query positions, and the oracle answers whether or not

$x[i] = x[j]$ (in case of an unordered alphabet) or $x[i] < x[j]$ (in case of an ordered alphabet). A malicious adversary will take over the role of the oracle, and its strategy is to construct (a family of) worst-case instances of the problem.

This process can be seen as a game between the algorithm and the adversary. In each round of the game, the algorithm asks a query and the adversary provides the answer. The algorithm aims to minimize the number of rounds needed to solve the problem, while the adversary aims to maximize it. The considered problems are decision problems, i.e., the algorithm ultimately has to decide whether it is working on a "yes" or on a "no" instance of the problem. Hence the adversary has to ensure that there is always at least one "yes" instance and one "no" instance that is consistent with the answers given so far. As long as the adversary can maintain this state, the algorithm is unable to solve the problem.

Of course, the adversary has to give answers consistently. Once it answers that $x[i] < x[j]$, it has to ensure that future answers do not contradict $x[i] < x[j]$ (either directly, or due to the transitivity of $<$ and $=$). Apart from that, there are no restrictions on the actions of the adversary. It has infinite computational power, i.e, we do not care about the time needed to answer a query. The same holds for the algorithm; we are only concerned about the number of queries asked, without paying attention to the time needed to choose the query positions. Thus, we will obtain lower bounds on the number of comparisons needed by the algorithm, which also trivially lower bounds the required time.

2.3.2 Lower Bounds for General Ordered Alphabet

Now we show the lower bounds from Theorem 2.5(i) and Theorem 2.3(i), respectively restated in Lemma 2.6 and Lemma 2.7.

Lemma 2.6. *For any fixed $\sigma, n \in \mathbb{N}^+$, there is no deterministic algorithm that solves Alphabet Set Testing (Problem 2.4) over general ordered alphabet in fewer than $\frac{1}{2} \cdot n \lceil \log_2 \sigma \rceil$ symbol order comparisons in the worst case.*

Proof. We use an adversary as described in Section 2.3.1. Internally, the adversary works with the integer alphabet $\Sigma \cup \Sigma' = [1, 2\sigma]$ with $\Sigma = \{2, 4, 6, \dots, 2\sigma\}$ (the even symbols) and $\Sigma' = \{1, 3, 5, \dots, 2\sigma - 1\}$ (the odd symbols). Let $\sigma' = 2^{\lfloor \log_2 \sigma \rfloor}$. In order to keep track of the given answers, the adversary annotates each position in $[1, n + 2\sigma]$ with a consecutive range of symbols that are consistent with the answers given so far. The initial annotation of each position $i \in (2\sigma, n + 2\sigma]$ is $[1, 2\sigma']$ (which indicates that $x[i]$ could be any symbol from $[1, 2\sigma']$). The initial annotation of each $i \in [1, \sigma]$ is $[2i, 2i]$, and the annotation of each $i \in (\sigma, 2\sigma]$ is $[2(i - \sigma) - 1, 2(i - \sigma) - 1]$, which indicates that positions $[1, 2\sigma]$ are already fixed to their respective symbols (as required by the problem definition). The adversary may update the range of a position by replacing it with a sub-range (which we describe in a moment). At all times, the cardinality of each range will be a power of two. Once the cardinality of a range becomes one, it will never change again. At any point in time, a string $x[1..n + 2\sigma]$ is *consistent with the answers of the adversary* if for every $i \in [1, n + 2\sigma]$ it holds $x[i] \in [i', i' + 2^p)$, where $[i', i' + 2^p)$ is the range that annotates position i. A position annotated with a range of cardinality one is fixed; there is only one symbol that can be assigned to such a position.

As mentioned earlier, the algorithm can only access x (or rather the simulated family of strings) by asking the adversary whether or not $x[i] < y[j]$, where $i, j \in [1, n + 2\sigma]$. When given such a query, the adversary first inspects the respective ranges $I = [i', i' + 2^p)$ and $J = [j', j' + 2^q)$ that currently annotate the query positions i and j. The adversary proceeds in three steps:

- First, if I has cardinality exactly two, i.e., $I = [i', i' + 1]$, then the adversary replaces I with $I \leftarrow [i', i']$ if i' is even, or $I \leftarrow [i' + 1, i' + 1]$ if i' is odd. This fixes position i to an even symbol. The same procedure is performed for J.
- From now on, $I = [i', i' + 2^p)$ and $J = [j', j' + 2^q)$ denote the (possibly new) ranges after the first step. In the second step, the adversary aims to make the ranges disjoint (if they are not disjoint already), which is required in order to decide the query answer. This is done by discarding half of each range in a way that depends on the positions of their respective centers. Assume that $i' + 2^{p-1} \leq j' + 2^{q-1}$ (i.e., the center of I is at most the center of J; the opposite case works analogously), then the adversary replaces I with its left half by assigning $I \leftarrow [i', i' + \lceil 2^{p-1} \rceil)$ and J with its right half by assigning $J \leftarrow [j' + \lfloor 2^{q-1} \rfloor, j' + 2^q)$. It can be readily verified that $i' + 2^{p-1} \leq j' + 2^{q-1}$ implies $i' + \lceil 2^{p-1} \rceil \leq j' + \lfloor 2^{q-1} \rfloor$, unless both p and q are 0. Hence the new ranges are either disjoint, or both of them have cardinality one. Also, the cardinality of each range was reduced by exactly half, unless the range already had cardinality one (in which case it has not been changed).
- Now either both ranges are of cardinality one, or they are entirely disjoint. Either way, by answering that $x[i] < x[j]$ if and only if $\min(I) < \min(J)$, the adversary behaves consistently with the ranges.

Assume that some algorithm performs fewer than $n \lceil \log_2 \sigma \rceil / 2$ symbol comparisons. After the algorithm terminates, every position is either fixed to an even symbol, or it is annotated with a range of cardinality at least two, which also contains an even symbol. Hence there is always a string over the even symbols that is consistent with the answers given by the adversary. Each comparison cuts the range of at most two positions in half (either with the first or the second step from the list above). A position from $(2\sigma, n + 2\sigma]$ has initial range $[1, 2\sigma']$, and it gets fixed as soon its range has cardinality one. Hence a position must be involved in $\log_2(2\sigma') = \lfloor \log_2 \sigma \rfloor + 1 \geq \lceil \log_2 \sigma \rceil$ comparisons before getting fixed. After fewer than $n \lceil \log_2 \sigma \rceil / 2$ comparisons, there must be fewer than $2 \cdot (n \lceil \log_2 \sigma \rceil / 2) / \lceil \log_2 \sigma \rceil = n$ fixed positions in $(2\sigma, n + 2\sigma]$. Thus, there is at least one position that is not fixed, and we can fix it to an odd symbol from its range of cardinality at least two. We have shown that, after fewer than $n \lceil \log_2 \sigma \rceil / 2$ comparisons, the set of strings consistent with the given answers contains both a string over the even symbols, and also a string that contains an odd symbol. Hence the algorithm has not solved the problem yet. □

Lemma 2.7. *For any fixed $\sigma, n \in \mathbb{N}^+$ with $\sigma \leq \frac{n}{2}$, there is no deterministic algorithm that solves Alphabet Size Testing (Problem 2.2) over general ordered alphabet in fewer than $\frac{1}{4} \cdot n \lfloor \log_2 \sigma \rfloor$ symbol order comparisons in the worst case.*

Proof. We use the same adversary as in the proof of Lemma 2.6. The internally used alphabet is $[1, 2\sigma]$, and the constructed string will be of length n. Initially, all the positions $i \in [1, n]$ are annotated with the full range $[1, 2\sigma']$, where $\sigma' = 2^{\lfloor \log_2 \sigma \rfloor}$. Apart from that, the adversary works exactly like in the proof of Lemma 2.6. Particularly, whenever a position gets fixed, it gets fixed to an even symbol. After the algorithm terminates, each position is fixed to an even symbol, or annotated with a range of cardinality at least two, which also contains an even symbol. Hence there is always a string over the even symbols that is consistent with the answers given by the adversary. Note that such a string contains at most σ distinct symbols.

Now assume that an algorithm performs fewer than $n \lceil \log_2 \sigma \rceil / 4$ symbol comparisons. A position must be involved in $\lceil \log_2 \sigma \rceil$ comparisons before it gets fixed, and each comparison involves only two positions. After fewer than $n \lceil \log_2 \sigma \rceil / 4$ comparisons, there must be fewer than $2 \cdot (n \lceil \log_2 \sigma \rceil / 4) / \lceil \log_2 \sigma \rceil = \frac{n}{2}$ fixed positions. Hence there are at least $\frac{n}{2}$ positions that are not fixed, i.e., positions that are annotated with a range of cardinality at least two. We retrospectively inflate the alphabet from $[1, 2\sigma]$ to $[n, 2n\sigma]$ as follows. Every position annotated with a range $[i', i' + 2^p - 1]$ is now annotated with a range $[ni', ni' + n2^p - n]$. A position is fixed after the inflation if and only if it was fixed before the inflation. Also, the inflated ranges are clearly still consistent with the given answers. A position that is not fixed is annotated with a range $[ni', ni' + n2^p - n] \supseteq [ni', ni' + n]$ of cardinality $n + 1$. Hence we can fix the position to a symbol that has no other occurrence in x. By doing this for the at least $\frac{n}{2}$ positions that are not fixed, we construct a string that contains at least $\frac{n}{2}$ distinct symbols. Hence the algorithm has not solved the problem yet. □

2.3.3 Lower Bounds for General Unordered Alphabet

Now we show the lower bounds from Theorem 2.5(*ii*) and Theorem 2.3(*ii*), respectively restated in Lemma 2.8 and Lemma 2.9. The general idea is the same as the one for general ordered alphabets. However, since inequality is not transitive, it is easier for the adversary to force a large number of comparisons, and the details of the proof are simpler.

Lemma 2.8. *For any fixed $\sigma, n \in \mathbb{N}^+$, there is no deterministic algorithm that solves Alphabet Set Testing (Problem 2.4) over general unordered alphabet in fewer than $\frac{1}{2} \cdot n\sigma$ symbol equality comparisons in the worst case.*

Proof. We use an adversary similar to the one in the proof of Lemma 2.6. Internally, the adversary again works with the integer alphabet $\Sigma \cup \Sigma' = [1, 2\sigma]$ with $\Sigma = \{2, 4, 6, \ldots, 2\sigma\}$ (the even symbols) and $\Sigma' = \{1, 3, 5, \ldots, 2\sigma - 1\}$ (the odd symbols). The adversary maintains a family of strings of length $n + 2\sigma$. In order to keep track of the given answers, it annotates each position $i \in [1, n + 2\sigma]$ with an initially empty set $\mathcal{L}_i$ of conflicting positions. For every $j \in [1, n + 2\sigma]$, it holds $j \in \mathcal{L}_i$ if and only if the adversary has previously answered that $x[i] \neq x[j]$. Additionally, each position i can get fixed to a symbol $\gamma(i) \in [1, 2\sigma]$, but only if no position $j \in \mathcal{L}_i$ has been fixed to the same symbol. Initially, the positions $i \in (2\sigma, n + 2\sigma]$ are not fixed, which is formally expressed by $\gamma(i) = 0$. The positions $i \in [1, \sigma]$ are fixed to $\gamma(i) = 2i$, and the positions $i \in (\sigma, 2\sigma]$ are fixed to $\gamma(i) = 2(i - \sigma) - 1$ (as required by the problem definition). A string $x[1..n + 2\sigma]$ over $\Sigma \cup \Sigma'$ is consistent with the given answers if for any two positions $i, j \in [1, n + 2\sigma]$ it holds $j \in \mathcal{L}_i \implies x[i] \neq x[j]$ and $\gamma(i) > 0 \implies x[i] = \gamma(i)$.

Whenever the adversary has to answer whether or not $x[i] = x[j]$ for some query positions $i, j \in [1, n]$ with $i \neq j$, it first checks if $|\mathcal{L}_i| = \sigma - 1$. If this is the case, and also i has not been fixed yet, then it fixes i to an even symbol such that afterwards $\gamma(i) \in \{2, 4, 6, \ldots, 2\sigma\} \setminus \{\gamma(j) \mid j \in \mathcal{L}_i\}$ (there is always at least one possible symbol due to $|\mathcal{L}_i| = \sigma - 1$). If $|\mathcal{L}_j| = \sigma - 1$, then j gets fixed in the same way. After this, the adversary answers that $x[i] = x[j]$ if and only if $\gamma(i) = \gamma(j) \neq 0$, i.e., if both positions are fixed to the same symbol. Otherwise, it answers that $x[i] \neq x[j]$ and inserts i into $\mathcal{L}_j$ and j into $\mathcal{L}_i$.

Assume that some algorithm performs fewer than $n\sigma/2$ symbol comparisons. After the algorithm terminates, every position is either fixed to an even symbol, or its set of conflicting positions is of cardinality less than σ. If we consider the non-fixed positions one at a time (in arbitrary order), and for each such position i fix it to an even symbol such that $\gamma(i) \in \{2, 4, 6, \ldots, 2\sigma\} \setminus \{\gamma(j) \mid j \in \mathcal{L}_i\}$, then we obtain a string over the even symbols that is consistent with the given answers. During the algorithm execution, a position gets fixed exactly when it is involved in a comparison for the σth time. Every comparison involves only two positions. Hence, after fewer than $n\sigma/2$ comparisons, there are fewer than $2 \cdot (n\sigma/2)/\sigma = n$ fixed positions. Since at least one position is not fixed, we can fix it to an odd symbol (and all remaining positions to even symbols in the same way as before). Thus, the algorithm has not solved the problem yet. □

Lemma 2.9. *For any fixed $\sigma, n \in \mathbb{N}^+$ with $\sigma \leq \frac{n}{2}$, there is no deterministic algorithm that solves Alphabet Size Testing (Problem 2.2) over general unordered alphabet in fewer than $\frac{1}{4} \cdot n\sigma$ symbol equality comparisons in the worst case.*

Proof. We use the adversary from the proof of Lemma 2.8. This time, it maintains a family of strings of length n, and initially no position is fixed. Apart from that, the adversary functions exactly as described before. Assume that some algorithm performs fewer than $n\sigma/4$ symbol comparisons. By the same reasoning as in the proof of Lemma 2.8, there is a string over the even symbols that is consistent with the given answers, and this string contains at most σ distinct symbols. During the algorithm execution, a position gets fixed exactly when it is involved in a comparison for the σth time. Every comparison involves only two positions. Hence, after fewer than $n\sigma/4$ comparisons, there are fewer than $2 \cdot (n\sigma/4)/\sigma = \frac{n}{2}$ fixed positions. We can fix each of the at least $\frac{n}{2}$ non-fixed positions to an entirely new symbol that has no other occurrence in the string. Hence there is a string that contains at least $\frac{n}{2}$ distinct symbols and is consistent with the answers given by the adversary. This means that the algorithm has not solved the problem yet. □

I

Lempel-Ziv Compression

Lempel-Ziv Compression

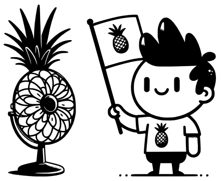
I

Introduction and Related Work

In this part of the dissertation, we consider the problem of lossless data compression. With the ever-growing amount of data generated, it has become almost inevitable to store the data in compressed form. One of the most evident examples of this is the amount of sequenced DNA. The human genome is a string of around three billion base pairs, and ambitious sequencing projects like the 100000 genome project [Weba] already sequenced the full genome of tens of thousands of individuals [Webb]. Repositories like the NCBI sequence read archive [Webe] maintain petabytes of DNA and are rapidly growing [Kat+21]. While a single human genome is not very well compressible (there are few long DNA strings that have multiple occurrences in the genome), a collection of multiple genomes is highly-compressible. The reasons for this is that, vaguely speaking, any two human individuals share the vast majority of their DNA. It is often stated that the similarity between two human genomes is as high as 99.9%, extrapolated from a difference rate of 1 in 1250 base pairs [Ven+01]. This does not capture larger structural variations in the genome, and more recent estimates claim a more conservative similarity of only around 99.5% [Lev+07]. Regardless of the precise number, applications like the compression of DNA collections can benefit from techniques that exploit repetitiveness. One of the most widely spread compression techniques with this property is Lempel-Ziv compression.

Compression by Lempel-Ziv Factorization The Lempel-Ziv (LZ) factorization [LZ76] of a string $x[1..n]$ decomposes it into a series of z phrases $x = f_1 f_2 \dots f_z$. Each phrase $f_{i'} = x[i..i + |f_{i'}|)$ with $i' \in [1, z]$ and $i = 1 + \sum_{k=1}^{i'-1} |f_k|$ is either a single symbol $x[i]$ that does not occur in $x[1..i)$ (a *literal phrase*), or otherwise it is the longest prefix of $x[i..n]$ that has at least one previous occurrence $x[j..j + |f_{i'}|) = f_{i'}$ with $j \in [1, i)$ (a *referencing phrase*). This is Storer and Szymanski's version of the factorization [SS82]. Position i is *the destination* of $f_{i'}$. If $f_{i'}$ is a referencing phrase, then j is *a source* of $f_{i'}$. For example, the string `ananas-fan` gets factorized as

1	2	3	4	5	6	7	8	9	10
`a`	`n`	`a`	`n`	`a`	`s`	`-`	`f`	`a`	`n`.
f_1	f_2		f_3		f_4	f_5	f_6	f_7	

[1]

[1] Two ananas-fans, envisioned by DALL·E [Bet+23].

Here, $f_3 = \texttt{ana}$ is a referencing phrase at destination 3 with unique source 1. We stress that a phrase is allowed to overlap its previous occurrence, as is the case for occurrences $x[1..3]$ and $x[3..5]$ of f_3. Phrase $f_7 = \texttt{an}$ is a referencing phrase at destination 9 with sources 1 and 3. All the other phrases are literal phrases. When factorizing the string from left to right, the next phrase is always uniquely defined due to the maximal length property of referencing phrases, and hence each string admits exactly one LZ factorization. In an LZ-*like* factorization, we relax the definition of referencing phrases such that they can be of arbitrary length. A string may admit multiple LZ-like factorizations, but the (exact) LZ factorization is the unique one that minimizes the number of phrases [SS82].

Compression can be achieved by encoding the phrases of an LZ(-like) factorization as follows. A length-ℓ referencing phrase at destination i gets encoded as (d, ℓ), where $d \in [1, i)$ is the distance to a source of the phrase, i.e., $x[i - d..i - d + \ell) = x[i..i + \ell)$. Each literal phrase $x[i]$ gets encoded as a pair $(0, x[i])$. For the previous example, we choose 3 as the source of f_6 and obtain the encoding shown below.

$$(0, \texttt{a})(0, \texttt{n})(2, 3)(0, \texttt{s})(0, \texttt{-})(0, \texttt{f})(6, 2)$$

Given this list of pairs, it is easy to reconstruct the string from left to right in $\mathcal{O}(n)$ time (interpreting each referencing phrase (d, ℓ) as an instruction to go back d positions and copy ℓ symbols). If the string is over polynomial integer alphabet, then each phrase (regardless if it is referencing or literal) is an integer pair that can be encoded naively in $\mathcal{O}(\log n)$ bits. This way, one can store the string in $\mathcal{O}(z \log n)$ bits of space, where z is the number of phrases. Of course, this requires that we know at least one source of each referencing phrase. Hence, whenever we talk about computing the LZ factorization, we actually mean the task of computing the factorization *and* finding a source for each referencing phrase.

Background and Related Work

The LZ factorization was first introduced in 1976, when Lempel and Ziv proposed the number z of phrases of the factorization as a complexity measure for strings (aimed at evaluating the “randomness” of a string) [LZ76]. Over 45 years later, it is still a standard measure for dictionary-based compression (see, e.g., [GNP18]). This is because z can be computed in linear time (e.g., [CI08a]), and because it lower-bounds other measures like the size of the smallest grammar that generates the string [Ryt03, Cha+05]. Many compressibility measures are within polylogarithmic factors of z, e.g., the size of the smallest bidirectional macro scheme [GNP18], the number of unary runs in the Burrows-Wheeler transform [KK22], the size of the smallest string attractor [KP18], and the normalized substring complexity [Ras+13, KNP20].

Apart from introducing z as a measure, Ziv and Lempel also used their factorization to derive the compression scheme now commonly known as LZ77 [ZL77]. Nowadays, the LZ factorization – despite being introduced in 1976 – is often referred to as the LZ77 factorization. Since then, the LZ factorization has become a cornerstone of practical compression; LZ-based techniques are a crucial ingredient of the most commonly used compressors (e.g., gzip, 7zip, rar, brotli) and compressed formats (e.g., PDF, PNG). There has been extensive work aimed at computing (versions of) the LZ factorization (and we list only a few examples). This includes parallel algorithms [CR91, Nao91, FM95, SZ13, Shu18], online and streaming algorithms [OS08, Sta12, Yam+14, PP15, Bil+17], external memory algorithms [KKP14, Bel+16], and approximation algorithms [Fis+15a, Kos+20]. Another line of research improves the compression rate by optimizing the encoding of phrases [ALU02, Cro+12, CLM13, FNV13, Lar14, BP16, KNP22, EFP23]. There are several text indices that rely on LZ compression [KS98, KN13, Fer+14, Gag+14, GGP15, Val16, Bil+18, BGS20, Nis+20, Sun+21]. Despite this plethora of results, the ever-increasing relevance of compression still drives the development of new ways to compute LZ(-like) factorizations [Köp21, Wu21, Gag22, NT22, SI22, HRB23].

Computing the Lempel-Ziv Factorization There are many word RAM algorithms for computing the LZ factorization. Most commonly, it is assumed that the string $x[1..n]$ is over polynomial integer alphabet. Whenever the string is packed over $[0, \sigma)$, it occupies $\mathcal{O}(n \log \sigma)$ bits of space. If the word-width is not too large with respect to the input, say, $w = \Theta(\log n)$, then the $\mathcal{O}(n \log \sigma)$ bits are equivalent to $\Theta(n / \log_\sigma n)$ words of space. In this case, $\Omega(n / \log_\sigma n)$ is a trivial lower bound on the time and words of space needed to compute the LZ factorization. Many algorithms take $\mathcal{O}(n)$ time (see, e.g., [CI08a, GB13, KKP13b, GB14, FIK15]) or $\mathcal{O}(n \log \sigma)$ bits of working space (see, e.g., [OS08, OG11, Sta12, KKP13a, Yam+14, Kos15a, BP16, KS16, Ell23]), and at least one algorithm achieves both [Fis+18]. Kempa [Kem19] introduced an algorithm that takes $\mathcal{O}(n / \log_\sigma n + r \log^9 n + z \log^9 n)$ time and $\mathcal{O}(n / \log_\sigma n + r \log^8 n)$ words of space, where $r = \mathcal{O}(z \log^2 n)$ [KK22] is the number of unary runs in the Burrows-Wheeler transform. The algorithm presented in Chapter 4 achieves $\mathcal{O}(n / \log_\sigma n + z \log^{3+\epsilon} z)$ time and $\mathcal{O}(n \log \sigma)$ bits of space (for arbitrary constant $\epsilon \in \mathbb{R}^+$). However, it remains unknown whether or not $\mathcal{O}(n / \log_\sigma n)$ time can be achieved. (Peculiarly, there is an $\mathcal{O}((n / \log_\sigma n) \cdot \text{polylog}(\log n))$ time algorithm [JSS15] for the related LZ78 factorization [ZL78]).

Rightmost Lempel-Ziv While the encoding of phrases as distance-length pairs as described before is indeed used by practical compressors, storing each pair in a fixed number of bits is far from optimal. Usually, the compression rate is further optimized by applying a variable-length code to the pairs. For example, one could use a universal code like Elias delta [Eli75] to encode the distance component of each referencing phrase. Such codes often assign longer code words to larger integers, and hence minimizing the distance component of the referencing phrases can improve the compression rate. This motivates the problem of computing the *rightmost* LZ factorization, where for each referencing phrases we have to find the maximal source (i.e., the one at minimal distance).

The first theoretical result on computing the rightmost LZ factorization is by Amir et al. [ALU02] and uses $\mathcal{O}(n \log n)$ time and words of space. Larsson et al. [Lar14] presented an online algorithm with the same time and space complexity. Crochemore et al. [CLM13] gave the first approximation algorithm, which runs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space and finds the rightmost *equal-cost* position for each phrase, meaning it takes the same number of bits to encode as the rightmost position (as long as the used code satisfies the right properties). Later, Bille et al. [Bil+17] gave an $(1+\epsilon)$-approximation algorithm for the rightmost factorization in $\mathcal{O}(n(\log z + \log \log n))$ time and linear working space, where each reported source is allowed to be $(1+\epsilon)$ times further away than the rightmost one. The first exact algorithm to achieve $o(n \log n)$ time is by Ferragina et al. [FNV13] and runs in $\mathcal{O}(n(1 + \log \sigma / \log \log n))$

time and $\mathcal{O}(n)$ words of space. This was improved by Belazzougui and Puglisi [BP16] with an algorithm using only $\mathcal{O}(n \log \sigma)$ bits of space and achieving $\mathcal{O}(n(\log\log\sigma + \log\sigma/\sqrt{\log n}))$ deterministic time or $\mathcal{O}(n(1+\log\sigma/\sqrt{\log n}))$ time with randomization, which is the current state of the art.

LZ-End Factorization The LZ factorization minimizes the number of phrases among all the LZ-like factorizations, but it has some disadvantages if the practical application requires efficient computation on the compressed text. In many cases, LZ compressed data has to be fully decompressed in order to perform even simple computational tasks. This motivates LZ-End [KN10, KN13], a family of LZ-like factorizations that supports efficient compressed indexing. That is, when combined with additional data structures, it allows fast random access to the string without decompressing it first. This is achieved by enforcing the following property. Each referencing phrase $f_{i'}$ must have a previous occurrence as a suffix of $f_1 f_2 \ldots f_{j'}$ for some $j' \in [1, i')$. We then call $j = 1 - |f_{i'}| + \sum_{k=1}^{j'} |f_k|$ an *LZ-End aligned* source of $f_{i'}$. This definition is slightly different from the original one, but it leads to a simpler description of algorithms (and the presented results can easily be adapted to the original definition). Computing a *rightmost* LZ-End factorization is the task of finding the maximal LZ-End aligned source of each referencing phrase.

LZ-End was introduced by Kreft and Navarro [KN10, KN13], who also provided a compressed index based on LZ-End that allows fast pattern matching and substring extraction. Their original definition requires referencing phrases to be of maximal length (like in the standard LZ factorization). This results in a uniquely defined factorization, which we call the *greedy* LZ-End factorization. It can be computed in linear time and space [KK17], and the number of its phrases is within an $\mathcal{O}(\log^2 n/\log\log n)$ factor of the exact LZ factorization [KS22, GKM23]. Unlike normal LZ, the greedy LZ-End factorization does not necessarily minimize the number of phrases among all the possible LZ-End factorizations. Bannai et al. [Ban+23] proved that computing the optimal LZ-End factorization (with minimal number of phrases) is NP-hard and gave a lower bound of 2 on the approximation ratio of optimal LZ-End to greedy LZ-End. There is a data structure that requires space linear in the number of greedy LZ-End phrases and allows polylogarithmic time random access on the string. A generalization of this data structure also answers LCE queries in polylogarithmic time [KS22].

Contributions

This part of the dissertation consists of three chapters, each of which introduces new results in the field of Lempel-Ziv compression. In Chapter 3, we show that computing the LZ factorization over general alphabet cannot be done faster than reducing the alphabet to effective integer alphabet with Lemma 2.1. Hence we do not need to design dedicated algorithms for LZ over general alphabets. Instead, we focus on the most practical case where the string $x[1..n]$ is packed over $[0, \sigma)$. In Chapter 4, we introduce a new algorithm that computes the LZ factorization in this setting in $\mathcal{O}(n/\log_\sigma n + z\log^{3+\epsilon} z)$ time and $\mathcal{O}(n\log\sigma)$ bits of working space (for arbitrary constant $\epsilon \in \mathbb{R}^+$), significantly improving the best previously known bounds. As part of the solution, we also show how to compute an LZ-like factorization of at most $3z$ phrases in $\mathcal{O}(n/\log_\sigma n)$ time and $\mathcal{O}(n\log\sigma)$ bits of working space. Finally, in Chapter 5, we consider the problem of computing rightmost LZ(-like) factorizations. We show that the rightmost sources of some non-trivial subsets of the referencing phrases can be computed in linear and sometimes even sublinear time. As a prime example, we can resolve all the phrases of length $\Omega(\log^{6.66} n / \log^2 \sigma)$ in $\mathcal{O}(n / \log_\sigma n)$ time. We also provide the first algorithms for computing rightmost LZ-End factorizations.

Chapter 3

Lower Bounds for LZ Over General Alphabets 3

In this short chapter, we show that computing the LZ factorization of a string $x[1..n]$ that contains σ distinct symbols takes $\Omega(n \log \sigma)$ time over general ordered alphabet, and $\Omega(n\sigma)$ time over general unordered alphabet. Note that σ is also the number of literal phrases in the LZ factorization. Thus, if we require the LZ factorization to be computed in a way that allows us to distinguish between literal phrases and referencing phrases of length one, then the lower bounds follow directly from the lower bounds for Alphabet Size Testing (Problem 2.2 and Theorem 2.3). However, the bounds then only hold if the number σ of distinct symbols is not given as part of the input. Therefore, we introduce a computationally easier version of Lempel-Ziv that expects the set of literal phrases as part of the input (Problem 3.1). We use a reduction from Alphabet Set Testing (Problem 2.4 and Theorem 2.5) to show that this version of LZ still has the same lower bounds (Theorem 3.2).

Problem 3.1 (Lempel-Ziv Factorization with Known Literals)**.** Given a string $x[1..n]$ and the destinations $\tilde{\Sigma} = \{i \in [1, n] \mid \nexists j \in [1, i) : x[j] = x[i]\}$ of all literal LZ phrases, output the destination of each phrase of the LZ factorization.

Theorem 3.2. *Let $\sigma, n \in \mathbb{N}^+$ with $8 \leq \sigma \leq \frac{n}{5}$ be fixed. There is no algorithm that computes the Lempel-Ziv Factorization with Known Literals (Problem 3.1) of a string of length n that contains σ distinct symbols over*

(i) *general ordered alphabet in fewer than $\frac{n \log_2 \sigma}{256}$ symbol order comparisons, and*

(ii) *general unordered alphabet in fewer than $\frac{n\sigma}{256}$ symbol equality comparisons*

in the worst case.

3.1 Reducing Alphabet Set Testing to Lempel-Ziv

Below, we show that any instance of Alphabet Set Testing (Problem 2.4) can be reduced to an instance of Lempel-Ziv with Known Literals (Problem 3.1) that is of similar size. Hence Lempel-Ziv is at least as hard as Alphabet Set Testing. This reduction is quite similar to the work of Kosolobov [Kos15b], who shows that computing the Lempel-Ziv factorization (with unknown literals) requires $\Omega(n \log \sigma)$

comparisons over general ordered alphabet. We merely extend Kosolobov's reduction such that the positions of literals depend solely on the alphabet size.

Lemma 3.3. *Let $\sigma, n \in \mathbb{N}^+$. Any instance $x = u[1..\sigma]v[1..\sigma]y[1..n]$ of Alphabet Set Testing (Problem 2.4) can be reduced to computing the Lempel-Ziv Factorization with Known Literals (Problem 3.1) of a string $x'[1..4n + 8\sigma + 13]$ that contains $2\sigma + 4$ distinct symbols. The reduction requires no symbol comparisons.*

Proof. Let $\Sigma = \{u[i] \mid i \in [1, \sigma]\}$ and $\Sigma' = \{v[i] \mid i \in [1, \sigma]\}$. We know that $y \in (\Sigma \cup \Sigma')^n$, and we have to decide whether or not $y \in \Sigma^n$. We show that the problem can be solved by computing the LZ factorization of the string x' below. The symbols $\$_1$, $\$_2$, $\$_3$, and $\texttt{\#}$ are distinct separators that do not occur in u or v. In case of general ordered alphabet, we define the order $\forall i \in [1, 2\sigma] : \$_1 < \$_2 < \$_3 < \texttt{\#} < x[i]$. It can be readily verified that x' is of length $4n + 8\sigma + 13$ and contains exactly $2\sigma + 4$ distinct symbols. Also, it is easy to perform a symbol comparison between some $x'[i]$ and $x'[j]$ by performing at most one symbol comparison between symbols of x.

$$\begin{aligned}
v_{\text{single}} &= \texttt{\#}v[1]\texttt{\#}v[2]\texttt{\#}v[3]\texttt{\#} \ldots v[\sigma]\texttt{\#} \\
v_{\text{square}} &= v[1]^2\, v[2]^2\, v[3]^2 \ \ldots v[\sigma]^2 \\
u_{\text{square}} &= \texttt{\#}^5 u[1]^2\texttt{\#\#}u[2]^2\texttt{\#\#}u[3]^2\texttt{\#\#}\ldots\texttt{\#\#}u[\sigma]^2\texttt{\#\#} \\
y_{\text{square}} &= \texttt{\#}y[1]^2\texttt{\#\#}y[2]^2\texttt{\#\#}y[3]^2\texttt{\#\#}\ldots\texttt{\#\#}y[n]^2\texttt{\#}^5 \\
x' &= v_{\text{single}} \cdot \$_1 \cdot v_{\text{square}} \cdot \$_2 \cdot u_{\text{square}} \cdot \$_3 \cdot y_{\text{square}}
\end{aligned}$$

Let $\hat{x} = |x'|$ and $\hat{y} = |y_{\text{square}}|$. Let $g_1 \ldots g_{z'} \cdot \$_3 \cdot f_1 \ldots f_z$ be the LZ factorization of x' (where $\$_3$ is a literal phrases because it is unique in x'). We start by showing that the destinations of literal phrases only depend on σ. First, note that every length-two substring of $x'(\hat{x} - \hat{y}..\hat{x}] = y_{\text{square}}$ has an occurrence in $x'[1..\hat{x} - \hat{y}]$. Thus, all the phrases $f_1, \ldots, f_{z-1}$ must be of length at least two. Next, we show that $|f_z| \geq 3$. Substring $x'[\hat{x} - 5..\hat{x} - 2] = y[n]\texttt{\#\#\#}$ has exactly one occurrence in x'. Thus $y[n]\texttt{\#\#\#}$ has no previous occurrence and cannot be contained in a single phrase, which means that there must be a phrase with destination $\hat{x} - i$ for some $i \in [2, 4]$. Since $x'[\hat{x} - i..\hat{x}] = \texttt{\#}^{i+1}$ has a previous occurrence as a prefix of u_{square}, it follows that $f_z = x'[\hat{x} - i..\hat{x}]$ and therefore $|f_z| = i + 1 \geq 3$. We have shown that there are no literal phrases in $x'(\hat{x} - \hat{y}..\hat{x}]$.

The literal phrases in $x'[1..\hat{x} - \hat{y}]$ are at fixed positions that depend only on σ. More precisely, the separator literals are at positions 1, $2\sigma + 2$, $4\sigma + 3$, and $8\sigma + 9$. The literals corresponding to symbols from Σ' are at positions $\{2j \mid j \in [1, \sigma]\}$ (which lie within the prefix v_{single} of x'). The literals corresponding to symbols from Σ are at positions $\{4j + 4\sigma + 5 \mid j \in [1, \sigma]\}$ (which lie within substring u_{square} of x'). Hence all literals can be computed without symbol comparisons.

We still have to show that computing the LZ factorization of x' actually solves the instance of Alphabet Set Testing. We claim that y contains a symbol from Σ' if and only if at least one of the phrases $f_1, \ldots, f_z$ has length exactly two. (Problem 3.1 gives the destination of all phrases as output, but it is trivial to compute the lengths from the destinations.) Assume that all symbols in y are from Σ. Then it is easy to see that every length-three substring of y_{square} has an occurrence in u_{square}. Hence all the phrases $f_1, \ldots, f_{z-1}$ are of length at least three (and we have already shown that also $|f_z| \geq 3$). It remains to be shown that, if y contains at least one symbol from Σ', then at least one of the phrases $f_1, \dots, f_{z-1}$ has length exactly two. Assume that some symbol $a \in \Sigma'$ occurs in y, and choose the minimal $i \in (\hat{x} - \hat{y}, \hat{x}]$ such that $x'[i] = a$. Then $x'[i-1..i+3] = \mathtt{\#}aa\mathtt{\#\#}$. Note that $x'[i-1..i+1] = \mathtt{\#}aa$ has no previous occurrence, and thus there must be some phrase with destination i or $i+1$. Since neither $x'[i..i+2] = aa\mathtt{\#}$ nor $x'[i+1..i+3] = a\mathtt{\#\#}$ has a previous occurrence, and both $x'[i..i+1] = aa$ and $x'[i+1..i+2] = a\mathtt{\#}$ do have a previous occurrence, it follows that a phrase with destination i or $i+1$ must be of length exactly two. □

Now we use the reduction to show Theorem 3.2 (*i*) and Theorem 3.2 (*ii*), respectively restated in Lemma 3.4 and Lemma 3.5.

Lemma 3.4. *Let $\sigma, n \in \mathbb{N}^+$ with $8 \leq \sigma \leq \frac{n}{5}$ be fixed. For general ordered alphabet, there is no algorithm that computes the Lempel-Ziv Factorization with Known Literals (Problem 3.1) of a string $x[1..n]$ that contains σ distinct symbols in fewer than $\frac{n \log_2 \sigma}{256}$ symbol order comparisons in the worst case.*

Proof. Let $\sigma' = \lfloor(\sigma - 4)/2\rfloor$ and $n' = \lfloor(n - 8\sigma' - 14)/4\rfloor$. By Lemma 3.3, any instance $x' = u[1..\sigma']v[1..\sigma']y[1..n']$ of Alphabet Set Testing can be reduced to computing the LZ factorization of some string $x''[1..n'']$ with $2\sigma' + 4 \in \{\sigma - 1, \sigma\}$ known literals, where $n'' = 4n' + 8\sigma' + 13 \leq n - 1$. In order to obtain a string of length exactly n, we use the separator symbol $\mathtt{\#}$ from the proof of Lemma 3.3 as padding. If $2\sigma' + 4 = \sigma$, then we use $x[1..n] = \mathtt{\#}^{(n-n'')} \cdot x''[1..n'']$. Otherwise, it holds $2\sigma' + 4 = \sigma - 1$, and we introduce an entirely new symbol $\mathsf{\S}$ that is smaller than all other symbols. We then use $x[1..n] = \mathsf{\S}^{(n-n'')} \cdot x''[1..n'']$. It is easy to see that this does not affect the reduction from Lemma 3.3. Hence the instance of Alphabet Set Testing can be solved by computing the LZ factorization of $x[1..n]$ with exactly σ known literals. By Theorem 2.5 (*ii*), this requires at least $n' \lceil \log_2 \sigma' \rceil / 2$ symbol comparisons, where $8 \leq \sigma \leq \frac{n}{5}$ implies

$$n' = \left\lfloor \frac{n - 8\left\lfloor \frac{\sigma-4}{2} \right\rfloor - 14}{4} \right\rfloor \geq \left\lfloor \frac{n - 4\sigma + 2}{4} \right\rfloor \geq \frac{n-1}{4} - \sigma \geq \frac{n-5}{20} \geq \frac{n}{24}, \text{ and}$$

$$\sigma' = \left\lfloor \frac{\sigma - 4}{2} \right\rfloor \geq \frac{\sigma - 5}{2} \geq \sigma^{3/16}.$$

Hence it holds $n' \lceil \log_2 \sigma' \rceil / 2 \geq \frac{n}{24} \cdot \frac{3 \log_2 \sigma}{16} \cdot \frac{1}{2} = \frac{n \log_2 \sigma}{256}$. □

Lemma 3.5. *Let $\sigma, n \in \mathbb{N}^+$ with $8 \leq \sigma \leq \frac{n}{5}$. For general unordered alphabet, there is no algorithm that computes the Lempel-Ziv Factorization with Known Literals (Problem 3.1) of a string $x[1..n]$ that contains σ distinct symbols in fewer than $\frac{n\sigma}{256}$ symbol equality comparisons in the worst case.*

Proof. We use the same reduction as in the proof of Lemma 3.4. This time, we observe that $\sigma' \geq \frac{\sigma-5}{2} \geq \frac{3\sigma}{16}$ for $\sigma \geq 8$. By Theorem 2.5 (*ii*), we have to perform at least $n'\sigma'/2 \geq \frac{n}{24} \cdot \frac{3\sigma}{16} \cdot \frac{1}{2} = \frac{n\sigma}{256}$ comparisons. □

3.2 Lower Bounds for Large Alphabets

The lower bound from Theorem 3.2 holds for $\sigma \leq \frac{n}{5}$, which naturally raises the question whether faster computation is possible for $\sigma > \frac{n}{5}$. With a slight modification of the proof, we can obtain lower bounds for $\sigma = \frac{(1+\epsilon)n}{5}$, where $\epsilon \in \mathbb{R}^+$ with $\epsilon < 4$ is an arbitrary constant. We create a string $x[1..n] = x'x''$ that consists of a prefix x' of length $k = \frac{\epsilon n}{4}$ and a suffix x'' of length $k' = n - k = \frac{(4-\epsilon)n}{4}$. The prefix $x'[1..k]$ contains k distinct symbols (with the purpose of inflating the size of the alphabet). The suffix $x''[1..k']$ contains $\frac{k'}{5}$ distinct symbols that are not present in x'. Hence x contains $k + \frac{k'}{5} = \sigma$ distinct symbols. Since x' and x'' are over disjoint alphabets, computing the LZ factorization of x requires at least as many comparisons as separately computing the respective LZ factorizations of x' and x''.

Computing the LZ factorization of x'' over general ordered alphabet requires at least $k' \cdot \log_2(k'/5)/256$ comparisons by Theorem 3.2. It holds $k' = \frac{(4-\epsilon)n}{4} = \frac{20-5\epsilon}{4+4\epsilon} \cdot \sigma$, and thus it is easy to see that the previous term can be bounded from below by $c_1 \cdot n \log_2 \sigma$, where $c_1 \in \mathbb{R}^+$ is a constant that depends solely on ϵ. By similar reasoning, computing the LZ factorization of x'' over general unordered alphabet requires at least $c_2 \cdot n\sigma$ comparisons, where $c_2 \in \mathbb{R}^+$ is a constant that depends solely on ϵ. We implicitly assumed that $\frac{(1+\epsilon)n}{5}$ and $\frac{\epsilon n}{4}$ are integers. This assumption can easily be avoided by rounding and adjusting the constants. By choosing the minimum $c = \min(c_1, c_2)$, we obtain the following result.

Corollary 3.6. *Let $\epsilon \in \mathbb{R}^+$ with $\epsilon < 4$ be constant. There are constants $c, n_0 \in \mathbb{R}^+$ such that the statement below holds for any $n \in \mathbb{N}^+$ with $n \geq n_0$.*

There is no algorithm that computes the Lempel-Ziv Factorization with Known Literals (Problem 3.1) of a string of length n that contains $\sigma = \left\lfloor \frac{(1+\epsilon)n}{5} \right\rfloor$ distinct symbols over

(i) *general ordered alphabet in fewer than $c \cdot n \log_2 \sigma$ symbol order comparisons, and*

(ii) *general unordered alphabet in fewer than $c \cdot n\sigma$ symbol equality comparisons*

in the worst case.

Chapter 4

4

Sublinear Time Lempel-Ziv Factorization of Packed Strings

As discussed in the introduction of this part of the dissertation, there are many linear time algorithms that compute the LZ factorization over polynomial integer alphabet. If the string $x[1..n]$ is packed over integer alphabet $[0, \sigma)$ (which is arguably the most practical setting), the number of words occupied by the string is $\mathcal{O}(n/\log_\sigma n)$. Hence reading the string takes only $\mathcal{O}(n/\log_\sigma n)$ time, which is *sublinear* in n for sufficiently small σ. Lempel and Ziv showed that the LZ factorization of a length-n string over an alphabet of size σ consists of $z = \mathcal{O}(n/\log_\sigma n)$ phrases (see [LZ76, Theorem 2] or Lemma 4.3 below). Thus $\mathcal{O}(n/\log_\sigma n)$ time suffices for writing the LZ factorization as a sequence of distance-length pairs (as defined in the beginning of Part I). This naturally raises the question whether $\mathcal{O}(n/\log_\sigma n)$ time is also enough for computing the factorization. So far, there is neither a lower bound nor an algorithm that answers this question. The only previous result in this direction is by Kempa [Kem19], who introduced an algorithm that computes the LZ factorization in $\mathcal{O}(n/\log_\sigma n + r\log^9 n + z\log^9 n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^8 n)$ words of space, where $r = \mathcal{O}(z\log^2 n)$ [KK22] is the number of unary runs in the Burrows-Wheeler transform. In this chapter, we present a more efficient solution.

Contributions We propose new deterministic algorithms for computing LZ(-like) factorizations, summarized by the theorems below.

Theorem 4.1. *Let $x[1..n]$ be packed over $[0, \sigma)$. If the LZ factorization of x consists of z phrases, then an LZ-like factorization of x that consists of at most $3z$ phrases can be computed in $\mathcal{O}(n/\log_\sigma n)$ time and $\mathcal{O}(n\log\sigma)$ bits of space.*

Theorem 4.2. *Let $x[1..n]$ be packed over $[0, \sigma)$, and let $\epsilon \in \mathbb{R}^+$ be an arbitrarily small positive constant. If the LZ factorization of x consists of z phrases, then it can be computed in $\mathcal{O}(n/\log_\sigma n + z\log^{3+\epsilon} z)$ time and $\mathcal{O}(n\log\sigma)$ bits of space.*

The space asymptotically matches the space for storing the string. The time for the exact factorization is optimal if $w = \Theta(\log n)$ and $z = \mathcal{O}(n\log\sigma/\log^{4+\epsilon} n)$, i.e., for strings that compress well, and on a word RAM with at most logarithmically sized words (which is a standard assumption). The remainder of the chapter is structured as follows. We first provide some auxiliary lemmas in Section 4.1. Next, we propose the new algorithm for computing the 3-approximate factorization in Section 4.2. It achieves its efficiency by combining string synchronizing sets [KK19], an LCE data structure [KK19], and a classic linear time algorithm for the exact LZ factorization [CI08a]. In Section 4.3, we introduce the new algorithm for the exact LZ factorization. It samples a small number of positions, and then (co-)lexicographically sorts the suffixes and prefixes that respectively start and end at these positions. The exact sampling scheme depends on the 3-approximate factorization. The sorted suffixes and prefixes can be used to reduce the computation of the LZ factorization to two-dimensional orthogonal range reporting, which ultimately results in the claimed bounds. Finally, we show how to adapt the algorithm to compute the LZ factorization without overlaps in Section 4.3.2.

4.1 Auxiliary Lemmas

As mentioned earlier, the number of LZ phrases is at most $\mathcal{O}(n/\log_\sigma n)$. Hence we can afford $\mathcal{O}(z)$ time and $\mathcal{O}(z \log n)$ bits of space for the proofs of Theorems 4.1 and 4.2.

Lemma 4.3 ([LZ76, Theorem 2])**.** *Let $x[1..n]$ be a string that contains σ distinct symbols. Then the LZ factorization of x contains $z = \mathcal{O}(n/\log_\sigma n)$ phrases.*

Proof. Let $m = \lfloor \log_\sigma n/2 \rfloor$. Since the phrases of the LZ factorization do not overlap, there are at most $\mathcal{O}(n/\log_\sigma n)$ phrases of length at least m. There are trivially at most $m = \mathcal{O}(n/\log_\sigma n)$ phrases with destination larger than $n - m$. Let $x[i..i + m')$ with $m' \in [1, m)$ be a phrase of length less than m at destination $i \in [1, n - m]$. By the definition of the LZ factorization, there is no $j \in [1, i)$ such that $x[j..j + m'] = x[i..i + m']$. Hence there is also no $j \in [1, i)$ such that $x[j..j + m] = x[i..i + m]$. Over an alphabet of size σ, there are only $\sigma^m \leq \sigma^{\log_\sigma n/2} = \sqrt{n}$ distinct strings of length m. Since each of the remaining phrases corresponds to the leftmost occurrence of one of these substrings, there cannot be more than $\sqrt{n} = \mathcal{O}(n/\log_\sigma n)$ of them. □

Lempel-Ziv and Longest Common Extensions LCEs are inherently related to LZ because a referencing phrase $f_{i'}$ with source j and destination i is of length $|f_{i'}| = \textsc{lce}(i, j) = \max_{j' \in [1,i)}(\textsc{lce}(i, j'))$. LCEs also reveal the lexicographical order of substrings, which we will exploit repeatedly. For any substrings $x[i..i + \ell_i)$ and $x[i'..i' + \ell_{i'})$, it holds $x[i..i + \ell_i) \prec x[i'..i' + \ell_{i'})$ if and only if either $\textsc{lce}(i, i') \geq \ell_i$ and $\ell_i < \ell_{i'}$, or $\textsc{lce}(i, i') < \min(\ell_i, \ell_{i'})$ and $x[i + \textsc{lce}(i, i')] < x[i' + \textsc{lce}(i, i')]$. A data structure by Kempa and Kociumaka provides constant time LCE queries, and thus also constant time lexicographical order testing of substrings.

Lemma 4.4 ([KK19, Theorem 5.4])**.** *Let $x[1..n]$ be packed over $[0, \sigma)^n$. A data structure that supports constant time LCEs (given $i, j \in [1, n]$, output $\textsc{lce}(i, j)$) and lexicographical order testing (given $i, j \in [1, n]$, output if $x[i..n] \prec x[j..n]$) can be computed in $\mathcal{O}(n/\log_\sigma n)$ time and $\mathcal{O}(n \log \sigma)$ bits of working space.*[a]

[a] The space complexity is not explicitly stated in [KK19, Theorem 5.4], but it is clear from the construction that the working space in words is linear in the required time.

4.2 Algorithm for 3-Approximate LZ-like Factorization

We accelerate the computation with precomputed lookup tables. We access the tables with short substrings of x. A (sub-)string $y[1..m]$ packed over $[0, \sigma)$ is a bitstring of length $m \cdot \lceil \log_2 \sigma \rceil$. Hence we can interpret y as an integer $\mathsf{int}(y) \in [0, 2^{m \cdot \lceil \log_2 \sigma \rceil})$. If $m \leq \log_2 n / \lceil \log_2 \sigma \rceil$, then y fits in a word of memory and can be extracted from x in constant time by using bit-shifts and bit-wise operations. We can then obtain $\mathsf{int}(y)$ and use it to access a lookup table in constant time. Unlike strings, we index lookup tables starting with 0. For example, $Q[0..n')$ is a lookup table that has n' entries, and the first entry is $Q[0]$.

As a warm-up result (and for later usage), we describe a lookup table that detects periodicities (Lemma 4.5) and a set of tables for leftmost pattern matching queries (Lemma 4.6). Recall that a string y is of period $p \in \mathbb{N}^+$ if and only if $y[1..|y| - p] = y[1 + p..|y|]$ (or equivalently if $\forall i \in [1, |y| - p] : y[i] = y[i + p]$). We say that p is *the* period of y if it is the minimal period of y.

Lemma 4.5. *Let $n \in [1, 2^w]$. There is a data structure that, given pattern $y[1..m]$ packed over $[0, \sigma)$ with $m \leq \log_2 n / (2 \lceil \log_2 \sigma \rceil)$, outputs the minimal period of y in constant time. It can be computed in $\mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}(n))$ time and space.*

Proof. Let $y[1..m]$ be packed over $[0, \sigma)$, then $\mathsf{int}(y) \in [0, n')$ with $n' \leq 2^{\log_2 n / 2} = \sqrt{n}$. For each $y \in [0, \sigma)^m$, we naively compute its period in $\mathcal{O}(m^2) \subseteq \mathcal{O}(\log^2 n)$ time, and store it in entry $Q_m[\mathsf{int}(y)]$ of a lookup table Q_m. There are $\mathcal{O}(\log n)$ tables (one per possible value of m), and each table has at most $\sqrt{n}$ entries. Hence the total time and words of space are bounded by $\mathcal{O}(n' \log^3 n) = \mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}(n))$. □

Lemma 4.6. *Let $x[1..n]$ be packed over $[0, \sigma)$. Let $\epsilon \in \mathbb{R}^+$ be constant. There is a data structure that, given a query pattern $y \in [0, \sigma)^m$ with $m \leq \log_2 n / ((2 + \epsilon) \lceil \log_2 \sigma \rceil)$, outputs the leftmost occurrence of y in x in constant time. It can be computed in $\mathcal{O}(n / \log_\sigma n)$ time and $o(n / \log n)$ bits of space.*

Proof. Let $k = \lfloor \log_2 n / ((2 + \epsilon) \lceil \log_2 \sigma \rceil) \rfloor$ be the maximal allowed pattern length. Let $u \in [0, \sigma)^{2k}$, then $\mathsf{int}(u) \in [0, n')$ with $n' \in \mathcal{O}(n^{1 - \hat{\epsilon}})$, where $\hat{\epsilon} = \epsilon / (2 + \epsilon) > 0$. In a table $M[0..n')$, we compute for every $u \in [0, \sigma)^{2k}$ the value $M[\mathsf{int}(u)] = ik + 1$, where $i \in [0, \frac{n}{k} - 2]$ is the minimal value with $u = x[ik + 1..ik + 2k + 1)$. If no such i exists, we store $M[\mathsf{int}(u)] = n + 1$. Computing the table takes $\mathcal{O}(n / \log_\sigma n)$ time. We simply iterate over the $\mathcal{O}(n / \log_\sigma n)$ possible values of $ik + 1$ in decreasing order. For each of them, we take constant time to assign $M[\mathsf{int}(x[ik + 1..ik + 2k + 1))] = ik + 1$.

Now we use M to compute the leftmost occurrence of each possible pattern of length at most k. We create k lookup tables $L_1, L_2, \ldots, L_k$. For $y \in [0, \sigma)^m$, entry $L_m[\mathsf{int}(y)]$ will contain the leftmost occurrence of y in x. We compute L_m as follows. Initially, all entries are set to $n + 1$. Now we consider each string $u \in [0, \sigma)^{2k}$. For every $j \in [0, 2k - m]$, we let $y = u[1 + j..1 + j + m)$ and assign $L_m[\mathsf{int}(y)] = \min(L_m[\mathsf{int}(y)], M[\mathsf{int}(u)] + j)$. The leftmost occurrence of any length-m pattern is fully contained in a length-$2k$ substring at some position $ik + 1$. Hence the computed values are correct. For each of the n' possible $u \in [0, \sigma)^{2k}$, we have to consider $\mathcal{O}(k^2)$ substrings, and for each of them we spend constant time to update some table L_m.

The time is $\mathcal{O}(n' \cdot k^2) \subset \mathcal{O}(n^{1-\hat{\epsilon}}\text{polylog}(n)) \subset \mathcal{O}(n/\log_\sigma n)$. There are $k+1$ lookup tables, and each has at most n' entries. Hence $\mathcal{O}(n^{1-\hat{\epsilon}}\text{polylog}(n)) \subset o(n/\log n)$ bits of space are sufficient. □

String Synchronizing Sets We will work with a small subset of sample positions that has convenient synchronizing properties.

Definition 4.7 ([KK19])**.** Let $x[1..n]$ be a string and let $\tau \in [1, \lfloor \frac{n}{2} \rfloor]$. A subset $S \subseteq [1, n-2\tau+1]$ of positions is τ-synchronizing (with respect to x) if and only if the following conditions hold.

- *Synchronizing condition:* For any $i, j \in [1, n-2\tau+1]$ with $x[i..i+2\tau) = x[j..j+2\tau)$, it holds $i \in S$ if and only if $j \in S$.
- *Density condition:* For any $i \in [1, n-3\tau+2]$, it holds $S \cap [i, i+\tau) = \emptyset$ if and only if the period of $x[i..i+3\tau-2]$ is at most $\frac{\tau}{3}$.

Lemma 4.8 ([KK19, Theorems 4.3 and 8.11])**.** *Let $x[1..n]$ be packed over $[0,\sigma)^n$. There is a $\lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil) \rfloor$-synchronizing set of size $\mathcal{O}(n/\log_\sigma n)$. It takes $\mathcal{O}(n/\log_\sigma n)$ time and $\mathcal{O}(n \log \sigma)$ bits of working space to compute the set, and to lexicographically sort all the suffixes that start at positions in the set.*[a]

[a] The space complexity is not explicitly stated in [KK19, Theorems 4.3 and 8.11], but it is clear from the construction that the working space in words is linear in the required time.

4.2.1 Computing Longest Previous Factors of Sample Positions

Let $\tau = \lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil) \rfloor$. We use a τ-synchronizing set of sample positions. We start by computing for each sample position the longest referencing phrase that could hypothetically start at that position. This is similar to computing longest previous factors [CI08a] in the sequential setting without word-packing.

We obtain a τ-synchronizing set $\{d_1, d_2, \ldots, d_N\}$ with $\forall h \in [1, N) : d_h < d_{h+1}$ and $N = \mathcal{O}(\frac{n}{\tau}) = \mathcal{O}(n/\log_\sigma n)$. We lexicographically sort the suffixes at synchronizing positions and obtain their sparse suffix array, which is the unique permutation suf of $[1, N]$ with $\forall h \in [1, N) : x[d_{\mathsf{suf}[h]}..n] \prec x[d_{\mathsf{suf}[h+1]}..n]$. This takes $\mathcal{O}(n/\log_\sigma n)$ time with Lemma 4.8. Next, we compute an array $\text{LPF}[1..N]$ (for longest previous factor), where entry $\text{LPF}[h]$ is a position from $[1, d_h)$ that maximizes $\textsc{lce}(d_h, \text{LPF}[h])$ (this position may not be unique, and it is not necessarily a sample position). We first use Lemma 4.6 to find the minimal j with $x[j..j+2\tau) = x[d_h..d_h+2\tau)$ in constant time. If $j = d_h$, then $x[d_h..d_h+2\tau)$ has no previous occurrence. In this case, we issue at most $\mathcal{O}(\tau)$ queries to Lemma 4.6 and find the maximal $\ell \in [0, 2\tau)$ such that $x[d_h..d_h+\ell)$ has a previous occurrence. This also reveals $\text{LPF}[h]$ (we can choose any position from $[1, d_h)$ if $\ell = 0$), but it takes $\mathcal{O}(\tau)$ time. However, this can only happen once per distinct length-2τ substring, which limits the total time to $\mathcal{O}(2^{2\tau \cdot \lceil \log_2 \sigma \rceil}\tau) \subset \mathcal{O}(n/\log n)$. If $j < d_h$, then $x[d_h..d_h+2\tau)$ has a previous occurrence, and the synchronizing property of Definition 4.7 guarantees that *all* previous occurrences of $x[d_h..d_h+2\tau)$ start at sample positions. Thus, we can compute LPF in the same way as it is usually done for the entire suffix array. A detailed description can be found, e.g., in [CI08a] (where our LPF corresponds to PrevOcc in [CI08a]), and we only give a brief summary. For each entry $\mathsf{suf}[h]$, we find

$$\begin{aligned} \mathsf{prev}[h] &= \max(\{h' \in [1, h) \quad\quad \mid \mathsf{suf}[h'] < \mathsf{suf}[h]\}) \quad \text{and} \\ \mathsf{next}[h] &= \min(\{h' \in [h+1, N] \mid \mathsf{suf}[h'] < \mathsf{suf}[h]\}), \end{aligned}$$

which takes $\mathcal{O}(N)$ time with an algorithm for nearest smaller values (see, e.g., [BFN11, Lemma 1]). We then use Lemma 4.4 to compute $\ell_1 = \textsc{lce}(d_{\mathsf{suf}[h]}, d_{\mathsf{suf}[\mathsf{prev}[h]]})$ and $\ell_2 = \textsc{lce}(d_{\mathsf{suf}[h]}, d_{\mathsf{suf}[\mathsf{next}[h]]})$, which are the respective maximal phrase lengths at destination $d_{\mathsf{suf}[h]}$ that can be achieved with a lexicographically smaller and a lexicographically larger suffix starting at an earlier sample position. If $\max(\ell_1, \ell_2) < 2\tau$, then we have already assigned $\mathsf{LPF}[\mathsf{suf}[h]]$ with Lemma 4.6 as described before. Otherwise, if $\ell_1 > \ell_2$, then we assign $\mathsf{LPF}[\mathsf{suf}[h]] = \mathsf{suf}[\mathsf{prev}[h]]$. If, however, $\ell_2 \geq \ell_1$, then we assign $\mathsf{LPF}[\mathsf{suf}[h]] = \mathsf{suf}[\mathsf{next}[h]]$. It is possible that $\mathsf{prev}[h]$ and/or $\mathsf{next}[h]$ are undefined, but treating this is trivial. The correctness follows from the synchronizing property and the correctness of the same technique for the full suffix array [CI08a]. The total time and space in words are $\mathcal{O}(N) = \mathcal{O}(n / \log_\sigma n)$.

4.2.2 Computing a Gapped Factorization

Now we compute a *gapped* LZ factorization $x = f_1 g_1 r_1 f_2 g_2 r_2 \dots f_{z'} g_{z'} r_{z'}$, where:

- Each $f_{i'}$ is a *perfect phrase* at destination $i = 1 + \sum_{h=1}^{i'-1} |f_h g_h r_h|$ defined just like in the exact factorization. It is either the leftmost occurrence of a symbol (a literal phrase), or the longest prefix of $x[i..n]$ with a previous occurrence $x[j..j + |f_{i'}|) = f_{i'}$ at some source $j \in [1, i)$ (a referencing phrase).
- Each $g_{i'}$ is a (possibly empty) *gap* at destination $i = 1 + |f_{i'}| + \sum_{h=1}^{i'-1} |f_h g_h r_h|$. A gap can be any string and does not necessarily have a previous occurrence.
- Each $r_{i'}$ is a *reference* at destination $i = 1 + |f_{i'} g_{i'}| + \sum_{h=1}^{i'-1} |f_h g_h r_h|$, which is either empty or it has a previous occurrence $x[j..j + |r_{i'}|) = r_{i'}$ at source $j \in [1, i)$ (with no requirement of maximal length).

Lemma 4.9. *Any gapped LZ factorization $x = f_1 g_1 r_1 f_2 g_2 r_2 \dots f_{z'} g_{z'} r_{z'}$ satisfies $z' \leq z$, where z is the number of phrases in the exact LZ factorization of x.*

Proof. A suffix $x[j..i + \ell)$ of an exact LZ phrase $x[i..i + \ell)$ at destination i has a previous occurrence. Hence, if j is the destination of a perfect phrase $f_{j'}$ in the gapped factorization, then this phrase is of length at least $\ell - j$. This means that a phrase of the exact LZ factorization contains the destination of at most one perfect phrase of the gapped LZ factorization, which implies $z' \leq z$. □

Computing *any* gapped factorization is trivial (e.g., $x = fgr$ with $f = x[1]$, $g = [x_2..n]$, $r = \varepsilon$ is a gapped factorization). We will compute a gapped factorization with the additional property that none of the gaps contains a position from the synchronizing set, which makes it easy to eliminate the gaps in a post-processing. We compute the factorization from left to right using LPF.

The first perfect phrase is literal phrase $f_1 = x[1]$. After creating some perfect phrase $f_{i'}$ at destination i, we iterate over the upcoming sample positions until we reach the first $d_h \geq i + |f_{i'}|$. The next gap is $g_{i'} = x[i + |f_{i'}|\,..d_h)$, and the next reference $r_{i'}$ is empty. The next perfect phrase $f_{i'+1}$ at destination d_h is a literal phrase if $\textsc{lce}(d_h, \mathrm{LPF}[h]) = 0$. Otherwise, it is a referencing phrase with source $\mathrm{LPF}[h]$ and length $\textsc{lce}(d_h, \mathrm{LPF}[h])$. As soon as we create a perfect phrase $f_{i'}$ at destination i with $i + |f_{i'}| > d_N$, we complete the factorization with gap $g_{i'} = x[i + |f_{i'}|\,..n]$ and empty reference $r_{i'}$. We spend constant time per sample position, and hence the time is $\mathcal{O}(N) = \mathcal{O}(n/\log_\sigma n)$.

Eliminating Long Gaps Now we eliminate the gaps by replacing them with references. We distinguish between short gaps of length at most 3τ, and long gaps of length more than 3τ. A long gap $g_{i'}$ at destination i is of length more than 3τ, and due to our method of computing the factorization it does not contain any of the synchronizing positions. By the density condition of Definition 4.7, $g_{i'}$ has period $p \leq \tau/3$. The reference $r_{i'}$ is empty (because all references in the initial gapped factorization are empty, and we only replace them with non-empty references when eliminating long gaps). We replace $g_{i'}r_{i'}$ with $g'_{i'}r'_{i'}$, where $g'_{i'} = g_{i'}[1..3\tau]$ and $r'_{i'} = g_{i'}[3\tau + 1..\,|g_{i'}|)$. Since $g_{i'}$ has period p, the new reference $r'_{i'}$ at destination $i + 3\tau$ has a previous occurrence at source $j = i + 3\tau - p$. Hence the replacement retains the properties of a gapped factorization.

If $p \leq \frac{\tau}{3}$ is the minimal period of $g_{i'}$ of length at least 3τ, then it is easy to see that also $g_{i'}[1..3\tau]$ has minimal period p (because $g_{i'}[1..3\tau]$ contains all the length $2p$ substrings of $g_{i'}$, and thus a smaller period would directly translate to the entire $g_{i'}$). Hence we can simply use Lemma 4.5 with query pattern $g_{i'}[1..3\tau]$ to look up p in constant time. This way, replacing a long gap takes constant time, and the total time needed for all long gaps is $\mathcal{O}(z) = \mathcal{O}(n/\log_\sigma n)$.

Eliminating Short Gaps and Finalizing the Factorization We have eliminated all long gaps, and from now on we simply say gap rather than short gap. A non-empty gap $g_{i'}$ at destination i is *referencing* if there is some $j < i$ with $x[j..j+|g_{i'}|) = g_{i'}$ (we could replace the gap with a reference). We first identify all the non-referencing non-empty gaps. For each non-empty gap, we use Lemma 4.6 to find the minimal j with $x[j..j + |g_{i'}|) = g_{i'}$ in constant time. If and only if $j = i$, then $g_{i'}$ is non-referencing. The total time needed is $\mathcal{O}(z) \subseteq \mathcal{O}(n/\log_\sigma n)$.

We process each non-referencing non-empty gap $g_{i'}$ separately. We find the maximal $\ell \in [0, |g_{i'}|)$ such that the prefix $x[i..i + \ell)$ of $g_{i'}$ has a previous occurrence. We do so by issuing $\mathcal{O}(|g_{i'}|)$ queries to Lemma 4.6, which takes $\mathcal{O}(\tau)$ time. This also reveals the source position $j \in [1, i)$ of the previous occurrence. We adjust the gapped factorization by re-factorizing the gap as $g_{i'} = grfg'$, where g is a new empty gap, r is a new empty reference, $f = x[i..i + \max(1, \ell))$ is a new perfect phrase (with source j if $\ell > 0$), and $g' = x[i + \ell..i + |g_{i'}|)$ is the remainder of the gap. Note that this replacement retains the properties of a gapped factorization, and by Lemma 4.9 there are still at most z factors of each type. If the new gap g' is still non-empty and non-referencing (we check this in the same way as before), then we replace g' by applying the same re-factorization procedure again, and we keep doing so until the remainder of the gap is either empty or referencing. Each application takes $\mathcal{O}(\tau)$ time and decreases the length of the remainder. Hence the total time for processing $g_{i'}$ is $\mathcal{O}(\tau^2)$.

A non-referencing gap $g_{i'}$ at destination i implies that $x[i..i+|g_{i'}|)$ is the leftmost occurrence of a substring of length at most 3τ. There are fewer than $2^{3\tau\cdot\lceil\log_2\sigma\rceil}\cdot 3\tau$ distinct substrings of this length, and hence re-factorizing all the non-referencing gaps takes $\mathcal{O}(2^{3\tau\cdot\lceil\log_2\sigma\rceil}\cdot\tau^3)\subseteq\mathcal{O}(n^{3/8}\log^3 n)$ time.

The remaining non-empty gaps are referencing, and we can use Lemma 4.6 to replace them with referencing phrases. We obtain an LZ-like factorization by discarding all empty factors. This leaves at most z perfect phrases and $2z$ referencing phrases. The total time is $\mathcal{O}(n/\log_\sigma n)$. Lemmas 4.4 to 4.6 and 4.8 require $\mathcal{O}(n\log\sigma)$ bits of memory. Apart from that, we only use arrays of size N, which also require $\mathcal{O}(N\log n)=\mathcal{O}(n\log\sigma)$ bits of memory. Hence we have shown Theorem 4.1. It is easy to see that, instead of first computing a gapped factorization and then closing the gaps, we could just as well directly compute the approximate factorization from left to right. This may result in a faster practical implementation.

4.3 Algorithm for Exact LZ Factorization

In the approximate algorithm, we create perfect phrases for which both source and destination are samples. For the exact LZ factorization, we have to allow arbitrary sources and destinations. We will define a new set of sample positions such that, if a phrase $f_{i'}$ has source j, there will be at least one sample position j' that is reasonably close to the source, say, $j'\in[j,j+\min(\delta,|f_{i'}|))$ for some parameter δ. We can conceptually divide the phrase into a head $x[j..j']$ and a tail $x[j'..j+|f_{i'}|)$. Computing a phrase means finding a sample position with matching head, and with tail of maximal length. If we co-lexicographically sort the prefixes that end at sample positions, then we group together samples that admit the same head. Similarly, if we lexicographically sort the suffixes that start at sample positions, then we group together samples that admit the same tail. This motivates a geometric interpretation of sample positions, in which each sample is represented by the lexicographical rank of its suffix and the co-lexicographical rank of its prefix. (This technique is similar to what was done in [BP16, Section 6.2].) Ultimately, we use geometric data structures for insertion-only orthogonal range one-reporting to handle most of the computational effort. (We could also use static data structures with an extra dimension or weighted points; however, there are few such data structures with known construction times.)

Definition 4.10. Let $N\in[1,2^w]$ and let π be a permutation of $[1,N]$. The task of insertion-only orthogonal range one-reporting is to maintain a set of points $\mathcal{P}\subseteq\{(i,\pi(i))\mid i\in[1,N]\}$ (initially empty) with the following operations:

(i) insert $p\in\{(i,\pi(i))\mid i\in[1,N]\}$ into $\mathcal{P}$

(ii) given $\mathcal{Q}=[a_1,a_2]\times[b_1,b_2]$, output any point from $\mathcal{Q}\cap\mathcal{P}$, or report $\mathcal{Q}\cap\mathcal{P}=\emptyset$

Now we show how to find previous occurrences of substrings by using orthogonal range reporting and an arbitrary set of sample positions.

Lemma 4.11. *Let $x[1..n]$ be packed over $[0, \sigma)$, and let $\mathcal{A}[1..N]$ be an array of N distinct samples from $[1, n]$ in increasing order. Let $u_{\mathcal{A}}$ and $q_{\mathcal{A}}$ be respectively the insertion and query time of a data structure for insertion-only orthogonal range one-reporting, and let $s_{\mathcal{A}}$ be the maximum number of words occupied by this data structure after N insertions. After an $\mathcal{O}(n/\log_\sigma n + N \log N)$ time preprocessing, and in $\mathcal{O}(n/\log_\sigma n + N + s_{\mathcal{A}})$ words of space, a subset $\mathcal{X}$ of sample positions can be maintained with the following operations.*

(*i*) *Given $h \in [1, N]$, insert $\mathcal{A}[h]$ into $\mathcal{X}$ in $\mathcal{O}(u_{\mathcal{A}})$ time*

(*ii*) *Given $i \in [1, n]$ and $k \in [0, n - i]$, find the (possibly not unique) $\hat{h} \in \mathcal{X} \cap (k, n]$ with $x[\hat{h} - k..\hat{h}] = x[i..i + k]$ and maximal $\ell = \textsc{lce}(\hat{h}, i + k)$ in $\mathcal{O}(\log \ell \cdot (\log N + q_{\mathcal{A}}))$ time, or report that $\hat{h}$ does not exist in $\mathcal{O}(\log N + q_{\mathcal{A}})$ time*

Proof. We start by preprocessing the sample positions. Let suf be the unique permutation of $[1, N]$ that lexicographically sorts the suffixes of x that start at sample positions, i.e., $\forall h \in [1, N) : x[\mathcal{A}[\mathsf{suf}[h]]..n] \prec x[\mathcal{A}[\mathsf{suf}[h + 1]]..n]$ (a sparse suffix array). We use comparison-based sorting with Lemma 4.4 for constant time lexicographical suffix comparisons and obtain suf in $\mathcal{O}(n/\log_\sigma n + N \log N)$ time and $\mathcal{O}(n/\log_\sigma n + N)$ words of space. Analogously, we obtain the unique permutation pref of $[1, N]$ that co-lexicographically sorts the prefixes of x that end at sample positions, i.e., $\forall h \in [1, N) : \mathsf{rev}(x[1..\mathcal{A}[\mathsf{pref}[h]]]) \prec \mathsf{rev}(x[1..\mathcal{A}[\mathsf{pref}[h+1]]])$. We compute $\mathsf{rev}(x)$ in $\mathcal{O}(n/\log_\sigma n)$ time and words of space with universal lookup tables (see, e.g., [BP16, Section 6.2]). By once more comparison-based sorting with the data structure from Lemma 4.4 constructed for $\mathsf{rev}(x)$, we obtain pref in $\mathcal{O}(n/\log_\sigma n + N \log N)$ time and $\mathcal{O}(n/\log_\sigma n + N)$ words of space. It is trivial to compute the respective inverse permutations $\mathsf{suf\text{-}rank}$ and $\mathsf{pref\text{-}rank}$ of suf and pref in $\mathcal{O}(N)$ time and words of space. This concludes the preprocessing.

Insertions. In order to insert $\mathcal{A}[h]$ into $\mathcal{X}$, we insert the two-dimensional point $(\mathsf{suf\text{-}rank}(h), \mathsf{pref\text{-}rank}(h))$ into the geometric data structure for orthogonal range reporting, which leads to the claimed insertion time and space complexity.

Queries. We first show a fast way to answer a slightly simpler type of query. Given suffix $x[i..n]$, offset k, and length estimate ℓ, we want to find some $\mathcal{A}[h] \in \mathcal{X}$ such that $x[\mathcal{A}[h] - k..\mathcal{A}[h] + \ell) = x[i..i + k + \ell)$ (if it exists). The lexicographical order groups together suffixes of x that share a long prefix. Thus, there is an interval $\mathsf{suf}[a_1..a_2]$ that contains exactly the $h \in [1, N]$ with $x[\mathcal{A}[h]..\mathcal{A}[h] + \ell) = x[i + k..i + k + \ell)$. We compute the interval boundary a_1 by binary searching in suf for the lexicographically minimal suffix that starts at a sample position and has prefix $x[i + k..i + k + \ell)$. This works similarly to pattern matching with the suffix array [MM93]. If $\mathsf{suf}[h']$ is the center of the search interval, then we compute $\ell' = \textsc{lce}(\mathcal{A}[\mathsf{suf}[h']], i + k)$. If $\ell' \geq \ell$, or if $x[i + k..n] \preceq x[\mathcal{A}[\mathsf{suf}[h']]..n]$, then we proceed in the left half of the search interval (including $\mathsf{suf}[h']$). Otherwise, it holds $\ell' < \ell$ and $x[i + k..n] \succ x[\mathcal{A}[\mathsf{suf}[h']]..n]$, and we continue in the right half of the interval (excluding $\mathsf{suf}[h]$). Computing LCEs and performing lexicographical comparisons takes constant time with Lemma 4.4, and hence the binary search takes $\mathcal{O}(\log N)$ time. Analogously, we compute a_2 in $\mathcal{O}(\log N)$ time. The co-lexicographical order groups together prefixes that share a long suffix. There is an interval $\mathsf{pref}[b_1..b_2]$ that contains exactly the $h \in [1, N]$ with $x[\mathcal{A}[h] - k..\mathcal{A}[h]] = x[i..i + k]$, and we can compute the interval borders in $\mathcal{O}(\log N)$ time (analogously to the computation of a_1 and a_2, but with the LCE data structure for $\mathsf{rev}(x)$). A sample $\mathcal{A}[h]$ satisfies $x[\mathcal{A}[h]-k..\mathcal{A}[h]+\ell) = x[i..i+k+\ell)$ if and only if $(\mathsf{suf\text{-}rank}(h), \mathsf{pref\text{-}rank}(h)) \in [a_1, a_2] \times [b_1, b_2]$. If we have already inserted a sample position that satisfies this condition, then a query to the geometric data structure returns a matching point $(\mathsf{suf\text{-}rank}(h), \mathsf{pref\text{-}rank}(h))$ in $q_{\mathcal{A}}$ time. Obtaining $\mathcal{A}[h]$ from the point takes constant time due to $h = \mathsf{suf}(\mathsf{suf\text{-}rank}(h))$. Otherwise, the data structure returns that the point does not exist. Thus, we can find some $\mathcal{A}[h] \in \mathcal{X}$ such that $x[\mathcal{A}[h]-k..\mathcal{A}[h]+\ell) = x[i..i+k+\ell)$, or report that such a sample does not exist, in $\mathcal{O}(\log N + q_{\mathcal{A}})$ time.

Finally, in order to answer a query of the type stated in the lemma, we use exponential search to find the maximal $\ell \in \mathbb{N}^+$ such that there is some $\mathcal{A}[h] \in \mathcal{X}$ with $x[\mathcal{A}[h]-k..\mathcal{A}[h]+\ell) = x[i..i+k+\ell)$. This way, we obtain $\mathcal{A}[h]$ in $\mathcal{O}(\log \ell \cdot (\log N + q_{\mathcal{A}}))$ time, or the first query with $\ell = 1$ to the geometric data structure comes back negative, and we report that no matching sample position exists in $\mathcal{O}(\log N + q_{\mathcal{A}})$ time. □

4.3.1 Computing the Exact LZ Factorization

Now we show how to compute the LZ factorization $x = f_1 f_2 \dots f_z$. We distinguish between short phrases of length less than δ and long phrases of length at least δ, where $\delta \in [1, n]$ is a parameter to be fixed later (it will be polylogarithmic in n). We compute the factorization one phrase at a time and in left-to-right order. When computing $f_{i'}$ at destination $i \in [1, \delta)$, we compute $\textsc{lce}(j, i)$ for each $j \in [1, i)$ with Lemma 4.4, which reveals the length and source of the phrase in $\mathcal{O}(\delta)$ time (or all LCEs are zero and $f_{i'}$ is a literal phrase). When computing a phrase $f_{i'}$ at destination $i \geq \delta$, we use three different methods depending on the leftmost source j of $f_{i'}$. We do not know j in advance, and thus we try each of the methods and then choose the result that yields the longest phrase.

Method 1: Close Sources If $j \in [i-\delta, i)$, then we obtain the phrase by computing $\textsc{lce}(j', i)$ for each $j' \in [i-\delta, i)$ and keeping track of the longest LCE.

Methods 2 and 3: Far Sources If $j \in [1, i-\delta)$, then we use two instances of the data structure from Lemma 4.11. The first one maintains a subset $\mathcal{X}$ of evenly spaced samples from an array $\mathcal{A}[1..N]$ with $N = \lfloor \frac{n}{\delta} \rfloor$ and $\forall h \in [1, N] : \mathcal{A}[h] = h\delta$. The second one maintains a subset $\mathcal{Y}$ of samples from an array $\mathcal{B}[1..M]$ of size $M = \mathcal{O}(z)$ that is sorted in increasing order. The samples are chosen such that, if $x[j'..j'+\ell)$ is the leftmost occurrence of any substring (e.g., the leftmost source occurrence of an LZ phrase), then $[j', j'+\ell)$ contains at least one sample position. This is achieved with the LZ-like factorization from Theorem 4.1, which we explain later. The space usage is $\mathcal{O}(n/\log_\sigma n + N + M + s_{\mathcal{A}} + s_{\mathcal{B}})$, and the preprocessing time is $\mathcal{O}(n/\log_\sigma n + N \log N + M \log M)$. For both instances, we maintain the following invariant. At the time at which we compute phrase $f_{i'}$ at destination i, we have inserted exactly the samples that satisfy $\mathcal{A}[h] < i$ into $\mathcal{X}$, and the samples that satisfy $\mathcal{B}[h] < i$ into $\mathcal{Y}$. Since we compute the phrases from left to right, we also insert the samples of each array in left-to-right order. Thus, there is no time overhead for finding the next sample to insert. The total insertion time is $\mathcal{O}(N \cdot u_{\mathcal{A}} + M \cdot u_{\mathcal{B}})$. Now we use $\mathcal{X}$ and $\mathcal{Y}$ to compute phrases.

Method 2: Long Phrases If $|f_{i'}| \geq \delta$, then $[j, j+\delta)$ contains the sample position $\mathcal{A}[h] = h\delta < j + \delta < i$ with $h = \lceil j/\delta \rceil$ (where $j + \delta < i$ due to the assumption that $j \in [1, i-\delta)$). By the invariant on $\mathcal{X}$, we have already inserted $\mathcal{A}[h]$ into $\mathcal{X}$. Let

$k = \mathcal{A}[h] - j \in [0, \delta)$, then it holds $x[j..j+k] = x[\mathcal{A}[h] - k..\mathcal{A}[h]] = x[i..i+k]$. Thus, if we query $\mathcal{X}$ with position i and offset k, then we obtain either $\mathcal{A}[h]$ or another sample position $\mathcal{A}[h'] < i$ with $\textsc{lce}(\mathcal{A}[h'] - k, i) = \textsc{lce}(\mathcal{A}[h] - k, i) = \textsc{lce}(j, i) = |f_{i'}|$. This way, we find both a source and the length of $f_{i'}$. Since we do not know k in advance, we issue one query for each possible value of k and keep track of the maximal LCE, which takes $\mathcal{O}(\delta \cdot \log |f_{i'}| \cdot (\log N + q_{\mathcal{A}}))$ time.

Method 3: Short Phrases This works analogously to the method for long phrases (but with $\mathcal{Y}$ instead of $\mathcal{X}$). If $|f_{i'}| < \delta$, then $[j, j + |f_{i'}|)$ contains at least one sample position $\mathcal{B}[h] < j + |f_{i'}| < j + \delta < i$ (where $j + \delta < i$ due to the assumption that $j \in [1, i - \delta)$, and a sample position is present because $x[j..j + |f_{i'}|)$ is the leftmost occurrence of a substring). By the invariant on $\mathcal{Y}$, we have already inserted $\mathcal{B}[h]$ into $\mathcal{Y}$. Let $k = \mathcal{B}[h] - j \in [0, |f_{i'}|)$, then it holds $x[j..j+k] = x[\mathcal{B}[h] - k..\mathcal{B}[h]] = x[i..i+k]$. Thus, if we query $\mathcal{Y}$ with position $x[i..n]$ and offset k, we obtain either $\mathcal{B}[h]$ or another sample position $\mathcal{B}[h'] < i$ for which it holds $\textsc{lce}(\mathcal{B}[h'] - k, i) = \textsc{lce}(\mathcal{B}[h] - k, i) = \textsc{lce}(j, i) = |f_{i'}|$. This way, we find both a source and the length of $f_{i'}$. Since we do not know k in advance, we issue one query for each possible value of k and keep track of the maximal LCE, which takes $\mathcal{O}(\delta \cdot \log |f_{i'}| \cdot (\log N + q_{\mathcal{B}}))$ time.

4.3.1.1 Analyzing the Procedure

Each method requires that j and possibly $|f_{i'}|$ satisfies some condition. It is easy to see that every scenario is covered by one of the conditions. Thus, there is always at least one method that computes a phrase of maximal length. If we run any of the methods even though the respective condition is not satisfied, then the result is still an LCE between i and a smaller position. Thus, we will never overestimate $|f_{i'}|$, and it is indeed correct to always run all three methods and produce the longest phrase admitted by any of them.

Now we analyze the time and space complexity (in words). We use $\delta = \Theta(\log^2 n)$ and two different geometric data structures with amortized time bounds. The first one [Mor06, CT18] has space complexity $s_{\mathcal{A}} = \mathcal{O}(N \log N) \subseteq \mathcal{O}(n/\log n)$ and performs insertions and queries in $u_{\mathcal{A}} = \mathcal{O}(\log n)$ and $q_{\mathcal{A}} = \mathcal{O}(\log n)$ time (the precise bounds are better, but we do not need them for our purposes). The second one [Nek09] uses linear space $s_{\mathcal{B}} = \mathcal{O}(M) = \mathcal{O}(z) \subseteq \mathcal{O}(n/\log_\sigma n)$ and performs insertions and queries in $u_{\mathcal{B}} = \mathcal{O}(\log^{3+\epsilon} n)$ and $q_{\mathcal{B}} = \mathcal{O}(\log n)$ time (where $\epsilon \in \mathbb{R}^+$ is an arbitrarily small constant). The total space usage is $\mathcal{O}(n/\log_\sigma n)$ words.

The preprocessing time is $\mathcal{O}(n/\log_\sigma n + N \log N + M \log M) \subseteq \mathcal{O}(n/\log_\sigma n + z \log z)$, and the total time for insertions is $\mathcal{O}(N \log n + M \log^{3+\epsilon} n) \subseteq \mathcal{O}(n/\log n + z \log^{3+\epsilon} n)$. The time for applying all three methods is $\mathcal{O}(\log^3 n \cdot \log |f_{i'}|)$ for phrase $f_{i'}$. If $|f_{i'}| = \Omega(\log^5 n)$, then the time amortizes to $\mathcal{O}(1/\log n)$ per symbol in $f_{i'}$, which results in $\mathcal{O}(n/\log n)$ time in total. If $|f_{i'}| = \mathcal{O}(\log^5 n)$, then the time is $\mathcal{O}(\log^3 n \cdot \log\log n)$, or $\mathcal{O}(z \log^{3+\epsilon} n)$ for all phrases. Thus, the overall time is $\mathcal{O}(n/\log_\sigma n + z \log^{3+\epsilon} n)$, and the space is $\mathcal{O}(n/\log_\sigma n)$ words or $\mathcal{O}(n \log \sigma)$ bits. For a purely cosmetic improvement, assume that $(z \log^{3+\epsilon} n) > (n/\log_\sigma n)$. Then $z > (n/\log^5 n)$ and $\log z = \Omega(\log n)$. Thus, the time bound is equivalent to $\mathcal{O}(n/\log_\sigma n + z \log^{3+\epsilon} z)$, and the complexities match the ones in Theorem 4.2.

4.3.1.2 Computing $\mathcal{B}$

We conclude the proof of Theorem 4.2 by showing how to compute $\mathcal{B}$ in $\mathcal{O}(n/\log_\sigma n)$ time and words of space. We compute an LZ-like factorization $x = f'_1 f'_2 \dots f'_M$ of $M = \Theta(z) \subseteq \mathcal{O}(n/\log_\sigma n)$ phrases with Theorem 4.1. Now it is easy to obtain $\mathcal{B}[1..M]$ with $\forall h \in [1, M] = \mathcal{B}[h] = \sum_{i'=1}^{h} |f'_{i'}|$ (the end positions of all phrases). Every leftmost occurrence of a symbol is a literal phrase in any LZ-like factorization. Thus, the leftmost occurrence of any symbol is a sample position. Now assume that $x[j'..j'+\ell)$ with $\ell > 1$ is the leftmost occurrence of a substring. If $[j', j'+\ell)$ contains no sample position, then $x[j'..j'+\ell)$ is fully contained within a referencing LZ-like phrase. However, every such phrase has a previous occurrence, which contradicts the fact that $x[j'..j'+\ell)$ has no previous occurrence. Thus, $\mathcal{B}$ functions as required by the algorithm, and we have shown Theorem 4.2.

4.3.2 Computing the Non-Overlapping LZ Factorization

A common variation of the LZ factorization [SS82] requires that the leftmost source j of a referencing phrase $f_{i'}$ at destination i satisfies $j + |f_{i'}| \le i$, i.e., there has to be at least one source occurrence that does not overlap the destination occurrence of the phrase. Theorem 4.2 can be adapted to compute this non-overlapping factorization. When using the first method, we avoid overlaps by simply truncating each LCE to $\min(\textsc{lce}(j, i), i - j)$.

For the second method, we repeatedly use the geometric data structure to decide if, for position i, offset k, and length estimate ℓ, there is any $\mathcal{A}[h] \in \mathcal{X}$ with $x[\mathcal{A}[h] - k..\mathcal{A}[h] + \ell) = x[i..i + k + \ell)$. To avoid overlaps, we have to ensure $\mathcal{A}[h] \le i - \ell$. We add a third dimension and represent each $\mathcal{A}[h]$ as a point $(\textsf{suf-rank}(h), \textsf{pref-rank}(h), \mathcal{A}[h])$. The query interval becomes a three-dimensional hyper-rectangle $[a_1, a_2] \times [b_1, b_2] \times [1, i - \ell]$. There is a three-dimensional geometric data structure with amortized times $u_{\mathcal{A}} = \mathcal{O}(\log^{8/5+\epsilon} N)$ and $q_{\mathcal{A}} = \mathcal{O}((\log N/\log\log N)^2)$ [CT18]. Its space complexity is $s_{\mathcal{A}} = \mathcal{O}(N \log^{8/5+\epsilon} N)$ words. To accommodate for the more expensive update time and higher space complexity, we use a larger $\delta = \Theta(\log_\sigma n \cdot \log^{8/5+\epsilon} n)$. This increases the time for computing a phrase to $\mathcal{O}(\log^{23/5+\epsilon} n \cdot \log |f_{i'}| / \log \sigma)$, and the total time to $\mathcal{O}(n/\log_\sigma n + z \log^{23/5+2\epsilon} z/\log\sigma)$. The space remains $\mathcal{O}(n/\log_\sigma n)$ words.

For avoiding overlaps in the third method, we adjust the first method such that it considers leftmost sources in $[i - 2\delta, i)$ rather than $[i - \delta, i)$ (which does not increase the complexity). Now we can change the invariant for $\mathcal{Y}$ such that, at the time at which we compute $f_{i'}$, we have inserted exactly the samples with $\mathcal{B}[h] < i - \delta$ into $\mathcal{Y}$. We do not miss any sources this way due to our previous adjustment of the first method. Now any source reported by the third method is from $[1, i - \delta)$. The third method is only used for phrases of length less than δ, and thus an overlap is impossible. If the third method reports a phrase of length at least δ, then we simply ignore it; it will be computed by the second method already. The increased value of δ increases the number of queries for the third method, but the time is still dominated by $\mathcal{O}(z \log^{23/5+\epsilon} z/\log\sigma)$.

Corollary 4.12. *Let $x[1..n]$ be packed over $[0, \sigma)$, and let $\epsilon \in \mathbb{R}^+$ be an arbitrarily small constant. If the non-overlapping LZ factorization of x consists of z phrases, then it can be computed in $\mathcal{O}(n / \log_\sigma n + z \log^{23/5+\epsilon} z / \log \sigma)$ time and $\mathcal{O}(n \log \sigma)$ bits of working space.*

4.4 Conclusion

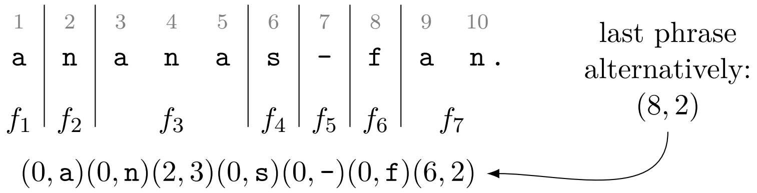
We presented sublinear time algorithms for LZ-like factorizations. While the algorithm for the exact LZ factorization achieves $\mathcal{O}(n / \log_\sigma n)$ time if the number z of phrases is sufficiently small, it remains an open question whether this time can also be achieved for all values of z. The presented solution relies heavily on geometric data structures with super-constant query times, and the computation of each phrase requires at least one query. While there is no obvious way of avoiding this cost per phrase, future advances in geometric data structures may directly improve the time complexity of the algorithm. The $\mathcal{O}(n / \log_\sigma n)$ time solution for the 3-approximate factorization is of independent interest. For example, it may be useful for computing squares and runs in sublinear time. This could be achieved by adapting existing algorithms that rely on the LZ factorization [Cro86, KK99]. Another potential application is the sublinear time construction of the block tree [Bel+21], a data structure based on the LZ factorization that allows fast random access to the compressed string.

Chapter 5

New Advances in Rightmost Lempel-Ziv

5

Recall the example from the introduction of Part I; the LZ factorization of string `ananas-fan` and its encoding as a list of pairs is given by


1 2 3 4 5 6 7 8 9 10
a n a n a s - f a n .
f_1 f_2 f_3 f_4 f_5 f_6 f_7
$(0,\mathtt{a})(0,\mathtt{n})(2,3)(0,\mathtt{s})(0,\mathtt{-})(0,\mathtt{f})(6,2)$
last phrase
alternatively:
$(8,2)$

We use the distance-length encoding for referencing phrases (which can be found in practical compressors like gzip [Webd]). In this encoding, a length-ℓ referencing phrase at destination i gets encoded as (d,ℓ), where $d \in [1,i)$ is the distance to a source of the phrase, i.e., $x[i-d..i-d+\ell) = x[i..i+\ell)$. For example, f_7 at destination 9 gets encoded as $(6,2)$ because it is of length 2 and has source 3 at distance $9-3=6$. However, f_7 also has source 1 at distance $9-1=8$, and we could alternatively encode the phrase as $(8,2)$. But which of the two options is better?

If we naively store each distance-length pair using fixed width integers, then choosing one source over another does not make a difference. However, practical compressors further encode the pairs using variable-length codes. That is, each integer is assigned a binary codeword, but the codewords are not necessarily all of the same length. Such codes often assign longer codewords to larger integers, as shown in Figure 5.1. In this case, we can improve the compression rate by minimizing the distance component of each pair. For the previous example, if we use Elias δ codes [Eli75], then the distance component of $(6,2)$ uses a codeword of length five, while the distance component of $(8,2)$ uses a codeword of length eight. Hence representing f_6 with the pair $(6,2)$ rather than $(8,2)$ saves three bits.

Rightmost Parsings of LZ-like and LZ-End Factorizations In order to minimize the distance component of a referencing phrase, we have to find its *rightmost* source. The problem of computing the *rightmost parsing* of a given LZ-like factorization is to annotate each referencing phrase with its rightmost source. Recall the definition of LZ-End [KN10, KN13]: an LZ-like factorization $f_1 \dots f_z$ is an LZ-End factorization if every referencing phrase $f_{i'}$ occurs as a suffix of $f_1 \dots f_{j'}$ for some $j' \in [1,i')$. We then call $j = 1 - |f_{i'}| + \sum_{k=1}^{j'} |f_k|$ an *LZ-End aligned* source of $f_{i'}$.

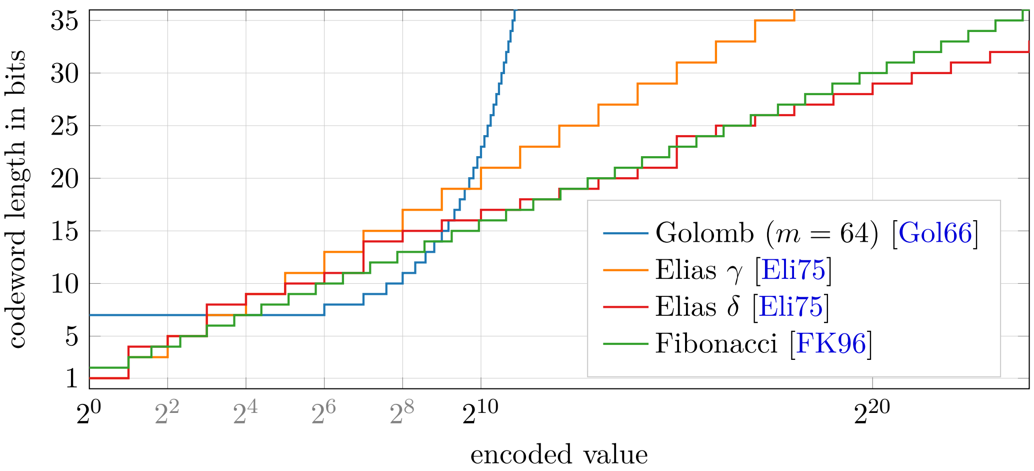


Figure 5.1: Common variable-length codes for positive integers.

We could define the rightmost parsing for LZ-End in the same way as for arbitrary LZ-like factorizations (i.e., annotate each phrase with its rightmost source), but this is undesirable because the rightmost source might not be LZ-End aligned. Hence the rightmost parsing of an LZ-End factorization annotates each referencing phrase with its rightmost LZ-End aligned source. The *greedy LZ-End factorization* is the uniquely defined LZ-like factorization in which each referencing phrase is of maximal length, i.e., a referencing phrase at destination i is the longest prefix if $x[i..n]$ that has an LZ-End aligned previous occurrence. (We again stress that this is not the original definition of LZ-End. The modified definition simplifies the proofs, and all presented results can be trivially adapted to the original definition.)

Contributions We present time-efficient deterministic algorithms for rightmost parsings, summarized by Theorems 5.1 and 5.2 below.

Theorem 5.1. *Let $x[1..n]$ be over linearly-sortable alphabet. Given an LZ-End factorization $x = f_1 \dots f_z$, we can compute its rightmost LZ-End parsing in $\mathcal{O}(n + z\sqrt{\log z})$ time and $\mathcal{O}(n)$ words of space. For the greedy LZ-End factorization, we achieve $\mathcal{O}(n)$ time and words of space.*

Theorem 5.2. *Let $x[1..n]$ be over linearly-sortable alphabet. Given an LZ-like factorization $x = f_1 \dots f_z$, we can compute the rightmost previous occurrence of all referencing phrases*

(a) *of length $\Omega(\log^{6.66} n / \log^2 \sigma)$ in $\mathcal{O}(n / \log_\sigma n)$ time and words of space, assuming that $x[1..n]$ is packed over integer alphabet $[0, \sigma)$,*

(b) *f_k with $k \in F \subseteq [1, z]$ in $\mathcal{O}(n + |F|\, d^\epsilon)$ time, where $d = |\{f_{k'} \mid k' \in F\}| \le |F|$ and $\epsilon \in \mathbb{R}^+$ is an arbitrarily small constant,*

(c) *f_k with $|\{k' \in [1, z] \mid f_{k'} = f_k\}| = \mathcal{O}(\log n)$ in $\mathcal{O}(n)$ time, and*

(d) *with rightmost previous occurrence at distance $\mathcal{O}(\log n)$ in $\mathcal{O}(n)$ time.*

Unless explicitly stated otherwise, the space complexity is $\mathcal{O}(n)$ words.

The remainder of the chapter is structured as follows. First, we introduce preliminary definitions and data structures (Section 5.1). We provide the solution for rightmost parsings of LZ-End factorizations (Theorem 5.1) in Section 5.2. The algorithms for partially solving general rightmost LZ-like parsings (Theorem 5.2) are presented in Section 5.3.

5.1 Preliminaries

We assume that the reader is familiar with tries [Fre60]. The *suffix tree* [Wei73] of x is the compact trie of all suffixes of $x\$$, where $\$$ is smaller than all symbols from the alphabet. Each leaf corresponds to a suffix of x and is labeled with the start position of this suffix. The outgoing edges of each node are arranged in increasing order of the first symbol of the respective edge label. Hence the leaves are ordered from left to right in lexicographical order of suffixes. In the present model of computation, the suffix tree can be computed in $\mathcal{O}(n)$ time and space (see, e.g., [Far97, FFM00]). The suffix array SA of x is the unique permutation of $[1, n]$ that lexicographically sorts the suffixes, i.e., $\forall i \in [1, n) : x[\mathsf{SA}[i]..n] \prec x[\mathsf{SA}[i + 1]..n]$ [MM93]. Equivalently, it consists of the leaf-labels of the suffix tree in left-to-right order and can therefore be constructed from the suffix tree in linear time.

From now on, we use z (commonly used to denote the number of phrases in the exact LZ factorization) to denote the number of phrases in the factorization at hand, even if it is an LZ-like or LZ-End factorization. Instead of saying that we compute the rightmost source of f_k, we simply say that we *resolve* f_k.

5.2 Computing Rightmost LZ-End Parsings

In this section, we provide the solutions for Theorem 5.1. We exploit the fact that an LZ-End phrase only has to choose from less than z sources, while a general LZ-like phrase has to consider up to $\Omega(n)$ possible sources. This makes the computation significantly easier for LZ-End factorizations.

5.2.1 Rightmost Greedy LZ-End Parsing

We start by computing an arbitrary LZ-End aligned source for each referencing phrase f_k with a technique that is similar to what was done in [Fis+18]. We compute the suffix tree of $\mathsf{rev}(x)$. Apart from the parent operation, we will not need any navigation in the trie. Hence we can construct it in $\mathcal{O}(n)$ deterministic time using standard techniques (e.g., bottom-up from the suffix array of $\mathsf{rev}(x)$). Now we process each $k \in [1, z]$ separately and in increasing order. We start at the leaf that corresponds to $\mathsf{rev}(f_1 \dots f_k)$, and traverse the path towards the root one node at a time. Whenever we see a node that has not been annotated yet, we annotate it with k. As soon as we see a node that already has some annotation k', we stop and, unless the reached node is the root, report that f_k has an LZ-End aligned source as a suffix of $f_1 \dots f_{k'}$. If we reach the root, then f_k is a literal phrase. It is easy to see that this indeed reports correct sources. The correctness follows readily from the fact that the lowest common ancestor of the leaves corresponding to a referencing phrase and its phrase aligned sources will be annotated by one of the sources. We annotate each node of the suffix tree at most once, hence the time is $\mathcal{O}(n)$.

The computed sources are already rightmost for all phrases that only have a single LZ-End aligned source. It remains to correct the sources of phrases that have multiple LZ-End aligned sources, for which we observe the following.

Proposition 5.3. *Let f_k be a referencing phrase in the greedy LZ-End factorization, and let $k', k'' \in [1, k)$ with $k'' < k'$ be such that f_k is a suffix of both $f_1 f_2 \dots f_{k'}$ and $f_1 f_2 \dots f_{k''}$. Then f_k is a suffix of $f_{k'-1} f_{k'}$.*

Proof. If f_k is a suffix of $f_1 f_2 \dots f_{k'}$ but not of $f_{k'-1} f_{k'}$, then $f_{k'-1} f_{k'}$ is a suffix of f_k. Since f_k is a suffix of $f_1 f_2 \dots f_{k''}$, this implies that $f_{k'-1} f_{k'}$ is a suffix of $f_1 f_2 \dots f_{k''}$. Hence $f_{k'-1} f_{k'}$ has a previous occurrence that satisfies the LZ-End property. Thus, $f_{k'-1}$ is not of maximal length, which contradicts the definition of the greedy LZ-End factorization. □

We compute a compacted trie over the set of strings defined by $\mathsf{rev}(f_{k'-1} f_{k'})$ for $k' \in [2, z]$. Note that the total length of the strings is less than $2n$. We make the respective nodes that spell $\mathsf{rev}(f_{k'})$ and $\mathsf{rev}(f_{k'-1} f_{k'})$ explicit (if they are not explicit already), and store pointers to these nodes. We will not need fast navigation on the trie; as before, we only need the parent operation. Hence we can construct the trie in $\mathcal{O}(n)$ deterministic time using standard techniques (e.g., from the suffix array of $\mathsf{rev}(f_1 f_2 \# f_2 f_3 \# \dots \# f_{z-1} f_z)$ where # is a special separator symbol). Now we process the phrase pairs $f_{k'-1} f_{k'}$ with $k' \in [2, z]$ from right to left. Whenever we finish processing a pair, we annotate the node that spells $\mathsf{rev}(f_{k'})$ with k' (indicating that the rightmost LZ-End aligned source of $f_{k'}$ has not been found yet). Before adding this annotation, we first check if $f_{k'-1} f_{k'}$ resolves other phrases. For this purpose, we traverse the path from the leaf that spells $\mathsf{rev}(f_{k'-1} f_{k'})$ to the root of the trie. For each node on the path, we check if it has been annotated with some value k. If we find such an annotation, then the corresponding node spells $\mathsf{rev}(f_k)$, and f_k is a suffix of $f_{k'-1} f_{k'}$. Hence we store $|f_1 f_2 \dots f_{k'}| - |f_k| + 1$ as the maximal LZ-End aligned source of f_k, and remove the annotation of the node. By Proposition 5.3 and the right-to-left order of processing, we correctly find the rightmost LZ-End aligned source of any phrase that has multiple LZ-End aligned sources.

A node might spell the reversal of a phrase that has multiple occurrences in the parsing. Nevertheless, each node has at most one annotation at any given point in time. This is because we annotate the node that spells $\mathsf{rev}(f_{k'})$ only after we finish processing pair $f_{k'-1} f_{k'}$. If the node is already annotated with some $k > k'$ (because $f_k = f_{k'}$), then we also find the source $|f_1 f_2 \dots f_{k'}| - |f_k| + 1$ of f_k while processing pair $f_{k'-1} f_{k'}$, and hence we remove annotation k before adding annotation k'.

We need $\mathcal{O}(n)$ time for computing the trie. Processing a pair $f_{k'-1} f_{k'}$ takes time linear in the depth of the node that spells $\mathsf{rev}(f_{k'-1} f_{k'})$. This is limited by $\mathcal{O}(|f_{k'-1} f_{k'}|)$, which sums to $\mathcal{O}(n)$ over all phrase pairs. The space for the trie is $\mathcal{O}(n)$. Hence we have shown Theorem 5.1 for the greedy LZ-End factorization.

5.2.2 Rightmost (Arbitrary) LZ-End Parsing

If the given LZ-End factorization does not satisfy the greedy property, then Proposition 5.3 no longer holds. However, each referencing phrase f_k is still a suffix of some $f_1 f_2 \dots f_{k'}$ with $k' \in [1, k)$, which limits the number of possible sources. We will again exploit properties of the co-lexicographical order of prefixes. We build the suffix array of the reversed text $\mathsf{rev}(x)$, and use filtering and rank reduction to obtain in $\mathcal{O}(n)$ time the unique permutation co of $[1, z]$ that satisfies $\forall k' \in [1, z) : \mathsf{rev}(f_1 f_2 \ldots f_{\mathsf{co}(k')}) \prec \mathsf{rev}(f_1 f_2 \ldots f_{\mathsf{co}(k'+1)})$. This permutation rearranges the prefixes that end at phrase boundaries in co-lexicographical order. We also compute its inverse permutation co^{-1}.

Next, we compute a compacted trie that contains for each $k' \in [1, z]$ the reversed prefix $\mathsf{rev}(f_1 f_2 \ldots f_{k'})$ of the text. We make the respective nodes that spell $\mathsf{rev}(f_{k'})$ and $\mathsf{rev}(f_1 f_2 \ldots f_{k'})$ explicit (if they are not explicit already), and store pointers to these nodes. We annotate the node that spells $\mathsf{rev}(f_1 f_2 \ldots f_{k'})$ with its co-lexicographical rank $\mathsf{co}^{-1}(k')$. Additionally, we annotate the node that spells $\mathsf{rev}(f_{k'})$ with its co-lexicographical range, which is given by the respectively smallest and largest co-lexicographical ranks $c_{k'}^{\min}$ and $c_{k'}^{\max}$ that were used to annotate any of its descendants (or itself). Again, we do not need fast navigation on the trie; for writing the annotations, it suffices if we can perform a preorder-traversal in linear time. Hence we can construct the trie and its annotations in $\mathcal{O}(n)$ deterministic time using standard techniques (e.g., from the suffix array of $\mathsf{rev}(x)$).

Now we show how to find the rightmost LZ-End aligned source of referencing phrase f_k. We have already annotated the node that spells $\mathsf{rev}(f_k)$ with the co-lexicographical range $[c_k^{\min}, c_k^{\max}]$, and we have also computed the permutation co. We assume that the permutation co is stored in an array. Note that, by design of the trie, the range $\mathsf{co}[c_k^{\min}..c_k^{\max}]$ contains exactly all the k' for which f_k is a suffix of $f_1 f_2 \ldots f_{k'}$. Hence finding the rightmost LZ-End aligned source of f_k is equivalent to answering the following so-called *range predecessor query*. Given the range $[c_k^{\min}, c_k^{\max}] \subseteq [1, z]$ and the threshold k, find the largest value $k' < k$ in $\mathsf{co}[c^{\min}..c^{\max}]$. Then, the rightmost LZ-End aligned source of f_k is $|f_1 f_2 \ldots f_{k'}| - |f_k| + 1$.

Belazzougui and Puglisi show how to compute a data structure in $\mathcal{O}(z\sqrt{\log z})$ time and $\mathcal{O}(z)$ space that answers range predecessor queries on a permutation of $[1, z]$ in $\mathcal{O}(\log^{\epsilon} z)$ time (for any constant $0 < \epsilon < 1$) [BP16]. We issue less than z queries, and thus the total construction and query time is $\mathcal{O}(z\sqrt{\log z})$. The total time for computing the rightmost parsing (including the construction of the trie) is $\mathcal{O}(n + z\sqrt{\log z})$, and the total space is $\mathcal{O}(n)$. Hence we have shown Theorem 5.1 for an arbitrary LZ-End factorization.

5.3 Partially Solving Rightmost LZ-Like Parsings

In this section, we show how to efficiently compute the rightmost sources for some subsets of the phrases of an LZ-like factorization (Theorem 5.2).

5.3.1 Long Phrases

Belazzougui and Puglisi [BP16] find the rightmost sources of all phrases of length $\Omega(\log^5 n)$ in $\mathcal{O}(n)$ time and $\mathcal{O}(n/\log_\sigma n)$ space. We show a similar result for resolving all phrases of length $\Omega(\log^{33/5+\epsilon} n/\log^2 \sigma)$ in $\mathcal{O}(n/\log_\sigma n)$ time and space. The main contribution here is that we achieve sublinear time. The solution works for an arbitrary LZ-like factorization $x = f_1 f_2 \ldots f_z$. The techniques are quite similar to the computation of long phrases in Chapter 4.

Let $\delta = \Omega(\log^2 n/\log \sigma)$ be a parameter to be fixed later. We start by performing a preprocessing as follows. In $\mathcal{O}(n/\log_\sigma n)$ time, we compute the reversed text $\mathsf{rev}(x)$

as described in [Bel+16, Section 6.2] (essentially, we use a precomputed lookup table to reverse the text one half-word rather than one symbol at a time). We consider a set $\mathcal{D} = \{d \in [1, n] \mid d \equiv 0 \pmod{\delta}\}$ of $m = |\mathcal{D}| = \mathcal{O}(\frac{n}{\delta})$ regularly sampled positions. We construct the respectively unique permutations pref and suf of $[1, m]$ such that for every $h \in [1, m)$ it holds $x[\mathsf{suf}(h)\delta..n] \prec x[\mathsf{suf}(h+1)\delta..n]$ and $\mathsf{rev}(x[1..\mathsf{pref}(h)\delta]) \prec \mathsf{rev}(x[1..\mathsf{pref}(h+1)\delta])$ (these are sparse suffix arrays of the string and its reversal). We use comparison sorting and obtain the permutations with $\mathcal{O}(m \log m) \subseteq \mathcal{O}(\frac{n}{\delta} \log n) \subset \mathcal{O}(n / \log_\sigma n)$ lexicographical comparisons between suffixes of either x or $\mathsf{rev}(x)$. With the LCE data structure by Kempa and Kociumaka Lemma 4.4 (constructed for both x and $\mathsf{rev}(x)$), each lexicographical comparison takes constant time. The data structure can be constructed in $\mathcal{O}(n / \log_\sigma n)$ time and space. We use $\mathcal{O}(m \log m) \subseteq \mathcal{O}(\frac{n}{\delta} \log n) \subset \mathcal{O}(n \log \sigma)$ bits of space to store pref, suf, and their respective inverse permutations $\mathsf{pref\text{-}rank}$ and $\mathsf{suf\text{-}rank}$.

A long phrase is of length at least $\gamma > \delta$, where γ is another parameter to be fixed later. When resolving a long phrase f_k with rightmost source j and destination i, we will use the fact that $j + q$ with $q = (\delta - (j \bmod \delta)) \in [1, \delta]$ is a sample position. For now, assume that we know the value of q in advance (we will later simply try all the possible values of q). Finding the rightmost source of f_k means that we have to find the rightmost sample position $h\delta < i + q$ with $x[h\delta - q..h\delta] = x[i..i + q]$ and $x[h\delta..h\delta - q + |f_k|) = x[i + q..i + |f_k|)$. Note that the co-lexicographical order groups together prefixes that share a long suffix, and hence all the values of h for which $x[i..i + q]$ is a suffix of $x[1....h\delta]$ form a consecutive interval $\mathsf{pref}[p_1..p_2]$ (we treat the permutations like arrays). We can find the boundaries p_1 and p_2 by binary searching in pref for the respectively co-lexicographically minimal and maximal prefixes of x that have suffix $x[i..i + q]$. This takes $\mathcal{O}(\log m)$ time because we can perform each LCE computation and lexicographical comparison in constant time using the same LCE data structure as before. Similarly, it takes $\mathcal{O}(\log m)$ time to compute the interval $\mathsf{suf}[s_1..s_2]$ that contains exactly the values of h for which $x[i + q..i + |f_k|)$ is a prefix of $x[h\delta..n]$.

We associate a three-dimensional point $(\mathsf{pref\text{-}rank}(h), \mathsf{suf\text{-}rank}(h), h)$ with each sample position. For resolving the phrase, we have to find the point $(p, s, \hat{h})$ with $p \in [p_1, p_2]$, $s \in [s_1, s_2]$, and maximal value $\hat{h}\delta < i + q$ (or equivalently $h < \frac{i+q}{\delta}$). Given this point, it is easy to compute the rightmost source $\hat{h}\delta - q$ of f_k. For solving the geometric query, we use a data structure for three-dimensional orthogonal range searching [CT18, Theorem 4]. For our m points from $[1, m]^3$, it can be constructed in $\mathcal{O}(m \log^{8/5+\epsilon} m)$ time and space (for any constant $\epsilon \in \mathbb{R}^+$). Given a three-dimensional six-sided orthogonal query range, it returns a point in the range or reports that it is empty in $\mathcal{O}(\log^2 m)$ time (the precise bound is slightly better, but not needed for our purposes). For our queries, we have to find the point with maximal coordinate in the third dimension. Thus, we binary search for this point with $\mathcal{O}(\log n)$ queries to the geometric data structure, which increases the query time to $\mathcal{O}(\log^3 n)$. Note that this dominates the $\mathcal{O}(\log m)$ time needed to compute the query range. Finally, we do not actually know the value of q in advance. Hence we try all the possible $q \in [1, \delta]$. For each of them, we compute the query range and find the rightmost admitted source in $\mathcal{O}(\log^3 n)$ time. Thus, the time needed per phrase is $\mathcal{O}(\delta \cdot \log^3 n)$.

We need $\mathcal{O}(n / \log_\sigma n)$ time for computing the (co-)lexicographically sorted permutations of samples, $\mathcal{O}(\frac{n}{\delta} \log^{8/5+\epsilon} n)$ time for computing the geometric data structure, and $\mathcal{O}(\frac{n\delta}{\gamma} \cdot \log^3 n)$ time for actually resolving the phrases. We want δ to be small in order to minimize the time for resolving phrases. On the other hand, the time needed for computing the geometric data structure should become $\mathcal{O}(n/\log_\sigma n)$. Hence we use $\delta = \Theta(\log^{13/5+\epsilon} n/\log\sigma)$, which achieves the desired construction time and implies that we take $\mathcal{O}(\frac{n}{\gamma} \cdot \log^{28/5+\epsilon} n/\log\sigma)$ time for resolving phrases. Thus, in order to achieve $\mathcal{O}(n/\log_\sigma n)$ time, long phrases have to be of length at least $\gamma = \Omega(\log^{33/5+\epsilon} n/\log^2\sigma) \subset \Omega(\log^{6.66} n/\log^2\sigma)$. For all steps (including the geometric data structure), the space is linear in the time spent, and hence it is $\mathcal{O}(n/\log_\sigma n)$. This concludes the proof of Theorem 5.2 (a).

5.3.2 Arbitrary Subsets of Phrases

Now we show how to solve an arbitrary subset of phrases of any LZ-like factorization $x = f_1 f_2 \dots f_z$. The subset is given by $F \subseteq [1, z]$, and the time complexity depends on $d = |\{f_k \mid k \in F\}| \leq F$, i.e., on the number of distinct phrases in the subset. In a slight abuse of terminology, we will say that f_k is a phrase from F if $k \in F$. We show how to resolve all phrases from F in $\mathcal{O}(\frac{n}{\epsilon} + |F|\, d^\epsilon)$ time and $\mathcal{O}(\frac{n}{\epsilon})$ space for arbitrary $\epsilon \in \mathbb{R}^+$ with $\epsilon \leq \frac{1}{2}$, or $\mathcal{O}(n + |F|\, d^\epsilon)$ time and $\mathcal{O}(n)$ space for constant ϵ. If the string is highly-compressible, say, $z = \mathcal{O}(n^{1-\epsilon})$, then the time for all phrases (i.e., $F = [1, z]$) is $\mathcal{O}(n)$. The idea is to use range maximum data structures to find the rightmost sources. We note that this solution is very similar to [FNV13], and mostly differs in the choice of the range maximum data structure.

We start with the following preprocessing. We arrange the distinct phrases of F into a tree of $d + 1$ nodes, and we start using the terms node and phrase interchangeably (even though multiple phrases may refer to the same node). The parent of phrase f_k is the longest phrase $f_{k'}$ from F that is a proper prefix of f_k (or the artificial root node ε if $f_{k'}$ does not exist), and we call this tree the *phrase trie*. This is a slight abuse of terminology, since the tree is only similar to a trie. An example is provided in Figure 5.2. We annotate f_k with its *preorder-number* p_k, which is the rank of f_k in a preorder-traversal of the phrase trie, as well as the maximal preorder-number q_k of a descendant of f_k. We also annotate each text position i with the preorder-number of the longest phrase from F that is a prefix of $x[i..n]$, if any. This concludes the preprocessing.

In order to resolve the phrases, we traverse x from left to right and track in an array $A[1..d]$ the last position at which we encountered each preorder-number as an annotation. When we reach the destination i of some phrase f_k from F, the rightmost previous occurrence will be at position $\max_{p\in[p_k,q_k]} A[p]$ (the solution of a *range maximum query*), as any occurrence of f_k is annotated with either p_k or the preorder-number $p_{k'}$ of a phrase $f_{k'}$ that is a descendant of f_k in the phrase trie. Hence, if we have a dynamic data structure for range maximum queries, then we can compute each rightmost occurrence with one query.

The phrase trie can be obtained as follows. We compute the suffix tree for the string $x' = x\#_0 f_1 \#_1 f_2 \#_2 \dots \#_{z-1} f_z \#_z$, where each $\#_k$ is a unique separator symbol. This takes $\mathcal{O}(n)$ time. For any f_k from F, the parent of the leaf that spells suffix $f_k \#_k \dots$ is exactly the node that spells f_k. Thus, we can mark the d nodes that spell phrases from F in $\mathcal{O}(|F|)$ time. It is then easy to compute the nearest marked ancestor of each node in $\mathcal{O}(n)$ time. The phrase trie is obtained by creating a new tree that contains only the marked nodes and an artificial root. The new parent of a marked node is its nearest marked ancestor (or the artificial root node if it does not exist). Finally, we compute the preorder-numbers in the phrase trie, and

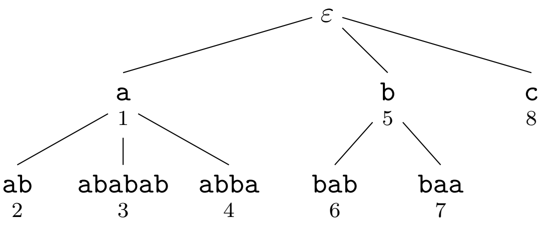


Figure 5.2: The phrase trie (where F contains all the distinct phrases) for the LZ factorization a|b|b|a|ab|ababab|bab|c|abba|baa. Below each node is the preorder-number.

also annotate the corresponding marked nodes in the suffix tree with these numbers. Then, the annotation of text position i is the annotation of the nearest marked ancestor of the leaf that corresponds to text position i in the suffix tree. Hence we obtain the annotations in $\mathcal{O}(n)$ time.

We have to solve dynamic range maximum queries (RMQ) for A. The updates are incremental in the sense that every update is the new global maximum (i.e., the rightmost text position processed so far). Therefore, we can maintain a dynamic RMQ data structure for A with $\mathcal{O}(\frac{1}{\epsilon})$ time updates and $\mathcal{O}(d^\epsilon)$ time queries using the standard technique of square-root decomposition, generalized to arbitrary ϵ. For $\epsilon = \frac{1}{2}$, we split A into blocks of size $\Theta(\sqrt{d})$ and maintain the maximum of each block, which we can update in constant time whenever we update an entry of A. To answer queries we need to scan at most $\mathcal{O}(\sqrt{d})$ elements in A that are in blocks that are only partially overlapped by the query range. Then, we also scan the $\mathcal{O}(\sqrt{d})$ maxima of blocks that are fully contained in the query range. Thus, we take $\mathcal{O}(\sqrt{d})$ time. This generalizes to smaller ϵ by recursively subdividing the blocks into $\frac{1}{\epsilon}$ layers, leading to $\mathcal{O}(\frac{1}{\epsilon})$ update time and $\mathcal{O}(d^\epsilon)$ query time. Each phrase in F incurs a range query and each text position an update. We perform $|F|$ range queries and n updates in $\mathcal{O}(\frac{n}{\epsilon} + |F|\, d^\epsilon)$ time. This concludes the proof of Theorem 5.2 (*b*).

5.3.3 Infrequent Phrases

Given an LZ-like parsing $x = f_1 \dots f_z$, we say that a phrase f_k is *infrequent* if $|\{k' \in [1, z] \mid f_{k'} = f_k\}| = \mathcal{O}(\log n)$, i.e., if it occurs at most $\mathcal{O}(\log n)$ times in the parsing. We stress that such a phrase can have more than $\mathcal{O}(\log n)$ occurrences in the text. We now show how to resolve all infrequent phrases in $\mathcal{O}(n)$ time, and we begin by establishing a data structure that is crucial for our solution.

Lemma 5.4. *Let $m, n \in [1, 2^w]$. For a tree of m nodes, labeled with preorder-numbers from $[1, m]$, after an $\mathcal{O}(m) + o(n)$ time preprocessing, and in $\mathcal{O}(m) + o(n)$ space, we can maintain a data structure for nearest marked ancestor queries with the following operations.*

- *mark/unmark a node $i \in [1, m]$ with d_i descendants in $\mathcal{O}(1 + d_i / \log n)$ time*
- *check if a node $i \in [1, m]$ is marked in $\mathcal{O}(1)$ time*
- *check if a node $i \in [1, m]$ has a marked ancestor in $\mathcal{O}(1)$ time*
- *output the nearest marked ancestor j of a node $i \in [1, m]$ in $\mathcal{O}(1 + d_j / \log n)$ time, where d_j is the number of descendants of j.*

Proof. We compute the balanced parenthesis sequence (BPS, see, e.g., [MR01, Nav16] or Chapters 8 and 9) $B[1..2m]$ of the tree by re-running the traversal used to obtain preorder-numbers (with an artificial parent edge for the root to start the traversal). When we walk down the edge to node i, we append i's opening parenthesis to B, when we walk up the edge from node i we append its closing one. The ith opening parenthesis (in left-to-right order) belongs to node i, and between i's opening and closing parentheses there are exactly all the parentheses corresponding to descendants of i. We preprocess B such that, given node $i \in [1, m]$, we can look up the positions $\mathsf{open}(i)$ and $\mathsf{close}(i)$ of its respective opening and closing parentheses in B in constant time. This is possible with a simply linear scan in $\mathcal{O}(m)$ time and space. For open, we also compute the inverse mapping $\mathsf{prenum}(\mathsf{open}(i)) = i$.

We use two additional bitvectors $A[1..2m]$ and $R[1..2m]$, both initialized with zeroes. When asked to mark node i, we set the bits $A[\mathsf{open}(i)]$ and $A[\mathsf{close}(i)]$ (marking the respective parentheses in B as *active*), and additionally we set the entire range $R[\mathsf{open}(i) + 1..\mathsf{close}(i)]$ one word at a time (indicating that nodes whose opening parentheses lie in this region have a marked ancestor). If i has d_i descendants, then it holds $\mathsf{close}(i) - \mathsf{open}(i) = 1 + 2d_i$, and thus the procedure takes $\mathcal{O}(1 + d_i/w) \subseteq \mathcal{O}(1 + d_i / \log n)$ time. A node i is marked if and only if $A[\mathsf{open}(i)]$ is set, and it has a marked ancestor if and only if $R[\mathsf{open}(i)]$ is set (we do not consider a node to be its own ancestor). Both can be tested in constant time. Finding the nearest marked ancestor of i is more involved, and we explain it later.

When unmarking a node i, we unset the bits $A[\mathsf{open}(i)]$ and $A[\mathsf{close}(i)]$. If i currently has a marked ancestor, then there is no need to unset the range in R associated with i. Otherwise, we cannot simply unset the entire range $R[\mathsf{open}(i) + 1..\mathsf{close}(i)]$ because it may have also been set by descendants of i. Hence we have to leave segments corresponding to marked nodes untouched. Starting at position $k = \mathsf{open}(i) + 1$, we scan $A[k..\mathsf{close}(i)]$ from left to right and keep track of the excess of opening active parentheses, which is initially $e = 0$. We perform the scan in blocks of size $w' = \lfloor \log n / 7 \rfloor$. Processing $A[k..k + w')$ works as follows. We scan the block from left to right. For each position $A[j]$ in the block, we first check if currently $e = 0$. If yes, then we unset bit $R[j]$. Afterwards, if $A[j] = 1$, we increment e if $B[j]$ is an opening parenthesis, and decrement e otherwise. Once we reach the end of the block, we increase k by w' and continue with the next block, until we reach position $\mathsf{close}(i)$. This way, we avoid unsetting parts of R that have to remain active. However, the procedure takes $\mathcal{O}(d_i)$ time, or $\mathcal{O}(w')$ time per block.

The processing of block $A[k..k+w')$ depends only on $A[k..k+w')$, $B[k..k+w')$, $R[k..k+w')$ and $\min(e, w')$ (if $e > w'$, then the excess cannot reach 0 while processing the block). Thus it depends on $3w' + \log w' \leq \log n/2$ bits of information, and in principle there are fewer than $2^{\log n/2} = \sqrt{n}$ distinct instances of the procedure. In a lookup table, we precompute for each possible $A[k..k+w')$, $B[k..k+w')$, $R[k..k+w')$, and $\min(e, w')$ the result of the procedure, i.e., the total increment or decrement that we have to apply to e, and the new value of $R[k..k+w')$. The lookup table has $\mathcal{O}(\sqrt{n})$ entries, and each of them can be computed naively in $\mathcal{O}(\text{polylog}(n))$ time. Using the table, an entire block $A[k..k+w')$ can be processed in constant time (and handling the last block that is possibly shorter than w' can be solved with additional lookup tables for each shorter block length). Thus, we can unmark a node in $\mathcal{O}(1 + d_i/w') = \mathcal{O}(1 + d_i/\log n)$ time.

We have already shown how to check if i has a marked ancestor in constant time. If we also want to output the nearest marked ancestor, then we start at position $o = \mathsf{open}(i)$. Similarly to the technique for unmarking nodes, we now scan $A[1..o]$ and $B[1..o]$ from *right to left* and keep track of the excess of active *closing* parentheses. As soon as the excess becomes negative, we have found the opening parenthesis of the nearest marked ancestor. If this parenthesis is at position o', then the ancestor is $j = \mathsf{prenum}(o')$. We can implement this procedure with lookup tables (similar to unmarking nodes), and thus it takes $\mathcal{O}(1 + d_j/\log n)$ time, where d_j is the number of descendants of j. □

Resolving the Phrases Now we are ready to resolve the infrequent phrases. We first build the phrase trie including only the infrequent phrases, and compute the mapping from phrases to preorder-numbers. We also annotate each text position i with the preorder-number corresponding to the longest infrequent phrase that is a prefix of $x[i..n]$ (this works just like in Section 5.3.2). We prepare the phrase trie for nearest marked ancestor queries with Lemma 5.4.

Now we scan x from right to left. For each text position i, we first try to resolve phrases, which we explain in a moment. After that, if i is the destination of a phrase f_k with preorder-number p_k, we mark node p_k in the phrase trie (indicating that the phrase needs to be resolved). We also store $u[p_k] = k$ in an array of size at most z. This is necessary because the preorder-numbers correspond to the *distinct* infrequent phrases, and thus the mapping from preorder-numbers to phrases is not necessarily injective. Later, we resolve f_k by discovering that node p_k is marked, and we will then need to be able to look up $k = u[p_k]$. Note that we never try to resolve two phrases with the same preorder-number at the same time, since the one further to the left would have already resolved the one further to the right.

For every text position i, if its annotation is q_i, we check if q_i has a marked ancestor. If this is the case, then we obtain the nearest marked ancestor p of q_i, which corresponds to phrase $f_{u[p]}$. By the construction of the phrase trie and the annotations of text positions, $f_{u[p]}$ is a prefix of $x[i..n]$. Since we have not unmarked the node yet, and due to the right-to-left processing order, it follows that i is the rightmost source of $f_{u[p]}$. We unmark node p.

Analyzing the Complexity The preprocessing for the nearest marked ancestor structure takes $\mathcal{O}(z) + o(n)$ time and space. For each text position, annotated with q_i, we check if q_i has a marked ancestor in overall $\mathcal{O}(n)$ time. Whenever this is the case, we also find its nearest marked ancestor. However, we will then also immediately unmark the nearest marked ancestor, and thus the total time for finding marked ancestors is the same as the time for unmarking nodes, which in turn is bounded by the time for marking them.

Now we analyze the total time for marking nodes. Let m be the number of nodes in the phrase trie (or equivalently the number of distinct infrequent phrases). We mark nodes $\mathcal{O}(z)$ times, and thus the total time is $\mathcal{O}(z)$ plus the sum of all the $\mathcal{O}(d_i/\log n)$ terms. For now, we assume that each node gets marked exactly once. Then the time is $\mathcal{O}(\frac{1}{\log n} \cdot \sum_{i=1}^{m} d_i)$. Let a_i denote the number of ancestors of a node i, and observe that $\sum_{i=1}^{m} d_i = \sum_{i=1}^{m} a_i$ (because in both sums each combination of descendant and ancestor contributes value 1 to the sum). If node i corresponds to a phrase f_k, then the number of ancestors of i is bounded by $a_i < |f_k|$, since each ancestor represents a phrase that is a proper prefix of f_k. Hence the time is $\mathcal{O}(\frac{1}{\log n} \cdot \sum_{i=1}^{z} |f_k|) = \mathcal{O}(n/\log n)$. We assumed that each node gets marked exactly once. Since we only consider infrequent phrases, each node gets marked $\mathcal{O}(\log n)$ times, and thus the time is $\mathcal{O}(n)$. This concludes the proof of Theorem 5.2(c).

5.3.4 Close Phrases

Given an LZ-like parsing $x = f_1 \dots f_z$, we say that a phrase f_k with destination i is *close* if its rightmost source is j and $i - j = \mathcal{O}(\log n)$. We now show how to resolve all close phrases in $\mathcal{O}(n)$ time. Let $\gamma = \Theta(\log n)$. If a phrase at destination i is of length at least γ, then we can afford $\mathcal{O}(\log n)$ time to resolve it. We consider each $j \in [i - r, i)$ with $r = \mathcal{O}(\log n)$ as a potential source. Checking if j is a source of i takes constant time with an LCE data structure (e.g., [KK19]). Thus we can resolve all close phrases of length at least γ in $\mathcal{O}(n)$ time.

For the phrases of length less than γ, we extract copies of overlapping segments $s_1, \dots, s_{\lfloor n/2\gamma \rfloor}$ of length 4γ where $\forall i \in [1, \lfloor n/2\gamma \rfloor] : s_i = x[1 + 2(i-1)\gamma \dots \min(2(i+1)\gamma, n)]$. We modify each segment s_i by rank-reducing the alphabet of s_i to (a subset of) $[1, 4\gamma]$, which takes $\mathcal{O}(n)$ total time by radix sorting all segments in batch. Then, we offset the alphabets such that s_i is over alphabet $[1 + 4(i-1)\gamma, 4i\gamma]$. We concatenate all segments s_i into $x' = s_1 s_2 \dots s_{\lfloor n/2\gamma \rfloor}$.

Each phrase of length less than γ is fully contained in the right half of at least one segment (apart from possible phrases with destination in the first 2γ position of x, which we solve with the LCE data structure in $\mathcal{O}(\text{polylog}(n))$ time). We map each phrase of length less than γ to a corresponding destination in x' such that if the destination is within some segment s_j then the phrase is fully contained in the right half of s_j. This results in a subset of an LZ-like factorization of x'. Since the segments have disjoint alphabets, all phrases in the subset are infrequent and can be solved with Theorem 5.2(c). We only have to map the sources back to original text positions, which is easily done in linear time. Hence we have shown Theorem 5.2(d).

5.4 Conclusion

We presented new algorithms for computing rightmost LZ-End and LZ-like parsings. Unfortunately, the question whether rightmost LZ-like parsings can be computed in linear time remains open. With Theorem 5.2, we identify the hard instances of the problem. As shown in Theorem 5.2(a) and (c), resolving a phrase is only problematic if it is short (with length polylogarithmic in n) *and* has many occurrences in the parsing (at least logarithmically many in n). The techniques for infrequent phrases can likely be generalized such that we can resolve all phrases with polylogarithmically many occurrences in the parsing. It remains to be shown if this sufficiently reduces the number of remaining phrases, as well as the complexity of their structure, such that they can be resolved in linear time.

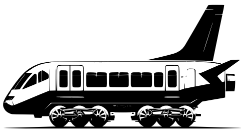
II

Computing the Lyndon Array

Computing the Lyndon Array

II Introduction and Related Work

A Lyndon word is a string that is lexicographically smaller than all of its non-trivial suffixes.[2] For example, `amtrak` is not a Lyndon word because of its suffix `ak` $\prec$ `amtrak`. On the other hand, `airbus` is a Lyndon word. While Lyndon words are named after and often attributed to the work of American mathematician Roger Lyndon (see [Lyn54, Lot83]), they were simultaneously discovered and researched by Soviet mathematician Anatoly Shirshov [Shi58, Shi09]. Hence they are sometimes called Lyndon-Shirshov words. They were described as *standard sequences* by Lyndon, and as *regular words* by Shirshov. By definition, Lyndon words assume that the alphabet is totally ordered, and, whenever such an order is present, they introduce additional structural elements in plain sequences of symbols. They have shown their usefulness for designing efficient string algorithms. For example, they underpinned the notion of critical positions [Lot83], the two-way string matching [CP91] and rotations of periodic strings [BJJ97].


[1]

Lyndon Trees and Arrays The Lyndon tree (see, e.g., [BCN02, HR03, BCN04]) of a Lyndon word x is a well-known combinatorial structure. It can be defined via the unique factorization $x = uv$ such that v is the longest proper Lyndon suffix of x. The string u is then also a Lyndon word. The Lyndon tree is a binary tree in which the root corresponds to the entire string x, and the left and respectively right subtree of the root are recursively defined as the Lyndon trees of u and v (where the Lyndon tree of a string of length one is a leaf node). The less common *left* Lyndon tree is based on the factorization $x = uv$ such that u is the longest proper Lyndon prefix of x [BC22] (we will not consider it any further and only mention it here for completeness). Hohlweg and Reutenauer [HR03] showed that the Lyndon tree is closely related to the lexicographical order of suffixes of a Lyndon word. More precisely, it is the Cartesian tree [Vui80] of the inverse suffix array [MM93] (see also [CR20]).

While the Lyndon tree stems from purely theoretical and combinatorial research, it has recently gained renewed attention due to its practical algorithmic applications. However, the more recent publications usually consider the Lyndon *array* (sometimes called Lyndon *table*), which stores for each position of a string the length of the longest Lyndon substring that starts at this position. In its essence, the Lyndon array

[1] Is it a train? Is it a plane? An Amtrak-Airbus, envisioned by DALL·E [Bet+23].

[2] In this context, the term word should not be confused with words of a word RAM.

is merely another representation of the Lyndon tree (see, e.g., [Ban+17, Lemma 5.4] or [Bad+22]) that can be more easily used for algorithmic applications. The perhaps most important application is as a tool for computing all maximal periodic substrings of a string [Ban+17], which is also the main topic of Chapter 10 in Part III. The Lyndon array has further been used to accelerate [OOB22] the computation of the suffix array [MM93], one of the major data structures in string algorithmics. Lyndon words are also used for computing the suffix array in [Man+13, Bai15, Bai16, BEF21]. A related topic of research is the reconstruction of a string from its Lyndon array or tree [Nak+17, Day+18, Nak+19].

Algorithms to Compute the Lyndon Array As mentioned before, the Lyndon tree is the Cartesian tree of the inverse suffix array. If the string is over linearly-sortable alphabet, then we can compute the Lyndon tree in linear time by first computing the inverse suffix array (e.g., using [KSB06]), and then constructing its Cartesian tree (in the obvious way using a stack). In order to obtain the Lyndon *array* instead, we only have to replace the Cartesian tree with the right-to-left minima tree [BFN11] of the inverse suffix array, which is equivalent to computing all next smaller values (see, e.g., [BSV93]) in the inverse suffix array. This approach was explicitly described in [Fra+16, Fig. 2], and it is by far not the only algorithm that computes the Lyndon array from the suffix array. Most notably, Baier's suffix array algorithm [Bai15, Bai16] produces the Lyndon array as a byproduct of computing the suffix array. This has lead to a closer analysis of the relation between the Lyndon array and suffix sorting [FPS17, FLS18, FL19, FL20]. There is at least one other suffix sorter that can be modified to simultaneously compute the suffix array and the Lyndon array [Lou+19]. The Lyndon array can also be constructed during inversion of the Burrows-Wheeler transform [Lou+18], which is closely related to the suffix array. A given string can be compressed into a grammar that efficiently simulates access to the Lyndon array [Tsu+20]. The deterministic construction algorithm of this grammar uses the suffix array.

Contributions

In Chapters 6 and 7, we introduce the first linear time algorithm for constructing the Lyndon array over general ordered alphabet. In Chapter 8, we improve this algorithm such that it computes the succinct version of the Lyndon array [Lou+18], which can be stored in around $2n$ bits of memory. The presented algorithm requires only $\mathcal{O}(n \log\log n / \log n)$ bits of additional working space, making it the so far most space efficient solution. These new algorithms are also the first linear time solutions that do not in any way rely on the suffix array. This results in significantly faster practical implementations, since computing the suffix array is relatively slow in practice. Finally, in Chapter 9, we show how to compute the succinct Lyndon array of a string packed over integer alphabet $[0, \sigma)$ in $\mathcal{O}(n / \log_\sigma n)$ time and $\mathcal{O}(n \log \sigma)$ bits of space. The new algorithms are useful tools for the computation of maximal periodic substrings (also known as runs). In fact, in Part III, Chapter 10, we use the algorithm from Chapters 7 and 8 to obtain the first linear time algorithm for computing runs over general ordered alphabet. We envision that the algorithm from Chapter 9 will lead to the first sublinear time algorithm for computing runs.

Chapter 6

6 The Lyndon Array and Nearest Smaller Suffixes

A Lyndon word is lexicographically smaller than all of its non-trivial suffixes. For example, `amtrak` is not a Lyndon word because of its suffix `ak`. On the other hand, `airbus` is a Lyndon word. This is only one of many equivalent definitions of Lyndon words. In this chapter, we show the equivalence of the most common definitions. We then introduce the Lyndon array, which identifies the longest Lyndon substring that starts at each position of a string (Section 6.1). Finally, we introduce the related nearest smaller suffix arrays, and show that they are (in some sense) equivalent to the Lyndon array (Section 6.2).

6.1 Lyndon Words and the Lyndon Array

We require the following self-evident properties of the lexicographical order.

Property 6.1. *For some alphabet Σ, let $u, v, s, t \in \Sigma^*$ be arbitrary strings.*

(*i*) *If $u \prec v$ and $|u| \geq |v|$, then $us \prec vt$.*

(*ii*) *If $u \prec v \preceq ut$, then $\exists w \in \Sigma^+ : v = uw$.*

Lemma 6.2 (below) shows the equivalence of the most common definitions of Lyndon words. Lemma 6.2 (*b*) is Lyndon's original definition of standard sequences. The equivalence of Lemma 6.2(*b*), (*c*) and (*e*) has previously been shown by Chen, Fox, and Lyndon [CFL58, Theorem 1.4; the sets $\mathfrak{A}'$, $\mathfrak{A}''$, and $\mathfrak{A}'''$ correspond to Lemma 6.2 (*b*), (*c*) and (*e*)]. The characterization of Lyndon words stated in Lemma 6.2 (*d*) is due to Charlier, Philibert, and Stipulanti [CPS19, Theorem 28 (iii)].

Lemma 6.2 (Lyndon Word [Lyn54, CFL58, CPS19])**.** *The following statements regarding a non-empty string $x \in \Sigma^+$ over totally ordered alphabet are equivalent:*

(*a*) *x is a Lyndon word.*

(*b*) *Every factorization $x = uv$ with $u, v \in \Sigma^+$ satisfies $x \prec v$.*
(x is lexicographically smaller than all of its non-trivial suffixes.)

(*c*) *Every factorization $x = uv$ with $u, v \in \Sigma^+$ satisfies $x \prec vu$.*
(x is lexicographically smaller than all of its non-trivial cyclic shifts.)

(*d*) *Every factorization $x = uv$ with $u, v \in \Sigma^+$ and Lyndon word v satisfies $x \prec v$. (x is lexicographically smaller than all of its non-trivial Lyndon suffixes.)*

(*e*) *$|x| = 1$, or $|x| > 1$ and there are Lyndon words u, v with $x = uv$ and $u \prec v$. (x is the concatenation of lexicographically strictly increasing Lyndon words.)*

Proof. By definition, (a) and (b) are equivalent (see [Lyn54, p. 203]; Lyndon words "have the property of preceding lexicographically all of their own proper terminal segments"). We show that each of the statements (c) to (e) is equivalent to (b).

Statement (*c*): It holds $(b) \implies (c)$ because $x \prec v$ always implies $x \prec vu$. For $(b) \impliedby (c)$, assume that x satisfies (c) but not (b). Then there is a factorization $x = uv$ with $u, v \in \Sigma^+$ such that $x \prec vu$, but $x \succ v$. It follows that v is a non-trivial prefix of x due to Property 6.1 (ii). Let $w \in \Sigma^+$ be the suffix of x such that $x = vw$, then it holds $x = vw \prec vu$. However, this implies $w \prec u$ and by Property 6.1 (i) also $wv \prec uv = x$, which contradicts the assumption that x satisfies (c).

Statement (*d*): Statements (d) and (e) recursively use shorter Lyndon words. For the remainder of the proof, we inductively assume that we have already proven the lemma for strings shorter than $|x|$. Strings of length 1 satisfy all statements.

Trivially, it holds $(b) \implies (d)$, and thus we only need to show $(b) \impliedby (d)$. By induction, any non-trivial suffix of x is a Lyndon word according to either all or none of the statements. Assume that x satisfies (d) but not (b). Then there is a non-trivial suffix v of x with $x \succ v$, but v is not a Lyndon word. Due to the inductive assumption and (d), there must be a non-trivial Lyndon suffix w of v with $v \succ w$. Note that w is also a non-trivial Lyndon suffix of x with $x \succ w$. This contradicts the assumption that x satisfies (d).

Statement (*e*): For $(b) \implies (e)$, assume that x satisfies (b) and let v be the longest non-trivial Lyndon suffix of x. Let u be the prefix of x such that $x = uv$. If $u \succeq v$ then also $uv \succ v$, which contradicts the assumption that x satisfies (b). Thus $u \prec v$, and it remains to be shown that u is a Lyndon word. Assume the opposite, then u does not satisfy (d) due to the inductive assumption. Consequently, there are $u', v' \in \Sigma^+$ such that $u = u'v'$, $v' \prec u$, and v' is a Lyndon word. Since $v' \prec u \prec v$ and both v' and v are Lyndon words, the string $v'v$ satisfies statement (e). This contradicts the fact that v is the longest non-trivial Lyndon suffix of x.

Finally, we show $(b) \impliedby (e)$. Assume that x satisfies (e), and let u and v be the Lyndon words used in (e). By induction, both u and v satisfy (b). We have to show that x is lexicographically smaller than all of its non-trivial suffixes. First, we show that $x = uv \prec v$. Assume the opposite, then it holds $u \prec v \prec uv$, and thus u is a proper prefix of v due to Property 6.1 (ii). Let w be the suffix of v such that $v = uw$, then $uw \prec uv$ and therefore $w \prec v$. However, w is a non-trivial suffix of v, which contradicts the fact that v satisfies (b). We have shown that $x \prec v$. Since v satisfies (b), all non-trivial suffixes of v are lexicographically larger than x. We still have to consider the suffixes $w'v$ of x, where w' is a non-trivial suffix of u. Since u satisfies (b), it holds $u \prec w'$ and by Property 6.1 (i) also $x = uv \prec w'v$. We have shown that all non-trivial suffixes of x are lexicographically larger than x. Thus, x satisfies (b). □

The Lyndon array identifies, for each position of a string, the longest Lyndon substring that starts at this position. We denote the Lyndon array by λ, which was also done in [Fra+16, Day+18, Bil+20, BE23]. Other common notations are *Lyn* in [CLR21, Bad+22], l in [Ban+17], and $\mathcal{L}$ in [FL19, FL20].

Definition 6.3 (Lyndon Array)**.**
The *Lyndon array* $\lambda_x[1..n]$ of a string $x[1..n]$ is defined by

$$\forall i \in [1,n] : \lambda_x[i] = \max\{m \in [1, n-i+1] \mid x[i..i+m) \text{ is a Lyndon word}\}.$$

We omit the subscript x whenever it is clear from context.

An example of the Lyndon array is provided in Figure 6.1a. We can use the Lyndon array to greedily factorize a string into a sequence of length-wise maximal Lyndon words. For example, in Figure 6.1a, the longest Lyndon substring at position 1 is of length $\lambda[1] = 4$, and thus $f_1 = \texttt{amtr}$ is the first factor. The second factor starts at position $1 + 4 = 5$ and is of length $\lambda[5] = 2$, i.e., $f_2 = \texttt{ak}$. The third and final factor starts at position $5 + 2 = 7$ and is of length $\lambda[7] = 6$, which yields $f_3 = \texttt{airbus}$. Hence we have factorized $x = f_1 f_2 f_3$ into a sequence of Lyndon words, and it holds $f_1 \succeq f_2 \succeq f_3$ (this readily follows from choosing length-wise maximal Lyndon words and Lemma 6.2(*e*)). The factorization of a string into lexicographically non-increasing Lyndon words is often called *Lyndon factorization* or *standard factorization*. It was introduced by Chen, Fox, and Lyndon [CFL58], and it is known that each string admits exactly one such factorization [Lot83, Theorem 5.1.5, attributed to Lyndon]. Duval's algorithm computes the factorization in linear time over general ordered alphabet [Duv83].

Theorem 6.4 (Lyndon Factorization [CFL58, Duv83, Lot83])**.**
Every non-empty string $x[1..n]$ has a unique factorization $x = f_1 f_2 \dots f_k$ such that each f_i is a Lyndon word, and $f_1 \succeq f_2 \succeq \dots \succeq f_k$. This factorization can be computed in $\mathcal{O}(n)$ time over general ordered alphabet.

6.2 Nearest Smaller Suffixes

In this section, we define the nearest smaller suffix arrays, which are strongly related to the Lyndon array. We also show structural properties of these arrays, which we will later exploit algorithmically. Throughout the remainder of Chapters 6 and 7, we use the variables ℓ and r to denote positions in the string. The intended meaning of these variables is *left* and *right*, i.e., when we use ℓ and r, it usually holds $\ell \leq r$.

Definition 6.5 (Nearest Smaller Suffix Arrays)**.** Let $x[1..n]$ be a string.

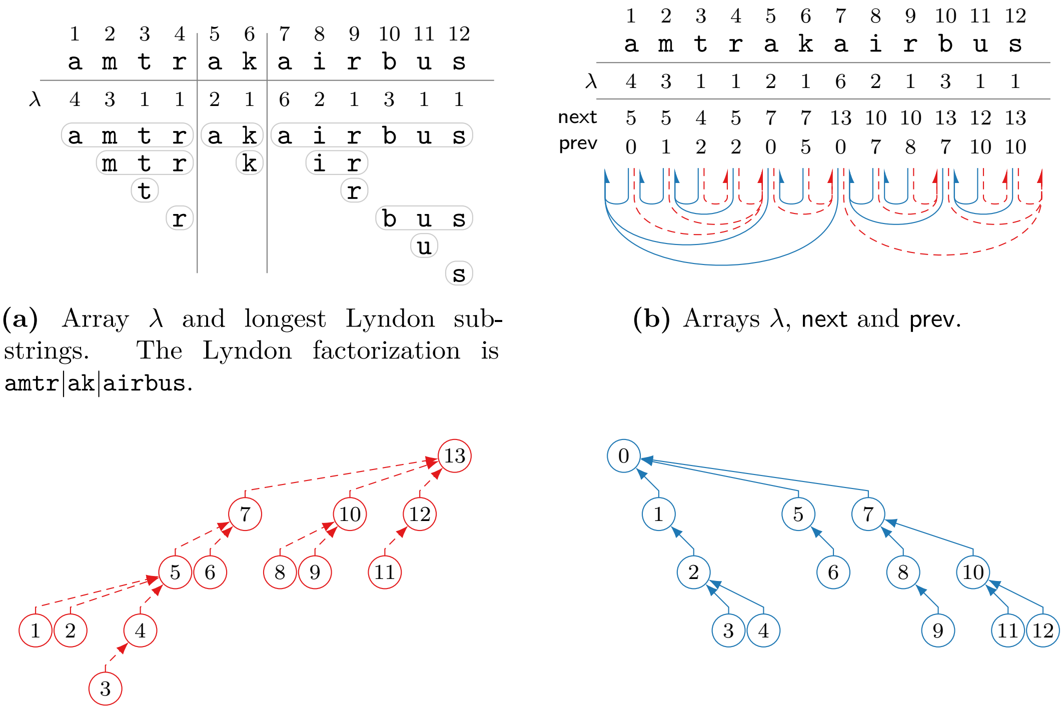
(PSS) The *previous smaller suffix (PSS) array* $\mathsf{prev}_x[1..n]$ of x is defined by

$$\forall r \in [1,n] : \mathsf{prev}_x[r] = \max\left(\{\ell \in [1,r) \mid x_\ell \prec x_r\} \cup \{0\}\right).$$

(NSS) The *next smaller suffix (NSS) array* $\mathsf{next}_x[1..n]$ of x is defined by

$$\forall \ell \in [1,n] : \mathsf{next}_x[\ell] = \min\left(\{r \in (\ell,n] \mid x_\ell \succ x_r\} \cup \{n+1\}\right).$$

We omit the subscript x whenever it is clear from context.



(a) Array λ and longest Lyndon substrings. The Lyndon factorization is `amtr`|`ak`|`airbus`.

(b) Arrays λ, next and prev.

(c) The next smaller suffix tree.

(d) The previous smaller suffix tree.

Figure 6.1: The Lyndon array, and nearest smaller suffix arrays and trees of the string `amtrackairbus`. Dashed edges point to next smaller suffixes, while solid edges point to previous smaller suffixes.

If $\mathsf{prev}_x[r] = \ell$, then we say that x_ℓ is the previous smaller suffix of x_r, or (in a slight abuse of terminology) that ℓ is the previous smaller suffix of r. Analogously, we use the term next smaller suffix for the relations expressed by next_x.

Figure 6.1b shows an example of the nearest smaller suffix arrays. In drawings, we use a directed edge between positions ℓ and r of a string to indicate that either $\ell = \mathsf{prev}[r]$ (whenever the edge is directed from right to left) or $r = \mathsf{next}[\ell]$ (whenever the edge is directed from left to right). We refer to these edges as PSS and NSS edges. The NSS edge at any position points either to some position further to the right or to $|x| + 1$. Hence the edges form a tree in which each node is a position, the parent of each position is its next smaller suffix, and $|x| + 1$ is the root. Analogously, the PSS edges form a tree with root 0. We refer to these trees as *NSS tree* and *PSS tree* respectively. Examples are provided in Figures 6.1c and 6.1d.

Equivalence of Lyndon Array and Nearest Smaller Suffixes

In Figure 6.1b it holds $\mathsf{next}[\ell] = \ell + \lambda[\ell]$ for all $\ell \in [1, |x|]$. For example, the longest Lyndon substring at position 2 is of length $\lambda[2] = 3$, and the next smaller suffix of 2 is $\mathsf{next}[2] = 2 + \lambda[2] = 5$. This is a fundamental property of the Lyndon array, which was first (indirectly in a different form) shown by Hohlweg and Reutenauer [HR03]. Subsequently, Franek et al. [Fra+16, Lemma 15] and Franek and Liut [FL20, Lemma 1][1]

[1] Lemma 1 (b) in [FL20] should state "$x[i..j]$ is proto-Lyndon" rather than "$x[i..n]$ is proto-Lyndon"

proved the property in the form stated above. Lemmas 6.6 and 6.7 (below) provide another proof of this result.

Lemma 6.6 (see also [HR03, Fra+16, FL20] and [Bil+20, Lemma 4])**.**

Let $x[1..n]$ be a string, and let either $r \in [1, n]$ with $\mathsf{prev}_x[r] = \ell > 0$, or $\ell \in [1, n]$ with $\mathsf{next}_x[\ell] = r$. Then $x[\ell..r)$ is a Lyndon word.

Proof. Recall that $x_r = \epsilon$ if $r = n + 1$. Let $w = x[\ell..r)$. If $|w| = 1$, then w is a Lyndon word. Otherwise, let v be any non-trivial suffix of w. By Lemma 6.2 (*b*), it suffices to show that $w \prec v$. For the sake of contradiction, assume that $v \prec w$. Note that vx_r is a suffix of x starting at some position in (ℓ, r), and thus either $\mathsf{next}[\ell] = r$ or $\mathsf{prev}[r] = \ell$ implies $x_\ell = wx_r \prec vx_r$. Since trivially $w \preceq wx_r$, we can combine the previous inequalities and obtain $v \prec w \preceq wx_r \prec vx_r$. By Property 6.1 (*ii*), v is a proper prefix of w, i.e., $w = vu$ for some $u \in \Sigma^+$. It follows $wx_r = vux_r \prec vx_r$, and thus $ux_r \prec x_r$. However, ux_r is a suffix of x starting at some position in (ℓ, r), such that either $\mathsf{next}[\ell] = r$ or $\mathsf{prev}[r] = \ell$ implies the contradiction $ux_r \succ x_r$. □

Lemma 6.7 (see also [HR03, Fra+16, FL20])**.**

For any string $x[1..n]$, it holds $\forall \ell \in [1, n] : \mathsf{next}_x[\ell] = \ell + \lambda_x[\ell]$.

Proof. Let $r = \mathsf{next}[\ell]$ and $u = x[\ell..r)$. Note that u is a Lyndon word due to Lemma 6.6, and thus $r \leq \ell + \lambda[\ell]$. It remains to be shown that $r \geq \ell + \lambda[\ell]$, i.e., that u is the *longest* Lyndon word starting at position ℓ. Assume the opposite, then there are strings $v \in \Sigma^+$ and $w \in \Sigma^*$ such that $x_\ell = uvw$, $x_r = vw$, and uv is a Lyndon word. By Lemma 6.2 (*b*), it holds $uv \prec v$. This implies $x_\ell = uvw \prec vw = x_r$ due to Property 6.1 (*i*), which contradicts the fact that $r = \mathsf{next}[\ell]$. □

Due to Lemma 6.7, instead of designing algorithms that compute the Lyndon array, we can design algorithms that compute the NSS array. Such algorithms are able to benefit from the rich structural properties of next and prev, which we explore in the following chapter. Even though the array next would suffice for the purpose of computing λ, many of the algorithms presented in Part II compute both next and prev (either explicitly or implicitly). This is due to the fact that next and prev are deeply intertwined, and computing one of them usually implies revealing the other one.

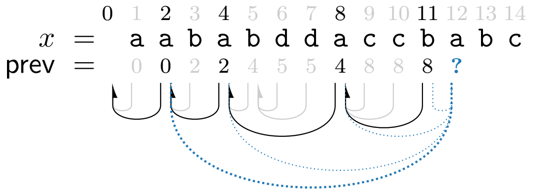
Chapter 7

7 A Simple Linear Time Algorithm for the Lyndon Array

In this chapter, we explore combinatorial structures of the nearest smaller suffix arrays. They imply a simple $\mathcal{O}(n)$ time algorithm that computes the Lyndon array of a string over linearly-sortable alphabet. With a few modifications, the same algorithm takes $\mathcal{O}(n)$ time even if the string is over general ordered alphabet, resulting in the main theorem below. The more advanced algorithms in Chapters 8 and 9 use the ideas from this chapter as a starting point.

Theorem 7.1. *The Lyndon array of a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of space.*

Before discussing the mechanism at play in full detail, we use a simple example to demonstrate the main algorithmic idea. Ultimately, the algorithms in this chapter will compute both next and prev simultaneously. For the sake of the example, we only aim to compute prev. The computation is performed in left-to-right order, i.e., at any given moment we have already computed a prefix prev$[1..i)$ of the PSS array. At this point in time, our goal is to compute prev$[i]$. In the example below, we already know the prefix prev$[1..11]$, and our goal is to determine prev$[12]$.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
x = a a b a b d d a c c b a b c
prev = 0 0 2 2 4 5 5 4 8 8 8 ?

A naive strategy for finding prev$[12]$ is to simply try all the $i \in [1, 11]$ in descending order. As soon as we reach i such that $x_i \prec x_{12}$, we know that prev$[12] = i$. In the example, it holds prev$[12] = 2$, which means that we have to perform $12 - 2 = 10$ lexicographical comparisons of suffixes. For a less naive strategy, we can use the already computed prefix prev$[1..11]$ to skip a significant number of suffix comparisons. Initially, we compare x_{11} and x_{12} and discover that $x_{11} \succ x_{12}$. We already know that prev$[11] = 8$, which implies $x_9 \succ x_{11}$ and $x_{10} \succ x_{11}$. By transitivity of the lexicographical order, we deduce that $x_9 \succ x_{12}$ and $x_{10} \succ x_{12}$. Hence we do not need to lexicographically compare x_{12} with either x_9 or x_{10}. Instead, we follow the NSS edge from 11 to prev$[11] = 8$ and use x_8 for the next comparison. It holds $x_8 \succ x_{12}$, and thus we have to keep searching for $\mathsf{prev}[12]$ further to the left. Like before, we use the already computed value $\mathsf{prev}[8] = 4$ to deduce that $\forall i \in (4, 8) : x_i \succ x_8 \succ x_{12}$. Hence the next comparison is between x_4 and x_{12}. It holds $x_4 \succ x_{12}$, and we follow the PSS edge from 4 to $\mathsf{prev}[4] = 2$. Finally, we discover that $x_2 \prec x_{12}$, which means that we have established $\mathsf{prev}[12] = 2$.

To summarize, we compute $\mathsf{prev}[12]$ by following the unique path of PSS edges originating at position 11, and searching for the first suffix that is lexicographically smaller than x_{12}. We only perform the four suffix comparisons indicated by the dotted lines in the drawing, which significantly improves upon the 10 comparisons of the naive approach.

7.1 Properties of Nearest Smaller Suffixes

As shown by the introductory example, following a path of PSS edges can be beneficial when computing prev from left to right. In this section, we formally describe the combinatorial properties that enable this algorithmic approach. We start by defining the set $\mathsf{prev}^*[r]$ that contains all the nodes on the path of PSS edges originating at position r. We also consider the analogous set $\mathsf{next}^*[\ell]$ for the path of NSS edges originating at ℓ, which can be used to compute next from right to left (see [Bad+22]). While the right-to-left computation is not described in the dissertation, it is based on combinatorial properties that are entirely symmetric to the ones used for the left-to-right computation. Hence there is no additional effort in proving the properties for both directions (rather than only one).

Definition 7.2. Let $x[1..n]$ be a string. For $\ell, r \in [1, n]$, we recursively define

(i) $\mathsf{prev}^*_x[0] = 0$ and $\mathsf{prev}^*_x[r] = \mathsf{prev}^*_x[\mathsf{prev}_x[r]] \cup \{r\}$, as well as

(ii) $\mathsf{next}^*_x[n+1] = n+1$ and $\mathsf{next}^*_x[\ell] = \mathsf{next}^*_x[\mathsf{next}_x[\ell]] \cup \{\ell\}$.

We omit the subscript x whenever it is clear from context.

In terms of the lexicographical order of suffixes, $\mathsf{prev}^*[r]$ contains the right-to-left lexicographical minima of suffixes with starting position in $[1, r]$, while $\mathsf{next}^*[\ell]$ contains the left-to-right lexicographical minima of suffixes with starting position in $[\ell, n]$. This is more formally expressed by the lemma below.

Lemma 7.3. *Let $x[1..n]$ be a string, and let $\ell, r \in [1, n]$ with $\ell \leq r$. Then*

(i) $\ell \in \mathsf{prev}^*_x[r] \iff \forall i \in (\ell, r] : x_\ell \prec x_i$*, and*

(ii) $r \in \mathsf{next}^*_x[\ell] \iff \forall i \in [\ell, r) : x_r \prec x_i$.

Proof. The lemma trivially holds if $\ell = r$. Assume $\ell < r$ and let $\mathsf{prev}^*[r] = \{\ell_1, \dots, \ell_q\}$ with $\ell_1 = 0$, $\ell_q = r$, and $\forall j \in [2, q] : \mathsf{prev}[\ell_j] = \ell_{j-1}$. First, assume that $\ell \in \mathsf{prev}^*[r]$, i.e., $\ell = \ell_p$ for some $p \in [2, q]$. It is easy to see that $x_\ell = x_{\ell_p} \prec x_{\ell_{p+1}} \prec \dots \prec x_{\ell_q}$. Now consider arbitrary $p' \in (p, q]$ and $i \in (\ell_{p'-1}, \ell_{p'})$. Due to $\mathsf{prev}[\ell_{p'}] = \ell_{p'-1}$, it holds $x_\ell = x_{\ell_p} \prec x_{\ell_{p'}} \prec x_i$. Hence we have shown $\forall i \in (\ell, r] : x_\ell \prec x_i$.

For the other direction, assume $\ell \notin \mathsf{prev}^*[r]$, i.e., there is some $p' \in [2, q]$ such that $\ell \in (\ell_{p'-1}, \ell_{p'}]$. Due to $\mathsf{prev}[\ell_{p'}] = \ell_{p'-1}$, it readily follows $x_{\ell_{p'}} \prec x_\ell$, which contradicts $\forall i \in (\ell, r] : x_\ell \prec x_i$. The proof of (ii) works analogously. □

Corollary 7.4. *Let $x[1..n]$ be a string, and let either $r \in [1, n]$ and $\ell = \mathsf{prev}_x[r]$, or $\ell \in [1, n]$ and $\mathsf{next}_x[\ell] = r$. Then $\ell \in \mathsf{prev}^*_x[r-1]$ and $r \in \mathsf{next}^*_x[\ell+1]$.*

Proof. If $\ell < r$ and $\ell \notin \mathsf{prev}^*_x[r-1]$ (respectively $r \notin \mathsf{next}^*_x[\ell+1]$), then by Lemma 7.3(*i*) (respectively Lemma 7.3 (*ii*)) there is some $i \in (\ell, r)$ such that $x_\ell \succ x_i$ (respectively $x_r \succ x_i$). This contradicts both $\ell = \mathsf{prev}[r]$ and $r = \mathsf{next}[\ell]$. □

By combining Definition 6.5 and Corollary 7.4, we obtain the characterization of nearest smaller suffixes that we algorithmically exploited in the introductory example.

Corollary 7.5. *Let $x[1..n]$ be a string and let $\ell, r \in [1, n]$. Then*

(*i*) $\mathsf{prev}_x[r] = \max(\{\ell' \in \mathsf{prev}^*_x[r-1] \mid \ell' = 0 \quad \vee x_{\ell'} \prec x_r\})$, *and*

(*ii*) $\mathsf{next}_x[\ell] = \min(\{r' \in \mathsf{next}^*_x[\ell+1] \mid r' = n+1 \vee x_{r'} \prec x_\ell\})$.

As mentioned before, we will compute next and prev simultaneously. In fact, the arrays are deeply intertwined; given one of them, it is possible to compute the other one without knowing the underlying string. The lemma below defines prev solely in terms of next and vice versa, which will be crucial for the simultaneous computation.

Lemma 7.6. *Let $x[1..n]$ be a string and let $\ell, r \in [1, n]$.*

(*i*) $\mathsf{next}_x[\ell] = r$ *if and only if* $\mathsf{prev}_x[r] < \ell$ *and* $\ell \in \mathsf{prev}^*_x[r-1]$.

$\mathsf{next}_x[\ell] = n+1$ *if and only if* $\ell \in \mathsf{prev}^*_x[n]$.

(*ii*) $\mathsf{prev}_x[r] = \ell$ *if and only if* $\mathsf{next}_x[\ell] > r$ *and* $r \in \mathsf{next}^*_x[\ell+1]$, *and*

$\mathsf{prev}_x[r] = 0$ *if and only if* $r \in \mathsf{next}^*_x[1]$.

Proof. For the first statement of (i), we show both directions separately. Assume $\mathsf{next}[\ell] = r$, then $\forall i \in [\ell, r) : x_r \prec x_\ell \preceq x_i$, which implies $\mathsf{prev}[r] < \ell$. Due to Corollary 7.4, it also holds $\ell \in \mathsf{prev}^*_x[r-1]$. For the other direction, assume $\mathsf{prev}[r] < \ell$ and $\ell \in \mathsf{prev}^*[r-1]$. Then $\forall i \in (\ell, r) : x_\ell \prec x_i$ because of Lemma 7.3 (*i*). Due to $\ell \in (\mathsf{prev}[r], r)$, it also holds $x_\ell \succ x_r$, and thus $\mathsf{next}[\ell] = r$. For the second statement of (i), it holds $\ell \in \mathsf{prev}^*[n]$ if and only if $\forall i \in (\ell, n] : x_\ell \prec x_i$ because of Lemma 7.3 (*i*). By Definition 6.5, it also holds $\mathsf{next}[\ell] = n+1$ if and only if $\forall i \in (\ell, n] : x_\ell \prec x_i$. The proof of (ii) works analogously. □

We conclude the section by observing that, whenever we draw NSS and PSS edges underneath the string, the edges can be embedded in the plane, i.e., we can draw them without intersections (like in Figure 6.1b). This is formally expressed by lemma below.

Lemma 7.7. *Let $x[1..n]$ be a string.*
Let either $r_1 \in [1, n]$ and $\ell_1 = \mathsf{prev}_x[r_1]$, or $\ell_1 \in [1, n]$ and $r_1 = \mathsf{next}_x[\ell_1]$.
Let either $r_2 \in [1, n]$ and $\ell_2 = \mathsf{prev}_x[r_1]$, or $\ell_2 \in [1, n]$ and $r_2 = \mathsf{next}_x[\ell_2]$.
Then it does not hold $\ell_1 < \ell_2 < r_1 < r_2$.

Proof. If $\ell_2 \in (\ell_1, r_1)$, then either $r_1 = \mathsf{next}[\ell_1]$ or $\ell_1 = \mathsf{prev}[r_1]$ implies $x_{\ell_2} \succ x_{r_1}$. However, by the same reasoning, $r_1 \in (\ell_2, r_2)$ implies $x_{\ell_2} \prec x_{r_1}$. □

7.2 A Simple Algorithm for Nearest Smaller Suffixes

Now we are well-prepared to more precisely describe and improve the algorithm from the introductory example, see Algorithm 7.1(a). We obtain two additional versions of this algorithm depending on how the lexicographical comparisons are implemented. Algorithm 7.1(c) uses naively computed LCEs. Algorithm 7.1(d) refines the LCE computation such that it is more time efficient. In the remainder of this section, we explain each version of the algorithm in detail. The Algorithms 7.1(c) and 7.1(d) require super-linear time, but they can be seen as incremental stepping stones towards the final solution. In Section 7.2.1, we modify Algorithm 7.1(d) such that it runs in $\mathcal{O}(n)$ time.

For all algorithms, we assume that the string $x[1..n]$ starts and ends with special *sentinel* symbols # and \$ such that $x = \#x(1..n)\$$ and $\forall i \in (1, n) : \# < \$ < x[i]$. This affects neither the asymptotic time or space complexity of the presented algorithms, nor the lexicographical order of suffixes; for any $i, j \in (1, n)$, it is easy to see that $x[i..n) \prec x[j..n)$ if and only if $x[i..n)\$ \prec x[j..n)\$$. The sentinels simplify the description by eliminating border cases. For example, they ensure that for $i \in (1, n)$ it holds $\mathsf{prev}[i] \geq 1$ and $\mathsf{next}[i] \leq n$. Also, for $i, j \in (1, n)$ with $i \neq j$ it holds $i + \text{LCE}(i, j) \leq n$.

Algorithm 7.1(a): The General Approach Due to the sentinels, we can directly assign $\mathsf{prev}[1]$, $\mathsf{next}[1]$, and $\mathsf{next}[n]$ (lines 1–2). We compute the arrays next and prev in $n - 1$ iterations of a simple for-loop (line 3). Immediately before iteration r of this loop, we have already computed $\mathsf{prev}[i]$ for $i \in [1, r)$, as well as $\mathsf{next}[i]$ for all i that satisfy $\mathsf{next}[i] < r$. Hence the goal of iteration r is to compute $\mathsf{prev}[r]$, while also identifying all indices ℓ with $\mathsf{next}[\ell] = r$.

A simple strategy for this follows from Corollary 7.5 (*i*) and Lemma 7.6 (*i*), which state that there is a path of PSS edges from $r - 1$ to $\mathsf{prev}[r]$, and all of the positions ℓ with $\mathsf{next}[\ell] = r$ lie on this path. We thus inspect the positions $\ell \in \mathsf{prev}^*[r - 1]$ one at a time and in decreasing order, starting with $\ell = r - 1$ (line 4). As long as $x_\ell \succ x_r$, we assign $\mathsf{next}[\ell] \leftarrow r$ (as dictated by Lemma 7.6 (*i*)), and then continue with the next position $\ell \leftarrow \mathsf{prev}[\ell]$ on the path of PSS edges (lines 5–7). As soon as $x_\ell \prec x_r$, we break out of the inner loop and finish the current iteration of the outer loop by assigning $\mathsf{prev}[r] \leftarrow \ell$ (line 8, as dictated by Corollary 7.5 (*i*)). The sentinel $x[1] = \#$ ensures $1 \in \mathsf{prev}^*[r]$ and $x_1 \prec x_r$; thus, we always reach some ℓ with $x_\ell \prec x_r$ eventually. The correctness of the algorithm follows directly from Corollary 7.5 and Lemma 7.6 (*i*). An example of an outer loop iteration is provided in Figure 7.1a (the arrays plce and nlce will be relevant later and can be ignored for now).

Algorithm 7.1 Computing nearest smaller suffixes over general ordered alphabet.

```
Require: String $x = x[1..n] = \#x(1..n)\$$ with $\forall i \in (1, n) : \# < \$ < x[i]$.
 Ensure: Previous and next smaller suffix arrays prev and next.
prev$[1..n] \leftarrow$ new array with prev$[1] = 0$
next$[1..n] \leftarrow$ new array with next$[1] =$ next$[n] = n + 1$
```

(a) Folklore

```
for $r = 2$ to $n$ do
    $\ell \leftarrow r - 1$
    while $x_\ell \succ x_r$ do
        next$[\ell] \leftarrow r$
        $\ell \leftarrow$ prev$[\ell]$
    prev$[r] \leftarrow \ell$
```

(b) LCE Functions for (c) and (d)

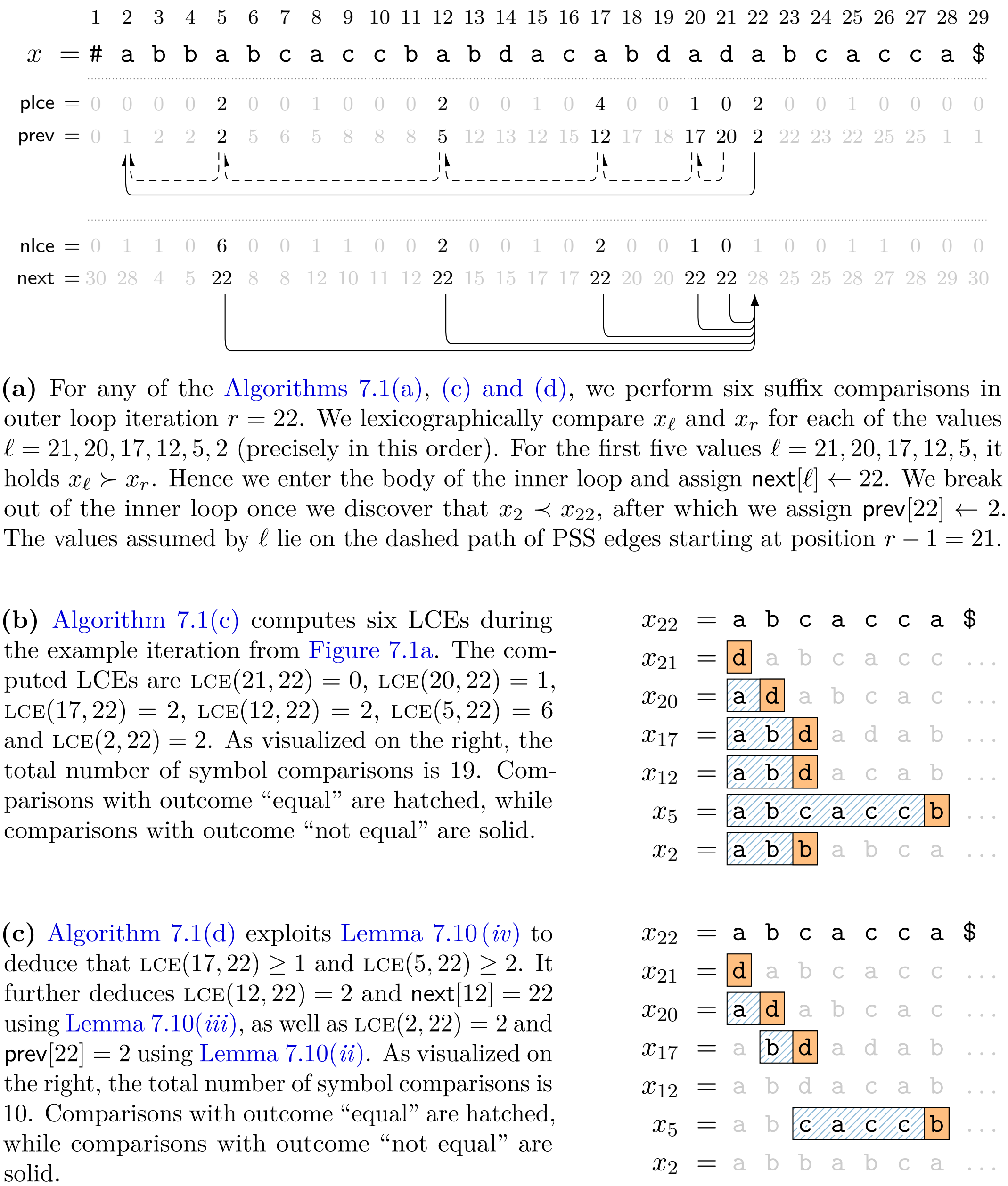
```
function LCE-SCAN$(\ell, r, m)$
    while $x[\ell + m] = x[r + m]$ do
        $m \leftarrow m + 1$
    return $m$

function LCE-SCAN$(\ell, r)$
    return LCE-SCAN$(\ell, r, 0)$
```

(c) Naive LCE-NSS

```
plce$[1..n] \leftarrow$ array filled with 0
nlce$[1..n] \leftarrow$ array filled with 0
for $r = 2$ to $n$ do
    $\ell \leftarrow r - 1$
    $m \leftarrow$ LCE-SCAN$(\ell, r)$
    while $x[\ell + m] > x[r + m]$ do
        next$[\ell]$, nlce$[\ell] \leftarrow r, m$
        $m \leftarrow$ LCE-SCAN(prev$[\ell], r)$
        —
        —
        —
        $\ell \leftarrow$ prev$[\ell]$
    prev$[r]$, plce$[r] \leftarrow \ell, m$
```

(d) Improved LCE-NSS

```
plce$[1..n] \leftarrow$ array filled with 0
nlce$[1..n] \leftarrow$ array filled with 0
for $r = 2$ to $n$ do
    $\ell \leftarrow r - 1$
    $m \leftarrow$ LCE-SCAN$(\ell, r)$
    while $x[\ell + m] > x[r + m]$ do
        next$[\ell]$, nlce$[\ell] \leftarrow r, m$
        if $m =$ plce$[\ell]$ then
            $m \leftarrow$ LCE-SCAN(prev$[\ell], r, m)$
        else if $m >$ plce$[\ell]$ then
            $m \leftarrow$ plce$[\ell]$
        $\ell \leftarrow$ prev$[\ell]$
    prev$[r]$, plce$[r] \leftarrow \ell, m$
```



(a) For any of the Algorithms 7.1(a), (c) and (d), we perform six suffix comparisons in outer loop iteration $r = 22$. We lexicographically compare x_ℓ and x_r for each of the values $\ell = 21, 20, 17, 12, 5, 2$ (precisely in this order). For the first five values $\ell = 21, 20, 17, 12, 5$, it holds $x_\ell \succ x_r$. Hence we enter the body of the inner loop and assign $\mathsf{next}[\ell] \leftarrow 22$. We break out of the inner loop once we discover that $x_2 \prec x_{22}$, after which we assign $\mathsf{prev}[22] \leftarrow 2$. The values assumed by ℓ lie on the dashed path of PSS edges starting at position $r - 1 = 21$.

(b) Algorithm 7.1(c) computes six LCEs during the example iteration from Figure 7.1a. The computed LCEs are $\textsc{lce}(21, 22) = 0$, $\textsc{lce}(20, 22) = 1$, $\textsc{lce}(17, 22) = 2$, $\textsc{lce}(12, 22) = 2$, $\textsc{lce}(5, 22) = 6$ and $\textsc{lce}(2, 22) = 2$. As visualized on the right, the total number of symbol comparisons is 19. Comparisons with outcome "equal" are hatched, while comparisons with outcome "not equal" are solid.

(c) Algorithm 7.1(d) exploits Lemma 7.10 (*iv*) to deduce that $\textsc{lce}(17, 22) \geq 1$ and $\textsc{lce}(5, 22) \geq 2$. It further deduces $\textsc{lce}(12, 22) = 2$ and $\mathsf{next}[12] = 22$ using Lemma 7.10(*iii*), as well as $\textsc{lce}(2, 22) = 2$ and $\mathsf{prev}[22] = 2$ using Lemma 7.10(*ii*). As visualized on the right, the total number of symbol comparisons is 10. Comparisons with outcome "equal" are hatched, while comparisons with outcome "not equal" are solid.

Figure 7.1: Computing nearest smaller suffixes with Algorithms 7.1(a), (c) and (d).

After every lexicographical comparison of suffixes in line 5, we correctly assign either $\mathsf{next}[\ell] \leftarrow r$ or $\mathsf{prev}[r] \leftarrow \ell$ immediately afterwards. Since each entry of prev and next gets assigned exactly once, we perform exactly $2n - 3$ suffix comparisons; we enter the body of the inner loop exactly $n - 2$ times (once per entry of next, but not for $\mathsf{next}[1] = \mathsf{next}[n] = n + 1$), and break out of the inner loop exactly $n - 1$ times (once per outer loop iteration, or equivalently once per entry of prev, but not for $\mathsf{prev}[1] = 0$). It is easy to see that the total time – apart from the time needed for suffix comparisons – is linear in the number of suffix comparisons. We have shown:

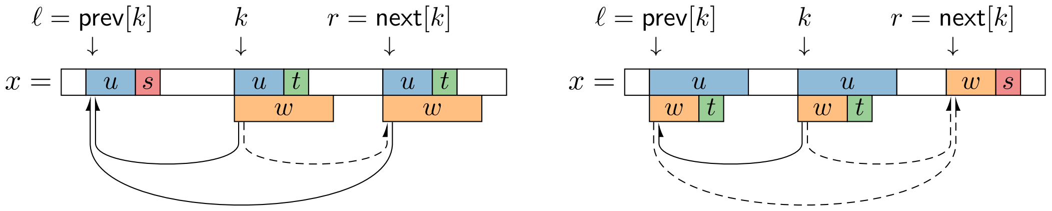
Proposition 7.8. *Algorithm 7.1(a) performs $2n - 3$ lexicographical comparisons of suffixes. Apart from these comparisons, the algorithm takes $\mathcal{O}(n)$ time.*

Algorithm 7.1(c): Computing LCEs with Simple Scanning Due to the sentinels, no suffix is a prefix of another suffix. Hence it holds $x_\ell \prec x_r \iff x[\ell + \textsc{lce}(\ell, r)] < x[r + \textsc{lce}(\ell, r)]$ for any $\ell, r \in (1, n)$ with $\ell \neq r$. The LCE can be computed by naive scanning, as shown in Algorithm 7.1(b), and the sentinels ensure that there is always a mismatching symbol eventually. Algorithm 7.1(c) is structurally identical to Algorithm 7.1(a), but it implements the lexicographical suffix comparisons with naively scanned LCEs (lines 7, 8, and 10). Additionally, it stores the computed LCEs in two arrays nlce and plce (lines 3–4, 9, and 15), where after termination it holds $\mathsf{plce}[i] = \textsc{lce}(\mathsf{prev}[i], i)$ and $\mathsf{nlce}[i] = \textsc{lce}(i, \mathsf{next}[i])$ for all $i \in (1, n)$. These arrays are of independent interest. For example, nlce is useful when computing maximal periodic substrings (which we explain in Chapter 10). The correctness of the algorithm follows from the correctness of Algorithm 7.1(a).

Computing $\textsc{lce}(\ell, r)$ by scanning takes $\textsc{lce}(\ell, r) + 1$ symbol equality comparisons, where the first $\textsc{lce}(\ell, r)$ comparisons have outcome “equal”, and the final comparison has outcome “not equal”. As shown in Figure 7.1b, this can quickly lead to a large number of symbol comparisons. In the worst case, a single LCE causes $\mathcal{O}(n)$ symbol comparisons, and thus Algorithm 7.1(c) takes $\mathcal{O}(n^2)$ time (the bound is tight, e.g., for the string $x = \mathtt{\#a}^{n-2}\mathtt{\$}$). If the input string is drawn uniformly at random from the set of length-n strings over Σ, where $|\Sigma| > 1$, then the expected running time of Algorithm 7.1(c) is $\mathcal{O}(n)$ (see [Bad+22, Theorem 7], where the expected running time is analyzed for the symmetric right-to-left algorithm). If the string is over linearly-sortable alphabet, then a data structure for constant time LCE queries can be precomputed in $\mathcal{O}(n)$ time (e.g., the data structure from Lemma 4.4). This reduces the time bound to $\mathcal{O}(n)$.

Proposition 7.9. *Algorithm 7.1(c) can be implemented such that it takes $\mathcal{O}(n)$ time for a string of length n over a linearly-sortable alphabet.*

Algorithm 7.1(d): Using LCEs with Improved Scanning In a single outer loop iteration r of Algorithm 7.1(c), we may compute $\textsc{lce}(\ell, r)$ for many different values of ℓ. So far, we always computed each new LCE entirely from scratch (line 10). For many of the LCEs, we can avoid (a part of) the scan by utilizing the additional properties of nearest smaller suffixes that are stated in the lemma below.



(a) Lemma 7.10 (*ii*). **(b)** Lemma 7.10 (*iii*).

Figure 7.2: Drawings for Lemma 7.10. NSS edges are dashed; PSS edges are solid.

Lemma 7.10. *Let* $x = \#x(1..n)\$$ *be a string with* $\forall i \in (1, n) : \# < \$ < x[i]$. *Let* $k \in (1, n)$ *be an arbitrary index, and let* $\ell = \mathsf{prev}_x[k]$ *and* $r = \mathsf{next}_x[k]$.

- ***(i)*** *It holds either* $\mathsf{prev}_x[r] = \ell$ *or* $\mathsf{next}_x[\ell] = r$.
- ***(ii)*** *If* $\textsc{lce}(\ell, k) < \textsc{lce}(k, r)$, *then* $\textsc{lce}(\ell, r) = \textsc{lce}(\ell, k)$ *and* $\mathsf{prev}_x[r] = \ell$.
- ***(iii)*** *If* $\textsc{lce}(\ell, k) > \textsc{lce}(k, r)$, *then* $\textsc{lce}(\ell, r) = \textsc{lce}(k, r)$ *and* $\mathsf{next}_x[\ell] = r$.
- ***(iv)*** *If* $\textsc{lce}(\ell, k) = \textsc{lce}(k, r)$, *then* $\textsc{lce}(\ell, r) \geq \textsc{lce}(k, r)$.

Proof. We start with (i). Due to $\ell = \mathsf{prev}[k]$ and $r = \mathsf{next}[k]$, it holds $x_\ell \prec x_k$, $x_r \prec x_k$, and $\forall i \in (\ell, k) \cup (k, r) : x_k \prec x_i$. Hence also $\forall i \in (\ell, r) : x_r \prec x_i \wedge x_\ell \prec x_i$. Thus, if $x_\ell \prec x_r$ then $\mathsf{prev}[r] = \ell$, and if $x_\ell \succ x_r$ then $\mathsf{next}[\ell] = r$.

For showing (ii), let $u = x[\ell..\ell + \textsc{lce}(\ell, k)) = x[k..k + \textsc{lce}(\ell, k))$, $s = x[\ell + \textsc{lce}(\ell, k)]$, and $t = x[k + \textsc{lce}(\ell, k)]$ (see Figure 7.2). By the definition of LCEs, it holds $s \neq t$, and due to $\ell = \mathsf{prev}[k]$ it can only be that $s < t$. Because of $\textsc{lce}(\ell, k) < \textsc{lce}(k, r)$, suffix x_r has prefix ut. Hence $x_\ell = us \cdot x_{\ell+|us|} \prec ut \cdot x_{r+|ut|} = x_r$. Due to (i), this means $\mathsf{next}[\ell] = r$. The proof of (iii) works analogously to the one of (ii).

Finally, (iv) is a trivial observation. Let $m = \textsc{lce}(\ell, k) = \textsc{lce}(k, r)$, then it is easy to see that $x[\ell..\ell + m) = x[k..k + m) = x[r..r + m)$, and hence $\textsc{lce}(\ell, r) \geq m$. □

Algorithm 7.1(d) uses the new insights from Lemma 7.10. It is identical to Algorithm 7.1(c), except for the highlighted computation of the LCE in lines 10–13. At the point in time at which we reach line 10, let $k' = \ell$, $\ell' = \mathsf{prev}[\ell]$, and $r' = r$. Note that $\ell' = \mathsf{prev}[k']$ and $r' = \mathsf{next}[k']$, and thus we can use ℓ', k', and r' to invoke Lemma 7.10. We already computed $\textsc{lce}(\ell', k') = \mathsf{plce}[k']$ (due to the iteration order of the algorithm) and $\textsc{lce}(k', r') = \mathsf{nlce}[k'] = m$ (this is the most recently computed LCE). Now we compute $\textsc{lce}(\ell', r')$ according to the cases of Lemma 7.10.

- If $\textsc{lce}(\ell', k') = \textsc{lce}(k', r')$ then Lemma 7.10 (*iv*) implies $\textsc{lce}(\ell', r') \geq \textsc{lce}(k', r')$. We compute $\textsc{lce}(\ell', r')$ by scanning, but we skip $m = \textsc{lce}(k', r')$ symbol comparisons (lines 10–11).

- If $\textsc{lce}(\ell', k') < \textsc{lce}(k', r')$ then Lemma 7.10 (*ii*) implies $\textsc{lce}(\ell', r') = \textsc{lce}(\ell', k')$. Since $\mathsf{plce}[k'] = \textsc{lce}(\ell', k')$, we can simply assign $m \leftarrow \mathsf{plce}[k']$ (lines 12–13). Note that Lemma 7.10 (*ii*) also implies $\mathsf{prev}[r'] = \ell'$, which means that we will immediately break out of the inner loop and finish the current iteration of the outer loop.

- If $\textsc{lce}(\ell', k') > \textsc{lce}(k', r')$ then Lemma 7.10(*iii*) implies $\textsc{lce}(\ell', r') = \textsc{lce}(k', r')$. It already holds $m = \textsc{lce}(k', r')$, and thus there is no need to do anything.

As shown in Figure 7.1c, the new approach may require significantly fewer symbol comparisons than Algorithm 7.1(c). However, Algorithm 7.1(d) still takes $\mathcal{O}(n^2)$ time in the worst case. In the next section, we slightly modify Algorithm 7.1(d) such that it achieves linear time.

A Note on the Space Complexity In order to store the arrays next, prev, nlce, and plce, Algorithms 7.1(c) and 7.1(d) require $4n \lceil \log_2 n \rceil$ bits of space. For a small practical improvement, it is possible to remove the array prev. This is because the only access to $\mathsf{prev}[\ell]$ occurs at the same time at which we assign $\mathsf{next}[\ell]$ (see lines 9 and 14). Thus, we only need to maintain access to the values $\mathsf{prev}[\ell]$ for positions with uninitialized $\mathsf{next}[\ell]$, which means that we can use a single array for storing both PSS and NSS information. The total working space (without the input string) then becomes $3n \lceil \log_2 n \rceil + \mathcal{O}(\log n)$ bits.

7.2.1 Achieving Linear Time

In order to achieve linear time, we use the function $\textsc{smart-lce}$ (Algorithm 7.2) to more efficiently compute LCEs. A call to $\textsc{smart-lce}(\ell, r, m)$ means that we want to compute $\textsc{lce}(\ell, r)$, and we have already established $\textsc{lce}(\ell, r) \geq m$. We modify Algorithm 7.1(d) by replacing line 7 with $m \leftarrow \textsc{smart-lce}(\ell, r, 0)$, and line 11 with $m \leftarrow \textsc{smart-lce}(\mathsf{prev}[\ell], r, m)$ (and leave everything else unchanged). In the remainder of the section, we show that $\textsc{smart-lce}$ works correctly, and that the total time spent for all invocations of $\textsc{smart-lce}$ is $\mathcal{O}(n)$. Then, it directly follows that the modified version of Algorithm 7.1(d) takes $\mathcal{O}(n)$ time. Note that Algorithm 7.2 is tailored to (and thus only works as a part of) Algorithm 7.1(d).

In the following description, whenever we use the variables ℓ, r, and m, we mean the arguments of the function $\textsc{smart-lce}$ (rather than the identically named variables from Algorithm 7.1(d)). Now we explain how the new LCE function works. Generally speaking, it computes LCEs with two different methods: naive scanning (as done before), and deduction from previously computed LCEs. Sometimes, a combination of both is necessary. Both methods rely on a global variable c (persistent between the function calls) that stores at all times the rightmost position of the string that we have already inspected (line 2).

Scanning LCEs We start by explaining the simpler method of naive scanning. If at the beginning of the function call it holds $r + m \geq c$ (line 4), then we simply scan the remainder of the LCE (lines 9–10; identical to what we did in $\textsc{lce-scan}$). Let m' be the initial value of m before the scan, and let $m'' = \textsc{lce}(\ell, r)$ be the final value of m after performing the scan. After the scan, the rightmost inspected position is $r + m''$, and we update c accordingly (line 11; the variable d is not relevant for now). Since we only perform the scan if $r + m' \geq c$, the assignment $c \leftarrow r + m''$ increases c by at least $m'' - m'$. Note that $m'' - m'$ is also exactly the number of times we execute line 10. Since c never exceeds n, we execute line 10 no more than n times during all the calls to $\textsc{smart-lce}$ that initially satisfy $r + m \geq c$. It follows that, for all of these calls together, we spend at most $\mathcal{O}(n)$ time.

Algorithm 7.2 Efficient LCE computation for Algorithm 7.1(d).

```
Require: String $x = x[1..n] = \#x(i..n)\$$ with $x[\ell..\ell + m) = x[r..r + m)$.
 Ensure: Longest common extension $\textsc{lce}(\ell, r)$.
 1: global variable $c \leftarrow 0$
 2: global variable $d \leftarrow 0$
 3: function SMART-LCE($\ell, r, m$)
 4:     if $r + m < c$ then
 5:         if $\mathsf{next}[\ell - d] = r - d$ then $m \leftarrow \mathsf{nlce}[\ell - d]$
 6:                                       else $m \leftarrow \mathsf{plce}[r - d]$
 7:         if $r + m < c$ then return $m$
 8:         $m \leftarrow c - r$
 9:     while $x[\ell + m] = x[r + m]$ do
10:         $m \leftarrow m + 1$
11:     $c, d \leftarrow r + m, r - \ell$
12:     return $m$
```

Deducing LCEs If at the beginning of the function call it holds $r + m < c$, then we try to deduce $\textsc{lce}(\ell, r)$ from previously computed LCEs (lines 4–8). Let r_c be the rightmost position for which we already computed some $\textsc{lce}(\ell_c, r_c)$ with $r_c + \textsc{lce}(\ell_c, r_c) = c$ (such a position must exist because otherwise we would not have inspected $x[c]$ yet). The global variable d contains at all times the distance $r_c - \ell_c$ (line 2; we update d together with c, see line 11). Let $\ell_* = \ell - d$ and $r_* = r - d$. The example in Figure 7.3a helps with understanding the notation. Later, we will show that (as suggested by the examples)

(i) it holds $r_c \le \ell < r < c$, and thus $\ell_c \le \ell_* < r_* < \ell_c + \textsc{lce}(\ell_c, r_c)$, and

(ii) either $\mathsf{prev}[r_*] = \ell_*$ (and thus $\mathsf{plce}[r_*] = \textsc{lce}(\ell_*, r_*)$)
or $\mathsf{next}[\ell_*] = r_*$ (and thus $\mathsf{nlce}[\ell_*] = \textsc{lce}(\ell_*, r_*)$).

When deducing LCEs, we first use (ii) to obtain $\textsc{lce}(\ell_*, r_*)$ (lines 5–6). Note that, by the definition of ℓ_* and r_*, the relative positions of ℓ_* and r_* within $x[\ell_c..\ell_c + \textsc{lce}(\ell_c, r_c))$ are the same as the relative positions of ℓ and r within $x[r_c..r_c + \textsc{lce}(\ell_c, r_c))$ (and the positions are indeed within these intervals due to (i)). If $r + \textsc{lce}(\ell_*, r_*) < c$ then

$$x[r..r + \textsc{lce}(\ell_*, r_*)] = x[r_*..r_* + \textsc{lce}(\ell_*, r_*)] \text{ and}$$
$$x[\ell..\ell + \textsc{lce}(\ell_*, r_*)] = x[\ell_*..\ell_* + \textsc{lce}(\ell_*, r_*)],$$

where both equalities follow from $x[\ell_c..\ell_c + \textsc{lce}(\ell_c, r_c)) = x[r_c..r_c + \textsc{lce}(\ell_c, r_c))$. This implies $\textsc{lce}(\ell, r) = \textsc{lce}(\ell_*, r_*)$, and we return $\textsc{lce}(\ell_*, r_*)$ in constant time (line 7). Since it holds $r + \textsc{lce}(\ell, r) < c$, there is no need to update c and d. In Figure 7.3a, we have $21 + \textsc{lce}(4, 7) = 23 < 29 = c$, and thus $\textsc{lce}(18, 21) = \textsc{lce}(4, 7) = 2$.

If, however, $r + \textsc{lce}(\ell_*, r_*) \ge c$ then we cannot immediately deduce the exact value of $\textsc{lce}(\ell, r)$ (as is the case in Figure 7.3c). We can still obtain some useful

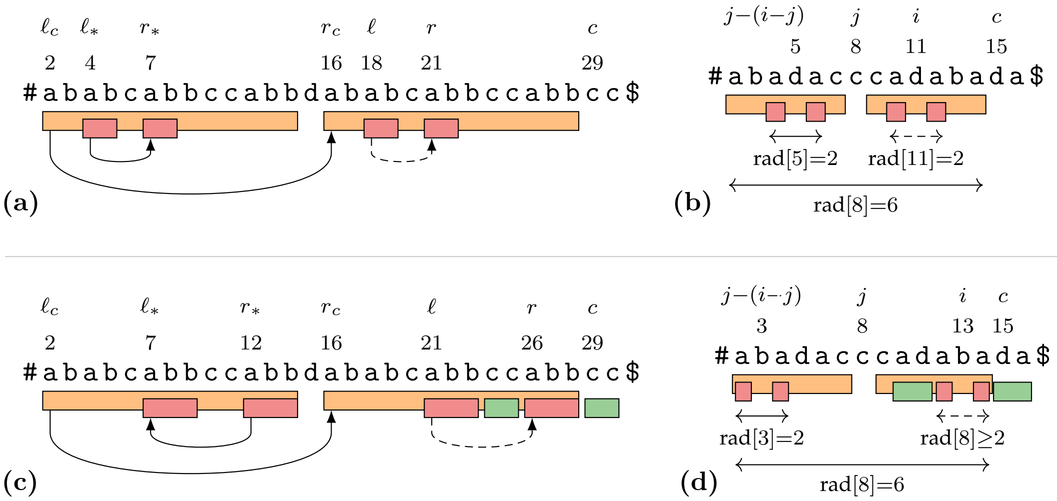


Figure 7.3: Deducing LCEs with Algorithm 7.2 in (a) and (c), and deducing longest palindromes with Manacher's algorithm in (b) and (d). Boxes of equal color indicate equal substrings. In (b) and (d), boxes of equal color sometimes indicate substrings that are the reverse of each other.

information because of

$$x[r..r+c-r) = x[r_*..r_*+c-r) = x[\ell_*..\ell_*+c-r) = x[\ell..\ell+c-r),$$

where the first and the third equality follow from $x[\ell_c..\ell_c+\textsc{lce}(\ell_c,r_c)) = x[r_c..r_c+\textsc{lce}(\ell_c,r_c))$, and the second equality follows from $r+\textsc{lce}(\ell_*,r_*) \geq c$, which is equal to $\textsc{lce}(\ell_*,r_*) \geq c-r$. The equation implies $\textsc{lce}(\ell,r) \geq c-r$, and we update m accordingly (line 8). In Figure 7.3c, we have $26+\textsc{lce}(7,12) = 29 = c$, and thus $\textsc{lce}(21,26) \geq 29-26 = 3$.

We compute the remaining part of $\textsc{lce}(\ell,r)$ by scanning (lines 9–10), and then update c and d (line 11). Since we assign $m \leftarrow (c-r)$ immediately before starting the scan, we can use the same argument as in the previous paragraph about scanning LCEs. For every symbol comparison of the scan (except for the last one), we will increase c by one. Therefore, the total number of symbol comparisons for all calls of SMART-LCE is $\mathcal{O}(n)$. In Figure 7.3c, the scan extends the LCE by two additional positions, and we obtain $\textsc{lce}(21,26) = 5$. We then have to update $c \leftarrow 26+5 = 31$ and $d \leftarrow 26-21 = 5$.

The correctness of the algorithm follows from its description and the properties (i) and (ii), which we will show in the next paragraphs.

Showing Property (i) The property states that, if we call SMART-LCE(ℓ,r,m) with $r+m < c$, then $r_c \leq \ell < r < c$. Since trivially $\ell < r \leq r+m < c$, we only have to show $r_c \leq \ell$. The property is readily proven for the call SMART-LCE$(\ell,r,0)$ in line 7 of Algorithm 7.1(d). It holds $\ell = r-1$, and this is the first LCE that we compute between r and any smaller index. Since we already computed $\textsc{lce}(\ell_c,r_c)$, it holds $r > r_c$ and $\ell = r-1 \geq r_c$.

Now we consider the call SMART-LCE(ℓ,r,m) in line 11. As seen in the description of Algorithm 7.1(d), for this call it holds $m = \textsc{lce}(\ell,k) = \textsc{lce}(k,r)$, where $k \in (\ell,r)$ with $\mathsf{prev}[k] = \ell$ and $\mathsf{next}[k] = r$. For every $h \in (\ell,r)$, the definition of prev and

next implies that $x_h \succeq x_k \succ x_r$. This also means that $m = \textsc{lce}(k, r) \geq \textsc{lce}(h, r)$. If $r = r_c$ then, because we already computed $\textsc{lce}(\ell_c, r)$, and due to the iteration order of the algorithm, it holds $\ell < \ell_c$. Then, however, $\ell_c \in (\ell, r)$ and thus $m = \textsc{lce}(k, r) \geq \textsc{lce}(\ell_c, r) = c - r$, which contradicts $r + m < c$. We have shown that $r > r_c$, which also implies $r_* > \ell_c$. If $\ell < r_*$ then $r_* \in (\ell, r)$ and thus $m \geq \textsc{lce}(r_*, r) = c - r$, which contradicts $r + m < c$. It follows that $\ell \geq r_* > \ell_c$. Finally, if $\ell \in (\ell_c, r_c)$ then $\ell_c < \ell < r_c < r$, which contradicts Lemma 7.7. The only remaining possibility is $\ell \geq r_c$, which is what we wanted to show.

Showing Property (*ii*) The property states that either $\mathsf{prev}[r_*] = \ell_*$, or $\mathsf{next}[\ell_*] = r_*$. By the definition of ℓ_* and r_*, and due to (i) and $x[\ell_c..\ell_c + \textsc{lce}(\ell_c, r_c)) = x[r_c..r_c + \textsc{lce}(\ell_c, r_c))$, it holds $x[\ell..r) = x[\ell_*..r_*)$. Since we want to compute $\textsc{lce}(\ell, r)$, it holds either $\mathsf{next}[\ell] = r$ or $\mathsf{prev}[r] = \ell$. Therefore, Lemma 6.6 implies that $x[\ell..r) = x[\ell_*..r_*)$ is a Lyndon word. Due to Lemma 6.7, we know that $\mathsf{next}[\ell_*] \geq r_*$. If $\mathsf{next}[\ell_*] = r_*$ or $\mathsf{prev}[r_*] = \ell_*$, then there is nothing left to show. Thus, assume that $\mathsf{next}[\ell_*] > r_*$ and $\mathsf{prev}[r_*] > \ell_*$ (it cannot be that $\mathsf{prev}[r_*] < \ell_*$ because then $\mathsf{prev}[r_*] < \ell_* < r_* < \mathsf{next}[\ell_*]$ contradicts Lemma 7.7). Let $p_r = r - (r_* - \mathsf{prev}[r_*])$. Due to Lemma 6.6, the substring $x[\mathsf{prev}[r_*]..r_*) = x[p_r..r)$ is a Lyndon word, and Lemma 6.7 implies $\mathsf{next}[p_r] \geq r$. Since $p_r \in (\ell, r)$ it holds $\mathsf{next}[p_r] \leq r$ (otherwise we contradict Lemma 7.7), and the only possible option is $\mathsf{next}[p_r] = r$.

We have shown that $\mathsf{next}[p_r] = r$ and thus $x_{p_r} \succ x_r$. By the definition of prev, it also holds $x_{\mathsf{prev}[r_*]} \prec x_{r_*}$. Since we chose r_c to be the rightmost index with $r_c + \textsc{lce}(\ell_c, r_c) = c$, it holds $r + \textsc{lce}(p_r, r) < c$ (otherwise we would have updated r_c already). Therefore, we have

$$x[\mathsf{prev}[r_*]..\mathsf{prev}[r_*] + \textsc{lce}(p_r, r)] = x[p_r..p_r + \textsc{lce}(p_r, r)], \text{ and}$$
$$x[r_*..r_* + \textsc{lce}(p_r, r)] = x[r..r + \textsc{lce}(p_r, r)].$$

This, however, means that $x_{\mathsf{prev}[r_*]} \prec x_{r_*} \iff x_{p_r} \prec x_r$, which contradicts our previous observation that $x_{p_r} \succ x_r$ and $x_{\mathsf{prev}[r_*]} \prec x_{r_*}$. It follows that the assumption $\mathsf{next}[\ell_*] > r_*$ and $\mathsf{prev}[r_*] > \ell_*$ was wrong, and it holds $\mathsf{next}[\ell_*] = r_*$ or $\mathsf{prev}[r_*] = \ell_*$.

We have shown that properties (i) and (ii) hold, which concludes the proof of correctness. The algorithm uses only a constant number of additional integer variables, apart from the space needed for the auxiliary arrays of size n. Hence we have shown the main theorem.

Theorem 7.1. *The Lyndon array of a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of space.*

7.3 Similarity to Manacher's Algorithm for Computing Maximal Palindromes

In this section, we want to briefly highlight the similarity between the technique of Section 7.2.1 and Manacher's algorithm for computing maximal palindromes [Man75]. For simplicity, we only consider odd palindromes. An *odd palindrome of radius* $|w| + 1$ is a string of the form $w \cdot s \cdot \mathsf{rev}(w)$, where s is a symbol and w is some possibly empty string. (Equivalently, an odd palindrome is a string of odd length that equals

Algorithm 7.3 Manacher's algorithm for odd palindromes.

```
Require: String $x[1..n] = \#x(1..n)\$$.
Ensure: Array rad containing the radii
        of longest odd palindromes.

rad$[1..n] \leftarrow$ new array with
        rad$[1] =$ rad$[n] = 1$

global variable $c \leftarrow 0$
global variable $j \leftarrow 0$

for $i = 2$ to $n - 1$ do
    rad$[i] \leftarrow$ SMART-PAL$(i, 1)$

function SMART-PAL$(i, m)$
    if $i + m < c$ then
        $m \leftarrow$ rad$[j - (i - j)]$
        if $i + m < c$ then return $m$
        $m \leftarrow c - i$

    while $x[i - m] = x[i + m]$ do
        $m \leftarrow m + 1$

    $c, j \leftarrow i + m, i$
    return $m$
```

its reversal.) For a string $x[1..n]$, the presented version of Manacher's algorithm computes an array $\mathsf{rad}[1..n]$, where $\forall i \in [1, n]$:

$$\mathsf{rad}[i] = \max\{m \in [1, \min(i, n - i + 1)] \mid x(i - m..i + m) \text{ is an odd palindrome}\}.$$

If we compute the entries of rad in left-to-right order, then we can sometimes fully or partially deduce an entry, see Figures 7.3b and 7.3d. This is very similar to our observations for LCEs in Figures 7.3a and 7.3c. A possible implementation of Manacher's algorithm is provided in Algorithm 7.3. It computes rad from left to right, while keeping track of the rightmost inspected position of the string. Whenever possible, the function SMART-PAL partially or fully deduces $\mathsf{rad}[i]$. Note that the functions SMART-LCE and SMART-PAL are structurally identical and use the same algorithmic ideas. We omit the proof of why Algorithm 7.3 functions as intended and why it takes $\mathcal{O}(n)$ time because it is much simpler than the proofs for Algorithm 7.1(d) and Algorithm 7.2. Specifically, it requires no complicated technicalities like properties (i) and (ii) in Section 7.2.1.

7.4 Conclusion and Practical Implementation

We presented a linear time construction algorithm for the Lyndon array over general ordered alphabet. While the proof is still quite involved and could possibly be simplified, the algorithm itself is surprisingly simple. Algorithm 7.1(d) augmented with the LCE computation from Algorithm 7.2 requires no additional complex data structures and can be directly implemented in practice. We provide a simple C++ implementation that is publicly available[1] and consists of less than 50 lines of code, using no additional libraries (neither external nor from the C++ standard library). A copy is provided in Listing 7.1. While we do not provide experimental results for its performance, it shall be noted that it is indeed quite fast in practice; this is also evident from the fact that it is a simpler version of the algorithm from Chapter 8, for which we provide a preliminary practical evaluation in Section 8.5. The simplicity is

[1] `https://github.com/jonas-ellert/simple-lyndon`, permanently archived in the Software Heritage Archive at `https://archive.softwareheritage.org/swh:1:dir:dbbf2b4ac2fa652eb0865cdc6719924ce8a81952;origin=https://github.com/jonas-ellert/simple-lyndon;visit=swh:1:snp:1a40ff8c462536624a348dbd651a5e66629fb09c;anchor=swh:1:rev:4b61d4a2500693886e0d69cd0c1c5ea68e48dd89`

a consequence of designing an algorithm for general ordered alphabet, leading to a solution that does not rely on the usual machinery like the suffix array.

```
void lyndon(char const *text, int n,                     // input
            int *nss, int *nlce, int *pss, int *plce) {  // output

  auto T_l = [&](int x)  // simulate left sentinel
             { return (x >= 0) ? ((int)text[x]) : ((int)-256); };
  auto T_r = [&](int x)  // simulate right sentinel
             { return (x < n) ? ((int)text[x]) : ((int)-256); };

  int d, rhs = -1;

  auto extend_lce = [&](int l, int r, int known_lce = 0) {
    if (r + known_lce < rhs) {
      known_lce = (nss[l - d] == r - d) ? nlce[l - d] : plce[r - d];
      if (r + known_lce < rhs) return known_lce;
      known_lce = rhs - r;
    }

    while (T_l(l + known_lce) == T_r(r + known_lce)) ++known_lce;

    rhs = r + known_lce;
    d = r - l;
    return known_lce;
  };

  for (int r = 0; r < n; ++r) {
    int l = r - 1;
    int lce = extend_lce(l, r, 0);
    while (T_l(l + lce) > T_r(r + lce)) {
      nss[l] = r;
      nlce[l] = lce;
      if (lce == plce[l]) {
        lce = extend_lce(pss[l], r, lce);
      } else if (lce > plce[l]) {
        lce = plce[l];
      }
      l = pss[l];
    }
    pss[r] = l;
    plce[r] = lce;
  }

  int l = n - 1;
  while (l >= 0) {
    nss[l] = n;
    nlce[l] = 0;
    l = pss[l];
  }
}
```

Listing 7.1: C++ implementation of Algorithm 7.1(d) and Algorithm 7.2.

Chapter 8

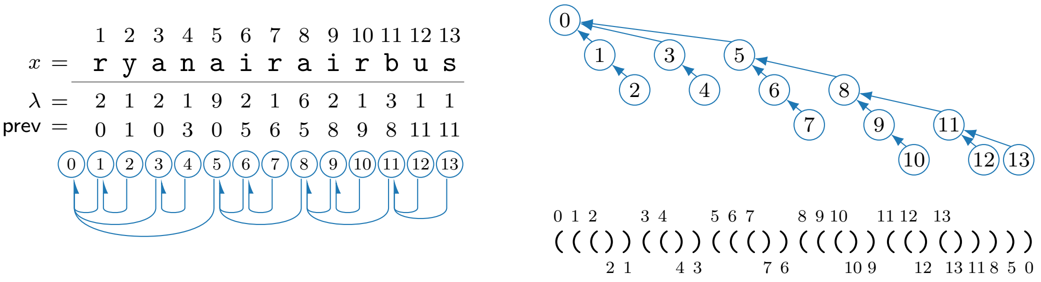
8 Computing the Succinct Lyndon Array in Small Working Space

The algorithm from Chapter 7 requires $\mathcal{O}(n)$ words or $\mathcal{O}(n \log n)$ bits of working space. This is unsatisfactory if the text is packed over $[0, \sigma)$, in which case it requires only $\mathcal{O}(n \log \sigma)$ bits of space. In this chapter, we significantly improve the working space with the following three main ideas. First, instead of storing the Lyndon array naively in $\Theta(n \log n)$ bits, we store it in its succinct representation [Lou+18], which requires only around $2n$ bits of space. Second, we show how to construct this representation in an append-only manner, while also maintaining auxiliary data structures that support fast operations on the already computed part. Third, we show that, unlike the algorithm presented in Chapter 7, which has to explicitly store the LCEs associated with each entry of the Lyndon array, we can indeed entirely avoid storing LCEs by instead using sophisticated amortization techniques. This ultimately leads to the following result, which is the most space efficient Lyndon array algorithm to date, even for linearly-sortable alphabet.

Theorem 8.1. *The succinct $2n + 2$ bit representation of the Lyndon array of a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and $\mathcal{O}(n \log \log n / \log n)$ bits of additional working space (apart from the space needed for input and output).*

8.1 Storing the Lyndon Array as a Balanced Parentheses Sequence

As mentioned before, naively storing the Lyndon array requires $\Theta(n \log n)$ bits (since each entry is an integer from $[1, n]$). In contrast to that, the succinct Lyndon array, first described in [Lou+18], requires only around $2n$ bits of space. This representation is a balanced parentheses sequence that describes the nested structure of Lyndon substrings (previously observed, e.g., in [SR03, Fra+16]). We provide a new semantic interpretation of the succinct Lyndon array; we show that it is a common tree encoding applied to the previous smaller suffix tree, which we already briefly mentioned in Section 6.2. Now we give a precise definition of the PSS tree and the associated terminology. An example is provided in Figure 8.1a.



(a) Lyndon array λ and PSS array prev. The edges drawn beneath form the PSS tree with root node 0.

(b) The PSS tree and its BPS. The numbers above and below the parentheses indicate the correspondence between nodes and parentheses.

Figure 8.1: Data structures for the string `ryanairairbus`.

Definition 8.2. Given a string $x[1..n]$, its previous smaller suffix (PSS) tree is a rooted ordered tree with nodes $[0, n]$. The unique *root* node is 0. Further it holds:

- Every node $r \in [1, n]$ is a *child* of its *parent* $\mathsf{prev}_x[r]$.
- A *leaf* is a node without children.
- The children of any node $\ell \in [0, n]$ are arranged in increasing order. If ℓ has children $r_1 < r_2 < \cdots < r_k$, then r_1 is the *leftmost child*, while r_k is the *rightmost child*.
- A node $r \in [1, n]$ is a *descendant* of a node $\ell \in [0, r)$ if and only if $\ell \in \mathsf{prev}^*_x[r]$. If r is a descendant of ℓ, then ℓ is an *ancestor* of r.

Now we show that the PSS tree inherently encodes all next smaller suffixes.

Lemma 8.3. *For a string $x[1..n]$ and a position $\ell \in [1, n]$, the descendants of ℓ in the PSS tree are exactly the nodes from $(\ell, \mathsf{next}_x[\ell])$.*

Proof. Assume that there is some node $r \in (\ell, \mathsf{next}[\ell])$ that is not a descendant of ℓ, i.e., $\ell \notin \mathsf{prev}^*[r]$. By Lemma 7.3 (*i*), there is some $i \in (\ell, r]$ such that $x_\ell \succ x_i$. However, this contradicts the fact that $i \in (\ell, \mathsf{next}[\ell])$. Next, assume that there is some $r \in [\mathsf{next}[\ell], n]$ that is a descendant of ℓ, i.e., $\ell \in \mathsf{prev}^*[r]$. By Lemma 7.3 (*i*), it holds $\forall i \in (\ell, r] : x_\ell \prec x_i$. However, this implies $x_\ell \prec x_{\mathsf{next}[\ell]}$ due to $\mathsf{next}[\ell] \in (\ell, r]$. □

We only need two operations on the PSS tree in order to simulate access to the arrays prev, next, and λ. Given a node $i \in [1, n]$, we need to be able to report its parent $\mathsf{Parent}(i)$ and the size $\mathsf{SubtreeSize}(i)$ of the subtree that is rooted in node i. By definition of the PSS tree, we can then report $\mathsf{prev}[i] = \mathsf{Parent}(i)$, and due to Lemmas 6.7 and 8.3 also $\mathsf{next}[i] = i + \mathsf{SubtreeSize}(i)$ and $\lambda[i] = \mathsf{SubtreeSize}(i)$. For example, in Figure 8.1 it holds $\lambda[8] = \mathsf{SubtreeSize}(8) = 6$.

Storing the PSS Tree as a Parentheses Sequence While a naive pointer-based representation of the PSS tree would lead to $\Theta(n)$ words of space, we can instead use a succinct tree encoding that requires only $2n + 2$ *bits* of space. One such encoding is the *balanced parentheses sequence* (see, e.g., [MR01]), in which we interpret each 1-bit as an opening parenthesis `(`, and each 0-bit as a closing parenthesis `)`.

Definition 8.4. Given a string x, the balanced parentheses sequence (BPS) of its PSS tree is recursively defined as $\mathcal{B}_x[1..2n+2] = \mathcal{B}_x(0)$, where

- for any leaf node ℓ of the PSS tree, it holds $\mathcal{B}_x(\ell) = \texttt{()}$, and
- for any non-leaf node ℓ of the PSS tree with children $r_1 < \cdots < r_k$, it holds $\mathcal{B}_x(\ell) = \texttt{(} \cdot \mathcal{B}_x(r_1) \cdot \ldots \cdot \mathcal{B}_x(r_k) \cdot \texttt{)}$.

We omit the subscript x whenever it is clear from context.

Alternatively, we can define $\mathcal{B}$ in the following constructive way. We write $\mathcal{B}$ in an append-only manner, starting with an empty sequence. We perform a depth-first traversal of the tree (see, e.g., [Cor+22, Section 20.3]), during which we visit the children of each node in increasing (i.e., left-to-right) order. Whenever we walk down an edge from $\mathsf{prev}[i]$ to i, we append the opening parenthesis of node i. Whenever we walk up an edge from i to $\mathsf{prev}[i]$, we write the closing parenthesis of node i. An example of the BPS is provided in Figure 8.1b. This description of the BPS also defines a one-to-one relation between nodes and opening parentheses, as well as nodes and closing parentheses. We say that an opening parenthesis and a closing parenthesis match, if and only if they belong to the same node.

During the traversal, we may assign preorder-numbers from $[0, n]$ to the nodes. The root has preorder-number 0. Whenever we walk down an edge from $\mathsf{prev}[i]$ to i, we assign the smallest so far unused preorder-number to node i. The order of preorder-numbers is induced by the order of children, as well as by the fact that every node is a child of a smaller node. This leads to Observation 8.5, which was also made by Fischer [Fis10] for the related 2d-min-heap, more commonly known under the name LRM (for left-to-right minima) tree [SN10, BFN11]). In fact, a reader familiar with the LRM tree and suffix arrays may notice that the PSS tree is merely the LRM tree of the inverse suffix array.

Observation 8.5. *In the PSS tree of $x[1..n]$, every node $i \in [0, n]$ has preorder-number i. The ith opening parenthesis in the BPS corresponds to node i.*

There are multiple data structures that, given a BPS encoding an ordered tree of n nodes, augment or replace the BPS with an index of size $2n + o(n)$ bits that supports fast tree operations (e.g., [MR01, SN10, NS14]). Given the preorder-number of a node, these data structures then return the preorder-number of its parent and also the size of the subtree rooted in the node in constant time. The construction time and working space in bits is $\mathcal{O}(n)$, and hence, given the BPS of the PSS tree, we can in $\mathcal{O}(n)$ time and bits of space obtain an index of size $2n + o(n)$ bits that simulates constant time access to λ, prev, and next.

8.2 Maintaining Operations on a BPS Prefix

In Section 8.3, we will show how to directly compute the BPS $\mathcal{B}$ of the PSS tree in an append-only manner, i.e., at any given point in time, we have already written a prefix of $\mathcal{B}$. For an efficient implementation of the construction algorithm, it is crucial that we maintain support for the following queries in constant time.

- Given the position o_i of an opening parenthesis in $\mathcal{B}$, determine the node i that belongs to the parenthesis. We have $i = \mathrm{rank}_{\mathrm{open}}(o_i) - 1$, where $\mathrm{rank}_{\mathrm{open}}(o_i)$ is the number of opening parentheses in $\mathcal{B}[1..o_i]$.
- Given a node i, find the index o_i of the corresponding opening parenthesis in $\mathcal{B}$. We have $o_i = \mathrm{select}_{\mathrm{open}}(i) = \min\{o \mid \mathrm{rank}_{\mathrm{open}}(o) > i\}$.
- Given an integer $k \geq 1$, find the index $o_{\mathrm{uncl}(k)} = \mathrm{select}_{\mathrm{uncl}}(k)$ of the kth *unclosed* parenthesis in $\mathcal{B}$. An opening parenthesis is called unclosed, if we have not written the matching closing parenthesis yet. For example, there are five opening parentheses in `(()((`, but only the first and the third one are unclosed.

There are space efficient data structures that can be built on top of $\mathcal{B}$ and support the required queries in constant time. However, such data structures are either static, or they are dynamic with super-constant query or update times. In the remainder of the section, we show how to use existing static data structures as a black box for append-only parentheses sequences.

8.2.1 Static Data Structures

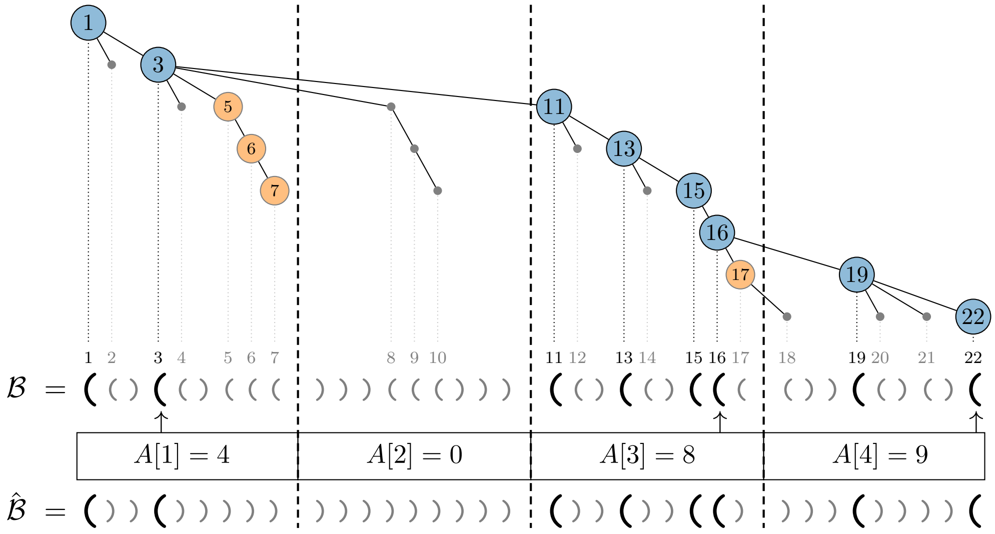
We start by describing the static data structures used by the solution for the append-only setting. In theory, we could simply use the range-min-max tree [NS14] as a static data structure for all queries. However, the range-min-max tree converts the BPS into a sequence of aB-trees [Pat08]. We prefer a data structure that merely *augments* the BPS with additional information (rather than replacing it), and thus our solution is based on the much simpler index by Golynski [Gol07].

Lemma 8.6 ([Gol07])**.** *Given a (not necessarily balanced) parentheses sequence $\mathcal{B}$ of length n, there is a data structure of size $n + \mathcal{O}(n \log\log n/\log n)$ bits that answers* $\mathrm{rank}_{\mathrm{open}}$ *and* $\mathrm{select}_{\mathrm{open}}$ *queries in constant time. The data structure consists of $\mathcal{B}$ augmented with additional tables, and it can be constructed on $\mathcal{O}(n)$ time and $\mathcal{O}(n \log\log n/\log n)$ bits of working space (apart from the space needed to store $\mathcal{B}$).*[a]

[a] Golynski does not explicitly describe the construction time and space. However, the data structure consists of arrays and universal lookup tables that can easily be computed in linear time, using negligible space apart from the space needed for the arrays and tables themselves.

Hence we have efficient data structures for $\mathrm{rank}_{\mathrm{open}}$ and $\mathrm{select}_{\mathrm{open}}$, and we only need a solution for $\mathrm{select}_{\mathrm{uncl}}$. Below, we show how to reduce $\mathrm{select}_{\mathrm{uncl}}$ to $\mathrm{select}_{\mathrm{open}}$, which means that we can simply use another instance of Lemma 8.6.

Lemma 8.7. *Given a (not necessarily balanced) parentheses sequence $\mathcal{B}$ of length n, there is a data structure of size $n + \mathcal{O}(n \log\log n/\log n)$ bits that answers* $\mathrm{select}_{\mathrm{uncl}}$ *queries in constant time. The data structure consists of $\mathcal{B}$ augmented with additional tables, and it can be constructed on $\mathcal{O}(n)$ time and $\mathcal{O}(n \log\log n/\log n)$ bits of working space (apart from the space needed to store $\mathcal{B}$).*



(a) Sequence $\mathcal{B}[1..n]$ contains eight unclosed parentheses, which correspond to the large nodes. We split $\mathcal{B}$ into blocks of length $\lceil \log_2 n/2 \rceil$, and for the sake of visualization we pretend that $\lceil \log_2 n/2 \rceil = 9$. Blocks $\mathcal{B}[1..9]$ and $\mathcal{B}[19..27]$ are identical, but they contain a different number of unclosed parentheses. This is because the opening parentheses corresponding to the medium size nodes are unclosed with respect to their block, but not with respect to the entire sequence $\mathcal{B}$. Array $A[1..4]$ stores the relative position of the rightmost unclosed parenthesis in each block. In $\hat{\mathcal{B}}$, it holds $\hat{\mathcal{B}}[i] =$ (if and only if $\mathcal{B}[i]$ is an unclosed parenthesis.

input block interpretation	input block $\mathcal{B}(ik-k..ik]$	input $A[i]$	output block $\hat{\mathcal{B}}(ik-k..ik]$
))))))))) $= (000000000)_2 = 0$	0	1	—
	⋮	⋮	⋮
$\mathcal{B}[1..9] = \mathcal{B}[19..27]$ = (()(()((($= (110110111)_2 = 439$	439	1	())))))))
	439	2	—
	439	3	—
	439	4	())()))))
	439	5	—
	439	6	—
	439	7	())())())
	439	8	())())(()
	439	9	())())(((
	⋮	⋮	⋮
((((((((($= (111111111)_2 = 511$	511	9	(((((((((

(b) Lookup table with $k = 9$ for retrieving blocks of the simulated sequence $\hat{\mathcal{B}}$.

Figure 8.2: Data structures used in the proof of Lemma 8.7.

Proof. Let $\hat{\mathcal{B}}$ be the length-n parentheses sequence such that $\hat{\mathcal{B}}[i] = \texttt{(}$ if and only if $\mathcal{B}[i]$ is unclosed (see Figure 8.2a). Then answering $\mathsf{select}_{\mathrm{uncl}}$ on $\mathcal{B}$ is equivalent to answering $\mathsf{select}_{\mathrm{open}}$ on $\hat{\mathcal{B}}$. Hence we use the data structure from Lemma 8.6 constructed for $\hat{\mathcal{B}}$. However, we cannot afford to actually store $\hat{\mathcal{B}}$. Instead, we compute a small auxiliary data structure that allows us to simulate access to $\hat{\mathcal{B}}$ using $\mathcal{B}$. In order to use Lemma 8.6 for the simulated sequence $\hat{\mathcal{B}}$, we have to be able to provide the following access in constant time. Given a position $i \in [1, n]$ and a length $\ell \in [0, \lceil \log_2 n \rceil)$ with $i + \ell \leq n$, return $\hat{\mathcal{B}}[i..i + \ell]$ (packed in a word).

Let $k = \lceil (\log_2 n)/2 \rceil$ and assume without loss of generality that k divides n. We conceptually divide $\mathcal{B}$ and $\hat{\mathcal{B}}$ into blocks of size k. In an array $A[1..n/k]$, we store $A[i] = 0$ whenever the block $\hat{\mathcal{B}}(ik - k..ik]$ contains no opening parenthesis, or equivalently when $\mathcal{B}(ik-k..ik]$ contains no unclosed parenthesis. Otherwise, we store the maximal value $A[i] \in [1, k]$ such that $\hat{\mathcal{B}}[ik - k + A[i]] = \texttt{(}$, i.e., $\mathcal{B}[ik - k + A[i]]$ is the rightmost unclosed parenthesis in $\mathcal{B}(ik - k..ik]$. An example is provided in Figure 8.2a. We compute A by enumerating the unclosed parentheses in right-to-left order as follows. We start with an all-zero array $A[1..n/k]$. We process $\mathcal{B}$ one parenthesis $\mathcal{B}[j]$ at a time and in right-to-left order. A counter h is initially 0 and keeps track of the excess of closing parentheses. Whenever $\mathcal{B}[j] = \texttt{)}$, we increase h by one. Whenever $\mathcal{B}[j] = \texttt{(}$ and $h > 0$, we decrease h by one. Whenever $\mathcal{B}[j] = \texttt{(}$ and $h = 0$, we leave h unchanged. In this case, we know that $\mathcal{B}[j]$ is an unclosed parenthesis in block $i = \lceil j/k \rceil$. If $A[i] = 0$, then we assign $A[i] \leftarrow j - (ik - k)$. It is easy to see that this computes A correctly and in $\mathcal{O}(n)$ time.

Now we describe how to extract a block $\hat{\mathcal{B}}(ik - k..ik]$ for some $i \in [1, n/k]$. If $A[i] = 0$, then we simply return $\texttt{)}^k$. Otherwise, note that the unclosed parentheses in $\mathcal{B}(ik-k..ik]$ can be enumerated with the same technique that we used for computing $A[i]$, i.e., starting with a counter $h = 0$ and scanning $\mathcal{B}(ik-k..ik-k+A[i]]$ from right to left. (We only need the array A to find a suitable position for starting the scan with $h = 0$.) Whenever we find an unclosed parentheses, we set the corresponding bit in the result word. When implemented naively, this approach takes $\mathcal{O}(k)$ time. Note that $\hat{\mathcal{B}}(ik-k..ik]$ depends solely on $\mathcal{B}(ik-k..ik]$ and $A[i]$. There are k possible values of $A[i]$, and the block $\mathcal{B}(ik - k..ik]$ is one of 2^k possible parentheses sequences of length k. The query result $\hat{\mathcal{B}}(ik - k..ik]$ is also a parentheses sequence of length k. Thus, we can precompute $\hat{\mathcal{B}}(ik - k..ik]$ for every possible $\mathcal{B}(ik - k..ik]$ and $A[i]$ in a lookup table of $k \cdot 2^k = \mathcal{O}(\sqrt{n} \cdot \log n)$ entries, each of size $k = \mathcal{O}(\log n)$ bits. An entry of the table can be computed naively in $\mathcal{O}(k) = \mathcal{O}(\log n)$ time. Hence $o(n/\log n)$ time and bits of space suffice for computing and storing the table. Using the table, a block of $\hat{\mathcal{B}}$ can be retrieved in constant time. See Figure 8.2b for an example.

In order to answer an arbitrary query asking for $\hat{\mathcal{B}}[i..i + \ell]$, we simply extract the at most three blocks of $\hat{\mathcal{B}}$ that overlap $\hat{\mathcal{B}}[i..i+\ell]$. Then, it is trivial to obtain $\hat{\mathcal{B}}[i..i+\ell)$ by applying bit-wise logical operations and bit-shifts on the three blocks. □

8.2.2 Dynamic Data Structures

As mentioned earlier, the algorithm for Theorem 8.1 requires that we maintain support for $\mathsf{rank}_{\mathrm{open}}$, $\mathsf{select}_{\mathrm{open}}$, and $\mathsf{select}_{\mathrm{uncl}}$ while writing $\mathcal{B}$ in an append-only manner. Now we explain how to use Lemmas 8.6 and 8.7 for this purpose, starting with the solution for $\mathsf{rank}_{\mathrm{open}}$. The general idea is to divide $\mathcal{B}[1..n]$ into chunks of size $m = \lceil n/\log^2 n \rceil$. At any given moment, we have already written a prefix $\mathcal{B}[1..bm+m']$, where $b \in [0, \lfloor n/m \rfloor]$ and $m' \in [0, m)$. There are b complete chunks $B_1, B_2, \ldots, B_b$

with $B_i = \mathcal{B}(im - m..im]$, and one incomplete chunk $B_{b+1} = \mathcal{B}(bm..bm + m']$. For every complete chunk B_i, we store the static data structure from Lemma 8.6 and the number O_i of opening parentheses that precede the chunk (i.e., the opening parentheses in $\mathcal{B}[1..im - m]$). If a query position of $\text{rank}_{\text{open}}$ lies within a complete chunk B_i, then the answer is the sum of O_i and the answer of the corresponding rank query within B_i, which can be obtained in constant time. For the only incomplete chunk B_{b+1}, we store the value O_{b+1} and the query answers for all the positions in B_{b+1} (which can easily be maintained when appending parentheses). Hence we can answer any query in constant time. Once we complete chunk B_{b+1}, we construct its static data structure and compute O_{b+2} for the new incomplete chunk.

The solution for $\text{select}_{\text{open}}$ is slightly more involved. In order to locate the kth opening parenthesis (counting from 1), we first determine the maximal i such that $O_i < k$. Since there are only $\mathcal{O}(\log^2 n)$ chunks, we can use a fusion node [FW93, PT14] to compute i in constant time (some trivial adjustments are needed if multiple O_i are identical). If $i < b + 1$, then we know that the kth opening parenthesis is the $(k - O_i)$th opening parenthesis in the complete chunk B_i. Hence we can use the static data structure from Lemma 8.6 constructed for chunk B_i to return the answer in constant time. Otherwise, the kth opening parenthesis is the $(k - O_{b+1})$th opening parenthesis in the incomplete chunk. We can afford to explicitly store the positions of all the opening parentheses in the incomplete chunk, and thus we can output the answer in constant time. Once we complete the chunk, we have to insert the newly computed O_{b+2} into the fusion node.

Finally, we solve $\text{select}_{\text{uncl}}$, which is again more involved. We construct the data structure from Lemma 8.7 for each chunk; however, this only allows us to select unclosed parentheses *with respect to each chunk*. The difficulty lies in the fact that an unclosed parenthesis with respect to a chunk may get closed when appending a new chunk, and thus it may not be unclosed with respect to the entire prefix $\mathcal{B}[1..bm + m']$. Hence we additionally maintain, for each complete chunk B_i, the number E_i of unclosed parentheses with respect to $\mathcal{B}[1..bm]$ that it contains (i.e., with respect to the prefix consisting exactly of the complete chunks). We additionally store the number $S_i = \sum_{j \in [1,i)} E_i$ of unclosed parentheses in all preceding chunks, also for the incomplete chunk B_{b+1} (where the S_i have the same purpose as the O_i for answering select queries). We use a stack to keep track of the unclosed parentheses in B_{b+1}. Whenever we append an opening parenthesis, we push its position onto the stack. Whenever we append a closing parenthesis, we either pop the topmost stack element, or, whenever the stack is empty, we increment a counter C by one. This counter indicates how many of the unclosed parentheses with respect to $\mathcal{B}[1..bm]$ have already been closed by B_{b+1}. We store the stack in an array of size m. When a query asks for the kth unclosed parenthesis (counting from 1), we first check if $S_{b+1} - C \geq k$, i.e., if the kth unclosed parenthesis lies in a complete chunk. If this is the case, then we use a fusion node to find the maximal i with $S_i < k$, and then use the static data structure to find the $(k - S_i)$th unclosed parenthesis in B_i. Otherwise, the kth unclosed parenthesis is the $(k - (S_{b+1} - C))$th unclosed parenthesis within the already written part of B_{b+1}, and we can simply look up its location by accessing the array that stores the stack. As soon as we complete the newly appended chunk B_{b+1}, we update the counters E_i as follows. We start with the rightmost chunk that was already complete, i.e., $i = b$. As long as $C > E_i$, we assign $C \leftarrow C - E_i$ and $E_i \leftarrow 0$, and then continue with $i \leftarrow i - 1$. As soon as $C \leq E_i$, we assign $E_i \leftarrow E_i - C$ and $C \leftarrow 0$. Since unclosed parentheses get closed in right-to-left order, it is easy to see that the described procedure correctly updates the number of unclosed parentheses in each chunk. We then recompute all the values S_i. Next, we construct the static data structures from Lemma 8.7 for chunk B_{b+1}, and compute E_{b+1} (the current size of the stack) and $S_{b+2} = S_{b+1} + E_{b+2}$. Finally, we empty the stack and recompute the fusion node that manages the values S_i from scratch.

Whenever we append a parenthesis, we have to update the explicitly stored answers for the incomplete chunk (e.g., the stack for $\text{select}_{\text{uncl}}$), which takes constant time. The space needed for the explicitly stored answers is $\mathcal{O}(m \log n) = \mathcal{O}(n/\log n)$ bits. Once we complete a chunk, we compute its static data structures from Lemmas 8.6 and 8.7 in $\mathcal{O}(m)$ time (summing to $\mathcal{O}(n)$ time for all chunks), permanently adding $\mathcal{O}(m \log\log m/\log m) = \mathcal{O}(m \log\log n/\log n)$ bits of space to the data structure, and summing to overall $\mathcal{O}(n \log\log n/\log n)$ bits for all chunks. There are only $\mathcal{O}(\log^2 n)$ values O_i, E_i, and S_i, and updating them and recomputing the fusion nodes for the O_i and S_i takes $\mathcal{O}(\text{polylog}(n))$ time and space [PT14]. The total space needed (on top of $\mathcal{B}$) is $\mathcal{O}(n \log\log n/\log n)$ bits, and the amortized time for appending a parenthesis is constant. Hence we can maintain constant time operations in the claimed complexity bounds.

8.3 Constructing the PSS Tree

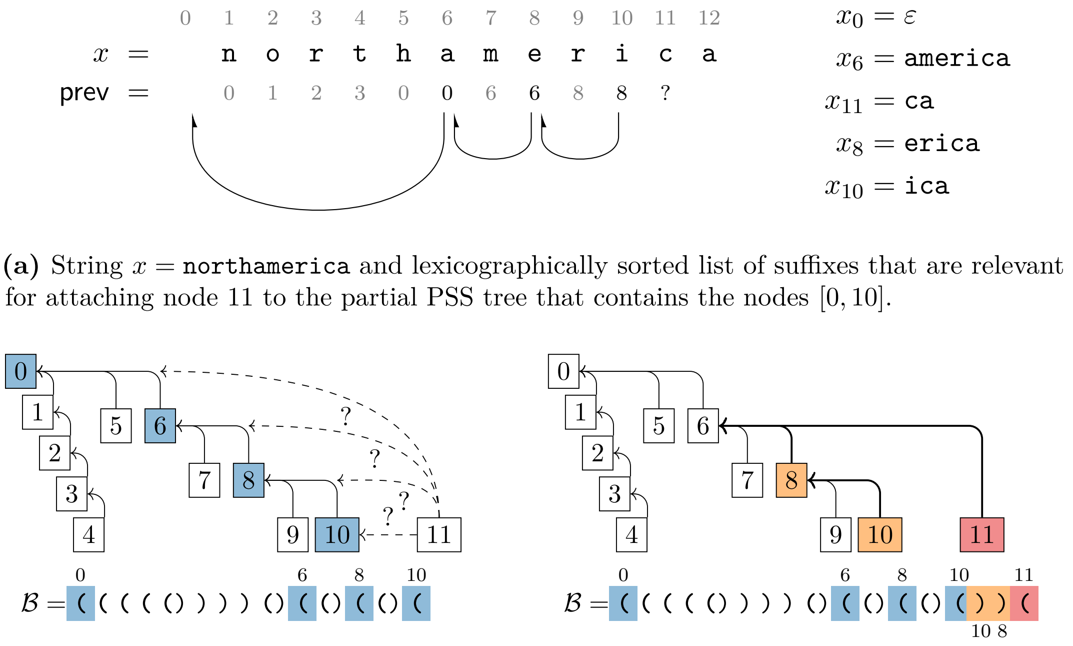
Equipped with fast operations on a BPS prefix, we now describe how to actually compute the BPS of the PSS tree in an append-only manner. We start with a simple algorithm that is a straight-forward adaptation of Algorithm 7.1(a) from Chapter 7. Suppose that we have already computed the subtree induced by nodes $[0, i)$. Attaching node i requires finding $\mathsf{prev}[i]$. A strategy for this follows from Corollary 7.5 (i), which states that

(a) $\mathsf{prev}[i]$ lies on the already computed path from $i-1$ to the root 0 (where the nodes on the path are exactly the elements of $\mathsf{prev}^*[i-1]$), and

(b) on this path, $\mathsf{prev}[i]$ is the deepest node (or equivalently the rightmost position) j such that either $j = 0$ or $x[j..n] \prec x[i..n]$.

These properties imply an algorithm in which the positions are inserted into the tree one by one and in left-to-right order, similarly to how the algorithm from Chapter 7 fills the PSS array in left-to-right order. This is also the approach of Algorithm 8.1, which directly computes the BPS of the tree.

At the time at which the algorithm starts processing position i, the sequence $\mathcal{B}$ contains the prefix of the BPS that ends with the opening parenthesis of node $i-1$, and the stack $\mathcal{Q}$ contains exactly the nodes on the path from $i-1$ (topmost stack element) to the root 0 (bottommost stack element). A loop is used to find the topmost element j on the stack that satisfies $j = 0$ or $x[j..n] \prec x[i..n]$ (lines 4–8). By properties (a) and (b), the final value of j is the previous smaller suffix of i, which means that node i will be attached as a child of j. Hence we pop the nodes on the path from $i-1$ to j (but excluding j) from the stack, and then push i on the stack (lines 7 and 10). As explained earlier, the BPS encodes a depth-first traversal of this tree. In terms of this traversal, we just moved from node $i-1$ up to node j, and then down to node i. Thus, we write one closing parenthesis for each step up (line 6), and then one opening parenthesis for moving down to node i (line 9). After


0 1 2 3 4 5 6 7 8 9 10 11 12
x = n o r t h a m e r i c a
prev = 0 1 2 3 0 0 6 6 8 8 ?
$x_0 = \varepsilon$
$x_6 =$ america
$x_{11} =$ ca
$x_8 =$ erica
$x_{10} =$ ica
0 1 2 3 4 5 6 7 8 9 10 11
? ? ? ?
$\mathcal{B} =$ ((((( )))) ( ) (( ) (( ) (
0 6 8 10
$\mathcal{B} =$ ((((( )))) ( ) (( ) (( ) ( )) (
0 6 8 10 11
10 8

(a) String $x =$ northamerica and lexicographically sorted list of suffixes that are relevant for attaching node 11 to the partial PSS tree that contains the nodes $[0, 10]$.

(b) PSS tree before attaching node 11.

(c) PSS tree after attaching node 11.

Figure 8.3: The partial PSS tree before and after processing index 11 of the string $x =$ northamerica during the execution of Algorithms 8.1 and 8.2. We have $p_1 = 10$, $p_2 = 8$, $p_3 = 6$, $p_4 = 0$, and $p_m = p_3$.

Algorithm 8.1 Simple construction of the PSS tree.

```
Require: String $x[1..n]$ over general ordered alphabet.
Ensure: BPS $\mathcal{B}$ of the PSS tree of $x$.
$\mathcal{B} \leftarrow$ (                                  ▷ opening parenthesis of node 0
$\mathcal{Q} \leftarrow$ stack that contains only 0
for $i = 1$ to $n$ do
    $j \leftarrow \mathcal{Q}.top()$
    while $j > 0$ and $x[i..n] \prec x[j..n]$ do
        append ) to $\mathcal{B}$                     ▷ closing parenthesis of node $j$
        $\mathcal{Q}.pop()$
        $j \leftarrow \mathcal{Q}.top()$
    append ( to $\mathcal{B}$                         ▷ opening parenthesis of node $i$
    $\mathcal{Q}.push(i)$
append $|\mathcal{Q}|$ times ) to $\mathcal{B}$   ▷ closing parentheses of nodes on path from $n$ to 0
```

processing position n, we write the closing parentheses of the nodes on the path from n to 0 (line 11). Figure 8.3 shows the BPS before and after an outer loop iteration.

The correctness follows from properties (*a*) and (*b*). However, the algorithm poses two challenges that make an efficient implementation difficult. First, we cannot efficiently perform a lexicographical comparison of suffixes over general ordered alphabet. Second, to achieve the desired space complexity, we cannot afford to explicitly store the stack. Nevertheless, we will still use Algorithm 8.1 as a starting point for our solution. An alternative description of Algorithm 8.1 is provided in Algorithm 8.2, which omits the inner loop and the stack by instead using the set $\mathsf{prev}^*[i-1]$, which contains exactly the elements that were previously stored on the stack. It should be easy to see that the positions contained in $\mathsf{prev}^*[i-1]$ also directly correspond to the so far unclosed parentheses. Hence we can use the data structures from Section 8.2 to obtain any $p_q = \text{rank}_{\text{open}}(\text{select}_{\text{uncl}}(k-q+1)) - 1$ in constant time. We only have to maintain the number k of unclosed parentheses, which is trivial. Hence the question is how to, given access to all the p_q, efficiently find $m \in [1,k]$ such that $p_m = \mathsf{prev}[i]$.

Algorithm 8.2 Constructing the BPS of the PSS tree.

```
Require: String $x[1..n]$ over general ordered alphabet.
 Ensure: BPS of the PSS Tree of $x$.
$\mathcal{B} \leftarrow$ (                                            ▷ open node 0
for $i = 1$ to $n$ do
    Let $\mathsf{prev}^*[i-1] = \{p_1,\dots,p_k\}$ with $\forall q \in [1,k) : \mathsf{prev}[p_q] = p_{q+1}$
    Determine $p_m = \mathsf{prev}[i]$
    Append $m-1$ closing parentheses to $\mathcal{B}$          ▷ close nodes $p_1,\dots,p_{m-1}$
    Append one opening parenthesis to $\mathcal{B}$                      ▷ open node $i$
Append $|\mathsf{prev}^*[n]|$ closing parentheses to $\mathcal{B}$          ▷ close rightmost path
```

8.3.1 Efficiently Computing p_m

Algorithm 8.1 computes $p_m = \mathsf{prev}[i]$ by iterating over the indices $p_1,\dots,p_k$ in descending order (i.e., p_1 first, p_k last; the p_q are exactly the stack elements). For each index p_q, it evaluates whether $x_{p_q} \prec x_i$. As soon as this is the case, we have found $p_m = \mathsf{prev}[i]$. The cost of this approach is high: A naive suffix comparison between x_{p_q} and x_i takes $\textsc{lce}(p_q,i)+1$ individual character comparisons, which means that we spend $\mathcal{O}(m + \sum_{q=1}^{m} \textsc{lce}(p_q,i))$ time to determine m. However, the following property will allow us to decrease this time bound significantly.

Property 8.8 (Bitonic LCE Values)**.** *Let $p_1,\dots,p_k$ be exactly the elements of $\mathsf{prev}^*[i-1]$ in descending order and let $p_m = \mathsf{prev}[i]$. Furthermore, let $\ell_q = \textsc{lce}(p_q,i)$ for all $q \in [1,k]$. We have $\ell_1 \le \ell_2 \le \dots \le \ell_{m-1}$ as well as $\ell_m \ge \ell_{m+1} \ge \dots \ge \ell_k$.*

Proof. Follows from $x_{p_1} \succ \ldots \succ x_{p_{m-1}} \succ x_i \succ x_{p_m} \succ \ldots \succ x_{p_k}$ and simple properties of the lexicographical order. □

From now on, we continue using the notation $\ell_q = \textsc{lce}(p_q,i)$ from the corollary. Note that the longest LCE between i and any of the p_q occurs either with p_m or with p_{m-1}. Let $\ell_{\max} = \max(\ell_{m-1},\ell_m)$ be this largest LCE value, then our more

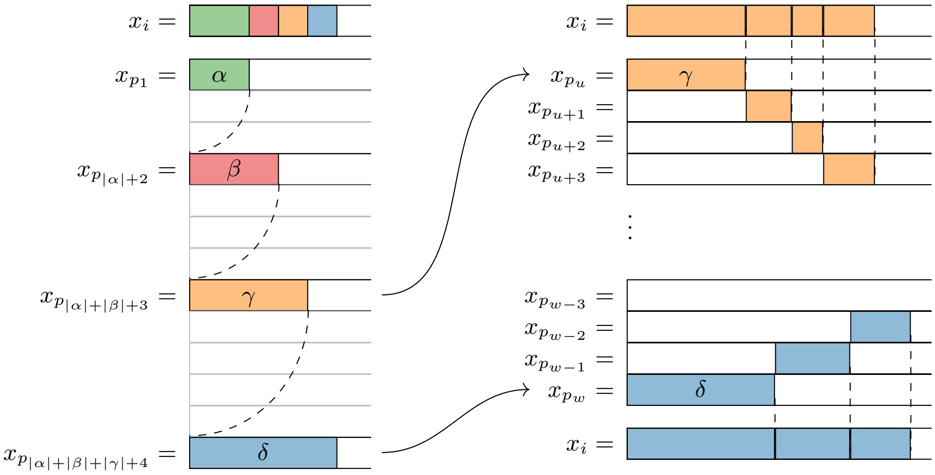


Figure 8.4: Matching character comparisons when determining p_m. On the left we have the suffix x_i as well as $x_{p_1}, x_{p_2}, \ldots, x_{p_w}$, which are relevant for the first step. Each prefix $\alpha, \beta, \gamma, \delta$ highlights the longest shared prefix between the respective suffix x_{p_q} and x_i. On the right side we have the suffixes $x_{p_u}, x_{p_{u+1}}, \ldots, x_{p_w}$, which are relevant for the second step.

sophisticated approach for determining m only takes $\mathcal{O}(m + \ell_{\max})$ time. It consists of two steps. First, we determine a candidate interval $(u, w] \subseteq [1, k]$ of size at most $\ell_{\max}$ that contains m. In the second step we gradually narrow down the borders of the candidate interval until the exact value of m is known.

Step 1: Find a candidate interval Our goal is to find $(u, w] = (u, u + \ell_u + 1]$ with $m \in (u, w]$. Initially, we naively compute $\ell_1 = \text{LCE}(p_1, i)$, allowing us to evaluate $x_{p_1} \prec x_i$. If this condition holds, then we have $m = 1$ and no further steps are necessary. Otherwise, let $u \leftarrow 1$ and (i) let $w \leftarrow u + \ell_u + 1$. We already know that $u < m$ holds. Now we have to evaluate if $m \leq w$ also holds, unless $w \geq k$, in which case we assign $w \leftarrow k$ and terminate. If $w < k$, we compute $\ell_w = \text{LCE}(p_w, i)$ naively, which allows us to check in constant time if $x_{p_w} \prec x_i$ and decide if $m \leq w$ holds. If this is not the case, then we assign $u \leftarrow w$ as well as $\ell_u \leftarrow \ell_w$ and continue at (i). If however $x_{p_w} \prec x_i$, then we have $m \leq w$ and therefore $m \in (u, w]$. Figure 8.4 (left) outlines the procedure.

Step 2: Narrow down $(u, w]$ to the exact value of m Now we gradually tighten the borders of the candidate interval. If ℓ_u is smaller than ℓ_w, then we try to increase u by one. Otherwise, we try to decrease w by one.

Assume that we have $\ell_u < \ell_w$, then it follows from Property 8.8 that $\ell_{u+1} \geq \ell_u$ holds. Therefore, when computing ℓ_{u+1} we can simply skip the first ℓ_u character comparisons. Now we use ℓ_{u+1} to evaluate in constant time if $x_{p_{u+1}} \succ x_i$ holds. If that is the case, then we have $u + 1 < m$ and thus we can assign $u \leftarrow u + 1$ and start Step 2 from the beginning. If however $x_{p_{u+1}} \prec x_i$ holds, then we have $m = u + 1$ and no further steps are necessary. In case of $\ell_u \geq \ell_w$ we proceed analogously. Once again, Figure 8.4 (right) visualizes the procedure.

Time Complexity Step 1 is dominated by computing LCE values. Determining the final LCE value ℓ_w takes $\ell_w + 1$ individual character comparisons and thus $\Theta(\ell_w + 1)$ time. Whenever we compute any previous value of ℓ_w, we increase w by $\ell_w + 1$ afterwards. Therefore, the time for computing all LCE values is bounded by $\Theta(w + \ell_w) = \Theta(u + \ell_u + \ell_w) \subseteq \mathcal{O}(m + \ell_{\max})$. Since initially $(u, w]$ has size at most $\ell_{\max}$, we call Step 2 at most $\mathcal{O}(\ell_{\max})$ times. With every call we increase ℓ_u or ℓ_w by exactly the number of matching character comparisons that we perform. Therefore, the total number of matching character comparisons is bounded by $2\ell_{\max}$. Thus, the total time needed for Step 2 is $\mathcal{O}(\ell_{\max})$. In sum, processing index i takes $\mathcal{O}(m + \ell_{\max})$ time. For the total processing time of all indices (and thus the execution time of Algorithm 8.2) we get:

$$\begin{aligned} &\sum_{i=1}^{n} \mathcal{O}\big(\overbrace{|\mathsf{prev}^*[i-1] \cap [\mathsf{prev}[i], i]|}^{m} \big) + \sum_{i=1}^{n} \mathcal{O}\big(\overbrace{\max_{p_q \in \mathsf{prev}^*[i-1]} \text{LCE}(p_q, i)}^{\ell_{\max}} \big) \\ = \quad & \mathcal{O}(n) \qquad\qquad\qquad\qquad + \quad \mathcal{O}(n^2) \end{aligned}$$

The $\mathcal{O}(m)$-terms sum to $\mathcal{O}(n)$ since $m - 1$ is exactly the number of closing parentheses that we write while processing i, and there are exactly $n + 1$ closing parentheses in the entire BPS. As it appears, the total time bound of the algorithm is still far from linear time. However, it is easy to identify the crucial time component that makes the algorithm too expensive. From now on we call the $\mathcal{O}(m)$ term of the processing time *negligible*, while the $\mathcal{O}(\ell_{\max})$ term is called *critical*.

Clearly, if we could somehow remove the critical terms, we would already achieve linear time. There exists a variety of data structures that could help us to achieve this goal by accelerating suffix comparisons, e.g., the (compressed or sparse) suffix tree, the (compressed) suffix array, or dedicated data structures for fast LCE queries (like the one from Lemma 4.4). However, none of them can be constructed in $\mathcal{O}(n)$ time over general ordered alphabet, and (even if we were considering a linearly-sortable alphabet) such data structures are too large to achieve the desired space complexity. This motivates the techniques that we describe in the following sections, which directly remove the critical terms without relying on any of the aforementioned data structures. This way, the execution time of Algorithm 8.2 decreases to $\mathcal{O}(n)$, while the additional working space remains unchanged.

8.3.2 Achieving Linear Time

The critical time component for processing index i is $\ell_{\max} = \max_{p_q \in \mathsf{prev}^*[i-1]} \text{LCE}(p_q, i)$. When processing i with the technique from Section 8.3.1, we inherently find out the exact value of $\ell_{\max}$, and we also discover the index $p_{\max}$ for which we have $\text{LCE}(p_{\max}, i) = \ell_{\max}$. From now on, we simply use $\ell = \ell_{\max}$ and $j = p_{\max}$. While discovering a large LCE value ℓ is costly, it yields valuable structural information about the input text: There is a repeating substring of length ℓ with occurrences $x[j..j + \ell)$ and $x[i..i + \ell)$. Intuitively, there is also a large repeating structure in the PSS tree, and consequently a repeating substring in $\mathcal{B}$. This motivates the techniques shown in this section, which conceptually alter Algorithm 8.2 as follows. Whenever we finish processing an index i with critical cost ℓ, we skip the next $\Omega(\ell)$ iterations of the loop by simply extending the BPS prefix with the copy of an already computed part, which means that the amortized critical cost per index becomes constant.

Depending on j and ℓ we choose either the *run extension* (Section 8.3.2.1) or the *amortized look-ahead* (Section 8.3.2.2) to perform the extension. Algorithm 8.3 outlines our final construction algorithm on a higher level, and complements the written description by showing when the special cases arise. Before going into detail, we point out that $x[j..i)$ is a Lyndon word. As mentioned earlier, it follows from Property 8.8 that j equals p_m or p_{m-1}. Regardless of which case applies, it definitely holds $\mathsf{next}[p_{m-1}] = i$ and $\mathsf{prev}[i] = p_m$, and thus Lemma 6.6 implies that $x[j..i)$ is a Lyndon word.

Algorithm 8.3 Linear time construction of the PSS tree.

```
Require: String $x[1..n]$ over general ordered alphabet.
 Ensure: BPS $\mathcal{B}$ of the PSS Tree of $x$.
 1: $\mathcal{B} \leftarrow$ (
 2: for $i = 1$ to $n$ do
 3:     Let $\mathsf{prev}^*[i-1] = \{p_1, \dots, p_k\}$ with $\mathsf{prev}[p_q] = p_{q+1}$
 4:     Determine $p_m = \mathsf{prev}[i]$,
                using the technique from Section 8.3.1, causing critical cost
                $\ell$ and discovering the index $j$ with $\textsc{lce}(j, i) = \ell$ as described
                in the beginning of Section 8.3.2.
 5:     Append $m - 1$ closing parentheses to $\mathcal{B}$          ▷ close node $p_1, \dots, p_{m-1}$
 6:     Append one opening parenthesis to $\mathcal{B}$                       ▷ open node $i$
        (For any $q$, let $o_q$ be the opening parenthesis of node $q$.)
 7:     if $\ell \geq 2(i - j)$ then
 8:         Apply the run extension as described in Section 8.3.2.1.
                  Let $t = \lfloor \ell/(i-j) \rfloor + 1$. Take $\mathcal{B}(o_j..o_i]$ and append it $(t-2)$
                  times to $\mathcal{B}$. Continue in line 2 with $i \leftarrow i + (t-2) \cdot (i-j)$.
 9:     else
10:         Apply the amortized look-ahead as described in Section 8.3.2.2.
                  Using Lemma 8.12, find the largest value $\chi \in [1, \lfloor \ell/4 \rfloor]$ that
                  satisfies $\mathcal{B}[o_j..o_{j+\chi-1}] = \mathcal{B}[o_i..o_{i+\chi-1}]$, and append a copy
                  of $\mathcal{B}(o_j..o_{j+\chi-1}]$ to $\mathcal{B}$. Continue in line 2 with $i \leftarrow i + \chi$.
                  If $\chi < \lfloor \ell/4 \rfloor$, then iteration $i + \chi$ will automatically skip
                  additional $\Omega(\ell)$ iterations by using the run extension.
11: Append $|\mathsf{prev}^*[n]|$ closing parentheses to $\mathcal{B}$
```

8.3.2.1 Run Extension

We apply the run extension whenever $\ell \geq 2(i - j)$. It is easy to see that in this case $x[j..j+\ell)$ and $x[i..i+\ell)$ overlap such that the Lyndon word $\mu = x[j..i)$ repeats itself at least three times, starting at index j. We call the substring $x[j..i+\ell)$ *Lyndon run with period* $|\mu|$. The number of *repeated occurrences* is $t = \lfloor \ell / |\mu| \rfloor + 1 \geq 3$, and the starting positions of the repeated occurrences are $r_1, \dots, r_t$ with $r_1 = j$, $r_2 = i$, and generally $r_q = r_{q-1} + |\mu|$. In a moment we will show that in this particular situation the following lemma holds.

Lemma 8.9. *Let o_q be the index of the opening parenthesis of node q in $\mathcal{B}$. Then we have $\mathcal{B}[o_{r_1}..o_{r_2}] = \mathcal{B}[o_{r_2}..o_{r_3}] = \cdots = \mathcal{B}[o_{r_{t-1}}..o_{r_t}]$.*

Expressed less formally, each repeated occurrence of μ — except for (possibly) the last one — induces the same substring in the BPS. Performing the run extension is as easy as taking the already written substring $\mathcal{B}(o_{r_1}..o_{r_2}] = \mathcal{B}(o_j..o_i]$, and appending it $t-2$ times to $\mathcal{B}$. Afterwards, the last parenthesis that we have written is the opening parenthesis of node r_t, and we continue the execution of Algorithm 8.2 with iteration $r_t + 1$. Thus, we have skipped the processing of $r_t - i$ indices. Since

$$r_t - i \;=\; (t-2)\cdot|\mu| \;\geq\; \frac{(t-2)\cdot|\mu|}{t\cdot|\mu|}\cdot\ell \;\geq\; \frac{1}{3}\cdot\ell \;=\; \Omega(\ell),$$

it follows that the average critical cost per index from $[i, r_t]$ is constant.

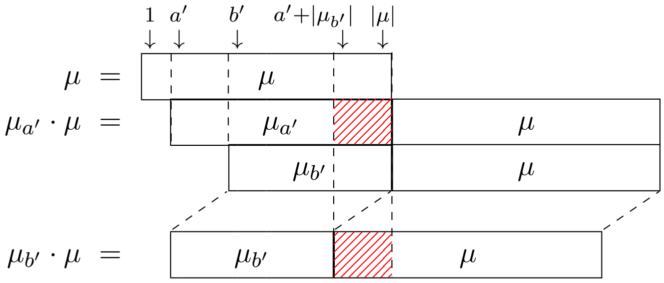
Proving the Lemma It remains to be shown that Lemma 8.9 holds. It is sufficient to prove the correctness for $t = 3$, since the correctness for the general case inductively follows by repeatedly applying the lemma with $t = 3$. Therefore, we only have to show $\mathcal{B}[o_{r_1}..o_{r_2}] = \mathcal{B}[o_{r_2}..o_{r_3}]$.

Isomorphic Subtrees Since μ is a Lyndon word, it is easy to see that the suffixes at the starting positions of the repeated occurrences are lexicographically smaller than the suffixes that begin in between the starting positions of the repeated occurrences, i.e., we have $\forall q \in (r_1, r_2) : x_{r_1} \prec x_q$ and $\forall q \in (r_2, r_3) : x_{r_2} \prec x_q$. Consequently, the indices from (r_1, r_2) are descendants of r_1 in the PSS tree, and the indices from (r_2, r_3) are descendants of r_2 in the PSS tree, i.e., each of the intervals $[r_1, r_2)$ and $[r_2, r_3)$ induces a tree.

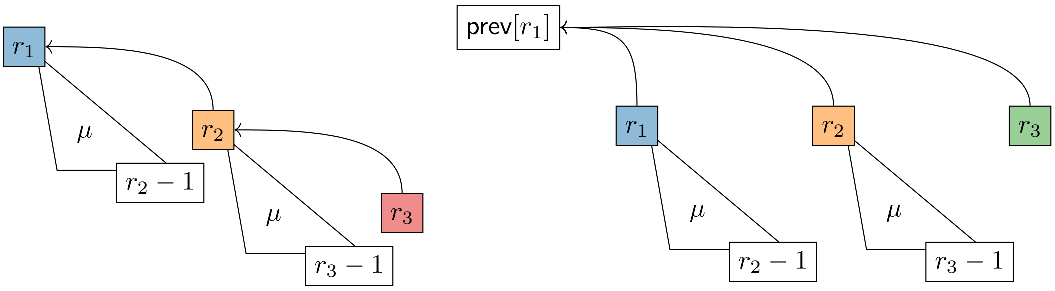
Next, we show that these trees are actually isomorphic. Clearly, the tree induced by $[r_1, r_2)$ solely depends on the lexicographical order of suffixes that begin within $[r_1, r_2)$, and the tree induced by $[r_2, r_3)$ solely depends on the lexicographical order of suffixes that begin within $[r_2, r_3)$. Assume that the trees are *not* isomorphic, then there must be a suffix comparison that yields different results in each interval, i.e., there must be offsets $a, b \in [0, |\mu|)$ such that $x_{r_1+a} \prec x_{r_1+b} \iff x_{r_2+a} \succ x_{r_2+b}$. However, this is impossible, as shown by the lemma below.

Lemma 8.10. *For $a, b \in [0, |\mu|)$ it holds $x_{r_1+a} \prec x_{r_1+b} \iff x_{r_2+a} \prec x_{r_2+b}$.*

Proof. The statement is trivial if $a = b$. Assume w.l.o.g. $a < b$, and let $a' = a + 1$ and $b' = b + 1$. We can show that the strings $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ have a mismatch:



Consider the two hatched areas in the drawing above. The top area highlights the suffix $\mu_{a'+|\mu_{b'}|}$ of μ, which has length $c = |\mu| - (a' + |\mu_{b'}|) + 1$. The bottom area



(a) Increasing run. **(b)** Decreasing run.

Figure 8.5: The run of the Lyndon word $\mu = x[r_1..r_2) = x[r_2..r_3) = x[r_3..r_3 + |\mu|)$ induces isomorphic subtrees in the PSS tree. If $x_{r_1} \prec x_{r_2}$, then the roots of the subtrees form a chain **(a)**. Otherwise, they have the same parent **(b)**.

highlights the prefix $\mu[1..c]$ of μ. Since μ is a Lyndon word, there is no non-trivial suffix of μ that is also a prefix of μ. It follows that the hatched areas cannot be equal, i.e., $\mu_{a'+|\mu_{b'}|} \neq \mu[1..c]$. This guarantees a mismatch between $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$. Therefore, appending an arbitrary string to $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ does not influence the outcome of a lexicographical comparison. The statement of the lemma directly follows by appending x_{r_3} and x_{r_4} respectively:

$$\begin{aligned}
\mu_{a'} \cdot \mu \prec \mu_{b'} \cdot \mu \quad &\iff \quad \underbrace{\mu_{a'} \cdot \mu \cdot x_{r_3}}_{= \; x_{r_1+a}} \quad \prec \quad \underbrace{\mu_{b'} \cdot \mu \cdot x_{r_3}}_{= \; x_{r_1+b}} \\
&\iff \quad \underbrace{\mu_{a'} \cdot \mu \cdot x_{r_4}}_{= \; x_{r_2+a}} \quad \prec \quad \underbrace{\mu_{b'} \cdot \mu \cdot x_{r_4}}_{= \; x_{r_2+b}}
\end{aligned}$$

□

Finally, we show that in the PSS tree the induced isomorphic trees are connected in a way that ultimately implies $\mathcal{B}[o_{r_1}..o_{r_2}] = \mathcal{B}[o_{r_2}..o_{r_3}]$. There are two possible scenarios for this connection, which depend on the so called *direction* of the Lyndon run. We call a run *increasing* if $x_{r_1} \prec x_{r_2}$, and *decreasing* otherwise.

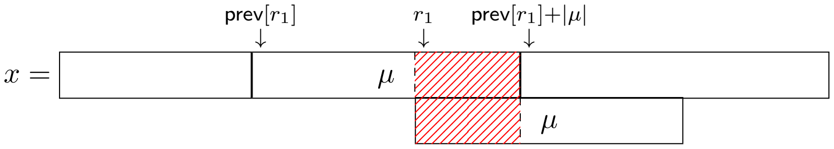
Increasing Runs First, we focus on increasing runs. It follows from $x_{r_1} \prec x_{r_2} \iff \mu \cdot x_{r_2} \prec \mu \cdot x_{r_3} \iff x_{r_2} \prec x_{r_3}$ that $x_{r_1} \prec x_{r_2} \prec x_{r_3}$. Since μ is a Lyndon word, we have $\forall q \in (r_1, r_2) : x_{r_2} \prec x_q$ as well as $\forall q \in (r_2, r_3) : x_{r_3} \prec x_q$. Therefore, we have $\mathsf{prev}[r_2] = r_1$ and $\mathsf{prev}[r_3] = r_2$, and the isomorphic subtrees are connected as visualized in Figure 8.5a. It is easy to see that a preorder-traversal from r_1 to r_2 yields the same sequence of parentheses as a preorder-traversal from r_2 to r_3. Therefore we have $\mathcal{B}[o_{r_1}..o_{r_2}] = \mathcal{B}[o_{r_2}..o_{r_3}]$, which means that Lemma 8.9 holds for increasing runs.

Decreasing Runs With the same argument as for increasing runs, we have $x_{r_1} \succ x_{r_2} \succ x_{r_3}$ in decreasing runs. We also have $\forall q \in (r_1, r_2) : x_{r_2} \prec x_q$ as well as $\forall q \in (r_2, r_3) : x_{r_3} \prec x_q$, which means that $\mathsf{prev}[r_2] \leq \mathsf{prev}[r_1]$ and $\mathsf{prev}[r_3] \leq \mathsf{prev}[r_1]$ hold. In Lemma 8.11, we show that $\mathsf{prev}[r_1] = \mathsf{prev}[r_2] = \mathsf{prev}[r_3]$, such that the isomorphic subtrees are connected as visualized in Figure 8.5b. A preorder-traversal from r_1 to r_2 yields the same sequence of parentheses as a preorder-traversal from

r_2 to r_3. Therefore we have $\mathcal{B}[o_{r_1}..o_{r_2}] = \mathcal{B}[o_{r_2}..o_{r_3}]$, which means that Lemma 8.9 holds for decreasing runs.

Lemma 8.11. *In decreasing runs we have* $\mathsf{prev}[r_1] = \mathsf{prev}[r_2] = \mathsf{prev}[r_3]$.

Proof. As explained previously, we have $\mathsf{prev}[r_2] \leq \mathsf{prev}[r_1]$ and $\mathsf{prev}[r_3] \leq \mathsf{prev}[r_1]$, and thus only need to show $x_{\mathsf{prev}[r_1]} \prec x_{r_2}$ and $x_{\mathsf{prev}[r_1]} \prec x_{r_3}$. We will show below that μ cannot be a prefix of $x_{\mathsf{prev}[r_1]}$, from which the statement of the lemma can be deduced easily because the suffixes x_{r_1}, x_{r_2}, and x_{r_3} begin with the prefix μ. Assume for the sake of contradiction that μ is a prefix of $x_{\mathsf{prev}[r_1]}$. If we also assume $\mathsf{prev}[r_1] + |\mu| > r_1$, then the situation is as visualized below.



As indicated by the hatched area, there is a non-trivial suffix of μ that is also a prefix of μ, which contradicts the fact that μ is a Lyndon word. Thus we have $\mathsf{prev}[r_1] + |\mu| \not> r_1$. Also, we cannot have $\mathsf{prev}[r_1] + |\mu| = r_1$, because then $\mathsf{prev}[r_1]$ would be the starting position of another repeated occurrence of μ, which would imply $x_{\mathsf{prev}[r_1]} \succ x_{r_1}$. It follows $\mathsf{prev}[r_1] + |\mu| < r_1$, i.e., $\mathsf{prev}[r_1] + |\mu| \in (\mathsf{prev}[r_1], r_1)$ and thus $x_{\mathsf{prev}[r_1]+|\mu|} \succ x_{r_1}$. However, as shown below, this leads to a contradiction.

$$\begin{aligned} x_{\mathsf{prev}[r_1]} \prec x_{r_1} &\iff \mu \cdot x_{\mathsf{prev}[r_1]+|\mu|} \prec \mu \cdot x_{r_2} \\ &\iff \quad x_{\mathsf{prev}[r_1]+|\mu|} \prec x_{r_2} \\ &\underset{x_{r_1} \succ x_{r_2}}{\Longrightarrow} \quad x_{\mathsf{prev}[r_1]+|\mu|} \prec x_{r_1} \end{aligned}$$

Hence μ is not a prefix of $x_{\mathsf{prev}[r_1]}$, and the correctness of the lemma follows. □

8.3.2.2 Amortized Look-Ahead

Finally, we show how to amortize the critical cost $\mathcal{O}(\ell)$ of processing index i if the run extension is not applicable, i.e., if we have $\ell < 2(i-j)$. Unfortunately, the trees induced by the nodes from $[j, j+\ell)$ and $[i, i+\ell)$ are not necessarily isomorphic. However, we can still identify a sufficiently large isomorphic structure. In a moment we will show that the following lemma holds.

Lemma 8.12. *Let o_q be the index of the opening parenthesis of node q in $\mathcal{B}$. We either have $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] = \mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$, or there is an integer $\chi < \lfloor \ell/4 \rfloor$ with $\mathcal{B}[o_j..o_{j+\chi-1}] = \mathcal{B}[o_i..o_{i+\chi-1}]$ and an index $h \in [i, i+\chi)$ such that $x[h..i+\ell)$ is a Lyndon run of the Lyndon word $x[h..i+\chi)$. We can determine which case applies, and also determine the value of χ (if applicable) in $\mathcal{O}(\ell)$ time and $\mathcal{O}(1)$ words of additional space.*

When performing the amortized look-ahead, we first determine which case of the lemma applies. Then, if $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] = \mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$, we extend the known prefix of the BPS by appending a copy of $\mathcal{B}(o_j..o_{j+\lfloor \ell/4 \rfloor -1}]$, and continue the execution of Algorithm 8.2 with iteration $i + \lfloor \ell/4 \rfloor$. Since this way we skip the processing

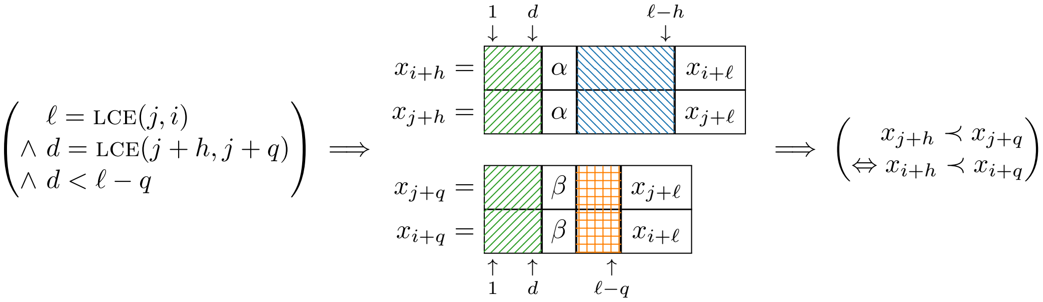


Figure 8.6: Proving Lemma 8.12. Equal patterns indicate equal substrings.

of $\lfloor \ell/4 \rfloor - 1 = \Omega(\ell)$ indices, the average critical cost per index from $[i, i + \lfloor \ell/4 \rfloor)$ is constant. If, however, the second case applies, then we determine the value of χ and extend the known prefix of the BPS by appending a copy of $\mathcal{B}(o_j..o_{j+\chi-1}]$, allowing us to continue the execution of Algorithm 8.2 with iteration $i + \chi$. We know that there is some $h \in [i, i+\chi)$ such that $x[h..i+\ell)$ is a Lyndon run of the Lyndon word $\mu = x[h..i+\chi)$. This run might even be longer; let $\ell' = \text{LCE}(h, i+\chi)$, then $x[h..i+\chi+\ell')$ is the longest run of μ that starts at index h. If the run is increasing, then $\mathsf{prev}[i+\chi] = h$ holds (see Section 8.3.2.1), and the longest LCE that we discover when processing index $i+\chi$ is ℓ'. If the run is decreasing, then $\mathsf{prev}[i+\chi] = \mathsf{prev}[h]$ holds. Also in this case, the longest LCE that we discover when processing index $i+\chi$ is ℓ', since $\text{LCE}(\mathsf{prev}[i+\chi], i+\chi)$ is less than $|\mu|$ (see proof of Lemma 8.11). Therefore, the critical cost of processing index $i+\chi$ will be $\mathcal{O}(\ell')$. However, since the Lyndon run has at least three repeated occurrences, we will also skip the processing of $\Omega(\ell')$ indices by using the run extension. The algorithmic procedure for the second case can be summarized as follows. We process index i with critical cost $\mathcal{O}(\ell)$ and skip $\chi - 1$ indices afterwards. Then, we process index $i+\chi$ with critical cost $\mathcal{O}(\ell')$ and skip another $\Omega(\ell')$ indices by using the run extension. Since we have $\ell' = \Omega(\ell)$, the total critical cost is $\mathcal{O}(\ell')$, and the total number of processed or skipped indices is $\Omega(\ell')$. Thus, the average critical cost per index is constant.

Proving Lemma 8.12 It remains to be shown that Lemma 8.12 holds. For this purpose, assume $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] \neq \mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$. From now on we refer to $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}]$ and $\mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$ as *left* and *right side*, respectively. Consider the first mismatch between the two, where w.l.o.g. we assume that the mismatch has an opening parenthesis on the left side, and a closing one on the right side. On the left side, the opening parenthesis corresponds to a node $j+q$ with $q \in [1, \lfloor \ell/4 \rfloor)$ that is a child of another node $j+h$. Since $x[j..i)$ is a Lyndon word, all nodes from $(j, j + \lfloor \ell/4 \rfloor) \subset (j, i)$ are descendants of j (where $j + \lfloor \ell/4 \rfloor < i$ because we only apply the amortized look-ahead if $\ell < 2(i-j)$). Consequently, we have $h \in [0, q)$. Now we look at the right side. Since we have a closing parenthesis instead of an opening one, we know that $i+q$ is not attached to $i+h$, but to a smaller node, i.e., we have $\mathsf{prev}[i+q] < i+h$. It follows that $x_{j+h} \prec x_{j+q}$ and $x_{i+h} \succ x_{i+q}$. Let $d = \text{LCE}(j+h, j+q)$ and assume $d \leq \ell - q$. Then due to $x_{j+h} \prec x_{j+q}$ we have $x[j+h+d] < x[j+q+d]$. However, since we have $x[j..j+\ell] = x[i..i+\ell]$, it follows $\text{LCE}(i+h, i+q) = d$ and $x[i+h+d] < x[i+q+d]$, which contradicts $x_{i+h} \succ x_{i+q}$ (see Figure 8.6). Thus, it holds $d = \text{LCE}(j+h, j+q) \geq \ell - q$, allowing us to show that $x[j+h..j+\ell)$ is a Lyndon run with period $q-h$. Since $\mathsf{prev}[j+q] = j+h$ holds, it follows from Lemma 6.6 that $x[j+h..j+q)$ is a Lyndon word. Due to

$\textsc{lce}(j+h, j+q) > \ell - q \geq 3(\ell/4) \geq 3(q-h)$ we know that the Lyndon word repeats at least four times, and the run extends all the way to the end of $x[j..j+\ell)$. Note that since the opening parenthesis of node $j+q$ causes the first mismatch between $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}]$ and $\mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$, we have $\mathcal{B}[o_j..o_{j+q-1}] = \mathcal{B}[o_i..o_{i+q-1}]$. Therefore, $\chi \leftarrow q$ already satisfies Lemma 8.12.

Finally, we explain how to determine $\chi = q$ in $\mathcal{O}(\ell)$ time. As described above, if $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] \neq \mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$, then there is some offset $h < \lfloor \ell/4 \rfloor$ such that $x[j+h..j+\ell)$ is a Lyndon run of at least four repeated occurrences of a Lyndon word μ with $|\mu| \leq \lfloor \ell/4 \rfloor$. Consequently, $x[j+\lfloor \ell/4 \rfloor..j+\ell)$ has the form $\text{suf}(\mu) \cdot \mu^t \cdot \text{pre}(\mu)$ with $t \geq 2$, where $\text{suf}(\mu)$ and $\text{pre}(\mu)$ are respectively a proper suffix and a proper prefix of μ (both possibly empty). A string of this form is called *extended Lyndon run*. In Section 8.3.2.3 we propose an algorithm that checks whether or not $x[j+\lfloor \ell/4 \rfloor..j+\ell)$ is an extended Lyndon run in $\mathcal{O}(\ell)$ time and constant additional space. If $x[j+\lfloor \ell/4 \rfloor..j+\ell)$ is not an extended Lyndon run, then we have $\mathcal{B}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] = \mathcal{B}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$ and no further steps are needed to satisfy Lemma 8.12. Otherwise, the algorithm from Section 8.3.2.3 also provides the period $|\mu|$ of the run, as well as $|\text{suf}(\mu)|$. In this case, we try to further extend the extended Lyndon run to the left: We are now not only considering $x[j+\lfloor \ell/4 \rfloor..j+\ell)$, but $x[j..j+\ell)$. We want to find the leftmost index $j+h$ that is the starting position of a repeated occurrence of μ. Given $|\mu|$ and $|\text{suf}(\mu)|$, this can be done naively by scanning $x[j..j+\lfloor \ell/4 \rfloor]$ from right to left, which takes $\mathcal{O}(\ell)$ time. If we have $h \geq \lfloor \ell/4 \rfloor - |\mu|$, then the first case of Lemma 8.12 applies and no further steps are necessary. Otherwise, we use $\chi \leftarrow h + |\mu|$. This concludes the proof of Lemma 8.12 and the description of the construction algorithm.

8.3.2.3 Detecting Extended Lyndon Runs

In this section, we propose a linear time algorithm that identifies extended Lyndon runs, i.e., strings of the form $\text{suf}(\mu) \cdot \mu^t \cdot \text{pre}(\mu)$ with $t \geq 2$, where $\text{suf}(\mu)$ and $\text{pre}(\mu)$ are a proper suffix and a proper prefix of μ. Our approach exploits properties of the Lyndon factorization, the definition of which is repeated below.

Theorem 6.4 (Lyndon Factorization [CFL58, Duv83, Lot83])**.**
Every non-empty string $x[1..n]$ has a unique factorization $x = f_1 f_2 \ldots f_k$ such that each f_i is a Lyndon word, and $f_1 \succeq f_2 \succeq \ldots \succeq f_k$. This factorization can be computed in $\mathcal{O}(n)$ time over general ordered alphabet.

Lemma 8.13. *Let $x = \text{suf}(\mu) \cdot \mu^t \cdot \text{pre}(\mu)$ be an extended Lyndon run. Let $q_1, \ldots, q_{k_1}$ be the Lyndon factorization of $\text{suf}(\mu)$, and let $y_1, \ldots, y_{k_2}$ be the Lyndon factorization of $\text{pre}(\mu)$. Then the Lyndon factorization of x is* $q_1, \ldots, q_{k_1}, \underbrace{\mu, \mu, \ldots, \mu}_{t \text{ times}}, y_1, \ldots, y_{k_2}$.

Proof. Clearly, the factorization given by the lemma consists solely of Lyndon words. Thus, we only have to show $q_1 \succeq \ldots \succeq q_{k_1} \succeq \mu \succeq y_1 \succeq \ldots \succeq y_{k_2}$. Since we defined $q_1, \ldots, q_{k_1}$ and $y_1, \ldots, y_{k_2}$ to be the Lyndon factorizations of $\text{suf}(\mu)$ and $\text{pre}(\mu)$ respectively, we already know that $\forall i \in [2, k_1] : q_{i-1} \succeq q_i$ and $\forall i \in [2, k_2] : y_{i-1} \succeq y_i$ hold. It remains to be shown that $q_{k_1} \succeq \mu \succeq y_1$ holds. Since q_{k_1} is a non-empty suffix of $\text{suf}(\mu)$ and thus also a non-trivial suffix of μ, it follows that $q_{k_1} \succ \mu$ (because μ is a

Lyndon word and hence lexicographically smaller than all of its non-trivial suffixes). Since y_1 is a prefix of $\mathrm{pre}(\mu)$ and thus also a prefix of μ, it follows (by definition of the lexicographical order) that $\mu \succ y_1$. □

Lemma 8.13 implies that the longest factor of the Lyndon factorization of an extended Lyndon run is exactly the repeating Lyndon word μ. This is the key insight that we use to detect extended Lyndon runs.

Lemma 8.14. *Let x be a string of length n. If x is an extended Lyndon run of the form $x = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$, then we can determine $|\mu|$ and $|\mathrm{suf}(\mu)|$ in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ words of additional space.*

Proof. Using Duval's algorithm [Duv83, Algorithm 2.1], we compute the Lyndon factorization of x in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ words of additional space. The algorithm computes and outputs the factors one at a time and in left-to-right order. Whenever it outputs a factor that is longer than all previous ones, we store its length l and its starting position d. Note that since we investigate each factor individually and then immediately discard it, we never need to store the entire factorization in memory. If x is an extended Lyndon run, then following Lemma 8.13 it must have the form $x = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ with $|\mathrm{suf}(\mu)| = d - 1$ and $|\mu| = l$. Since we know d and l, checking whether $x = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ holds can be achieved by performing a simple scan over x. □

8.4 Algorithmic Summary & Adaptation to the Lyndon Array

We now summarize our construction algorithm for the PSS tree. We process the indices from left to right using the techniques from Section 8.3.1, where processing an index means attaching it to the PSS tree. Whenever the critical time of processing an index is $\mathcal{O}(\ell)$, we skip the next $\Omega(\ell)$ indices by using the run extension (Section 8.3.2.1) or the amortized look-ahead (Section 8.3.2.2). Thus, the critical time per index is constant, and the total worst-case execution time is $\mathcal{O}(n)$. In terms of working space, we only need $\mathcal{O}(n \log\log n / \log n)$ bits to support the operations described in Section 8.2. The correctness of the algorithm follows from the description. Hence we have shown the main result of the chapter.

Theorem 8.1. *The succinct $2n + 2$ bit representation of the Lyndon array of a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and $\mathcal{O}(n \log\log n / \log n)$ bits of additional working space (apart from the space needed for input and output).*

The algorithm can easily be adapted to compute the plain Lyndon array instead of the PSS tree. For this purpose, we use a single array $\mathcal{A}$ (which later becomes the Lyndon array), and no further auxiliary data structures. We maintain the following invariant: At the time we start processing index i, we have $\mathcal{A}[j] = \mathsf{prev}[j]$ for $j \in \mathsf{prev}^*[i-1]$, and $\mathcal{A}[j] = \lambda[j]$ for $j \in [1, i) \setminus \mathsf{prev}^*[i-1]$. As before, we determine $p_m = \mathsf{prev}[i]$ with the techniques from Section 8.3.1. In Step 1 and Step 2 we require some access on elements of $\mathsf{prev}^*[i-1]$, which we can directly retrieve from $\mathcal{A}$. Apart from that, the algorithm remains unchanged. Once we computed p_m, we set $\mathcal{A}[i] \leftarrow p_m$ $(= \mathsf{prev}[i])$. Additionally, it follows that i is the first node that is not a descendant of any of the nodes $p_1, \ldots, p_{m-1}$, which means that we have $\mathsf{next}[p_q] = i$ for any such node. Therefore, we assign $\mathcal{A}[p_q] \leftarrow i - p_q$ $(= \lambda[p_q])$. The run extension and the amortized look-ahead remain essentially unchanged, with the only difference being that we copy and append respective array intervals instead of BPS substrings (some trivial shifts on copied values are necessary). Once we have processed index n, we have $\mathcal{A}[j] = \mathsf{prev}[j]$ for $j \in \mathsf{prev}^*[n]$, and $\mathcal{A}[j] = \lambda[j]$ for $j \in [1, n] \setminus \mathsf{prev}^*[n]$. Clearly, all indices $p_q \in \mathsf{prev}^*[n]$ do not have a next smaller suffix, and we set $\mathcal{A}[p_q] \leftarrow n - p_q + 1 = \lambda[p_q]$. After this, we have $\mathcal{A} = \lambda$. Since at all times we only use $\mathcal{A}$ and no auxiliary data structures, the additional working space needed (apart from input and output) is constant. The linear execution time and correctness of the algorithm follow from the description.

Theorem 8.15. *Given a string $x[1..n]$ over general ordered alphabet, we can compute its Lyndon array λ in $\mathcal{O}(n)$ time using $\mathcal{O}(1)$ words of working space apart from the space needed for x and λ.*

8.5 Experimental Results

We implemented the construction algorithm for both the succinct and the plain Lyndon array (LA-Succ and LA-Plain). The C++ implementation is publicly available at GitHub[1]. As a baseline, we compared the throughput of our algorithms with the throughput of DivSufSort[2], which is one of the fastest suffix array construction algorithms in practice [FK17]. Thus, it can be seen as a natural lower bound for any Lyndon array construction algorithm that depends on the suffix array. Additionally we consider LA-ISA-NSV, which builds the Lyndon array by computing next smaller values on the inverse suffix array (see [Fra+16], we use DivSufSort to construct the suffix array). For LA-Succ we only construct the succinct Lyndon array without the support data structure for fast queries. Instead of the described data structures for fast selection of unclosed parentheses, we simply maintain a space efficient stack that contains the positions of all the unclosed parentheses. All experiments were conducted on the LiDO3 cluster[3], using an Intel Xeon E5-2640v4 processor and 64GiB of memory. We repeated each experiment five times and use the median as the final result. All texts are taken from the Pizza & Chili text corpus[4].

Table 8.1 shows the throughput of the different algorithms, i.e., the number of input bytes that can be processed per second. We are able to construct the plain Lyndon array at a speed of between 41 MiB/s (`fib41`) and 82 MiB/s (`xml`), which is on average 9.9 times faster than LA-ISA-NSV, and 8.1 times faster than DivSufSort. Even in the worst case, LA-Plain is still 6.8 times faster than LA-ISA-NSV, and 5.2 times faster than DivSufSort (`pitches`). When constructing the succinct Lyndon

[1] `https://github.com/jonas-ellert/nearest-smaller-suffixes`, permanently archived in the Software Heritage Archive at `https://archive.softwareheritage.org/swh:1:dir:e0423be2932248664f948e01c4aa9083929f2ce9;origin=https://github.com/jonas-ellert/nearest-smaller-suffixes;visit=swh:1:snp:14acaf39fa4ccd00ef9d1746465bb13d34966f09;anchor=swh:1:rev:72cf8906b29199476e32fdd295d792ddada40ebe`

[2] `https://github.com/y-256/libdivsufsort`

[3] `https://www.lido.tu-dortmund.de/cms/de/LiDO3/index.html`

[4] `http://pizzachili.dcc.uchile.cl/`

array we achieve around 86% of the throughput of LA-Plain on average, but never less than 81% (`pitches`). In terms of memory usage, we measured the additional working space needed apart from the space for the text and the (succinct) Lyndon array. Both LA-Plain and LA-Succ never needed more than 0.002 bytes of additional memory per input character (or 770 KiB of additional memory in total), which is why we do not list the results in detail.

8.6 Conclusion

We showed how to construct the succinct Lyndon array for a string over general ordered alphabet in linear time using $\mathcal{O}(n \log\log n / \log n)$ bits of working space. The construction algorithm can also produce the (non-succinct) Lyndon array in linear time using only $\mathcal{O}(1)$ words of working space. Regardless of the computed representation, the algorithm performs very well in practice. We envision practical applications of this result in full-text indexing. For example, it has already been used to accelerate Baier's algorithm for constructing the suffix array [Bai16, OOB22].

Table 8.1: Throughput in MiB/s. All numbers are truncated to two decimal places.

	(normal corpus)						(repetitive corpus)			
	`english` *1024 MiB*	`dna` *385 MiB*	`pitches` *53 MiB*	`proteins` *1024 MiB*	`sources` *201 MiB*	`xml` *282 MiB*	`cere` *439 MiB*	`einstein.de` *89 MiB*	`fib41` *255 MiB*	`kernel` *247 MiB*
LA-Plain	60.57	50.83	60.58	62.18	66.13	82.10	53.08	59.09	41.71	62.27
LA-Succ	52.81	46.03	49.49	52.77	57.31	68.56	48.20	50.35	35.30	54.42
LA-ISA-NSV	4.61	4.86	9.13	4.40	7.41	7.11	5.44	6.72	3.81	6.79
DivSufSort	5.53	5.76	11.61	5.21	9.25	8.62	6.57	8.45	4.20	8.45

Chapter 9

9 Computing the Succinct Lyndon Array in Sublinear Time

The algorithms from Chapters 7 and 8 compute the Lyndon array of a length-n string over general ordered alphabet in $\mathcal{O}(n)$ time. It is easy to see that this time is optimal; computing the Lyndon array requires that each symbol gets inspected at least once (i.e., each symbol must be involved in at least one symbol comparison). However, this $\Omega(n)$ time lower bound does not apply if the string is packed over integer alphabet $[0, \sigma)$. In this case, each symbol is encoded in $\lceil \log_2 \sigma \rceil$ bits, and the string occupies only $n \lceil \log_2 \sigma \rceil$ bits of space, which is at most $\mathcal{O}(n / \log_\sigma n)$ words. Hence $\mathcal{O}(n / \log_\sigma n)$ time also suffices to read the string and inspect each symbol. Since the succinct Lyndon array from Chapter 8 consists of merely $\mathcal{O}(n)$ bits, it can also be written in a word-wise manner in $\mathcal{O}(n / \log n) \subseteq \mathcal{O}(n / \log_\sigma n)$ time. This raises the question whether $\mathcal{O}(n / \log_\sigma n)$ time also suffices for computing the succinct Lyndon array. In this chapter, we positively answer this question.

Theorem 9.1. *The succinct $2n+2$ bit representation of the Lyndon array of a length-n string packed over $[0, \sigma)$ can be computed in $\mathcal{O}(n / \log_\sigma n)$ time and $\mathcal{O}(n \log \sigma)$ bits of working space.*

The plain BPS of the PSS tree, without additional support data structures, cannot be used directly to efficiently simulate the Lyndon array or the nearest smaller suffix arrays. However, as discussed in Chapter 8, we only have to support Parent and SubtreeSize operations, which can be implemented with simple primitives on the parentheses sequence, namely rank, select, find-close, and enclose (see, e.g., [MR01]). Some of the primitives, particularly rank and select, can be implemented in constant time after an $\mathcal{O}(n / \log n)$ time preprocessing [Bab+15, MNV16]. We leave the sublinear time construction of support data structures for the remaining operations as future work.

Simple Construction Algorithm for the PSS Tree We use Algorithm 8.1 from Chapter 8 as a starting point, and (for the sake of convenience) the algorithm and its description are reproduced below. Suppose that we have already computed the subtree induced by nodes $[0, i)$. Attaching node i requires finding $\mathsf{prev}[i]$. A strategy for this follows from Corollary 7.5 (i), which states that

(a) $\mathsf{prev}[i]$ lies on the already computed path from $i-1$ to the root 0, and

(b) on this path, $\mathsf{prev}[i]$ is the deepest node (or equivalently the rightmost position) j such that either $j = 0$ or $x[j..n] \prec x[i..n]$.

These properties motivate Algorithm 8.1, which directly computes the BPS of the PSS tree by inserting the nodes in left-to-right order, which means that it writes the BPS in an append-only manner.

Algorithm 8.1 Simple construction of the PSS tree.

```
Require: String $x[1..n]$ over general ordered alphabet.
 Ensure: BPS $\mathcal{B}$ of the PSS tree of $x$.
 1: $\mathcal{B} \leftarrow$ (                          ▷ opening parenthesis of node 0
 2: $\mathcal{Q} \leftarrow$ stack that contains only 0
 3: for $i = 1$ to $n$ do
 4:     $j \leftarrow \mathcal{Q}.top()$
 5:     while $j > 0$ and $x[i..n] \prec x[j..n]$ do
 6:         append ) to $\mathcal{B}$                ▷ closing parenthesis of node $j$
 7:         $\mathcal{Q}.pop()$
 8:         $j \leftarrow \mathcal{Q}.top()$
 9:     append ( to $\mathcal{B}$                    ▷ opening parenthesis of node $i$
10:     $\mathcal{Q}.push(i)$
11: append $|\mathcal{Q}|$ times ) to $\mathcal{B}$   ▷ closing parentheses of nodes on path from $n$ to 0
```

At the time at which the algorithm starts processing position i, the sequence $\mathcal{B}$ contains the prefix of the BPS that ends with the opening parenthesis of node $i-1$, and the stack $\mathcal{Q}$ contains exactly the nodes on the path from $i-1$ (topmost stack element) to the root 0 (bottommost stack element). A loop is used to find the topmost element j on the stack that satisfies $j = 0$ or $x[j..n] \prec x[i..n]$ (lines 4–8). By properties (a) and (b), the final value of j is the previous smaller suffix of i, which means that node i will be attached as a child of j. Hence we pop the nodes on the path from $i-1$ to j (but excluding j) from the stack, and then push i on the stack (lines 7 and 10). As explained earlier, the BPS encodes a depth-first traversal of this tree. In terms of this traversal, we just moved from node $i-1$ up to node j, and then down to node i. Thus, we write one closing parenthesis for each step up (line 6), and then one opening parenthesis for moving down to node i (line 9). After processing position n, we write the closing parentheses of the nodes on the path from n to 0 (line 11).

The correctness follows from properties (a) and (b). Each line takes constant time, except for the lexicographical suffix comparison in line 5. It holds $x[i..n] \prec x[j..n]$ if and only if either $\textsc{lce}(i,j) = n-i+1$ or $x[i+\textsc{lce}(i,j)] < x[j+\textsc{lce}(i,j)]$. Thus, the

LCE data structure by Kempa and Kociumaka [KK19] (see Lemma 4.4) suffices to lexicographically compare suffixes in constant time (we use this technique repeatedly throughout the chapter). The number of inner loop iterations is less than the number of closing parentheses, and hence the total time needed by the algorithm is $\mathcal{O}(n)$.

9.1 A Blockwise Algorithm for the PSS Tree

In this section, we modify Algorithm 8.1 such that instead of processing a single index at a time, it processes blocks (i.e., consecutive intervals) of indices in each step. The block size $k = \left\lfloor \frac{\log_2 n}{8\lceil \log_2 \sigma \rceil} \right\rfloor$ is approximately one eighth of the number of symbols that fit into one word of memory, and hence there are $N = \lceil \frac{n}{k} \rceil = \Theta(n/\log_\sigma n)$ blocks. Let $B_1, \ldots, B_N$ with $\forall b \in [1, N] : B_b = (bk - k, bk]$ be the sequence of blocks (where without loss of generality we assume that k divides n).

In the PSS tree, each block B_b induces a forest that contains exactly the nodes that are members of the block. For any node $j \in B_b$, if $\mathsf{prev}[j] \in B_b$, then $\mathsf{prev}[j]$ is the parent of j in the forest induced by B_b. Otherwise, j is the root of a tree in the forest. We call these trees *small trees*, and their roots *small roots*. The small roots are exactly the left-to-right lexicographical minima of suffixes starting in B_b, i.e., $i \in B_b$ is a root if and only if $\forall i' \in B_b : i' < i \implies x[i'..n] \succ x[i..n]$. Just like in the PSS tree, we arrange the children of each node in increasing order. The BPS of the forest is the concatenation of the BPSs of its small trees in left-to-right order.

High-Level Description of the Blockwise Algorithm We process the blocks one at a time in left-to-right order. At the time at which we process block B_b, we have already computed the partial PSS tree induced by all previous blocks, i.e., by $[0, bk - k]$. For B_b, we first obtain its induced PSS forest. Our goal is to attach the small roots (including their small trees) to the respective previous smaller suffixes, which lie on the path from $bk - k$ to 0 in the partial PSS tree. This is schematically shown in Figure 9.1a. Note that small roots further to the right will be attached further up in the path. This is because suffixes on the path are lexicographically decreasing towards the root, while the suffixes corresponding to small roots are lexicographically decreasing from left to right. Hence our task is to lexicographically *interleave* the path with the small roots. For an efficient implementation of this interleaving process, it is crucial that we maintain the path from $bk - k$ to 0 in a blockwise manner. Just like in Algorithm 8.1, we maintain a stack of the nodes on the path. However, each stack element is a pair $(a, \mathcal{L})$, where a indicates that we consider block B_a, and $\mathcal{L}[1..k]$ is a bitvector indicating which of the positions in the block are relevant. For $j' \in [1, k]$, it holds $\mathcal{L}[j'] = 1$ if and only if $ak - k + j'$ lies on the path from $bk - k$ to 0. The stack then contains exactly the blocks with at least one 1-bit in the bitvector. This is visualized in Figure 9.1a.

9.1.1 Detailed Description of the Blockwise Algorithm

So far, we described the algorithm in terms of the PSS tree. However, we want to directly compute its BPS. After $\mathcal{O}(N)$ preprocessing time, we can obtain the BPS of the forest induced by any block in $\mathcal{O}(1)$ time. This follows directly from the lemma below.

Lemma 9.2. *Let $x[1..n]$ be packed over $[0, \sigma)$ and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, the following type of query can be answered in $\mathcal{O}(1)$ time. Given a range $[i, i + \ell) \subseteq [1, n]$ of length $\ell \leq \frac{\log_2 n}{(2+\epsilon)\lceil \log_2 \sigma \rceil}$, output the BPS of the PSS forest induced by $[i, i + \ell)$, as well as a bitvector $\mathcal{R}[1..\ell]$ such that for $j \in [1, \ell]$ it holds $\mathcal{R}[j] = 1$ if and only if $i + j - 1$ is the root of a tree in the forest.*

The proof of the lemma is provided in Section 9.2. When we lexicographically interleave the suffixes, we will repeatedly encounter another type of query. Given a small root, we have to find its previous smaller suffix within a block on the stack. A solution for this is provided by the lemma below, which we prove in Section 9.3.

Lemma 9.3. *Let $x[1..n]$ be packed over $[0, \sigma)$ and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, we can answer the following type of query in $\mathcal{O}(1)$ time. Given a position $i \in [1, n]$ and a non-empty interval $[j, j+\ell) \subseteq [1, n]$ of length $\ell \leq \frac{\log_2 n}{(5+\epsilon)\cdot\lceil \log_2 \sigma \rceil}$, find the position $j_{\max} = \max(\{j' \in [j, j + \ell) \mid x[j'..n] \prec x[i..n]\} \cup \{j - 1\})$.*

Now we have all the tools needed to describe the algorithm. We start with an empty stack $\mathcal{Q}$ and $\mathcal{B} =$ (, i.e., with the opening parenthesis of the artificial root node 0. Now we process the blocks $B_1, \ldots, B_N$ in left-to-right order. At the time at which we start processing B_b, the stack $\mathcal{Q}$ contains the nodes on the path from $bk - k$ to 0 in the previously described blockwise manner, and $\mathcal{B}$ contains the prefix of the BPS of the PSS tree that ends with the opening parenthesis of node $bk - k$. We start by querying Lemma 9.2 with B_b and obtain the BPS $\mathcal{F}$ of the forest induced by B_b, as well as the bitvector $\mathcal{R}$ indicating the small roots. We then find the rightmost 1-bit in $\mathcal{R}$ in constant time (there are only $2^k = \mathcal{O}(\sqrt[8]{n})$ possible values of $\mathcal{R}$, hence a lookup table for rightmost or leftmost 1-bits can be precomputed in $o(n/\log n)$ time). If this bit is at position $\mathcal{R}[r'_b]$, then $r_b = r'_b + bk - k$ is the rightmost small root in the forest induced by B_b. Note that $x[r_b..n]$ is the lexicographically smallest suffix starting in B_b. The state after this step is visualized in Figure 9.1a. Now we repeatedly run the interleaving main routine described below, during which we will alter $\mathcal{F}$, $\mathcal{R}$, $\mathcal{Q}$, and $\mathcal{B}$.

Main Routine The goal of this routine is to interleave (the remaining small trees of) B_b with the topmost block on the stack. If $\mathcal{F}$ is empty (which happens if and only if $\mathcal{R}$ contains only zeroes), then we have attached all small trees and the main routine terminates. Otherwise, if $\mathcal{Q}$ is empty, the remaining small trees need to be attached to the root of the PSS tree, and we append $\mathcal{F}$ to $\mathcal{B}$. This takes $\mathcal{O}(1)$ time and also terminates the main routine.

If neither $\mathcal{F}$ nor $\mathcal{Q}$ are empty, then we retrieve and pop the topmost pair $(a, \mathcal{L})$ from $\mathcal{Q}$. We use Lemma 9.3 to obtain $r_a = \max(\{j' \in B_a \mid x[j'..n] \prec x[r_b..n]\} \cup \{ak - k\})$, and the corresponding within-block offset $r'_a = r_a - ak + k$. If $\mathsf{prev}[r_b] \in B_a$ then $r_a = \mathsf{prev}[r_b]$, and all remaining small trees have to be attached to nodes from $B_a \cap [r_a, n]$. Since r_b will be attached to r_a, none of the nodes from $B_a \cap (r_a, n]$ will remain on the stack. Hence we compute a bitvector $\mathcal{L}'[1..k]$ where for $j' \in [1, k]$ it holds $\mathcal{L}'[j'] = 1$ if and only if $\mathcal{L}[j'] = 1$ and $j' \leq r'_a$ (this takes constant time using bit-wise operations). We then push $(a, \mathcal{L}')$ back onto the stack. If however $\mathsf{prev}[r_b] \notin B_a$, then $r_a = ak - k$ (the first position to the left of B_a) and r_b will be

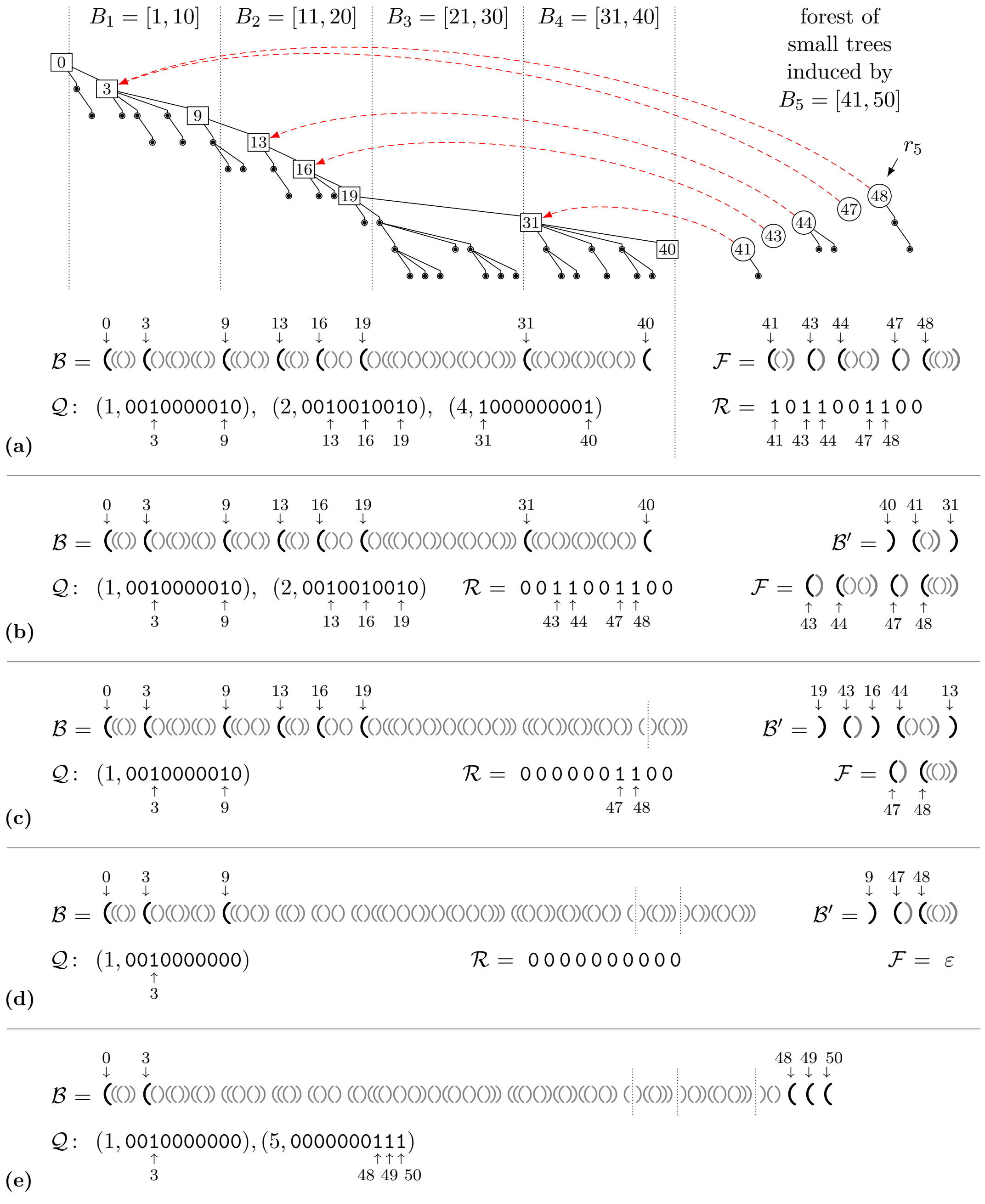

$B_1 = [1, 10]$
$B_2 = [11, 20]$
$B_3 = [21, 30]$
$B_4 = [31, 40]$
forest of small trees induced by $B_5 = [41, 50]$
r_5
0
3
9
13
16
19
31
40
41
43
44
47
48
(a)
$\mathcal{B} =$
$\mathcal{Q}$: $(1, 0010000010)$, $(2, 0010010010)$, $(4, 1000000001)$
$\mathcal{F} =$
$\mathcal{R} =$ 1 0 1 1 0 0 1 1 0 0
(b)
$\mathcal{Q}$: $(1, 0010000010)$, $(2, 0010010010)$
$\mathcal{R} =$ 0 0 1 1 0 0 1 1 0 0
$\mathcal{B}' =$
(c)
$\mathcal{Q}$: $(1, 0010000010)$
$\mathcal{R} =$ 0 0 0 0 0 0 1 1 0 0
(d)
$\mathcal{Q}$: $(1, 0010000000)$
$\mathcal{R} =$ 0 0 0 0 0 0 0 0 0 0
$\mathcal{F} = \varepsilon$
(e)
$\mathcal{Q}$: $(1, 0010000000), (5, 0000000111)$
49
50

Figure 9.1: Data structures during the execution of the blockwise algorithm with block size $k = 10$. While processing $B_5 = [41..50]$, the dashed edges will be inserted into the partial PSS tree induced by $[0..40]$. The drawings show the state of the relevant data structures before calling the subroutine during the first iteration of the main routine (a), and after calling the subroutine in the first (b), second (c), and third (d) iteration of the main routine. The state after finalizing B_5 is shown in (e).

attached to a node in a block to the left of B_a. This means that block B_a will no longer be on the stack. Note that either way $\forall j' \in (r_a, r_b) : x[j'..n] \succ x[r_b..n]$.

Our next task is as follows. We have to attach some (possibly none, possibly all) of the remaining small trees to nodes in B_a. We reflect this change in $\mathcal{F}$ and $\mathcal{R}$ by removing the corresponding prefix of $\mathcal{F}$, and setting the corresponding bits in $\mathcal{R}$ to 0. Simultaneously, we extend $\mathcal{B}$ such that it contains the newly attached small trees, possibly interleaved with additional closing parentheses of nodes from B_a. This is realized by the following interleaving subroutine, which we run in a loop (and which will later be replaced by a single constant-time table lookup). A sequence $\mathcal{B}'$ is used to buffer the parentheses that we will append to $\mathcal{B}$.

Subroutine If either $\mathcal{L}$ or $\mathcal{R}$ consists only of 0-bits, we terminate the subroutine. Otherwise, we obtain the rightmost 1-bit of $\mathcal{L}$ (with a lookup table). If this bit is at position $\mathcal{L}[j']$, then the corresponding absolute position is $j = j' + ak - k$. If $j' = r'_a$ (which is equivalent to $j = r_a = \mathsf{prev}[r_b]$), then all remaining small trees need to be attached to j, and we append $\mathcal{F}$ to $\mathcal{B}'$. We replace $\mathcal{F}$ with ε and $\mathcal{R}$ with an all-zero bitvector. This terminates the subroutine.

Otherwise (i.e., if $j' > r'_a$ or equivalently $j > r_a$), we obtain the leftmost 1-bit of $\mathcal{R}$ (with a lookup table). If this bit is at position $\mathcal{R}[i']$, then $i = i' + bk - k$ is the leftmost small root that we still have to attach. Now we have to determine if $x[i..n] \prec x[j..n]$. (This state is equivalent to reaching the head of the inner loop of Algorithm 8.1 with the current values of i and j.) It holds $x[i..n] \prec x[j..n]$ if and only if $x[i..r_b) \preceq x[j..j + r_b - i)$. This is because $j + r_b - i \in (r_a, r_b)$, and hence we have already established that $x[r_b..n] \prec x[j + r_b - i..n]$. Thus, if $x[i..r_b) = x[j..j + r_b - i)$, it immediately follows that $x[i..n] \prec x[j..n]$. Note that $x[i..r_b)$ and $x[j..j + r_b - i)$ are substrings of $x(bk - k..bk]$ and $x(ak - k..ak + k]$ respectively, which will later be relevant for an efficient implementation. If $x[i..n] \prec x[j..n]$, then we append) to $\mathcal{B}'$ (this is the closing parenthesis of node j), and we assign $\mathcal{L}[j'] = 0$. If, however, $x[i..n] \succ x[j..n]$, then $\mathsf{prev}[i] = j$. In this case, we take the prefix of $\mathcal{F}$ that corresponds to the small tree rooted in i (which is the shortest balanced prefix of $\mathcal{F}$), and append it to $\mathcal{B}'$. We remove this prefix from $\mathcal{F}$ and assign $\mathcal{R}[i'] = 0$. We then continue with the next iteration of the subroutine. After the subroutine terminates, we append $\mathcal{B}'$ to $\mathcal{B}$ and continue with the next iteration of the main routine. Figures 9.1b to 9.1d shows the result of the subroutine in three consecutive iterations of the main routine.

Finalizing the Block Once the main routine terminates for block B_b, we have attached all the small trees of B_b to $\mathcal{B}$. The stack $\mathcal{Q}$ contains the blockwise representation of all the nodes on the path from bk to 0, except for the ones in block B_b. Before we can continue with the next iteration of the main routine, we have to push $(b, \mathcal{L}'')$ on the stack, where the 1-bits in $\mathcal{L}''$ correspond to the nodes on the path from bk to r_b. Note that this information can be obtained from the state of $\mathcal{F}$ at the beginning of the main routine iteration. Since $\mathcal{F}$ is a bitvector of length $2k$, a lookup table $W[0..2^{2k})$ suffices to store the bitvector $\mathcal{L}''$ for each possible $\mathcal{F}$. The table has $\mathcal{O}(\sqrt[4]{n})$ entries and can be filled naively in $\mathcal{O}(\sqrt[4]{n} \cdot \operatorname{polylog}(n)) \subset \mathcal{O}(n/\log n)$ time. Once we need $\mathcal{L}''$, we simply look up $W[\mathsf{int}(\mathcal{F})]$ in constant time. Here, just like in Chapter 4, $\mathsf{int}(\mathcal{F})$ is the value obtained by interpreting the binary representation of $\mathcal{F}$ as an integer. Finally, in order to continue, the last written parenthesis needs to be the opening parenthesis of bk. Hence we remove the at most k trailing closing parentheses of $\mathcal{B}$ (in constant time, using another lookup table), and then continue by processing block B_{b+1}. Figure 9.1e shows the running example after finalizing the processed block.

After block B_N has been processed, we finish the algorithm execution by appending the $2n + 2 - |\mathcal{B}|$ closing parentheses of the nodes on the path from n to 0. This can be done in $\mathcal{O}(n/\log n)$ time by appending them one word (rather than one parenthesis) at a time.

9.1.2 Analyzing the Time and Space Complexity

The initial and final processing of each block (i.e., computing $\mathcal{F}$, $\mathcal{R}$, r_b, and the pair $(b, \mathcal{L}'')$ to push on the stack) takes constant time. There are exactly N terminal iterations of the main routine, i.e., iterations where either $\mathcal{F}$ or $\mathcal{Q}$ is empty. Each terminal iteration takes constant time. In each of the non-terminal iterations, we pop a pair $(a, \mathcal{L})$ from the stack. If we do not push an updated pair $(a, \mathcal{L}')$ back onto the stack, then block B_a will never participate in the stack again, and hence this case occurs at most N times. If, however, we do push an updated pair $(a, \mathcal{L}')$ back onto the stack, then during the same main routine iteration we will also attach all remaining small trees of B_b to the partial PSS tree, which can also occur only N times. Hence the total number of iterations of the main routine is $\mathcal{O}(N)$. In each non-terminal iteration of the main routine, we call the subroutine exactly once (even though a single call may lead to multiple iterations of the subroutine). Apart from this call, each iteration of the main routine takes constant time.

It remains to be shown how to implement the subroutine such that the $\mathcal{O}(N)$ calls take $\mathcal{O}(n/\log_\sigma n)$ time in total. A straightforward naive implementation takes $\mathcal{O}(\text{poly}(k)) \subseteq \mathcal{O}(\text{polylog}(n))$ time per call. Note that the subroutine only accesses the following information: $\mathcal{L}$, r'_a, $\mathcal{R}$ (which allows access to r'_b), $\mathcal{F}$, and substrings $x_a = x(ak-k..ak+k]$ and $x_b = x(bk-k..bk]$. Bitvectors $\mathcal{L}$ and $\mathcal{R}$ are of length k bits each; r'_a is an integer from $[0, k]$ and hence can be encoded in $\lceil \log_2(k+1) \rceil \leq \lfloor 0.99k \rfloor$ bits (for sufficiently large k); sequence $\mathcal{F}$ is of length at most $2k$ bits; strings x_a and x_b in packed representation require $2k \lceil \log_2 \sigma \rceil$ and $k \lceil \log_2 \sigma \rceil$ bits respectively. This motivates a lookup table

$$M[0..2^k)[0..2^{\lfloor 0.99k \rfloor})[0..2^k)[0..2^{2k})[0..2^{2k\lceil \log_2 \sigma \rceil})[0..2^{k\lceil \log_2 \sigma \rceil}).$$

In entry $M[\mathsf{int}(\mathcal{L})][r'_a][\mathsf{int}(\mathcal{R})][\mathsf{int}(\mathcal{F})][\mathsf{int}(x_a)][\mathsf{int}(x_b)]$, we store $\mathcal{B}'$ as well as the new values of $\mathcal{R}$ and $\mathcal{F}$ after running the subroutine. Note that $\mathcal{B}'$ is of length at most $3k$ because it contains at most all the parentheses from $\mathcal{F}$ and one closing parenthesis per 1-bit in $\mathcal{L}$. Hence the information stored in each table entry fits in a constant number of words and can be retrieved in constant time. We fill the table in a lazy manner. Initially, we mark each entry as uninitialized. When accessing $M[\mathsf{int}(\mathcal{L})][r'_a][\mathsf{int}(\mathcal{R})][\mathsf{int}(\mathcal{F})][\mathsf{int}(x_a)][\mathsf{int}(x_b)]$, we check if this entry is marked. If it is, then we run the naive $\mathcal{O}(\text{polylog}(n))$ time algorithm for the subroutine, store the result in the entry, and remove its marking. Otherwise, the entry already contains the values of $\mathcal{B}'$, $\mathcal{R}$, and $\mathcal{F}$ after running the subroutine, and we return them in constant time. The lookup table has at most $2^{7.99k\lceil \log_2 \sigma \rceil} \leq 2^{\log_2 n \cdot 7.99/8} = n^{7.99/8}$ entries. Computing one entry takes $\mathcal{O}(\text{polylog}(n))$ time. Thus, the entire time spent on filling the table (i.e., on running the subroutine naively) is $\mathcal{O}(n^{7.99/8} \cdot \text{polylog}(n)) \subset \mathcal{O}(n/\log n)$.

Additional $\mathcal{O}(n/\log_\sigma n)$ preprocessing time is needed for Lemmas 9.2 and 9.3. Hence we have shown that the entire algorithm runs in $\mathcal{O}(n/\log_\sigma n)$ time.

For analyzing the space complexity, we observe that no uninitialized memory or similar techniques are used (this includes the lookup tables used in the proofs of Lemmas 9.2 and 9.3), and thus the working space measured in memory words is linear in the time spent. Without loss of generality, we can assume that words are of width $\Theta(\log n)$ bits (since $\Omega(\log n)$ bits are indeed sufficient, and, in case that the actual width is much larger than $\mathcal{O}(\log n)$ bits, we can simply simulate smaller words). Therefore, the total space is $\mathcal{O}(n \log \sigma)$ bits. The correctness of the algorithm follows from the description. It remains to be shown that Lemmas 9.2 and 9.3 hold.

9.2 Proving Lemma 9.2

A key insight for the proof of Lemma 9.2 is that the lexicographical order of suffixes starting in a small range either depends entirely on a short substring, or there is a periodic substring that can still be exploited in order to determine the lexicographical order of suffixes. This is formally expressed by the auxiliary lemma below.

Lemma 9.4. *Let $x[1..n]$ be a string over totally ordered alphabet, and let $[i, i+2\ell) \subseteq [1, n]$ be a non-empty interval of even length. Then at least one of the following properties holds:*

- $\forall a, b \in [i, i+\ell) : x[a..n] \prec x[b..n] \iff x[a..i+2\ell) \quad \prec x[b..i+2\ell)$*, or*
- $\forall a, b \in [i, i+\ell) : x[a..n] \prec x[b..n] \iff x[a..i+2\ell)\# \prec x[b..i+2\ell)\#$*, where $\#$ is an infinitely large symbol, i.e., $\forall i' \in [1, n] : x[i'] < \#$.*

Proof. Let $\tilde{n} = i + 2\ell$. Assume that neither of the properties holds, then there are indices $a_1, a_2, b_1, b_2 \in [i, i+\ell)$ such that $x[a_1..n] \prec x[b_1..n]$ but $x[a_1..\tilde{n}) \succ x[b_1..\tilde{n})$, and $x[a_2..n] \prec x[b_2..n]$ but $x[a_2..\tilde{n})\# \succ x[b_2..\tilde{n})\#$. It is easy to see that this implies

$$\begin{array}{lllllllll} x[b_1..\tilde{n}) & \prec & x[a_1..\tilde{n}) & \prec & x[a_1..n] & \prec & x[b_1..n] & = & x[b_1..\tilde{n})x[\tilde{n}..n], \text{ and} \\ x[a_2..\tilde{n})\# & \succ & x[b_2..\tilde{n})\# & \succ & x[b_2..n] & \succ & x[a_2..n] & = & x[a_2..\tilde{n})x[\tilde{n}..n]. \end{array}$$

Due to first condition and Property 6.1 *(ii)*, $x[b_1..\tilde{n})$ is a proper prefix of $x[a_1..\tilde{n})$, which implies $a_1 < b_1$. Note that $x[b_1..\tilde{n})$ is therefore also a proper suffix (and hence a border) of $x[a_1..\tilde{n})$, and thus $x[a_1..\tilde{n})$ has period $p_1 = (b_1 - a_1)$. By the same reasoning, the second condition implies that $x[a_2..\tilde{n})$ is a border of $x[b_2..\tilde{n})$. Hence $b_2 < a_2$, and $x[b_2..\tilde{n})$ has period $p_2 = (a_2 - b_2)$. By combining these observations with the initial assumption, we obtain

$$\begin{array}{lllllll} x[b_1..\tilde{n})x[\tilde{n} - p_1..n] & = & x[a_1..n] & \prec & x[b_1..n] & = & x[b_1..\tilde{n})x[\tilde{n}..n], \qquad \text{and} \\ x[a_2..\tilde{n})x[\tilde{n}..n] & = & x[a_2..n] & \prec & x[b_2..n] & = & x[a_2..\tilde{n})x[\tilde{n} - p_2..n]. \end{array}$$

The former implies $x[\tilde{n} - p_1..n] \prec x[\tilde{n}..n]$, the latter implies $x[\tilde{n}..n] \prec x[\tilde{n} - p_2..n]$. Hence

$$x[\tilde{n} - p_1..\tilde{n})x[\tilde{n}..n] \prec x[\tilde{n}..n] \prec x[\tilde{n} - p_2..\tilde{n})x[\tilde{n}..n]. \tag{9.1}$$

Since $x[\max(a_1, b_2)..\tilde{n})$ is a suffix of both $x[a_1..\tilde{n})$ and $x[b_2..\tilde{n})$, it has periods p_1 and p_2. Note that $a_1 < i+\ell-p_1$ and $b_2 < i+\ell-p_2$, and hence $x[\max(a_1, b_2)..\tilde{n})$ is of length

$\tilde{n}-\max(a_1,b_2) > \tilde{n}-i-\ell+\min(p_1,p_2) = \ell+\min(p_1,p_2) > p_1+p_2$. Therefore, it follows from the periodicity lemma [FW65] that $x[\max(a_1,b_2)..\tilde{n})$ has period $p_0 = \gcd(p_1,p_2)$. Since both $x[\tilde{n}-p_1..\tilde{n})$ and $x[\tilde{n}-p_2..\tilde{n})$ are suffixes of $x[\max(a_1,b_2)..\tilde{n})$, they also have period p_0. Let $\alpha = x[\tilde{n}-p_0..\tilde{n})$, $k_1 = p_1/p_0$ and $k_2 = p_2/p_0$. Then both k_1 and k_2 are positive integers, and it holds $x[\tilde{n}-p_1..\tilde{n}) = \alpha^{k_1}$ and $x[\tilde{n}-p_2..\tilde{n}) = \alpha^{k_2}$. Let k_0 be the largest integer (possibly 0) such that $x[\tilde{n}..n] = \alpha^{k_0}x[\tilde{n}+k_0p_0..n]$, and let $\beta = x[\tilde{n}+k_0p_0..n]$. Then α is not a prefix of β. The inequality above (Equation (9.1)) can be written as $\alpha^{k_1+k_0}\beta \prec \alpha^{k_0}\beta \prec \alpha^{k_2+k_0}\beta$. However, this is equivalent to $\alpha^{k_1}\beta \prec \beta \prec \alpha^{k_2}\beta$, which implies that α is a prefix of β. Due to this contradiction, the initial assumption must be false, and the lemma holds. □

Now we are ready to show Lemma 9.2, which is restated below.

Lemma 9.2. *Let $x[1..n]$ be packed over $[0,\sigma)$ and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, the following type of query can be answered in $\mathcal{O}(1)$ time. Given a range $[i,i+\ell) \subseteq [1,n]$ of length $\ell \le \frac{\log_2 n}{(2+\epsilon)\lceil\log_2\sigma\rceil}$, output the BPS of the PSS forest induced by $[i,i+\ell)$, as well as a bitvector $\mathcal{R}[1..\ell]$ such that for $j \in [1,\ell]$ it holds $\mathcal{R}[j] = 1$ if and only if $i+j-1$ is the root of a tree in the forest.*

Proof. The answer to any query $[i,i+\ell)$ is a parentheses sequence of length exactly 2ℓ and a bitvector of length ℓ. Hence it fits in a constant number of words. Let $\ell_{\max} = \left\lfloor\frac{\log_2 n}{(2+\epsilon)\cdot\lceil\log_2\sigma\rceil}\right\rfloor$. We precompute a two-dimensional lookup table $E[0..2\ell_{\max})[0..\ell_{\max})$ with the purpose of answering the subset of queries that satisfy $i+2\ell_{\max} > n$. For any such query $[i,i+\ell)$, it holds $n-i \in [0,2\ell_{\max})$, and we explicitly store its answer in $E[n-i][\ell-1)$. Since these queries only consider suffixes of length $\mathcal{O}(\log n)$, each of the $\mathcal{O}(\log^2 n)$ table entries can be computed naively in $\mathcal{O}(\text{polylog}(n))$ time. Using the lookup table, the corresponding queries can be answered in constant time.

We answer the remaining queries using the LCE data structure from Lemma 4.4 and additional lookup tables. For each possible value of ℓ, we construct tables $A_\ell[0..2^{2\ell\lceil\log_2\sigma\rceil})$ and $B_\ell[0..2^{2\ell\lceil\log_2\sigma\rceil})$. For every string $y[1..2\ell]$ packed over $[0,\sigma)$, we store at position $A_\ell[\mathsf{int}(y)]$ the BPS of the PSS forest of y that is induced by $[1,\ell]$, as well as the bitvector that indicates the roots. At position $B_\ell[\mathsf{int}(y)]$, we store the BPS of the PSS forest of $y\#$ that is induced by $[1,\ell]$, as well as the bitvector that indicates the roots. As before, # is an infinitely large symbol.

When answering query $[i,i+\ell)$, we first extract $x' = x[i,i+2\ell)$. Due to Lemma 9.4, the answer to the query is either $A_\ell[\mathsf{int}(x')]$ or $B_\ell[\mathsf{int}(x')]$. A table $C_\ell[0..2^{2\ell\lceil\log_2\sigma\rceil})$ is used to decide which answer is correct. For every string $y[1..2\ell]$ packed over $[0,\sigma)$, we store at position $C_\ell[\mathsf{int}(y)]$ an integer pair $(a,b) \in [1,\ell]^2$ such that $y[a..2\ell] \prec y[b..2\ell]$ and $y[a..2\ell]\# \succ y[b..2\ell]\#$ (or $a = b = 1$ if such a pair does not exist). This is a witness pair of suffixes for which y and $y\#$ disagree on the lexicographical order. At query time, we look up $(\hat{a},\hat{b}) = C_\ell[\mathsf{int}(x')]$. If $x[i+\hat{a}-1..n] \prec x[i+\hat{b}-1..n]$, then we return $A_\ell[\mathsf{int}(x')]$, and otherwise we return $B_\ell[\mathsf{int}(x')]$. The correctness follows from Lemma 9.4. Testing $x[i+\hat{a}-1..n] \prec x[i+\hat{b}-1..n]$ takes constant time with the LCE data structure from Lemma 4.4. Extracting x' and performing table lookups also takes constant time because x' fits in a single word of memory.

A single lookup table entry can be computed naively in $\mathcal{O}(\text{polylog}(n))$ time. There are $\mathcal{O}(\log n)$ tables, each storing at most $2^{2\ell_{\max}\lceil\log_2\sigma\rceil} \le 2^{\log_2 n/(1+\epsilon/2)} = \sqrt[1+\epsilon/2]{n}$ entries. Thus, the precomputation of lookup tables takes $\mathcal{O}(\sqrt[1+\epsilon/2]{n}\cdot\text{polylog}(n))$ time, which is dominated by the $\mathcal{O}(n/\log_\sigma n)$ time needed to construct the LCE data structure. $\square$

9.3 Proving Lemma 9.3

The proof of Lemma 9.3 relies on the properties of periodic substrings that are stated below.

Proposition 9.5. *Let α, β, and γ be arbitrary strings. The following properties hold.*

(1) *If $\alpha\beta \succ \beta$ and $\alpha\gamma \prec \gamma$ then $\beta \prec \gamma$.*

(2) *If $\alpha\gamma \prec \gamma$ and α is not a prefix of γ, then $\forall g, h \in \mathbb{N}^0 : g > h \implies \alpha^g\beta \prec \alpha^h\gamma$.*

Proof. We start with (1). Let $k \in \mathbb{N}^0$ be the maximal value such that both $\beta = \alpha^k\beta'$ and $\gamma = \alpha^k\gamma'$ for some (possibly empty) strings β' and γ'. If $\alpha^{k+1}\beta' \succ \alpha^k\beta'$ and $\alpha^{k+1}\gamma' \prec \alpha^k\gamma'$, then $\beta' \prec \alpha\beta'$ and $\alpha\gamma' \prec \gamma'$. Now assume that $\gamma' \preceq \beta'$, then $\alpha\gamma' \prec \gamma' \preceq \beta' \prec \alpha\beta'$. However, this implies that α is a prefix of both β and γ, which contradicts the definition of k. Thus $\beta' \prec \gamma'$, which also implies $\beta = \alpha^k\beta' \prec \alpha^k\gamma' = \gamma$. For (2), consider any $g, h \in \mathbb{N}^0$ with $g > h$, and assume that $\alpha\gamma \prec \gamma$. Since α is not a prefix of γ, it follows from $\alpha\gamma \prec \gamma$ that $\alpha\delta \prec \gamma$ for every string δ. Hence also $\alpha^{g-h}\beta \prec \gamma$, which implies $\alpha^g\beta = \alpha^h\alpha^{g-h}\beta \prec \alpha^h\gamma$. $\square$

Now we are ready to show Lemma 9.3, which is restated below.

Lemma 9.3. *Let $x[1..n]$ be packed over $[0, \sigma)$ and let $\epsilon \in \mathbb{R}^+$. After $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, we can answer the following type of query in $\mathcal{O}(1)$ time. Given a position $i \in [1, n]$ and a non-empty interval $[j, j+\ell) \subseteq [1, n]$ of length $\ell \leq \frac{\log_2 n}{(5+\epsilon)\cdot\lceil\log_2\sigma\rceil}$, find the position $j_{\max} = \max(\{j' \in [j, j+\ell) \mid x[j'..n] \prec x[i..n]\} \cup \{j-1\})$.*

Proof. Without loss of generality, we assume $j + 3\ell < n$ and $i + 2\ell < n$ (otherwise, we can simply add simulated padding $0^{3\ell}$ to the end of the string, which does not affect the query result or access time). We focus on the set

$$C = \{j' \in [j, j+\ell) \mid x[j'..j'+2\ell) = x[i..i+2\ell)\} = \{c_1, c_2, \ldots, c_h\}$$

with $c_1 < c_2 < \cdots < c_h$. This set contains exactly the positions $j' \in [j, j+\ell)$ for which we cannot easily determine whether $x[j'..n] \prec x[i..n]$ by inspecting only a small number of symbols. Hence it captures the difficult part of answering a query, and we treat it separately from the rest. We answer the query using the following subsets of $[j, j+\ell)$:

- $D' = \{j' \in C \mid x[j'..n] \prec x[i..n]\}$ (the hard subset), and
- $D'' = \{j' \in [j, j+\ell) \setminus C \mid x[j'..n] \prec x[i..n]\}$ (the easy subset).

The result of the query is $j_{\max} = \max(j'_{\max}, j''_{\max})$, where $j'_{\max} = \max(D' \cup \{j-1\})$ and $j''_{\max} = \max(D'' \cup \{j-1\})$. We start with the significantly harder task of computing $j'_{\max}$. First, we outline the algorithmic approach and the combinatorial properties of the present substrings (without giving details of an efficient implementation). Later, we describe lookup tables that achieve the claimed preprocessing and query times.

Periodicity of $x[i..i+2\ell)$ and $x[c_1..c_h+2\ell)$ We show that, if $|C| \geq 2$, then there is some p such that $x[c_1..c_h+2\ell)$ has period p, and $\forall k \in [1,h) : c_{k+1} - c_k = p$. This is similar, e.g., to [MST97, Lemma 1] and [Kid+03, Lemma 2]. Assume that $|C| \geq 2$. For $k \in [1,h)$, let $p_k = c_{k+1} - c_k < \ell$. By design of C, it holds $x[c_k..c_k+2\ell-p_k) = x[c_{k+1}..c_{k+1}+2\ell-p_k) = x[c_k+p_k..c_k+2\ell)$. This means that $x[i..i+2\ell) = x[c_k..c_k+2\ell)$ has a border of length $2\ell - p_k$, and therefore it has period p_k. Let p be the minimal period of $x[i..i+2\ell)$. If there was some $k \in [1,h)$ such that $p_k < p$, then p would not be the minimal period of $x[i..i+2\ell)$. Hence $p_k \geq p$. Now we show that $\forall k \in [1,h) : p_k = p$. For the sake of contradiction, assume $p_k > p$. By definition of C, it holds $p_k < \ell$, which means that $x[c_k..c_k+2\ell)$ and $x[c_{k+1}..c_{k+1}+2\ell)$ overlap by $c_k + 2\ell - c_{k+1} = 2\ell - p_k > \ell > p$ symbols. Due to this overlap, and because the identical substrings $x[c_k..c_k+2\ell)$ and $x[c_{k+1}..c_{k+1}+2\ell)$ have period p, it is clear that also their union $x[c_k..c_{k+1}+2\ell)$ has period p. However, this implies $x[c_k..c_k+2\ell) = x[c_k+p..c_k+p+2\ell)$, which means that $c_k + p$ should be in C. Due to this contradiction, it holds $p_k = p$. It also follows that $x[c_1..c_h+2\ell)$ has period p.

Computing $j'_{\max}$ from c_1, c_h, and p We will later introduce lookup tables that output c_1, c_h, and p for any query in constant time. The tables might return that c_1 and c_h do not exist (i.e., $|C| = 0$), in which case we report $j'_{\max} = j - 1$. Otherwise, it might be that $c_1 = c_h$ (i.e., $|C| = 1$). In this case, we report that $j'_{\max} = c_1$ if $x[i..n] \succ x[c_1..n]$ (using an LCE query for the comparison). Otherwise, we report $j'_{\max} = j - 1$. It remains to be shown how to compute $j'_{\max}$ if $c_1 \neq c_h$ (i.e., if $|C| \geq 2$, and the previously described periodicity exists).

We evaluate $x[i..n] \prec x[i+p..n]$ and $x[c_h..n] \prec [c_h+p..n]$ (using LCE queries). Due to the periodicity of $x[c_1..c_h+2\ell)$, for $k \in [1,h)$ it holds $x[c_h..n] \prec x[c_h+p..n]$ if and only if

$$x[c_k..n] \quad = \quad x[i..i+p)^{h-k}x[c_h..n] \quad \prec \quad x[i..i+p)^{h-k}x[c_h+p..n] \quad = \quad x[c_{k+1}..n].$$

Hence either $x[c_1..n] \prec x[c_2..n] \prec \ldots \prec x[c_h..n]$ or $x[c_1..n] \succ x[c_2..n] \succ \ldots \succ x[c_h..n]$, and we already know which of the two applies. Depending on the outcome of the lexicographical comparisons, we report $j'_{\max}$ according to one of the following three cases.

Case 1: $x[c_1..n] \succ x[c_2..n] \succ \ldots \succ x[c_h..n]$.

For the computation of $j'_{\max}$, we are only interested in the rightmost $k \in [1,h]$ such that $x[i..n] \succ x[c_k..n]$. Since $x[c_h..n]$ is both rightmost and lexicographically minimal among all the possible $x[c_k..n]$, we simply use another LCE query to check if $x[i..n] \succ x[c_h..n]$. If yes, then we report $j'_{\max} = c_h$. Otherwise, we report $j'_{\max} = j - 1$.

Case 2: $x[i..n] \succ x[i+p..n]$ and $x[c_1..n] \prec x[c_2..n] \prec \ldots \prec x[c_h..n]$.

Let $\alpha = x[i..i+p)$, $\beta = x[i+p..n]$, and $\gamma = x[c_2..n]$. The precondition of this case means that $\alpha\beta \succ \beta$ and $\alpha\gamma \prec \gamma$. Proposition 9.5 (1) implies $\beta \prec \gamma$, and thus also $x[i..n] = \alpha\beta \prec \alpha\gamma = x[c_1..n] \prec x[c_2..n] \prec \ldots \prec x[c_h..n]$. Hence we report $j'_{\max} = j - 1$.

Case 3: $x[i..n] \prec x[i+p..n]$ and $x[c_1..n] \prec x[c_2..n] \prec \ldots \prec x[c_h..n]$.

Let $\alpha = x[i..i+p)$. We start by computing $r = \lfloor \textsc{lce}(i, i+p)/p \rfloor + 1$ and $s = \lfloor \textsc{lce}(c_1, c_1+p)/p \rfloor + 1$, i.e., the respectively maximal integer powers with $x[i..n] = \alpha^r\beta$ and $x[c_1..n] = \alpha^s\gamma$, where $\beta = x[i+rp..n]$ and $\gamma = x[c_1+sp..n]$. The precondition of this case means that $\alpha^s\gamma \prec \alpha^{s-1}\gamma$, and thus also $\alpha\gamma \prec \alpha$. Note that α is not a prefix of γ. Hence Proposition 9.5 (2) implies that $\alpha^r\beta \prec \alpha^{s'}\gamma$ for any $s' < r$. Thus, for $k \in [1,h]$, if $s-k+1 < r$ then $x[i..n] = \alpha^r\beta \prec \alpha^{s-k+1}\gamma = x[c_k..n]$. Hence we only have to consider $k \leq s-r+1$. On the other hand, the precondition of the case also implies $\alpha^r\beta \prec \alpha^{r-1}\beta$, and thus $\alpha\beta \prec \alpha$. Also, α is not a prefix of β. Hence Proposition 9.5 (2) (with swapped roles of β and γ) implies that $\alpha^r\beta \succ \alpha^{s'}\gamma$ for any $s' > r$. For $k \in [1,h]$, if $k \leq s-r$ then $x[i..n] = \alpha^r\beta \succ \alpha^{s-k+1}\gamma = x[c_k..n]$.

This motivates the following strategy. If $s-r+1 < 1$, then there is no suitable choice of k and we report $j'_{\max} = j-1$. If $h \leq s-r$, then $x[i..n] \succ x[c_h..n]$ and we report $j'_{\max} = c_h$. We are left with the case where $s-r+1 \in [1,h]$. If $x[i..n] \succ x[c_{s-r+1}..n]$, then we report $j'_{\max} = c_{s-r+1}$ (we use another LCE query to achieve constant time). If we still have not reported anything, then we report $j'_{\max} = c_{s-r}$ if and only if $s-r \in [1,h]$ (we have already established that $x[i..n] \succ x[c_{s-r}..n]$). If, however, $s-r \notin [1,h]$, then we report $j'_{\max} = j-1$.

The three cases are exhaustive, and it takes constant time to determine which case applies. Regardless of the case, we report $j'_{\max}$ in constant time. We require the LCE data structure from Lemma 4.4, and hence the preprocessing time is $\mathcal{O}(n/\log_\sigma n)$.

Lookup Tables for c_1, c_h, p, and $j''_{\max}$ As described above, we can compute $j'_{\max}$ in constant time if we can determine c_1, c_h, and p in constant time. Note that these values depend solely on the substrings $x[i..i+2\ell)$ and $x[j..j+3\ell)$. This also holds for $j''_{\max}$, which can be written as

$$j''_{\max} = \max(\{j' \in [j, j+\ell) \mid x[j'..j'+2\ell) \prec x[i..i+2\ell)\} \cup \{j-1\}).$$

For each possible value of ℓ, we compute a lookup table $L_\ell[0..2^{2\ell\lceil\log_2\sigma\rceil})[0..2^{3\ell\lceil\log_2\sigma\rceil})$. Let $y_1[1..2\ell]$ and $y_2[1..3\ell]$ be packed over $[0,\sigma)$. In entry $L_\ell[\mathsf{int}(y_1)][\mathsf{int}(y_2)]$, we store the quadruple $\langle\hat{p}, \hat{c}_{\min}, \hat{c}_{\max}, \hat{g}_{\max}\rangle$, where

- $\hat{p}$ is the minimal period of y_1,
- $\hat{c}_{\min} = \min(\{k' \in [1,\ell] \mid y_2[k'..k'+2\ell) = y_1\} \cup \{\infty\})$,
- $\hat{c}_{\max} = \max(\{k' \in [1,\ell] \mid y_2[k'..k'+2\ell) = y_1\} \cup \{-\infty\})$,
- $\hat{g}_{\max} = \max(\{k' \in [1,\ell] \mid y_2[k'..k'+2\ell) \prec y_1\} \cup \{-\infty\})$.

A single entry $L_\ell[\mathsf{int}(y_1)][\mathsf{int}(y_2)]$ can be computed naively in $\mathcal{O}(\mathrm{poly}(\ell)) \subseteq \mathcal{O}(\mathrm{polylog}(n))$ time. Table L_ℓ has $2^{5\ell\lceil\log_2\sigma\rceil} \leq 2^{\log_2 n/(1+\epsilon/5)} = \sqrt[1+\epsilon/5]{n}$ entries, and there are $\mathcal{O}(\log n)$ tables. Thus, the entire preprocessing time is $\mathcal{O}(\sqrt[1+\epsilon/5]{n} \cdot \mathrm{polylog}(n)) \subset \mathcal{O}(n/\log_\sigma n)$. Whenever we have to answer a query i, $[j, j+\ell)$, we extract $x' = x[i..i+2\ell)$ and $x'' = x[j..j+3\ell)$ and look up $\langle p, c_{\min}, c_{\max}, g_{\max}\rangle = L_\ell[\mathsf{int}(x')][\mathsf{int}(x'')]$. This takes constant time because x' and x'' fit in a single word of memory. From the construction of L_ℓ, it is clear that

- p is the minimal period of $x[i..i + 2\ell)$.
- If $c_{\min} \neq \infty$ then $c_1 = j + c_{\min} - 1$. Otherwise, c_0 does not exist.
- If $c_{\max} \neq -\infty$ then $c_h = j + c_{\max} - 1$. Otherwise, c_h does not exist.
- If $g_{\max} \neq -\infty$ then $j''_{\max} = j + g_{\max} - 1$. Otherwise, $j''_{\max} = j - 1$.

Hence we can compute $j'_{\max}$ in constant time as described above, and output the query result $j_{\max} = \max(j'_{\max}, j''_{\max})$ in constant time. □

9.4 Conclusion

We presented an algorithm that computes the succinct Lyndon array in $\mathcal{O}(n/\log_\sigma n)$ time, which is optimal on a word RAM of width $w = \mathcal{O}(\log n)$. The working space is $\mathcal{O}(n \log \sigma)$ bits, dominated by the LCE data structure and the stack $\mathcal{Q}$. The working space may possibly be improved by adapting the techniques from Chapter 8. However, this seems challenging due to the technical nature of Chapters 8 and 9. We envision that the new algorithm will lead to the first sublinear time algorithm that computes all runs, which can be done in linear time by using the Lyndon array (which we discuss in Chapter 10). This will require overcoming multiple smaller hurdles. Most notably, we will need support data structures that allow fast operations on the BPS after a sublinear time preprocessing. Also, since a string may contain $\Omega(n)$ runs, we need a non-trivial encoding of runs in order to report them in sublinear time.

III

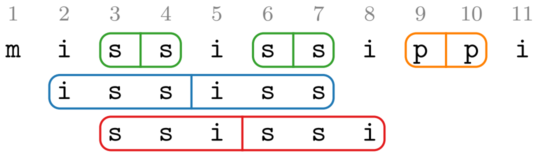
Computing Maximal Periodic Substrings

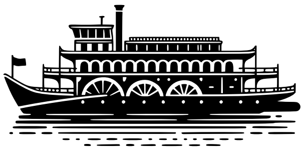
Computing Maximal Periodic Substrings

III

Introduction and Related Work

The notion of repetition is a central concept in combinatorics on words and algorithms on strings. The simplest type of repetition is a *square*, which is a string of the form $y \cdot y$ (for some non-empty string y). A fundamental algorithmic task is to detect square substrings in a longer string $x[1..n]$, i.e., substrings of the form $x[i..i+2\ell) = x[i..i+\ell)^2$. For example, the string `mississippi` contains the squares `pp`, `ss` (twice), `ississ`, and `ssissi`, which is visualized below.




[1]

The number of squares in a length-n string can be $\Theta(n^2)$. For example, every even-length substring of $x = x[1]^n$ is a square. Hence, if we want to efficiently report all square substrings, we cannot explicitly output each square separately. However, one might argue that a square $y \cdot y$ is of less interest if y itself is already non-primitive, i.e., if there is a string w and integer $k \geq 2$ such that $y = w^k$. This is because a square $y \cdot y = w^k \cdot w^k$ is covered by $2k - 1$ overlapping occurrences of the square w^2, and thus we indirectly report occurrences of $y \cdot y$ by reporting occurrences of $w \cdot w$. Hence we are only interested in *primitively rooted* squares, i.e., squares $y \cdot y$ where y is primitive.

A length-n string contains at most $\mathcal{O}(n \log n)$ primitively rooted squares. This bound is tight, as some strings, e.g., Fibonacci strings, contain $\Theta(n \log n)$ primitively rooted squares [Cro81]. Thus, if we explicitly output each primitively rooted square separately, then we still cannot achieve linear time. A more efficient approach is to report groups of squares that form a periodic substring. A run (also called maximal periodic substring) is a substring $x[i..j]$ of minimal period $p \leq \frac{j-i+1}{2}$ such that neither $x[i-1..j]$ nor $x[i..j+1]$ has period p (respectively if $i > 0$ and $j < n$). We represent the run as a triple $\langle i, j, p \rangle$. For example, the substring `ississi` of `mississippi` is a run $\langle 2, 8, 3 \rangle$. In a run of period p, every length $2p$ substring is a primitively rooted square. Conversely, every primitively rooted square of length $2p$ is contained in exactly one run of period p, and thus the runs in a string fully capture the structure of squares. Conveniently, there are less than n runs in any length-n string [Ban+17], which means that we can hope to compute and report the runs in linear time.

[1] Classic Mississippi steamboat, envisioned by DALL·E [Bet+23].

Background and Related Work

Bioinformatics Runs in DNA sequences are called *tandem repeats* and have a profound biological meaning. For example, the human HTT gene contains the CAG triplet repeat, i.e., a substring of the form $(\mathtt{CAG})^k$. The exponent k is inversely correlated with the probability of developing Huntington's disease. The higher k is, the earlier the onset of the disease (see, e.g., [BM13]). There are numerous other pathogenic conditions that are caused by abnormally long tandem repeats in the DNA, e.g., myotonic dystrophy, fragile X syndrome, spinocerebellar ataxias, and Friedreich's ataxia (see, e.g., [Cie+17]).

Another notable application of runs is in forensic DNA analysis, where short tandem repeat typing is one of the primary methods (see, e.g., [Tre12, Udo+20]). In the human genome, many tandem repeats and their genetic locations are well-known and understood. Forensic laboratories frequently use the selective amplification of these DNA locations, which allows them to determine the lengths of the runs. The lengths exhibit significant variability among individuals. Consequently, even a few genetic locations are sufficient to generate a unique fingerprint that identifies an individual. Such DNA fingerprints streamline the process of comparing a DNA sample from a crime scene with that of a subject. The FBI originally used 13 different runs as a core requirement for their DNA database, with seven new runs added in 2017 [Webc]. It is debated whether the runs used for forensic analysis can reveal sensitive information about an individual that goes beyond mere identity [WBM20].

Due to their significance in DNA, the computation of runs has been a topic of practical research. For example, there is an efficient practical implementation for computing all runs (either exactly or allowing some error) [KBK03].

Combinatorics on Words We will study runs primarily from a theoretical perspective. The study of squares in strings goes back to the work of Thue published in 1906 [Thu06, Ber94], who considered the question of constructing an infinite string with no squares. It is easy to see that any sufficiently long binary string must contain a square, and Thue proved that there exists an infinite ternary string with no squares. His result has been rediscovered multiple times, and in 1979 Bean, Ehrenfeucht and McNulty [BEM79] started a systematic study of the so-called avoidable repetitions, see for example the survey by Currie [Cur05].

A basic tool in the area of combinatorics on words is the so-called periodicity lemma, which states that, if p and q are distinct periods of $x[1..n]$ and $p + q - \gcd(p, q) \leq n$, then $\gcd(p, q)$ is also a period [FW65] (which we already used in the proof of Lemma 9.3). This was generalized in a myriad of ways, for example for strings [CMR99, Jus00, TZ03], partial words (strings with “don't cares”) [BB99, SK01, BH02, SG04, BBS08, IS14, Koc+22], Abelian periods[CI06, Bla+13], parametrized periods [AG08], order-preserving periods [Mat+16, Gou+20], and approximate periods [AEL10, AL12, AEL15].

As mentioned earlier, a length-n string may contain $\Omega(n^2)$ squares. While considering runs rather than squares is one way of avoiding this lower bound, from the combinatorial point of view it would also be natural to count only distinct squares. Fraenkel and Simpson [FS98] showed an upper bound of $2n$ and a lower bound of $n - \Theta(\sqrt{n})$ on the maximum number of distinct squares in a length-n string. After a sequence of improvements [Ili07, DFT15, Thi20], the upper bound was recently improved to n [BL22, BL23]. The last result was already generalized to higher powers [LPR22]. However, counting or reporting distinct squares inherently requires some form of sorting, which means that we cannot accomplish it in linear time over general alphabets.

Computing all runs does not require sorting and, as discussed earlier, it fully captures the structure of all square substrings. Kolpakov and Kucherov [KK99] showed an upper bound of $\mathcal{O}(n)$ on the number of runs in a length-n string, which started a long line of work on determining the exact constant [Ryt06, CI08b, Gir08, PSS08, Gir09, CIT11], culminating in the celebrated paper by Bannai et al. [Ban+17] showing an upper bound of n, and followed by even better upper bounds for binary strings [Fis+15b, Hol17]. This was complemented by a sequence of lower bounds [FY08, Mat+08, Mat+09, Sim10]. Apart from showing the $\mathcal{O}(n)$ upper bound, Kolpakov and Kucherov [KK99] also provided an algorithm that computes all runs over linearly-sortable alphabet in $\mathcal{O}(n)$ time.

Computing Repetitions Over General Alphabets In this part of the dissertation, we are interested in the algorithmic aspects of detecting repetitions in strings, particularly over general ordered and unordered alphabet. The most basic problem is checking if a given string contains at least one square, while the most general problem is computing all the runs. Testing square-freeness was considered by Main and Lorentz [ML84] around 40 years ago, who designed an $\mathcal{O}(n \log n)$ time algorithm based on a divide-and-conquer approach and a linear-time procedure for finding all new squares obtained when concatenating two strings. In fact, their algorithm can be used to find (a compact representation of) all squares in a given string within the same time complexity. Their algorithm is designed for general unordered alphabet, i.e., it performs only equality comparisons of symbols. They also proved that any algorithm based on such comparisons needs $\Omega(n \log n)$ operations to test square-freeness in the worst case. However, to obtain the lower bound they had to consider instances consisting of up to n distinct characters, that is, over alphabet of size n. This is somewhat unsatisfactory; indeed, Main and Lorentz [ML84] explicitly asked whether there is a faster algorithm that tests square-freeness over general unordered alphabet if the size of the alphabet is restricted. In Chapter 11, we positively answer this question with an $\mathcal{O}(n \log \sigma)$ time algorithm, where σ is the number of distinct symbols in the string.

Another $\mathcal{O}(n \log n)$ time algorithm for finding all repetitions was provided by Crochemore [Cro81], who also showed that testing square-freeness can be done in $\mathcal{O}(n)$ time for constant-size alphabet [Cro86]. In fact, the latter algorithm works in $\mathcal{O}(n \log \sigma)$ time for general ordered alphabet of size σ, i.e., it requires order comparisons of symbols. Later, Kosaraju [Kos94] showed that, assuming constant-size alphabet, $\mathcal{O}(n)$ time is enough to find the shortest square starting at each position of the input string. Apostolico and Preparata [AP83] provide another $\mathcal{O}(n \log n)$ time algorithm assuming a general ordered alphabet, based more on data structure considerations than combinatorial properties of strings. Finally, a number of alternative $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log \sigma)$ time algorithms (respectively for general unordered and general ordered alphabet) can be obtained from the work on online [HC08, Kos14, Kos15c] and parallel [Apo92, AB96] square detection (interestingly, this cannot be done efficiently in the related streaming model [MS19, MS22]). Breslauer explicitly asked whether testing square-freeness in linear time is possible over general ordered alphabet [Bre92, Section 4.4] around 30 years ago.

The first steps towards answering Breslauer's question were made as a byproduct of the more general results on finding all runs (since a string is square-free if and only if it is run-free). For general ordered alphabet, Kosolobov [Kos15b] showed that the decision tree complexity of this problem is indeed only $\mathcal{O}(n)$, and later complemented this with an efficient $\mathcal{O}(n(\log n)^{2/3})$ time algorithm [Kos16a] (still using only $\mathcal{O}(n)$ comparisons, i.e., the time is dominated by other word RAM operations). The algorithm is based on an earlier solution by Bannai et el. [Ban+17] that reduces the computation of all the runs to answering an online sequence of $\mathcal{O}(n)$ LCE queries. Kosolobov conjectured that $\mathcal{O}(n)$ time could be achieved, which inspired a line of work aimed at accelerating the LCEs. The time complexity for computing all runs was first improved to $\mathcal{O}(n \log \log n)$ by providing a general mechanism for LCEs over general ordered alphabet [Gaw+16], and then to $\mathcal{O}(n\alpha(n))$ by observing that the LCE queries have additional structure [Cro+16]. In Chapter 10, we show how to compute the LCEs in $\mathcal{O}(n)$ time by exploiting the rich structure of Lyndon words in the string.

Contributions

We fully resolve the complexity of computing runs (and thus detecting squares) over general alphabets. For general ordered alphabet, we provide an algorithm that computes all runs in $\mathcal{O}(n)$ time and words of space (Chapter 10). This confirms Kosolobov's conjecture [Kos16a] and positively answers the long-standing question whether testing square-freeness over general ordered alphabet is possible in linear time (see [Bre92, Section 4.4]). The algorithm is much simpler than previous near-linear time solutions, and we complement the theoretical result with a fast practical implementation.

For general unordered alphabet, we provide an algorithm that computes all runs in $\mathcal{O}(n \log \sigma)$ time, and a matching $\Omega(n \log \sigma)$ time lower bound for testing square-freeness (Chapter 11). The algorithm positively answers the long-standing question whether square-freeness can be tested in less than $\Omega(n \log n)$ time for general unordered alphabet of restricted size [ML84].

Chapter 10

Computing Runs Over General Ordered Alphabet

10

In this chapter, we show how to compute all runs in a length-n string over general ordered alphabet in $\mathcal{O}(n)$ time and words of space. The algorithm is based on the solution by Bannai et al. [Ban+17], which computes the runs using the Lyndon array and a data structure for constant time LCE queries. We can compute the Lyndon array in linear time over general ordered alphabet (see Chapters 7 and 8 or [Bil+20, Ell22]), but such a result is not known for an LCE data structure. Instead, we exploit combinatorial properties of the Lyndon array that allow us to directly compute the required LCEs in overall linear time. A string is square-free if and only it is run-free. Therefore, the new algorithm positively answers the question whether or not testing square-freeness is possible in linear time over general ordered alphabet.

Theorem 10.1. *All the runs contained in a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of working space.*

Corollary 10.2. *Testing whether or not a length-n string over general ordered alphabet contains a square can be done in $\mathcal{O}(n)$ time and words of space.*

The chapter is structured as follows. First, we introduce the algorithmic tools and combinatorial properties that we use throughout the chapter (Section 10.1). Then, we give a simplified description of the runs algorithm by Bannai et al. (Section 10.2). In Section 10.3, we show how to compute the LCEs required by the runs algorithm in overall linear time, which directly yields the linear time bound for the entire algorithm. Even though the main contribution is the improved asymptotic time bound, it is worth mentioning that the algorithm is also very fast in practice. On commodity hardware, computing all runs for a text of length $10^7 (= 10\text{MB})$ takes only one second. We discuss additional practical aspects and experimental results in Section 10.4.

10.1 Algorithmic Toolbox

In this section, we introduce the main algorithmic tools and properties that we use for computing runs.

10.1.1 Lyndon Array and Nearest Smaller Suffixes

We use the Lyndon array and the next smaller suffix array from Chapter 6 as the main algorithmic tool for computing runs. For the sake of convenience, their definitions are repeated below. Examples can be found in Chapter 6, Figure 6.1.

Definition 6.3 (Lyndon Array)**.**
The *Lyndon array* $\lambda_x[1..n]$ of a string $x[1..n]$ is defined by

$$\forall i \in [1, n] : \lambda_x[i] = \max\{m \in [1, n - i + 1] \mid x[i..i + m) \text{ is a Lyndon word}\}.$$

We omit the subscript x whenever it is clear from context.

Definition 6.5 (Nearest Smaller Suffix Arrays)**.** Let $x[1..n]$ be a string.

(PSS) The *previous smaller suffix (PSS) array* $\mathsf{prev}_x[1..n]$ of x is defined by

$$\forall j \in [1, n] : \mathsf{prev}_x[j] = \max\left(\{i \in [1, j) \mid x_i \prec x_j\} \cup \{0\}\right).$$

(NSS) The *next smaller suffix (NSS) array* $\mathsf{next}_x[1..n]$ of x is defined by

$$\forall i \in [1, n] : \mathsf{next}_x[i] = \min\left(\{j \in (i, n] \mid x_i \succ x_j\} \cup \{n + 1\}\right).$$

We omit the subscript x whenever it is clear from context.

Lemma 6.7 (see also [HR03, Fra+16, FL20])**.**
For any string $x[1..n]$, it holds $\forall i \in [1, n] : \mathsf{next}_x[i] = i + \lambda_x[i]$.

A fundamental property of next smaller suffixes is that they do not intersect. We have previously shown this property in Lemma 7.7, and we will use it for the computation of runs in the slightly different form stated below.

Corollary 10.3. *Let $x[1..n]$ be a string, let $i \in [1, n]$, and $i' \in [i, \mathsf{next}_x[i])$. Then it holds $\mathsf{next}_x[i'] \leq \mathsf{next}_x[i]$.*

10.1.2 Relation Between Runs and Lyndon Words

We start by showing that every run of minimal period p contains a Lyndon word of length p. This is a well-known property, and we only add the proof for completeness. Recall that a run in a string $x[1..n]$ is a triple $\langle i, j, p\rangle$ such that $x[i..j]$ has minimal period $p \leq \frac{j-i+1}{2}$, and neither $x[i - 1..j]$ nor $x[i..j + 1]$ has period p (respectively if $i > 0$ and $j < n$). In a slight abuse of terminology, we may then refer to both $\langle i, j, p\rangle$ and $x[i..j]$ as a run. We will need the notion of primitive strings and cyclic shifts. A string w is primitive if there is no string u and integer $k \geq 2$ such that $w = u^k$. A string w is a cyclic shift of another string w' if there are strings u and v (one of which is possibly empty) such that $w = uv$ and $w' = vu$. We say that w and w' are cyclically equivalent if w is a cyclic shift of w'. This is indeed an equivalence relation; every string is a cyclic shift of itself, the definition of a cyclic shift is inherently symmetric, and the transitivity is easy to show.

Lemma 10.4. *If $\langle i, j, p\rangle$ is a run in a string $x[1..n]$, then every length-p substring of $x[i..j]$ is primitive.*

Proof. Let u be a length-p substring of $x[i..j]$, then $x[i..j]$ is a substring of u^h with $h = \lfloor (j - i + 1)/p\rfloor + 2$ (this readily follows from the fact that $x[i..j]$ has period p). If u is not primitive, then $u = v^k$ for some string v and integer $k \geq 2$. This means that $x[i..j]$ is a substring of $u^h = v^{hk}$. However, then it is easy to see that $x[i..j]$ has period $|v| < p$, which contradicts the fact that p is the minimal period of $x[i..j]$. □

Lemma 10.5. *For a run $\langle i', j', p\rangle$ in a string x, there is exactly one position $i_0 \in [i', i' + p)$ such that $x[i_0..i_0 + p)$ is a Lyndon word.*

Proof. Let $u = x[i..i + p)$ and $y = x[i..i + 2p - 1) = uu[1..p)$. It is easy to see that the length-p substrings of y are exactly the p cyclic shifts of u.

Now we show that all of the cyclic shifts are distinct, i.e., no length-p substring of y has more than one occurrence in y. We will show that, if a length-p substring has two occurrences, then it cannot be primitive. Assume that there are $i, j \in [1, p]$ with $i < j$ and $y[i..i + p) = y[j..j + p)$. Then $y[i..i + p)$ is a length-p border of $y[i..j + p)$, and thus $y[i..j + p)$ has period $j - i$. Since $y[i..j + p)$ is a substring of the run, it also has period p. Hence it has periods p and $j - i$, and by the periodicity lemma [FW65] also period $p' = \gcd(p, j - i) < p$. Trivially, if p' is a period of $y[i..j + p)$, then it is also a period of $y[i..i+p)$. Since p' divides p, it is clear that $y[i..i+p) = y[i..i+p')^{p/p'}$, where p/p' is an integer larger than one. Hence $y[i..i + p)$ is a non-primitive substring of $x[i'..j']$, but this contradicts Lemma 10.4.

We have shown that the length-p substrings of y are exactly the p cyclic shifts of u, and all the cyclic shifts are distinct. A Lyndon word is lexicographically smaller than all of its non-trivial cyclic shifts, see Lemma 6.2 (*c*). Hence the unique lexicographically minimal cyclic shift of u is the only length-p Lyndon substring of y. □

Finally, we can make a stronger statement about Lyndon words in runs if we distinguish between lexicographically increasing and lexicographically decreasing runs (like previously done in Chapter 8).

Definition 10.6. Let $\langle i, j, p\rangle$ be a run in some string x. We say that $\langle i, j, p\rangle$ is *(lexicographically) decreasing* if and only if $x_i \succ x_{i+p}$. Otherwise, $\langle i, j, p\rangle$ is *(lexicographically) increasing.*

Lemma 10.7. *Let $\langle i, j, p\rangle$ be a decreasing run, then there is exactly one index $i_0 \in [i, i + p)$ such that $\lambda[i_0] = p$.*

Proof. Let $i_0 \in [i, i + p)$ be the unique position such that $x[i_0..i_0 + p)$ is a Lyndon word, where the existence and uniqueness are due to Lemma 10.5. It is clear that $\lambda[i_0] \geq p$, and it remains to be shown that $\lambda[i_0] \leq p$. Since $x[i..j]$ has period p, it holds $x[i..i_0) = x[i+p..i_0+p)$. The run is decreasing, which implies $x[i..i_0)x_{i_0} = x_i \succ x_{i+p} = x[i + p..i_0 + p)x_{i_0+p}$ and consequently $x_{i_0} \succ x_{i_0+p}$. Hence $\mathsf{next}[i_0] \leq i_0 + p$ and by Lemma 6.7 also $\lambda[i_0] \leq p$. □

10.1.3 Longest Common Extensions

As mentioned before, a key component of the runs algorithm is the computation of LCEs. In the previous chapters, we used LCEs that extend from left to right, i.e., the LCE between i and j indicates the length of the longest shared prefix between $x[i..n]$ and $x[j..n]$. For the computation of runs, we also need LCEs in the opposite direction, i.e., the length of the longest shared suffix between $x[1..i]$ and $x[1..j]$. Formally, we define R-LCEs and L-LCEs for a string $x[1..n]$ and positions $i, j \in [1, n]$ as follows.

$$\begin{aligned} \textsc{lce}_r(i,j) &= \max(\{m \in [0, n - \max(i,j) + 1] \mid x[i..i+m) = x[j..j+m)\}) \\ \textsc{lce}_\ell(i,j) &= \max(\{m \in [0, \min(i,j)] \qquad\qquad \mid x(i-m..i] = x(j-m..j]\}) \end{aligned}$$

Given three suffixes, we can deduce properties of their R-LCEs from their lexicographical order.

Lemma 10.8. *For any three suffixes $x_i \prec x_j \prec x_k$ of some string x, it holds $\textsc{lce}_r(i,k) \leq \textsc{lce}_r(i,j)$ and $\textsc{lce}_r(i,k) \leq \textsc{lce}_r(j,k)$.*

Proof. Assume $\ell = \textsc{lce}_r(i,j) < \textsc{lce}_r(i,k)$, then $x_i[1..\ell] = x_j[1..\ell] = x_k[1..\ell]$ and $x_j[\ell+1] \neq x_i[\ell+1] = x_k[\ell+1]$. This implies $x_i \prec x_j \Leftrightarrow x_k \prec x_j$, which contradicts $x_i \prec x_j \prec x_k$. The proof of $\textsc{lce}_r(i,k) \leq \textsc{lce}_r(j,k)$ works analogously. □

10.2 The Runs Algorithm Revisited

In this section, we recapitulate the main ideas of the algorithm by Bannai et al. [Ban+17] that computes all runs in a string. This will be the basis of our solution for general ordered alphabet. We have already shown that every decreasing run is *rooted* in a longest Lyndon word (Lemma 10.7).

Definition 10.9 (Root of a Run)**.** Let $\langle i, j, p \rangle$ be a decreasing run, and let $i_0 \in [i, i+p)$ be the unique index with $\lambda[i_0] = p$ (as described in Lemma 10.7). We say that $\langle i, j, p \rangle$ is *rooted in* i_0.

Note that our notion of a root differs from the L-roots introduced by Crochemore et al. [Cro+14]. While an L-root is *any* length-p Lyndon word contained in the run, our root is exactly the starting position of the *leftmost* one. Now we explain how to compute a run from its root.

For a longest Lyndon word $x[i_0..\mathsf{next}[i_0])$ of length $p = \mathsf{next}[i_0] - i_0 = \lambda[i_0]$, it is easy to determine whether i_0 is the root of a decreasing run. We simply try to extend the periodicity as far as possible to both sides by using the LCE functions. For this purpose, we only need to compute $\ell = \textsc{lce}_\ell(i_0, \mathsf{next}[i_0])$ and $r = \textsc{lce}_r(i_0, \mathsf{next}[i_0])$. Let $i = i_0 - \ell + 1$ and $j = \mathsf{next}[i_0] + r - 1$, then clearly the substring $x[i..j]$ has minimal period p, and we cannot extend the substring to either side without breaking the periodicity. Thus, if $j - i + 1 \geq 2p$ then $\langle i, j, p \rangle$ is a run. Note that this run is only rooted in i_0 if additionally $i_0 \in [i, i+p)$ (or equivalently $\ell \leq p$) holds. An example of the LCEs and their role in the computation of a run is provided in Figure 10.1.

A Simple Runs Algorithm Since each decreasing run is rooted in exactly one index, we can find all decreasing runs by checking for each index whether it is the

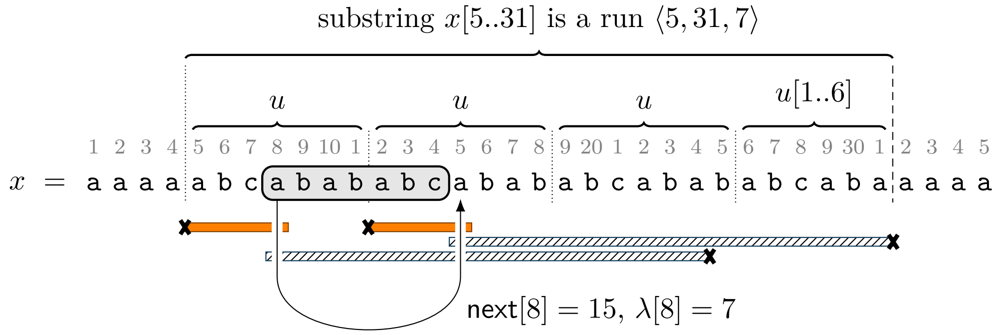

substring $x[5..31]$ is a run $\langle 5, 31, 7\rangle$
u
u
u
$u[1..6]$
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1 2 3 4 5
x = a a a a a b c a b a b a b c a b a b a b c a b a b a b c a b a a a a a
next[8] = 15, $\lambda[8] = 7$

Figure 10.1: Decreasing run $\langle 5, 31, 7\rangle$ with $x[5..31] = (\texttt{abcabab})^3\texttt{abcaba}$. The run is a repetition of the substring $u = \texttt{abcabab}$, and is rooted in position 8 with longest Lyndon word $x[8..\mathsf{next}[8]) = x[8..15) = u_4u[1..3] = \texttt{abababc}$. The solid boxes underneath the string indicate $\text{LCE}_\ell(8, 15) = 4$, while the hatched boxes indicate $\text{LCE}_r(8, 15) = 17$. The leftmost position of the run is $8 - \text{LCE}_\ell(8, 15) + 1 = 5$, while the rightmost position is $\mathsf{next}[8] + \text{LCE}_r(8, 15) - 1 = 31$.

root of a run. This procedure is outlined in Algorithm 10.1. First, we compute the NSS array (line 2), for example with the algorithm from Chapter 7. Then, we investigate one index $i_0 \in [1, n]$ at a time (line 3), and consider it as the root of a run with period $p = \mathsf{next}[i_0] - i_0$ (line 4). If the left-extension covers an entire period (i.e., $\text{LCE}_\ell(i_0, \mathsf{next}[i_0]) > p$), then we have already investigated the root of the run in an earlier iteration of the for-loop, and no further action is required (line 5). Otherwise, we compute the left and right border of the potential run as described earlier (lines 6–7). If the resulting interval has length at least $2p$, then we have discovered a run that is rooted in i_0 (lines 8–9). Note that we do not need to consider i_0 with $\mathsf{next}[i_0] = n + 1$, since the definition of a root implies that position $\mathsf{next}[i_0]$ is contained in the run.

Algorithm 10.1 Computing all decreasing runs.

```
Require: String $x[1..n]$ over general ordered alphabet.
Ensure: Set $R$ of all decreasing runs in $x$.
1: $R \leftarrow \emptyset$
2: compute array next (the NSS array of $x$)
3: for $i_0 \in [1, n]$ with $\mathsf{next}[i_0] \neq n + 1$ do
4:     $p \leftarrow \mathsf{next}[i_0] - i_0$
5:     if $\text{LCE}_\ell(i_0, \mathsf{next}[i_0]) \leq p$ then
6:         $i \leftarrow i_0 - \text{LCE}_\ell(i_0, \mathsf{next}[i_0]) + 1$
7:         $j \leftarrow \mathsf{next}[i_0] + \text{LCE}_r(i_0, \mathsf{next}[i_0]) - 1$
8:         if $j - i + 1 \geq 2p$ then
9:             $R \leftarrow R \cup \{\langle i, j, p\rangle\}$
```

Time and Space Complexity The NSS array can be computed in $\mathcal{O}(n)$ time and space for general ordered alphabet (see Chapters 7 and 8 or [Bil+20, Ell22]). Assume for now that we can answer L-LCE and R-LCE queries in constant time, then clearly the rest of the algorithm also requires $\mathcal{O}(n)$ time and space. The correctness of the algorithm follows from Lemma 10.7 and the description.

Lemma 10.10. *Let x be a string of length n over general ordered alphabet. We can compute all decreasing runs of x in $\mathcal{O}(n) + t(n)$ time and $\mathcal{O}(n) + s(n)$ space, where $t(n)$ and $s(n)$ are the time and space needed to compute* $\textsc{lce}_\ell(i, \mathsf{next}_x[i])$ *and* $\textsc{lce}_r(i, \mathsf{next}_x[i])$ *for all $i \in [1, n]$ with $\mathsf{next}_x[i] \neq n + 1$.*

In order to also find all *increasing* runs, we only need to rerun the algorithm with reversed alphabet order. This way, previously increasing runs become decreasing.

10.3 Algorithm for Computing the LCEs

In this section, we show how to precompute the LCEs required by Algorithm 10.1 in linear time and space. If we use the algorithm from Chapter 7 for computing the Lyndon array, then we already obtain all the R-LCEs as a byproduct (they are exactly the values stored in the array nlce). However, the algorithm from Chapter 7 requires a large amount of working space, which we can reduce if we use the solution from Chapter 8 instead. Hence we explicitly provide a solution for computing the R-LCEs from the precomputed Lyndon array. Our approach is asymmetric in the sense that we require different algorithms for L-LCEs and R-LCEs (whereas previous approaches usually compute L-LCEs by applying the R-LCE algorithm to the reverse text). However, for both directions we use similar properties of the Lyndon array that are shown in Lemmas 10.11 and 10.12 and visualized in Figure 10.2a.

Lemma 10.11. *Let $i \in [1, n]$ and $j = \mathsf{next}[i] \neq n + 1$. If $\textsc{lce}_r(i, j) \geq (j - i)$, then it holds $\textsc{lce}_r(j, j + (j - i)) = \textsc{lce}_r(i, j) - (j - i)$ and $\mathsf{next}[j] = j + (j - i)$.*

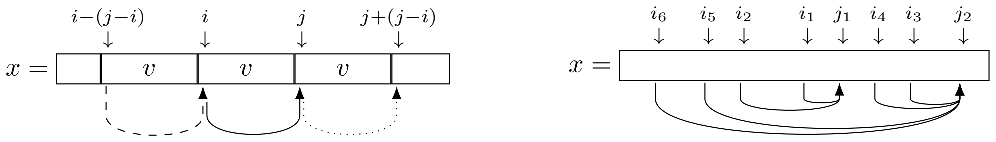
Proof. From $\textsc{lce}_r(i, j) \geq (j - i)$ follows $\textsc{lce}_r(i, j) = (j - i) + \textsc{lce}_r(j, j + (j - i))$, which is equivalent to $\textsc{lce}_r(j, j + (j - i)) = \textsc{lce}_r(i, j) - (j - i)$. It remains to be shown that $\mathsf{next}[j] = j + (j - i)$. Since x_i and x_j share a prefix of length at least $(j - i)$, and since $\mathsf{next}[i] = j$ implies $x_i \succ x_j$, it is easy to see that $x_{i+(j-i)} \succ x_{j+(j-i)}$. This means that $\mathsf{next}[j] \leq j + (j - i)$. Note that $x[i..j) = x[j..j + (j - i))$ is a Lyndon word due to $\mathsf{next}[i] = j$ and Lemma 6.7. Hence it holds $\lambda[j] \geq (j - i)$, or equivalently $\mathsf{next}[j] \geq j + (j - i)$. □

Lemma 10.12. *Let $i \in [1, n]$ and $j = \mathsf{next}[i] \neq n + 1$. If $\textsc{lce}_\ell(i, j) > (j - i)$, then it holds $\textsc{lce}_\ell(i - (j - i), i) = \textsc{lce}_\ell(i, j) - (j - i)$ and $\mathsf{next}[i - (j - i)] = i$.*

Proof. Analogous to Lemma 10.11. □

10.3.1 Computing the R-LCEs

First, we will briefly describe our general technique for computing LCEs, and our method of showing the linear time bound. Assume for this purpose that we want to compute $\ell = \textsc{lce}_r(i, j)$ with $i < j$. It is easy to see that we can determine ℓ by performing $\ell + 1$ individual symbol comparisons (by simultaneously scanning the suffixes x_i and x_j from left to right until we find a mismatch). Whenever we use this naive way of computing an LCE, we *charge* one symbol comparison to each of the indices in the interval $[j, j + \ell)$. This way, we account for ℓ symbol comparisons. Since we want to compute $\mathcal{O}(n)$ R-LCE values in $\mathcal{O}(n)$ time, we can afford a constant



(a) Lemmas 10.11 and 10.12. The dotted edge follows from $\text{LCE}_r(i,j) \geq (j-i)$ (Lemma 10.11). The dashed edge follows from $\text{LCE}_\ell(i,j) > (j-i)$ (Lemma 10.12).

(b) Relative order of R-LCE computations from first to last: $\text{LCE}_r(i_1,j_1)$, $\text{LCE}_r(i_2,j_1)$, $\text{LCE}_r(i_3,j_2)$, $\text{LCE}_r(i_4,j_2)$, $\text{LCE}_r(i_5,j_2)$, $\text{LCE}_r(i_6,j_2)$.

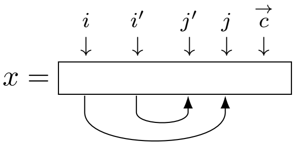
Figure 10.2: An edge from text position a to text position b indicates $\mathsf{next}[a] = b$.

time overhead (i.e., a constant number of unaccounted symbol comparisons) for each LCE computation. Thus, there is no need to charge the $(\ell+1)$th comparison to any index. At the time at which we want to compute ℓ, we may already know some lower bound $k \leq \ell$. In such cases, we simply skip the first k symbol comparisons and compute $\ell = k + \text{LCE}_r(i+k, j+k)$. This requires $\ell - k + 1$ symbol comparisons, of which we charge $\ell - k$ to the interval $[j+k, j+\ell)$.

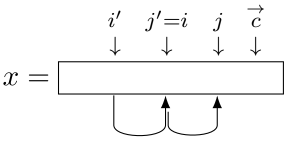
Ultimately, we will show that all R-LCE values $\text{LCE}_r(i,j)$ with $i \in [1,n]$ and $j = \mathsf{next}[i] \neq n+1$ can be computed in a way such that each text position gets charged at most once, which results in the desired linear time bound. From now on, we refer to i as the *left index* and j as the *right index* of the R-LCE computation. Our algorithm computes the R-LCEs in the following order (a visualization is provided in Figure 10.2b): We consider the possible right indices $j \in [2,n]$ one at a time and in *increasing* order. For each right index j, we then consider the corresponding left indices i with $\mathsf{next}[i] = j$ in *decreasing* order (we will see how to efficiently deduce this order from the Lyndon array later).

Assume that we are computing the R-LCEs in the previously described order, and let $\ell = \text{LCE}_r(i,j)$ with $j = \mathsf{next}[i] \neq n+1$ be the next value that we want to compute. The set of indices that we have already considered as left indices for LCE computations is $I = \{i' \mid (\mathsf{next}[i'] < j) \vee ((\mathsf{next}[i'] = j) \wedge (i < i'))\}$. For example, when we compute $\text{LCE}_r(i_4, j_2)$ in Figure 10.2b it holds $\{i_1, i_2, i_3\} \subseteq I$. At this point in time, the rightmost text position that we have already inspected is $\vec{c} = \max_{i' \in I}(\mathsf{next}[i'] + \text{LCE}_r(i', \mathsf{next}[i']))$ if $I \neq \emptyset$, or $\vec{c} = 1$ otherwise. Due to the nature of our charging method, we have not charged any indices from the interval $[\vec{c}, n]$ yet. Thus, in order to show that we can compute all LCEs without charging any index twice, it suffices to show how to compute $\ell = \text{LCE}_r(i,j)$ without charging any index from the interval $[1, \vec{c})$. If $j \geq \vec{c}$ then we naively compute ℓ and charge the symbol comparisons to the interval $[j, j+\ell)$, thus only charging previously uncharged indices. The new value of $\vec{c}$ is $j + \ell$. If however $j < \vec{c}$, then the computation of ℓ depends on the previously computed LCEs, which we describe in the following.

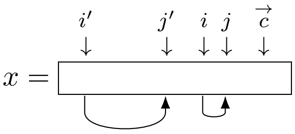
Let $\ell' = \text{LCE}_r(i', j')$ with $j' = \mathsf{next}[i']$ be the *most recently* computed R-LCE that satisfies $j' + \ell' = \vec{c}$. Our strategy for computing ℓ depends on the position of i relative to i' and j'. First, note that $i \notin [i', j')$ because otherwise Corollary 10.3 implies $j \leq j'$, which contradicts our order of computation. This leaves us with three possible cases (just like in Part II, a directed edge from text position a to text position b indicates $\mathsf{next}[a] = b$):



Case R1: $\boldsymbol{i < i'}$
(possibly $j' = j$)



Case R2: $\boldsymbol{i = j'}$

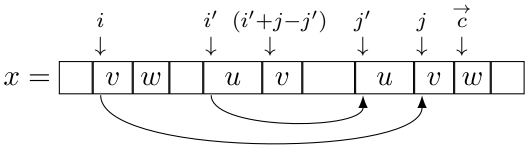


Case R3: $\boldsymbol{i > j'}$

Now we explain the cases in detail. Each case is accompanied by a schematic drawing. We kindly advise the reader to study the drawings alongside the description, since they are essential for an easy understanding of the matter.

Case R1: $\boldsymbol{i < i'}$ **(and** $j' \le j < \vec{c}$**).**

$|u| = j - j'$, $|v| = \vec{c} - j$
$\ell' = |uv|$, $\ell = |vw|$

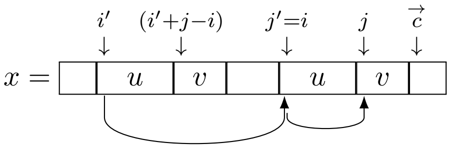


Due to $i < (i' + j - j') < j = \mathsf{next}[i]$ we have $x_j \prec x_i \prec x_{i'+j-j'}$. From Lemma 10.8 follows $\vec{c} - j = \text{LCE}_r(i' + j - j', j) \le \text{LCE}_r(i, j) = \ell$, i.e., both x_i and x_j start with v. Since now we know a lower bound $\vec{c} - j \le \ell$ on the desired LCE value, we can skip symbol comparisons during its computation. Later, we will see that the same bound also holds for most of the other cases. Generally, whenever we can show $\vec{c} - j \le \ell$ we use the following strategy. We compute $\ell = (\vec{c} - j) + \text{LCE}_r(i + (\vec{c} - j), \vec{c})$ using $\ell - (\vec{c} - j) + 1$ symbol comparisons, of which we charge $\ell - (\vec{c} - j)$ to the interval $[\vec{c}, j + \ell)$. Thus we only charge previously uncharged positions. We continue with $i' \leftarrow i$, $j' \leftarrow j$, $\ell' \leftarrow \ell$, and $\vec{c} \leftarrow j + \ell$.

Case R2: $\boldsymbol{i = j'}$**.** We divide this case into two subcases.

Case R2a: $\boldsymbol{\ell' < j' - i'}$.

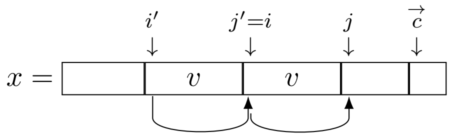
$|u| = j - j'$, $|v| = \vec{c} - j$



From $j < \vec{c} \implies j - i < \vec{c} - i = \ell'$ and $\ell' < j' - i'$ follows $i' + j - i < j' = i$. Therefore, $\mathsf{next}[i'] = i$ and the definition of next smaller suffixes imply $x_i \prec x_{i'+j-1}$. Due to $\mathsf{next}[i] = j$ we also have $x_j \prec x_i$, such that it holds $x_j \prec x_i \prec x_{i'+j-1}$. It is easy to see that $x_{i'+j-i}$ and x_j share a prefix v of length $\text{LCE}_r(i' + j - i, j) = \vec{c} - j$. In fact, also x_i has prefix v because Lemma 10.8 implies that $\text{LCE}_r(i' + j - i, j) \le \text{LCE}_r(i, j) = \ell$. Thus it holds $\vec{c} - j \le \ell$, and we can use the strategy from Case R1.

Case R2b: $\boldsymbol{\ell' \geq j' - i'}$.

$|v| = j' - i'$, $\ell = \ell' - |v|$

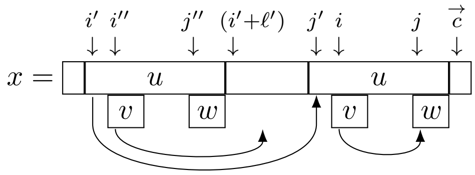


Due to $\ell' \geq j' - i'$, Lemma 10.11 implies $j = i + (j' - i')$ and $\ell = \ell' - (j' - i')$. Since i', j', and ℓ' are known, we can compute ℓ in constant time without performing symbol comparisons. We continue with $i' \leftarrow i$, $j' \leftarrow j$, and $\ell' \leftarrow \ell$ (leaving $\vec{c}$ unchanged).

Case R3: $\boldsymbol{i > j'}$. This is the most complicated case, and it is best explained by dividing it into three subcases. Let $d = j' - i'$, $i'' = i - d$, $j'' = j - d$, and $\ell'' = \textsc{lce}_r(i'', j'')$.
(In this situation it is implied that $j'' \leq j'$ because otherwise $\ell' = \textsc{lce}_r(i', j')$ would not be the *most recently* computed R-LCE that satisfies $j' + \ell' = \vec{c}$. However, since our proof does not rely on this property, we will not explain it in more detail.)

Case R3a: $\mathsf{next}[\boldsymbol{i''}] \boldsymbol{\neq j''}$**:**

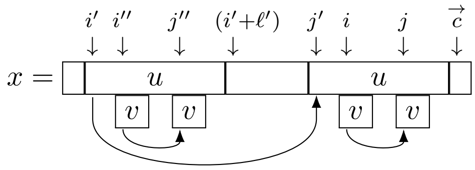
$|u| = \ell'$, $|v| = |w| = \vec{c} - j$
$\ell'' \geq |v|$, $\ell \geq |v|$



First, note that $x[i'..i' + \ell') = x[j'..\vec{c})$ implies $x[i..j) = x[i''..j'')$. From $\mathsf{next}[i] = j$ follows that $x[i..j) = x[i''..j'')$ is a Lyndon word. Thus, due to Lemma 6.7 and $\mathsf{next}[i''] \neq j''$ it holds $\mathsf{next}[i''] > j''$, which implies $x_{i''} \prec x_{j''}$. Let $v = x[i''..i'' + \vec{c} - j) = x[i..i + \vec{c} - j)$ and let $w = x[j''..i' + \ell') = x[j..\vec{c})$. From $x_{i''} \prec x_{j''}$ follows $v \preceq w$, while $x_i \succ x_j$ implies $v \succeq w$. Thus it holds $v = w$, and therefore $\textsc{lce}_r(i, j) \geq |w| = \vec{c} - j$. This means that we can use the strategy from Case R1.

Case R3b: $\mathsf{next}[\boldsymbol{i''}] \boldsymbol{= j''}$ **and** $\boldsymbol{(j'' + \ell'') < (i' + \ell')}$**:**

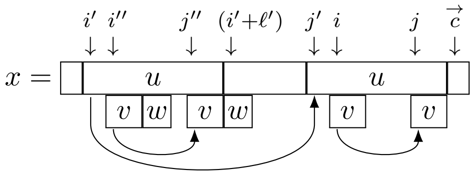
$|u| = \ell'$, $|v| = \ell'' = \ell$



Due to $\ell'' = \textsc{lce}_r(i'', j'')$, there is a shared prefix $v = x[i''..i'' + \ell'') = x[j''..j'' + \ell'')$ between $x_{i''}$ and $x_{j''}$, and the first mismatch between the two suffixes is $x[i'' + \ell''] \neq x[j'' + \ell'']$. Because of $(j'' + \ell'') < (i' + \ell')$, both the shared prefix and the mismatch are contained in $x[i'..i' + \ell')$ (i.e., in the first occurrence of u). If we consider the substring $x[j'..j' + \ell')$ instead (i.e., the second occurrence of u), then x_i and x_j clearly also share the prefix $v = x[i..i + \ell'') = x[j..j + \ell'')$, with the first mismatch occurring at $x[i + \ell''] \neq x[j + \ell'']$. Thus it holds $\ell = \ell''$. Due to $\mathsf{next}[i''] = j''$ and our order of R-LCE computations, we have already computed ℓ''. Therefore, we can simply assign $\ell \leftarrow \ell''$ and continue without changing i', j', ℓ', and $\vec{c}$.

Case R3c: $\mathsf{next}[i''] = j''$ and $(j'' + \ell'') \geq (i' + \ell')$:

$|u| = \ell'$, $|v| = \overrightarrow{c} - j$, $|vw| = \ell''$
$\ell \geq |v|$



This situation is similar to Case R3b. There is a shared prefix $v = x[i''..i'' + \overrightarrow{c} - j) = x[j''..i' + \ell')$ between the suffixes $x_{i''}$ and $x_{j''}$. They may share an even longer prefix vw, but only the first $|v| = \overrightarrow{c} - j$ symbols of their shared prefix are contained in $x[i'..i' + \ell')$ (i.e., in the first occurrence of u). If we consider the substring $x[j'..j' + \ell')$ instead (i.e., the second occurrence of u), then x_i and x_j clearly also share at least the prefix $v = x[i..i + \overrightarrow{c} - j) = x[j..\overrightarrow{c})$. Thus it holds $\overrightarrow{c} - j \leq \ell$, and we can use the strategy from Case R1.

We have shown how to compute ℓ without charging any index twice. It follows that the total number of symbol comparisons for all R-LCEs is $\mathcal{O}(n)$.

A Simple Algorithm for R-LCEs While the detailed differentiation between the six subcases helps to show the correctness of our approach, it can be implemented in a significantly simpler way (see Algorithm 10.2). At all times, we keep track of j', $\overrightarrow{c}$ and the distance $d = j' - i'$ (line 1). We consider the indices $j \in [2, n]$ in increasing order (line 2). For each index j, we then consider the indices i with $\mathsf{next}[i] = j$ in decreasing order (line 3). Each time we want to compute an R-LCE value $\ell = \textsc{lce}_r(i, j)$, we first check whether Case R3b applies (line 4). If it does, then we simply copy the previously computed R-LCE value $\textsc{lce}_r(i - d, j - d)$ (line 5). Otherwise, we either compute the LCE naively (if $j \geq \overrightarrow{c}$), or we have to apply the strategy from Case R1 (since all other cases except for Case R2b use this strategy; in Case R2b it holds $\overrightarrow{c} - j = \ell$, which means that it can also be solved with the strategy from Case R1). If $j \geq \overrightarrow{c}$ then in lines 7–8 we have $k = 0$, and thus we naively compute $\textsc{lce}_r(i, j)$ by scanning. If however $j < \overrightarrow{c}$, then we have $k = \overrightarrow{c} - j$, and we skip k symbol comparisons. In any case, we update the values j', $\overrightarrow{c}$, and d accordingly (line 9).

Algorithm 10.2 Computing all R-LCEs.

```
Require: String $x[1..n]$ and its NSS array $\mathsf{next}$.
 Ensure: R-LCE value $\textsc{lce}_r(i, \mathsf{next}[i])$ for each index $i \in [1, n]$ with $\mathsf{next}[i] \neq n + 1$.
1: $j' \leftarrow 0$;   $\overrightarrow{c} \leftarrow 1$;   $d \leftarrow 0$
2: for $j \in [2, n]$ in increasing order do
3:     for $i$ with $\mathsf{next}[i] = j \neq n + 1$ in decreasing order do
4:         if $\left(\begin{array}{l} \quad i, j \in (j', \overrightarrow{c}) \\ \wedge\ \mathsf{next}[i - d] = j - d \\ \wedge\ j + \textsc{lce}_r(i - d, j - d) < \overrightarrow{c} \end{array}\right)$ then
5:             $\textsc{lce}_r(i, j) \leftarrow \textsc{lce}_r(i - d, j - d)$        ▷ retrieve LCE in constant time
6:         else
7:             $k \leftarrow \max(\overrightarrow{c}, j) - j$
8:             $\textsc{lce}_r(i, j) \leftarrow k + \textsc{naive-scan-lce}_r(i + k, j + k)$
9:             $j' \leftarrow j$;   $\overrightarrow{c} \leftarrow j + \textsc{lce}_r(i, j)$;   $d \leftarrow j - i$
```

The correctness of the algorithm follows from the description of Cases 1–3. Since for each left index i we have to store at most one R-LCE, we can simply maintain the LCEs in a length-n array, where the ith entry is $\textsc{lce}_r(i, \mathsf{next}[i])$. This way, we use linear space and can access the R-LCE that is required in line 5 in constant time. Apart from the at most n symbol comparisons that we charge to the indices, we only need a constant number of additional primitive operations per computed R-LCE. The order of iteration can be realized by first generating all $(i, \mathsf{next}[i])$-pairs, and then using a linear time radix sorter to sort the pairs in increasing order of their second component and decreasing order of their first component. We have shown:

Lemma 10.13. *Given a string $x[1..n]$ and its NSS array next, we can compute $\textsc{lce}_r(i, \mathsf{next}[i])$ for all indices $i \in [1, n]$ with $\mathsf{next}[i] \neq n + 1$ in $\mathcal{O}(n)$ time and space.*

10.3.2 Computing the L-LCEs

Our solution for the L-LCEs is similar to the one for R-LCEs, but differs in subtle details. We generally compute $\ell = \textsc{lce}_\ell(i, j)$ by simultaneously scanning the prefixes $x[1..i]$ and $x[1..j]$ from right to left until we find the first mismatch. This takes $\ell + 1$ symbol comparisons, of which we charge ℓ comparisons to the interval $(i - \ell, i]$. As before, if some lower bound $k \leq \ell$ is known then we skip k symbol comparisons. In this case, we compute the L-LCE as $\ell = k + \textsc{lce}_\ell(i - k, j - k)$, and charge $\ell - k$ comparisons to the interval $(i - \ell, i - k]$.

Again, we will show how to compute all values $\textsc{lce}_\ell(i, \mathsf{next}[i])$ with $i \in [1, n]$ and $\mathsf{next}[i] \neq n + 1$ such that each index gets charged at most once. In contrast to the more complex R-LCE iteration order, we can simply compute the L-LCE values in *decreasing* order of i. Thus, when we want to compute $\ell = \textsc{lce}_\ell(i, j)$ with $j = \mathsf{next}[i] \neq n + 1$, we have already considered the indices $I = \{i' \mid i' \in (i, n] \wedge \mathsf{next}[i'] \neq n + 1\}$ as left indices of L-LCE computations. The leftmost text position that we have already inspected so far is $\overleftarrow{c} = \min_{i' \in I}(i' - \textsc{lce}_\ell(i', \mathsf{next}[i']))$ if $I \neq \emptyset$, or $\overleftarrow{c} = n$ otherwise. Due to our charging method, we have not charged any index from the interval $[1, \overleftarrow{c}]$ yet. Thus, we only have to show how to compute ℓ without charging indices from $(\overleftarrow{c}, n]$. Let $\ell' = \textsc{lce}_\ell(i', j')$ be the most recently computed L-LCE that satisfies $i' - \ell' = \overleftarrow{c}$. If $i \leq \overleftarrow{c}$ then we compute ℓ naively and charge the symbol comparisons to the interval $(i - \ell, i]$ (thus only charging previously uncharged indices). If however $i > \overleftarrow{c}$, then our strategy is more complicated. Before explaining it in detail, we show three important properties that hold in the present situation.

First, we show that $i \geq i' - (j' - i')$. Assume the opposite (as visualized in Figure 10.3a), then from $\overleftarrow{c} = i' - \ell' < i$ follows $\ell' > j' - i'$. Thus, Lemma 10.12 implies $\mathsf{next}[i' - (j' - i')] = i'$ (dashed edge) and $\textsc{lce}_\ell(i' - (j' - i'), i') = \ell' - (j' - i')$. Due to our order of computation and $i < i' - (j' - i')$ we must have already computed this L-LCE. However, it holds $i' - (j' - i') - \textsc{lce}_\ell(i' - (j' - i'), i') = \overrightarrow{c}$, which contradicts the fact that $\ell' = \textsc{lce}_\ell(i', j')$ is the *most recently* computed L-LCE with $i' - \ell' = \overleftarrow{c}$.

Next, we show that $j \leq i'$. First, note that $j \notin (i', j')$, since due to $i < i'$ we would otherwise contradict Corollary 10.3. Thus we only have to show $j < j'$. Assume for this purpose that $j \geq j'$ (as visualized in Figure 10.3b). From $j' - i' + i \in (i, \mathsf{next}[i])$ and the definition of next smaller suffixes follows $x_i \prec x_{j' - i' + i}$. Because

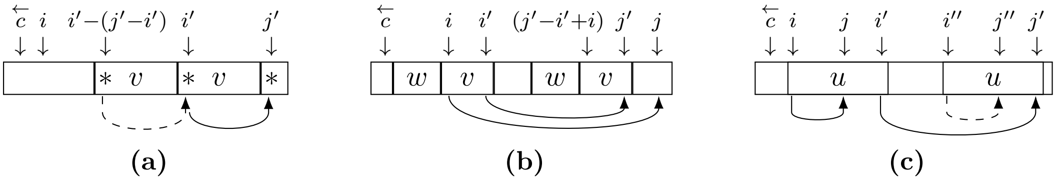

$\overleftarrow{c}$ i $i'-(j'-i')$ i' j'
$*$ v $*$ v $*$
(a)
$\overleftarrow{c}$ i i' $(j'-i'+i)$ j' j
w v w v
(b)
$\overleftarrow{c}$ i j i' i'' j'' j'
u u
(c)

Figure 10.3: Illustration of the proofs of the three properties in Section 10.3.2.

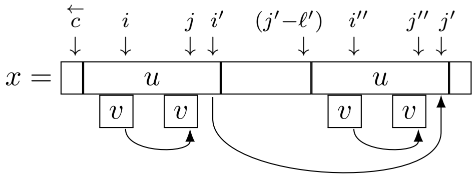
of $\text{LCE}_\ell(i', j') > (i' - i)$ it holds $x[i..i'] = x[j' - i' + i..j']$. Thus $x_i \prec x_{j'-i'+i}$ implies $x_{i'} \prec x_{j'}$, which contradicts the fact that $\mathsf{next}[i'] = j'$.

Lastly, let $d = j' - i'$, $i'' = i + d$, and $j'' = j + d$ (as visualized in Figure 10.3c). Now we show that $\mathsf{next}[i''] = j''$ (dashed edge in the figure). Because of $x(\overleftarrow{c}..i'] = x(j' - \ell'..j']$ it holds $x[i..j) = x[i''..j'')$. From $\mathsf{next}[i] = j$ and Lemma 6.7 follows that $x[i''..j'')$ is a Lyndon word, and thus $\mathsf{next}[i''] \geq j''$. We have already shown that $i \geq i' - (j' - i')$, which implies $i'' \geq i'$. Due to $\mathsf{next}[i'] = j'$ and $i'' \in [i', j')$ it follows from Corollary 10.3 that $\mathsf{next}[i''] \leq j'$. Now assume $\mathsf{next}[i''] \in (j'', j']$, then $x[i''..\mathsf{next}[i'']) = x[i..j + (\mathsf{next}[i''] - j''))$ is a Lyndon word, which contradicts the fact that $x[i..j)$ is the longest Lyndon word starting at position i. Thus, we have ruled out all possible values of $\mathsf{next}[i'']$ except for j''.

Now we show how to compute ℓ. We keep using the definition of i'' and j'' from the previous paragraph. Furthermore, let $\ell'' = \text{LCE}_\ell(i'', j'')$. There are two possible cases.

Case L1: $(i'' - \ell'') > (j' - \ell')$.

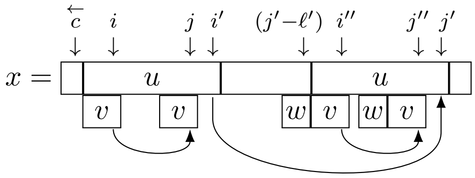
$\ell' = |u|$, $\ell = \ell'' = |v|$


$\overleftarrow{c}$ i j i' $(j'-\ell')$ i'' j'' j'
$x =$
u u
v v v v

Due to $\ell'' = \text{LCE}_\ell(i'', j'')$, the prefixes $x[1..i'']$ and $x[1..j'']$ share the suffix $v = x(i'' - \ell''..i''] = x(j'' - \ell''..j'']$, and the first (from the right) mismatch between these prefixes is $x[i'' - \ell''] \neq x[j'' - \ell'']$. Both the shared suffix and the mismatch are contained in $x(j' - \ell'..j']$ (i.e., in the right occurrence of u). If we consider the substring $x(\overleftarrow{c}..i']$ instead (i.e., the left occurrence of u), then $x[1..i]$ and $x[1..j]$ clearly also share the suffix $v = x(i - \ell''..i] = x(j - \ell''..j]$, with the first mismatch occurring at $x[i - \ell''] \neq x[j'' - \ell]$. Thus it holds $\ell = \ell''$. Due to $\mathsf{next}[i''] = j''$ and the order of L-LCE computations, we have already computed ℓ''. Therefore, we can simply assign $\ell \leftarrow \ell''$ and continue without changing i', j', ℓ', and $\overleftarrow{c}$. (Note that possibly $i'' \neq i' \wedge j'' = j'$, which requires no special handling. We provide a sketch in Figure 10.4a at the end of the chapter.)

Case L2: $(i'' - \ell'') \leq (j' - \ell')$.

$\ell' = |u|$, $\ell'' = |vw|$, $\ell \geq |v|$


$\overleftarrow{c}$ i j i' $(j'-\ell')$ i'' j'' j'
$x =$
u u
v v w v w v

This situation is similar to Case L1. There is a shared suffix $v = x(j' - \ell'..i''] = x(j'' - (i - \overleftarrow{c})..j'']$ between the prefixes $x[1..i'']$ and $x[1..j'']$. They may share an

even longer suffix wv, but only the rightmost $|v| = i' - \overleftarrow{c}$ symbols of this suffix are contained in $x(j' - \ell'..j']$ (i.e., in the right occurrence of u). If we consider the substring $x(\overleftarrow{c}..i']$ instead (i.e., the left occurrence of u), then $x[1..i]$ and $x[1..j]$ clearly also share the suffix $v = x(\overleftarrow{c}..i] = x(j - (i - \overleftarrow{c})..j]$. Thus it holds $i - \overleftarrow{c} \leq \ell$, and we can skip the first $i - \overleftarrow{c}$ symbol comparisons by computing the LCE as $\ell = (i - \overleftarrow{c}) + \textsc{lce}_\ell(\overleftarrow{c}, j + \overleftarrow{c} - i)$. We charge $\ell - (i - \overleftarrow{c})$ symbol comparisons to the previously uncharged interval $(i - \ell, \overleftarrow{c}]$, and continue with $i' \leftarrow i$, $j' \leftarrow j$, $\ell' \leftarrow \ell$, and $\overleftarrow{c} \leftarrow i - \ell$.
(Note that possibly $i'' \neq i' \wedge j'' = j'$ or even $i'' = i' \wedge j'' = j'$, which requires no special handling. We provide schematic drawings in Figures 10.4b and 10.4c at the end of the chapter.)

We have shown how to compute ℓ without charging any index twice. It follows that the total number of symbol comparisons for all LCEs is $\mathcal{O}(n)$. For completeness, we outline a simple implementation of our approach in Algorithm 10.3. Lines 4–5 correspond to Case L1. If $i \leq \overleftarrow{c}$, then lines 7–9 compute the LCE naively. Otherwise, they correspond to Case L2.

Algorithm 10.3 Computing all L-LCEs.

```
Require: String $x[1..n]$ and its NSS array next.
Ensure: L-LCE value $\textsc{lce}_\ell(i, \mathsf{next}[i])$ for each index $i \in [1, n]$ with $\mathsf{next}[i] \neq n + 1$.
1: $i' \leftarrow 0$;  $\overleftarrow{c} \leftarrow n$;  $d \leftarrow 0$
2: for $i \in [1, n]$ with $\mathsf{next}[i] \neq n + 1$ in decreasing order do
3:     $j \leftarrow \mathsf{next}[i]$
4:     if $i \in (\overleftarrow{c}, i') \wedge i - \textsc{lce}_\ell(i + d, j + d) > \overleftarrow{c}$ then
5:         $\textsc{lce}_\ell(i, j) \leftarrow \textsc{lce}_\ell(i + d, j + d)$        ▷ retrieve LCE in constant time
6:     else
7:         $k \leftarrow i - \min(\overleftarrow{c}, i)$
8:         $\textsc{lce}_\ell(i, j) \leftarrow k + \textsc{naive-scan-lce}_\ell(i - k, j - k)$
9:         $i' \leftarrow i$;  $\overleftarrow{c} \leftarrow i - \textsc{lce}_\ell(i, j)$;  $d \leftarrow j - i$
```

Lemma 10.14. *Given a string of length n and its NSS array* next, *we can compute $\textsc{lce}_\ell(i, \mathsf{next}[i])$ for all indices $i \in [1, n]$ with $\mathsf{next}[i] \neq n + 1$ in $\mathcal{O}(n)$ time and space.*

The main theorem is a corollary of Lemmas 10.10, 10.13 and 10.14. These lemmas state that all decreasing runs can be computed in linear time and space. For the increasing runs, we only have to reverse the order of the alphabet and rerun the algorithm. This way, previously increasing runs become decreasing. Hence we have shown Theorem 10.1.

Theorem 10.1. *All the runs contained in a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of working space.*

10.4 Practical Implementation

We implemented the algorithm for the runs computation in C++17 and evaluated it by computing all runs on texts from the natural, real repetitive, and artificial repetitive text collections of the Pizza-Chili corpus[1]. Additionally, we used the binary run-rich strings proposed by Matsubara et al. [Mat+09] as input. Table 10.1 shows the achieved throughput, i.e., the number of input bytes (or equivalently input symbols) processed per second. On the string `tm29` we achieve the highest throughput of 15.6 MiB/s. The lowest throughput of 8.8 MiB/s occurs on the text `dna`. Generally, we perform better for run-rich strings.

Lastly, it is noteworthy that our new method of LCE computation leads to a remarkably simple implementation of the runs algorithm. In fact, the entire implementation *including the computation of the NSS array* needs only 250 lines of code. We achieve this by interleaving the computation of the R-LCEs with the computation of the NSS array (as described in Chapter 7), which also improves the practical performance. For technical details we refer to the source code, which is publicly available on GitHub[2].

10.5 Conclusion

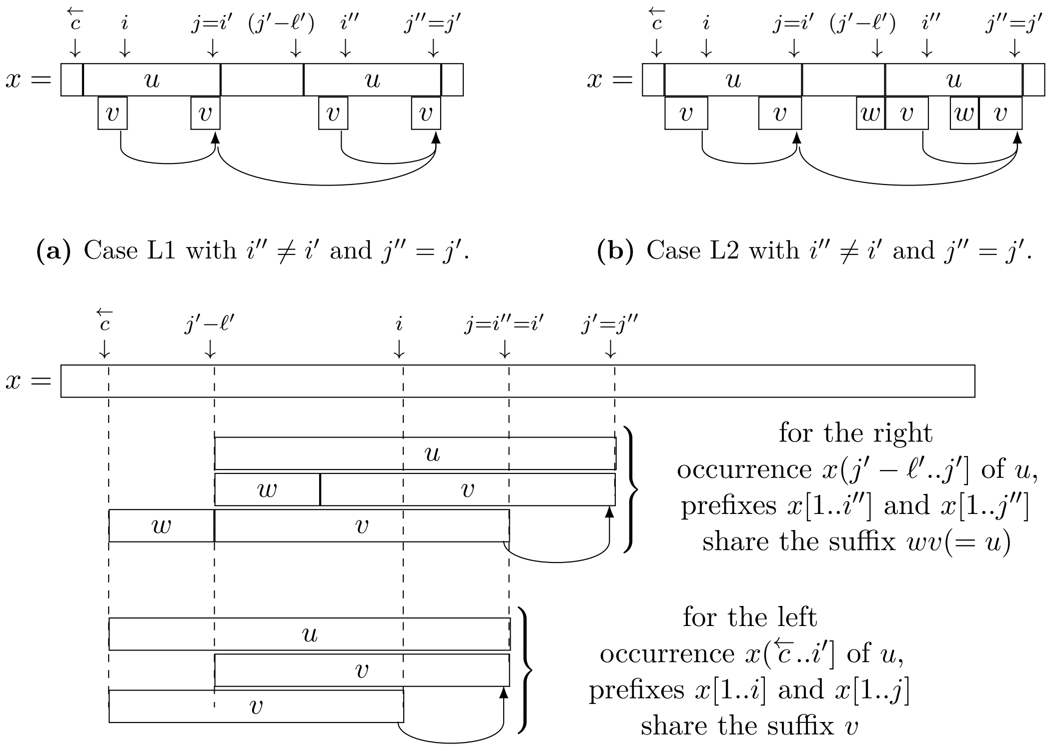
We have shown the first linear time algorithm for computing all runs over general ordered alphabet. The algorithm is also very fast in practice and remarkably easy to implement. It is an open question whether our techniques could be used for the computation of runs on tries, where the best known algorithms require super-linear time, even for linearly-sortable alphabets (see, e.g., [Sug+21]).

[1] http://pizzachili.dcc.uchile.cl/texts.html, http://pizzachili.dcc.uchile.cl/repcorpus.html

[2] https://github.com/jonas-ellert/linear-time-runs, permanently archived in the Software Heritage Archive at https://archive.softwareheritage.org/swh:1:dir:6117eb0b8f1f857e585efc1e359809e680776f4e;origin=https://github.com/jonas-ellert/linear-time-runs;visit=swh:1:snp:aa7b1dc6293d939b5ec6e554d3102b15a518b7e7;anchor=swh:1:rev:065dfae01cccc8fa7fa8b116cfa91f272c0b22ba

Table 10.1: Throughput achieved by the new runs algorithm using an AMD EPYC 7452 processor. The run-density is the number of runs divided by the length of the respective string, scaled up by a factor of 100. We repeated each experiment five times and use the median throughput as the final result (the minimum and maximum throughputs were almost identical to the median). All numbers are truncated to one decimal place.

Text *n in MiB*	t_{49} [Mat+09] *1077 MiB*	**sources** *201 MiB*	**pitches** *53 MiB*	**proteins** *1024 MiB*	**dna** *385 MiB*	**english** *1024 MiB*	**xml** *282 MiB*	**ecoli** *107 MiB*	**cere** *439 MiB*	**fib41** *255 MiB*	**rs.13** *206 MiB*	**tm29** *256 MiB*
run-density	94.4	4.7	11.7	7.0	25.3	2.4	3.4	24.4	23.6	76.3	92.7	83.3
MiB/s	15.0	11.4	11.0	10.9	8.8	10.5	12.8	9.0	9.2	15.4	15.1	15.6



(a) Case L1 with $i'' \neq i'$ and $j'' = j'$.

(b) Case L2 with $i'' \neq i'$ and $j'' = j'$.

(c) Case L2 with $i'' = i'$ and $j'' = j'$.

Figure 10.4: Additional drawings for Cases L1 and L2. The notation is the same as in the description of the cases.

Chapter 11

11 Computing Runs Over General Unordered Alphabet

In Chapter 10, we resolved the complexity of computing runs over general ordered alphabet. In this chapter, we show a similar result for general unordered alphabet, i.e., using only equality comparisons of symbols. We start by analyzing the decision tree complexity of the problem. That is, we only consider the required and necessary number of comparisons, without worrying about an efficient implementation. We show that, even if the value of σ is assumed to be known, $\Omega(n \log \sigma)$ comparisons are required for testing whether a length-n string that contains σ distinct symbols is square-free. Then, we also provide an efficient algorithm that not only tests square-freeness, but actually computes all runs in $\mathcal{O}(n \log \sigma)$ time. Here (just like in the previous chapters), we do not only count the number of performed symbol comparison, but also the additional word RAM operations performed by the algorithm.

Theorem 11.1. *Let $n, \sigma \in \mathbb{N}^+$ with $8 \leq \sigma \leq n$ be fixed. There is no deterministic algorithm that performs at most $n \ln \sigma - 3.6n = \mathcal{O}(n \log \sigma)$ equality comparisons in the worst case, and determines whether a length-n string that contains at most σ distinct symbols over general unordered alphabet is square-free.*

Theorem 11.2. *All the runs contained in a length-n string over general unordered alphabet can be computed in $\mathcal{O}(n \log \sigma)$ time, where σ is the number of distinct symbols in the string, which is not known in advance.*

We stress again that the value of σ is not assumed to be known. Considering the lower bounds for Alphabet Size Testing (Problem 2.2) from Chapter 2, this may seem surprising. In Theorem 2.3 (*ii*), we essentially showed that finding a sublinear multiplicative approximation of σ requires $\Omega(n\sigma)$ comparisons. However, this does not contradict the claimed upper bound in Theorem 11.2, as we are only saying that the number of comparisons used on a particular input string is at most $\mathcal{O}(n \log \sigma)$, but might actually be smaller. Thus, it is not possible to extract any meaningful approximation of the value of σ from the number of performed comparisons. The algorithm for Theorem 11.2 is quite involved, and we describe it in an incremental manner. First, we consider the easier problem of testing square-freeness, and provide a solution that performs an asymptotically optimal number of equality comparisons.

Lemma 11.3. *Testing square-freeness of a length-n string that contains σ distinct symbols over general unordered alphabet can be done using $\mathcal{O}(n \log \sigma)$ comparisons.*

The above result is not efficient in the sense that it only restricts the overall number of comparisons, and not the time to actually figure out which comparisons should be used. A direct implementation results in a quadratic time algorithm. We first show how to improve this to $\mathcal{O}(n \log \sigma + n \log^* n)$ time (while still keeping the asymptotically optimal $\mathcal{O}(n \log \sigma)$ number of comparisons), and finally to $\mathcal{O}(n \log \sigma)$.

Lemma 11.4. *Testing square-freeness of a length-n string that contains σ distinct symbols over general unordered alphabet can be implemented in $\mathcal{O}(n \log \sigma)$ time.*

Finally, we also generalize this result to the computation of runs, which results in Theorem 11.2. Altogether, our results fully resolve the open question of Main and Lorentz [ML84] for the case of general unordered alphabet and deterministic algorithms. We leave extending the lower bound to randomized algorithms as an open question.

Overview of the Methods

As mentioned before, Main and Lorentz [ML84] designed an $\mathcal{O}(n \log n)$ time algorithm for testing square-freeness of length-n strings over general unordered alphabet. The high-level idea of their algorithm goes as follows. They first designed a procedure for checking, given two strings x and y, if their concatenation contains a square that is not fully contained in x nor y in $\mathcal{O}(|x| + |y|)$ time. Then, a divide-and-conquer approach can be used to detect a square in the whole input string in $\mathcal{O}(n \log n)$ total time. For general unordered alphabet of unbounded size this cannot be improved, but Crochemore [Cro86] showed that, for general ordered alphabet of size σ, a faster $\mathcal{O}(n \log \sigma)$ time algorithm exists. The gist of his approach is to first obtain the so-called f-factorization of the input string, which in a certain sense “discovers” repetitive fragments (this factorization is the same as Storer and Szymanski’s version of the Lempel-Ziv factorization [LZ76, SS82], which we also used in Part I). Then, the factorization can be used to apply the procedure of Main and Lorentz on appropriately selected fragments of the input strings in such a way that the leftmost occurrence of every distinct square is detected, and the total length of the strings on which we apply the procedure is only $\mathcal{O}(n)$. The factorization can be found in $\mathcal{O}(n \log \sigma)$ time for general ordered alphabet of size σ, e.g., by reducing the string to effective integer alphabet with Lemma 2.1, and then using a linear time algorithm for such alphabet (e.g., [CI08a]).

For general unordered alphabet, computing the Lempel-Ziv factorization (or anything similar) is problematic, as we have already shown in Chapter 3 that it requires $\Omega(n\sigma)$ equality tests. Thus, we need another approach. Additionally, the $\mathcal{O}(n)$ time algorithm from Chapter 10 hinges on the notion of Lyndon words, which is simply not defined for strings over general unordered alphabet. Thus, at first glance it might seem that $\Theta(n\sigma)$ is the right time complexity for testing square-freeness over length-n strings over general unordered alphabet of size σ. However, due to the $\Omega(n \log n)$ lower bound by Main and Lorentz for testing square-freeness of length-n strings consisting of up to n distinct symbols, one might hope for an $\mathcal{O}(n \log \sigma)$ time algorithm when there are only σ distinct symbols.

We begin the chapter with a lower bound of $\Omega(n \log \sigma)$ equality comparisons for such strings. Intuitively, we show that testing square-freeness has the direct sum property: $\frac{n}{\sigma}$ instances over length-σ strings can be combined into a single instance over a length-n string. As in the proof of Main and Lorentz, we use the adversarial method, which we also used for the lower bounds in Chapters 2 and 3. While the underlying calculation is essentially the same as the one by Main and Lorentz, we need to appropriately combine the smaller instances, which is done using the infinite square-free Prouhet-Thue-Morse sequence, and we have to use significantly more complex rules for resolving the subsequent equality tests.

We then move to designing an algorithm that uses $\mathcal{O}(n \log \sigma)$ equality comparisons to test square-freeness. As discussed earlier, one way of detecting squares uses the LZ factorization of the string. However, as shown in Chapter 3, we cannot compute the factorization over general unordered alphabet in $o(n\sigma)$ comparisons. Therefore, we will instead use a novel type of factorization, the Δ-*approximate LZ factorization*, which can be seen as an approximate version of the LZ factorization (but not to be confused with the LZ-like factorizations from Part I). Intuitively, its goal is to "capture" all sufficiently long squares, while the original LZ factorization (or f-factorization) captures all squares. Each phrase in a Δ-approximate LZ factorization consists of a head of length at most Δ and a tail (possibly empty) that must occur at least once before, such that the whole phrase is at least as long as the standard LZ phrase starting at the same position. Contrary to the standard LZ factorization, this factorization is not unique. The advantage of our modification is that there are fewer phrases (and there is more flexibility as to what they should be), and hence one can hope to compute such a factorization more efficiently.

To design an efficient construction method for a Δ-approximate LZ factorization, we first show how to compute a sparse suffix tree while trying to use only a few symbol comparisons. This is then applied on a set of positions from a so-called difference cover with some convenient synchronizing properties. Then, a Δ-approximate LZ factorization allows us to detect squares of length $\geq 8\Delta$.

The first warm-up algorithm fixes Δ depending on n and σ (assuming that σ is known), and uses the approximate LZ factorization to find all squares of length at least 8Δ. It then finds all the shorter squares by dividing the string into blocks of length 8Δ, and applying the original algorithm by Main and Lorentz on each block pair. Our choice of Δ leads to $\mathcal{O}(n(\log \sigma + \log \log n))$ comparisons.

The improved algorithm does not need to know σ, and instead starts with a large $\Delta = \Omega(n)$, and then progressively decreases Δ in at most $\mathcal{O}(\log \log n)$ phases, where later phases detect shorter squares. As soon as we notice that there are many distinct symbols in the alphabet, by carefully adjusting the parameters, we can afford switching to the approach of Main and Lorentz on sufficiently short fragments of the input string. Since we cannot afford $\Omega(n)$ comparisons per phase, we use a deactivation technique, where whenever we perform a large number of comparisons in a phase, we will discard a large part of the string in all following phases. More precisely, during a given phase, we avoid looking for squares in a fragment fully contained in a tail from an earlier phase. This leads to optimal $\mathcal{O}(n \log \sigma)$ comparisons.

The above approach uses an asymptotically optimal number of equality tests in the worst case, but does not result in an efficient algorithm. The main bottleneck is constructing the sparse suffix trees. However, it is not hard to provide an efficient implementation using the general mechanism for answering LCE queries for strings over general unordered alphabet [Gaw+16]. Unfortunately, the best known approach for answering such queries incurs an additional $\mathcal{O}(n \log^* n)$ in the time complexity, even if the size of the alphabet is constant. We overcome this technical hurdle by once more carefully deactivating fragments of the text to account for the performed work.

Many of our techniques can easily be modified to compute all runs rather than detecting squares. We exploit that the approximate factorization reveals long substrings with a previous occurrence. Hence we compute runs only for the first occurrence of such substrings, while for later occurrences we simply copy the already computed runs. By carefully arranging the order of the computation, we ensure that the total time for copying is bounded by the number of runs, which is known to be $\mathcal{O}(n)$. This way, we achieve $\mathcal{O}(n \log \sigma)$ time and comparisons to compute all runs.

11.1 Preliminaries

Unlike in the rest of the dissertation, we use the letter T (rather than x) to denote the input string. As in the previous chapters, the *substring* $T[i..j]$ is the string $T[i] \dots T[j]$. However, we may also refer to the *fragment* $T[i..j]$ whenever we talk about the specific occurrence of $T[i..j]$ starting at position i in T. We say that a fragment $T[i'..j']$ is properly contained in another fragment $T[i..j]$ if $i < i' \leq j' < j$. A substring is properly contained in $T[i..j]$, if it equals a fragment that is properly contained in fragment $T[i..j]$. Recall that we refer to the minimal period of a string as *its period* or *the period* of the string, and a string is called periodic if its period is at most half its length.

Computational Model We assume a general unordered alphabet as described in Chapter 2, and the time taken by an algorithm is the combined number of symbol equality comparisons and word RAM operations. Since we do not consider general ordered alphabets in this chapter, we simply write comparison rather than equality comparison. Our algorithms will internally use strings over linearly-sortable alphabet (for example by reducing parts of the input string with techniques similar to Lemma 2.1). We stress that in such strings the symbols are not the symbols from the input string, but simply integers calculated by the algorithm.

Squares and Runs Recall that a square in a string is a length-2ℓ fragment of period ℓ. The following theorem is a classical result by Main and Lorentz [ML84].

Theorem 11.5. *Testing square-freeness of a length-n string over general unordered alphabet can be implemented in $\mathcal{O}(n \log n)$ time.*

The proof of the above theorem is based on running a divide-and-conquer procedure using the following lemma.

Lemma 11.6. *Given two strings x and y over general unordered alphabet, we can test if there is a square in xy that is not fully contained in x nor y in $\mathcal{O}(|x| + |y|)$ time and comparisons.*

Recall that a repetition in $T[1..n]$ is a triple $\langle s, e, p \rangle$ with $s, e, p \in [1, n]$ such that $T[s..e]$ is a periodic substring of minimal period $p \leq \frac{e-s+1}{2}$, and a run is a repetition

$\langle s, e, p\rangle$ that cannot be extended to the left nor to the right with the same period, in other words $s = 1$ or $T[s-1] \neq T[s-1+p]$ and $e = n$ or $T[e+1] \neq T[e+1-p]$. The celebrated runs conjecture, proven by Bannai et al. [Ban+17], states that the number of runs is any length-n string is less than n. As shown in Chapter 10, computing runs (and thus testing square-freeness) over general ordered alphabet takes $\mathcal{O}(n)$ time.

Theorem 10.1. *All the runs contained in a length-n string over general ordered alphabet can be computed in $\mathcal{O}(n)$ time and words of working space.*

Corollary 10.2. *Testing whether or not a length-n string over general ordered alphabet contains a square can be done in $\mathcal{O}(n)$ time and words of space.*

Lempel-Ziv Factorization In contrast to Part I, we will use Lempel and Ziv's original definition of the LZ factorization. The unique LZ phrase starting at position s of $T[1..n]$ is a fragment $T[s..e]$ such that $T[s..(e-1)]$ occurs at least twice in $T[1..(e-1)]$ (where the empty string ε occurs twice in every string) and either $e = n$ or $T[s..e]$ occurs only once in $T[1..e]$. The Lempel-Ziv factorization of T consists of z phrases $f_1, \ldots, f_z$ such that the concatenation $f_1 \ldots f_z$ is equal to $T[1..n]$ and each f_i is the unique LZ phrase starting at position $1 + \sum_{j=1}^{i-1} |f_j|$.

Tries Given a collection $\mathcal{S} = \{T_1, \ldots, T_k\}$ of strings over some alphabet Σ, its trie is a rooted tree with edge labels from Σ. For any node v, the concatenation of the edge labels from the root to the node *spells* a string. The string-depth of a node is the length of the string that it spells. No two nodes spell the same string, i.e., for any node, the labels of the edges to its children are pairwise distinct. Each leaf spells one of the T_i, and each T_i is spelled by either an internal node or a leaf.

The compacted trie of $\mathcal{S}$ can be obtained from its (non-compacted) trie by contracting each path between a leaf or a branching node and its closest branching ancestor into a single edge (i.e., by contraction we eliminate all non-branching internal nodes). The label of the new edge is the concatenation of the edge labels of the contracted path in root to leaf direction. Since there are at most k leaves and all internal nodes are branching, there are $O(k)$ nodes in the compacted trie. Each edge label is some substring $T_i[s..e]$ of the string collection, and we can avoid explicitly storing the label by instead storing the reference (i, s, e). Thus $O(k)$ words are sufficient for storing the compacted trie. Consider a string T' that is spelled by a node of the non-compacted trie. We say that T' is explicit, if and only if it is spelled by a node of the compacted trie, in which case this node is also called explicit. Otherwise T' is implicit, and we say that it is spelled by an implicit node.

The suffix tree of a string $T[1..n]$ is the compacted trie containing exactly its suffixes, i.e., a trie over the string collection $\{T[i..n] \mid i \in \{1, \ldots, n\}\}$. It is one of the most fundamental data structures in string algorithmics, and is widely used, e.g., for compression and indexing [Gus97]. The suffix tree can be stored in $\mathcal{O}(n)$ words of memory, and for linearly-sortable alphabets it can be computed in $\mathcal{O}(n)$ time [Far97]. The sparse suffix tree of T for some set $B \subseteq \{1, \ldots, n\}$ of sample positions is the compacted trie containing exactly the suffixes $\{T[i..n] \mid i \in B\}$. It can be stored in $\mathcal{O}(|B|)$ words of memory.

We assume that T is terminated by some special symbol $T[n] = \$$ that occurs nowhere else in T. This ensures that, in a (sparse) suffix tree, each suffix is spelled by a leaf, and we label the leaves with the respective starting positions of the suffixes. Note that for any two leaves $i \neq j$, their lowest common ancestor (i.e., the deepest node that is an ancestor of both i and j) spells a string of length $\textsc{lce}(i, j)$.

11.2 Lower Bound for Testing Square-Freeness Over General Unordered Alphabet

In this section, we show that $\Omega(n \log \sigma)$ symbol comparisons are needed to test square-freeness over general unordered alphabet. We use the adversarial method as described in Section 2.3.1, which we briefly recall now. The present model of computation may be interpreted as follows. An algorithm working on a string over general unordered alphabet has no access to the actual string. Instead, it can only ask an oracle whether or not there are identical symbols at two positions. The number of questions asked is exactly the number of performed comparisons. In order to show a lower bound on the number of comparisons required to solve some problem, we describe an adversary that takes over the role of the oracle, forcing the algorithm to perform as many symbol comparisons as possible.

Similar to the conflict lists in the proof of Theorem 2.5 *(ii)* and Theorem 2.3 *(ii)*, we use a conflict graph $G = (V, E)$ with $V = \{1, \dots, n\}$ and $E \subseteq V^2$ to keep track of the answers given by the adversary. The nodes directly correspond to the positions of the string. Initially, we have $E = \emptyset$ and all nodes are colorless, which formally means that they have color $\gamma(i) = \perp$. During the algorithm execution, the adversary may assign colors from the set $\Sigma = \{0, \dots, n-1\}$ to the nodes, which can be seen as permanently fixing the alphabet symbol at the corresponding position (i.e., each node gets colored at most once). The rule used for coloring nodes will be described later. Apart from the coloring rule, the general behavior of the adversary is as follows. Whenever the algorithm asks whether $T[i] = T[j]$ holds, the adversary answers "yes" if and only if $\gamma(i) = \gamma(j) \neq \perp$. Otherwise, it answers "no" and inserts an edge (i, j) into E. Whenever the adversary assigns the color of a node, it has to choose a color that is not used by any of the adjacent nodes in the conflict graph. This ensures that the coloring does not contradict the answers given in the past.

Let us define a set $\mathcal{T} \subseteq \Sigma^n$ of strings that is consistent with the answers given by the adversary. A string $T \in \Sigma^n$ is a member of $\mathcal{T}$ if

$$\forall i \in V : \gamma(i) \in \{\perp, T[i]\} \quad \wedge \quad \forall i, j \in V : (T[i] = T[j]) \implies (i, j) \notin E.$$

Note that $\mathcal{T}$ changes over time. Initially (before the algorithm starts), we have $\mathcal{T} = \Sigma^n$. With every question asked, the algorithm might eliminate some strings from $\mathcal{T}$. However, there is always at least one string in $\mathcal{T}$, which can be obtained, e.g., by coloring each colorless node in a previously entirely unused color.

Testing Square-Freeness In order to obtain a lower bound for testing square-freeness, we will ensure that $\mathcal{T}$ always contains a square-free string with at most σ distinct symbols. At the same time, we try to ensure that $\mathcal{T}$ also contains a string with at least one square. We will show that we can maintain this state until at least $n \ln \sigma - 3.6n$ comparisons have been performed.

The string (or rather family of strings) constructed by the adversary is organized in $\left\lceil \frac{4n}{\sigma} \right\rceil$ non-overlapping blocks of length $\frac{\sigma}{4}$ (we assume $\frac{\sigma}{4} \in \mathbb{N}$ and $8 \leq \sigma \leq n$). Each

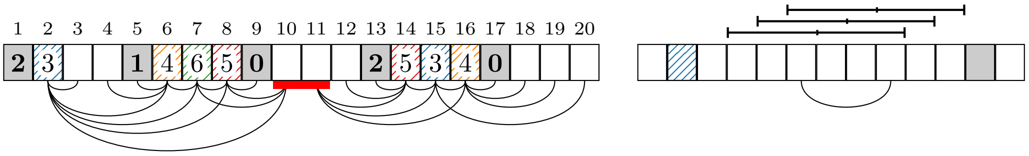


Figure 11.1: Example conflict graph of the adversary described in Section 11.2. The alphabet $\{0,\ldots,15\}$ is of size $\sigma = 16$. The blocks are of length $\frac{\sigma}{4} = 4$. The gray nodes are exactly the starting positions of the blocks and contain the symbols of the ternary Thue-Morse sequence $v = 2,1,0,2,0,1,2,\ldots$, which is square-free. We assume that the colored nodes were colored in the following order: $2,6,8,7,15,16,14$. At the time of coloring node 8, we had to avoid colors $0,1,2$ (because they are reserved for the separator positions), 3 (because the adjacent node 2 already has color 3), and 4 (because node 6 is in the same block and already has color 4). The algorithm has not eliminated all squares yet. For example, nodes 10 and 11 with absent edge $(10,11) \notin E$ are adjacent to nodes of colors $\{3,6,5\} \cup \{5,3,4\}$. Thus, any of the colors $\{0,1,2\} \cup \{7,\ldots,15\}$ can be assigned to both nodes, enforcing the square $T[10..11]$. As visualized on the right, an edge of length ℓ eliminates at most ℓ squares.

block begins with a special separator symbol. More precisely, the first symbol of the kth block is the kth symbol of a ternary square-free string over the alphabet $\{0,1,2\}$ (e.g., the distance between the kth and $(k+1)$th occurrence of 0 in the Prouhet-Thue-Morse sequence, also known as the ternary Thue-Morse-Sequence, see [AS98, Corollary 1] or [Webf]). Initially, the adversary colors the nodes that correspond to the separator positions in their respective colors from $\{0,1,2\}$. All remaining nodes will later get colors other than $\{0,1,2\}$. Any fragment crossing a block boundary can be projected on the colors $\{0,1,2\}$, and by construction the string cannot contain a square. Thus, the separator symbols ensure that there is no square crossed by a block boundary, which implies that the string is square-free if and only if each of its blocks is square-free.

During the algorithm execution, we use the following coloring rule. The available colors are $\{3,\ldots,\sigma-1\}$. Whenever the degree of a node becomes $\frac{\sigma}{4}$, we assign its color. We avoid not only the at most $\frac{\sigma}{4}$ colors of already colored neighbors in the conflict graph, but also the less than $\frac{\sigma}{4}$ colors of nodes within the same block (due to $\sigma \geq 8$, there are at least $\sigma - 3 - \frac{\sigma}{2} \geq 1$ colors available). An example of the conflict graph is provided in Figure 11.1. At any moment in time, we could hypothetically complete the coloring by assigning one of the colors $\{3,\ldots,\sigma-1\}$ to each colorless node, avoiding colors of adjacent nodes and colors of nodes in the same block. Afterwards, each node holds one of the σ colors, but no two nodes within the same block have the same color. Hence each block is square-free, and therefore $\mathcal{T}$ always contains a square-free string with at most σ distinct symbols.

Now we consider the state of the conflict graph *after the algorithm has terminated.* We are particularly concerned with consecutive ranges of colorless nodes. The following lemma states that for each such range, the algorithm either performed many comparisons, or we can enforce a square within the range.

Lemma 11.7. *Let $R = [i, j] \subset V$ be a range of $m = j - i + 1$ colorless nodes in the conflict graph. Then either $|E \cap R^2| \geq \sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$, or there is a string $T \in \mathcal{T}$ with at most σ distinct symbols such that $T[i..j]$ contains a square.*

Proof. We say that an integer interval $[x, x + 2\ell - 1]$ with $i \leq x < (x + 2\ell - 1) \leq j$ has been *eliminated* if for some y with $x \leq y < x + \ell$ there is an edge $(y, y + \ell)$ in the conflict graph. If such an edge exists, then (by the definition of $\mathcal{T}$) all strings $T \in \mathcal{T}$ satisfy $T[y] \neq T[y + \ell]$. Thus $T[x..x + 2\ell - 1]$ is not a square for any of them.

Now we show that, if $[x, x + 2\ell - 1]$ has not been eliminated, then there exists a string $T \in \mathcal{T}$ such that $T[x..x + 2\ell - 1]$ is a square. For this purpose, consider any position y with $x \leq y < x + \ell$, i.e., a position in the first half of the potential square. Since $[x, x + 2\ell - 1]$ has not been eliminated, $(y, y + \ell)$ is not an edge in the conflict graph. It follows that we could assign the same color to y and $y + \ell$. We only have to avoid the at most $2 \cdot (\frac{\sigma}{4} - 1)$ colors of adjacent nodes of both y and $y + \ell$ in the conflict graph. Thus there are $\frac{\sigma}{2} + 2$ appropriate colors that can be assigned to both nodes. Unlike during the algorithm execution, we do not need to avoid the special separator colors or the colors in the same block; since we are trying to enforce a square, we do not have to worry about accidentally creating one. By applying this coloring scheme for all possible choices of y, we enforce that all strings $T \in \mathcal{T}$ have a square $T[x..x + 2\ell - 1]$. Note that by coloring additional nodes after the algorithm terminated, we only remove elements from $\mathcal{T}$. Thus, the strings with square $T[x..x + 2\ell - 1]$ were already in $\mathcal{T}$ when the algorithm terminated. It follows that, if the algorithm actually guarantees square-freeness, then it must have eliminated all possible intervals $[x, x + 2\ell - 1]$ with $i \leq x < (x + 2\ell - 1) \leq j$.

While each interval needs at least one edge to be eliminated, a single edge eliminates multiple intervals. However, all the intervals eliminated by an edge must be of the same length. Now we give a lower bound on the number of edges needed to eliminate all intervals of length 2ℓ. Any edge $(y, y + \ell)$ eliminates ℓ intervals, namely the intervals $[x, x + 2\ell - 1]$ that satisfy $x \leq y < x + \ell$. Within R, we have to eliminate $m - 2\ell + 1$ intervals of length 2ℓ, namely the intervals $[x, x + 2\ell - 1]$ that satisfy $i \leq x \leq j - 2\ell + 1$ (see right side of Figure 11.1). Thus, we need at least $\frac{m-2\ell+1}{\ell}$ edges to eliminate all squares of length 2ℓ. Finally, by summing over all possible values of ℓ, we need at least $\sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$ edges to eliminate all intervals in R. Note that the edges used for elimination have both endpoints in R, and are thus contained in $E \cap R^2$. Consequently, if $|E \cap R^2| < \sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$, then not all intervals have been eliminated, and there is a string in $\mathcal{T}$ that contains a square. □

Finally, we show that the algorithm either performed at least $\Omega(n \log \sigma)$ comparisons, or there is a string $T \in \mathcal{T}$ that contains a square. Let $c_1, c_2, \ldots, c_k$ be exactly the colored nodes. Initially (before the algorithm execution), the adversary colored $\left\lceil \frac{4n}{\sigma} \right\rceil$ nodes (corresponding to the separators of the Prouhet-Thue-Morse sequence). Thus $k \geq \left\lceil \frac{4n}{\sigma} \right\rceil$, and there are $k - \left\lceil \frac{4n}{\sigma} \right\rceil$ nodes that have been colored after their degree reached $\frac{\sigma}{4}$. Therefore, the sum of degrees of all colored nodes is at least $(k - \left\lceil \frac{4n}{\sigma} \right\rceil) \cdot \frac{\sigma}{4} \geq \frac{\sigma k - 4n - \sigma}{4} \geq \frac{\sigma k - 5n}{4}$. Each comparison may increase the degree of two nodes by one. Thus, the colored nodes account for at least $\frac{\sigma k - 5n}{8}$ comparisons. There are k non-overlapping maximal colorless ranges of nodes, namely the range $\{c_i + 1, \ldots, c_{i+1} - 1\}$ for every $i \in [1, k]$, with auxiliary value $c_{k+1} = n + 1$. If all the

$T \in \mathcal{T}$ are square-free, then each range accounts for $e_i = \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell}$ edges by Lemma 11.7, where $m_i = c_{i+1} - c_i - 1$. No edge gets counted more than once because the ranges are non-overlapping, and both endpoints of the respective edges are within the range. Thus, to verify square-freeness, the algorithm must have performed at least $\sum_{i=1}^{k} e_i + \frac{\sigma k - 5n}{8}$ comparisons. The remainder of the proof consists of simple algebra. First, we provide a lower bound on e_i (explained below).

$$
\begin{aligned}
e_i = \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell} = \sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{m_i - 2\ell + 1}{\ell} &\geq (m_i + 1) \left(\left(\sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{1}{\ell} \right) - 1 \right) \\
&> (m_i + 1) \cdot (\ln \frac{m_i}{2} - \frac{1}{2}) \\
&= (m_i + 1) \cdot \ln \frac{m_i}{2\sqrt{e}} \\
&\geq (m_i + 1) \cdot \ln \frac{m_i + 1}{2.5\sqrt{e}}
\end{aligned}
$$

We can replace $\lfloor m_i/2 \rfloor$ with $\lceil m_i/2 \rceil$ because if m_i is odd the additional summand equals zero. The first inequality uses simple arithmetic operations. The second inequality uses the classical lower bound $(\ln x + \frac{1}{2}) < H_x$ on harmonic numbers. The last inequality holds for $m_i \geq 4$. For $m_i < 4$ the result becomes negative and is thus still a correct lower bound on the number of comparisons. We obtain:

$$
\begin{aligned}
\underbrace{\sum_{i=1}^{k} (m_i + 1) \cdot \ln \frac{m_i + 1}{2.5\sqrt{e}}}_{\text{comparisons within colorless ranges}} + \underbrace{\frac{\sigma k - 5n}{8}}_{\text{comparisons for colored nodes}} &\geq n \cdot \ln \frac{n}{2.5\sqrt{e}k} + \frac{\sigma k - 5n}{8} \\
&= n \cdot \ln \frac{\sigma}{2.5\sqrt{e}x} + \frac{xn - 5n}{8} \\
&= n \cdot \ln \sigma + n \cdot \left(\frac{x - 5}{8} - \ln(2.5\sqrt{e}x) \right) \\
&> n \cdot \ln \sigma - 3.12074n
\end{aligned}
$$

The first step follows from $\sum_{i=1}^{k} (m_i + 1) = n$ and the log sum inequality (see [CT06, Theorem 2.7.1]). In the second step we replace k by using $x = \frac{\sigma k}{n}$. The third step uses simple arithmetic operations. The last step is reached by substituting $x = 8$, which minimizes the equation. Finally, we assumed that σ is divisible by 4. We account for this by adjusting the lower bound to $n \ln(\sigma - 3) - 3.12074n$, which is larger than $n \ln \sigma - 3.6n$ for $\sigma \geq 8$.

Theorem 11.1. *Let $n, \sigma \in \mathbb{N}^+$ with $8 \leq \sigma \leq n$ be fixed. There is no deterministic algorithm that performs at most $n \ln \sigma - 3.6n = \mathcal{O}(n \log \sigma)$ equality comparisons in the worst case, and determines whether a length-n string that contains at most σ distinct symbols over general unordered alphabet is square-free.*

11.3 Testing Square-Freeness in $\mathcal{O}(n \log \sigma)$ Comparisons

In this section, we consider the problem of testing square-freeness. We introduce an algorithm that decides whether a given string is square-free using $\mathcal{O}(n \log \sigma)$

comparisons, matching the lower bound from Theorem 11.1. Note that this algorithm is not yet time efficient because, apart from the performed symbol comparisons, it uses other operations that are expensive in the word RAM model. A time efficient implementation of the algorithm will be presented in Section 11.4, where we first achieve $\mathcal{O}(n \log \sigma + n \log^* n)$ time, and then improve this to $\mathcal{O}(n \log \sigma)$ time. In Section 11.5, we generalize the result to compute all runs in the same time complexity.

11.3.1 Sparse Suffix Trees and Difference Covers

As mentioned earlier, we will detect squares using an approximate version of the LZ factorization. In order to compute this factorization efficiently, we need to be able to construct sparse suffix trees in few symbol comparisons. From now on, whenever we say that some cost is shared by all invocations of some lemma, we implicitly mean that the cost is shared by all invocations of this lemma *on the same text T*.

Lemma 11.8. *The sparse suffix tree containing any b suffixes $T[i_1..n], \dots, T[i_b..n]$ of $T[1..n]$ can be constructed using $\mathcal{O}(b\sigma \log b)$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

Proof. We maintain a union-find structure over the positions of $T[1..n]$. Initially, each position is in a separate component. Whenever we have to perform an equality comparison between symbols $T[x]$ and $T[y]$, we check if x and y are in the same component of the union-find structure, and if so immediately return that $T[x] = T[y]$ without performing any comparisons. Otherwise, we perform the comparison and, if the outcome of the comparison is $T[x] = T[y]$, we merge the components of x and y. We merge components at most n times, which means that the total number of issued queries with positive answer, over all invocations of the lemma, is less than n, and it remains to bound the number of issued queries with negative answer.

We insert the suffixes $T[i_j..n]$ one-by-one into an initially empty sparse suffix tree. To insert the next suffix, we descend from the root of the tree to identify the node u that corresponds to the longest common prefix between $T[i_j..n]$ and any of the already inserted suffixes. We then make u explicit unless it is explicit already, and add an edge from u to a new leaf corresponding to the whole $T[i_j..n]$. We say that the insertion procedure terminates at u. Node u can be identified using only $\mathcal{O}(\sigma \log b)$ comparisons with negative answers as follows. Let v be the current node (initially the root), and let $v_1, \dots, v_d$ be its children, where $d \leq \sigma$. Here, v can be either explicit or implicit, in the latter case $d = 1$. We arrange the children of v so that the number of leaves in the subtree rooted at v_1 is at least as large as the number of leaves in the subtree rooted at any other child of v. Then, we compare the symbol on the edge leading to v_1 with the corresponding symbol of the current suffix. If they are equal we continue with v_1, otherwise we compare the symbols on the edges leading to $v_2, \dots, v_d$ with the corresponding symbol of the current suffix one-by-one. Then, we either continue with some v_j, $j \geq 2$, or terminate at v. To bound the number of comparisons with negative answer, observe that such comparisons only occur when we either terminate at v or continue with v_j, $j \geq 2$. Whenever we continue with v_j, $j \geq 2$, the number of leaves in the current subtree rooted at v_j decreases at least by a factor of 2 compared to subtree rooted at v (as the subtree rooted at v_1 had the largest number of leaves). Thus, during the whole descent from the root performed during an insertion this can happen only at most $1 + \log_2 b$ times. Every time we do

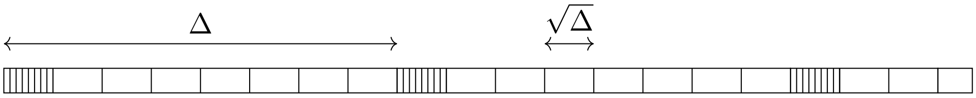


Figure 11.2: Positions in a Δ-difference cover.

not continue in the subtree v_1 we might have up to $d \leq \sigma$ comparisons with negative answer, thus the total number of such comparisons is as claimed[1]. □

Now we describe the sample positions that we will later use to compute the approximate LZ factorization. A set $S \subseteq \mathbb{N}$ is called a t-cover of $\{1, \ldots, n\}$ if there is a constant-time computable function h such that, for any $i, j \in [1, n-t+1]$, we have $h(i,j) \in [0,t)$ and $i + h(i,j), j + h(i,j) \in S$. A possible construction of t-covers is given by the lemma below, and visualized in Figure 11.2.

Lemma 11.9. *For any n and $t \leq n$, there exists a t-cover $D(t)$ of $\{1, \ldots, n\}$ with size $\mathcal{O}(n/\sqrt{t})$. Furthermore, its elements can be enumerated in time proportional to their number.*

Proof. We use the well-known combinatorial construction known as difference covers, see, e.g. [Mae85]. Let $r = \lfloor \sqrt{t} \rfloor$ and define $D(t) = \{i \in \{1, \ldots, n\} : i \bmod r = 0 \text{ or } i \bmod r^2 \in \{0, \ldots, r-1\}\}$. By definition, $|D(t)| \leq \lfloor n/r \rfloor + \lfloor n/r^2 \rfloor r = \mathcal{O}(n/r) = \mathcal{O}(n/\sqrt{t})$. The function $h(i,j)$ is defined as $a + b \cdot r$, where $a = (r - i) \bmod r$ and $b = (r - \lfloor (j+a)/r \rfloor) \bmod r$. Note that $i + h(i,j) \leq n$ and $j + h(i,j) \leq n$. Then, $i + (a + b \cdot r) = 0 \pmod r$, while $\lfloor (j + (a + b \cdot r))/r \rfloor = \lfloor (j+a)/r + b \rfloor = 0 \bmod r$ implies $j + h(i,j) \bmod r^2 \in \{0, \ldots, r-1\}\}$, thus $i + h(i,j), j + h(i,j) \in D(t)$ as required. □

11.3.2 Detecting Squares with a Δ-Approximate LZ Factorization

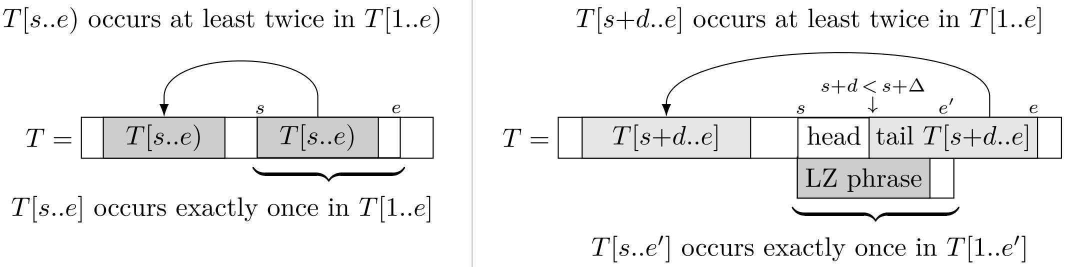
Now we are ready to introduce the new approximate version of the Lempel-Ziv factorization.

Definition 11.10 (Δ-approximate LZ factorization)**.** For a positive integer parameter Δ, the fragment $T[s..e]$ is a Δ-approximate LZ phrase if it can be split into a head and a tail $T[s..e] = \mathsf{head}(T[s..e])\mathsf{tail}(T[s..e])$ such that $|\mathsf{head}(T[s..e])| < \Delta$ and additionally

- $\mathsf{tail}(T[s..e])$ is either empty or occurs at least twice in $T[1..e]$, and
- the unique (standard) LZ phrase $T[s..e']$ starting at position s satisfies $e' - 1 \leq e$.

In a Δ-approximate LZ factorization $T = b_1 b_2 \ldots b_z$, each factor b_i is a Δ-approximate phrase $T[s..e]$ with $s = 1 + \sum_{j=1}^{i-1} |b_j|$ and $e = \sum_{j=1}^{i} |b_j|$.

[1] In the descent, if all children are sorted according to their subtree size, the number of comparisons decreases to $\mathcal{O}(b(\sigma/\log\sigma)\log b)$, but this is irrelevant for our final algorithm.


$T[s..e)$ occurs at least twice in $T[1..e)$
$T =$
$T[s..e)$
s
e
$T[s..e)$
$T[s..e]$ occurs exactly once in $T[1..e]$
$T[s{+}d..e]$ occurs at least twice in $T[1..e]$
$s{+}d < s{+}\Delta$
s
e'
e
$T =$
$T[s{+}d..e]$
head
tail $T[s{+}d..e]$
LZ phrase
$T[s..e']$ occurs exactly once in $T[1..e']$

(a) A standard LZ phrase $T[s..e]$.

(b) A Δ-approximate phrase $T[s..e]$. The (standard) LZ phrase at position s is $T[s..e']$, and it holds $e' - 1 \leq e$.

Figure 11.3: Illustration of an LZ-phrase and a Δ-approximate phrase.

Note that a standard LZ phrase is not a Δ-approximate phrase. Also, while the LZ phrase starting at each position (and thus also the LZ factorization) is uniquely defined, there may be multiple different Δ-approximate phrases starting at each position. This also means that a single string can have multiple different Δ-approximate factorizations. The definitions of both standard and Δ-approximate LZ phrases are illustrated in Figure 11.3.

The intuition behind the above definition is that constructing the Δ-approximate LZ factorization becomes easier for larger values of Δ. In particular, for $\Delta = n$ one phrase is enough. We formalize this in the following lemma, which is made more general for the purpose of obtaining the final result in this section.

Lemma 11.11. *For any parameter $\Delta \in [1, m]$, a Δ-approximate LZ factorization of any fragment $T[x..y]$ of length m can be computed using $\mathcal{O}(m\sigma \log m/\sqrt{\Delta})$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

Proof. By Lemma 11.9, there is a Δ-cover $D(\Delta)$ of $\{1, \ldots, n\}$ with size $\mathcal{O}(n/\sqrt{\Delta})$. Let $S = (D(\Delta) \cap \{x, x+1, \ldots, y\}) = \{i_1, i_2, \ldots, i_b\}$. It is straightforward to verify that the construction additionally guarantees $b = \mathcal{O}(m/\sqrt{\Delta})$. We apply Lemma 11.8 on the suffixes $T[i_1..n], \ldots, T[i_b..n]$ to obtain their sparse suffix tree using $\mathcal{O}(b\sigma \log b)$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma. The sparse suffix tree allows us to obtain the longest common prefix of any two fragments $T[i..y]$ and $T[j..y]$, for $i, j \in S$, with no additional comparisons. By the properties of $D(\Delta)$, for any $i, j \in [x, y - \Delta + 1]$ we have $h(i, j) \in [0, \Delta)$ and $i + h(i, j), j + h(i, j) \in S$.

We compute the Δ-approximate LZ factorization of $T[x..y]$ phrase-by-phrase. Denoting the remaining suffix of the whole $T[x..y]$ by $T[x'..y]$, we need to find $y' \in [x', y]$ such that $T[x'..y']$ is a Δ-approximate phrase. This is done as follows. We iterate over every $x'' \in [x', x' + \Delta) \cap S$. For every such x'', we consider every $a' \in [x, x') \cap S$, and compute the length ℓ of the longest common prefix of $T[x''..y]$ and $T[a'..y]$. Among all such x'', a' we choose the pair that results in the largest value of $x'' - x' + \ell - 1$ and choose the next phrase to be $T[x'..(x'' + \ell - 1)]$, with the head being $T[x'..(x'' - 1)]$ and the tail $T[x''..(x'' + \ell - 1)]$. Finally, if there is no such pair, or the value of $x'' - x' + \ell - 1$ corresponding to the found pair is less than $\Delta - 2$, we take the next phrase to be $T[x'.. \min\{x' + \Delta - 1, y\}]$ (with empty tail). Selecting such a pair requires no extra comparisons, as for every $x'', a' \in S$ we can use the sparse suffix tree to compute ℓ. While it is clear that the generated Δ-approximate phrase has the required form, we need to establish that it is sufficiently long.

Let $T[x'..y'']$ be the (unique) standard LZ phrase of $T[x..y]$ that is prefix of $T[x'..y]$. If $y'' < x' + \Delta - 1$ then we only need to ensure that the generated Δ-approximate phrase is of length at least $\min\{\Delta - 1, y - x' + 1\}$, which is indeed the case. Therefore, it remains to consider the situation when $y'' \geq x' + \Delta - 1$. Let $T[a..b]$ be a previous occurrence of $T[x'..(y''-1)]$ in $T[x..y]$ (because $T[x'..y'']$ is a phrase this is well-defined). Thus, $T[a..b] = T[x'..(y''-1)]$ and $a < x'$. Because $y'' \geq x' + \Delta - 1$ and $y'' \leq y$, as explained above $0 \leq h(a, x') < \Delta$ and $a + h(a, x'), x' + h(a, x') \in S$. We will consider $x'' = x' + h(a, x')$ and $a' = a + h(a, x')$ in the above procedure. Next, $T[a'..b] = T[x''..(y''-1)]$, so when considering this pair we will obtain $\ell \geq |T[x''..(y''-1)]|$. Thus, for the found pair we will have $x'' + \ell - 1 \geq y'' - 1$ as required in the definition of a Δ-approximate phrase. □

Next, we show that even though the Δ-approximate LZ factorization does not capture all distinct squares (as it is the case for the standard LZ factorization), it is still helpful in detecting all squares that are sufficiently long relative to Δ. A crucial component is the following property of the Δ-approximate LZ factorization.

Lemma 11.12. *Let $b_1 b_2 \dots b_z$ be a Δ-approximate LZ factorization of a string T. For every square $T[s..s+2\ell-1]$ of length $2\ell \geq 8\Delta$, there is at least one phrase b_i with $|tail(b_i)| \geq \frac{\ell}{4} \geq \Delta$ such that $tail(b_i)$ and the right-hand side $T[s+\ell..s+2\ell-1]$ of the square intersect.*

Proof. Assume that all tails that intersect $T[s+\ell..s+2\ell-1]$ are of length less than $\frac{\ell}{4}$, then the respective phrases of these tails are of length at most $\frac{\ell}{4} + \Delta - 1$ (because each head is of length less than Δ). This means that $T[s+\ell..s+2\ell-1]$ intersects at least $\left\lceil \ell/(\frac{\ell}{4} + \Delta - 1) \right\rceil \geq \left\lceil \ell/(\frac{\ell}{2} - 1) \right\rceil = 3$ phrases. Thus, there is some phrase $b_i = T[x..y]$ properly contained in $T[s+\ell..s+2\ell-1]$, formally $s + \ell < x \leq y < s + 2\ell - 1$. However, this contradicts the definition of the Δ-approximate LZ factorization because $T[x..s+2\ell]$ is the prefix of a standard LZ phrase (due to $T[x..s+2\ell-1] = T[x-\ell..s+\ell-1]$), and the Δ-approximate phrase $b_i = T[x..y]$ must satisfy $y \geq s + 2\ell - 1$. The contradiction implies that $T[s+\ell..s+2\ell-1]$ intersects a tail of length at least $\frac{\ell}{4} \geq \Delta$. □

Lemma 11.13. *Given a Δ-approximate LZ factorization $T = b_1 b_2 \dots b_z$, we can detect whether there is at least one square of length $\geq 8\Delta$ and report such a square in $\mathcal{O}\left(z + \sum_{|tail(b_i)| \geq \Delta} |tail\,(b_i)|\right)$ time and $\mathcal{O}\left(\sum_{|tail(b_i)| \geq \Delta} |tail\,(b_i)|\right)$ comparisons.*

Proof. We consider each phrase $b_i = T[a_1..a_3]$ with $\mathsf{head}(b_i) = T[a_1..a_2-1]$ and $\mathsf{tail}(b_i) = T[a_2..a_3]$ separately. Let $k = |\mathsf{tail}(b_i)|$. If $k \geq \Delta$, we apply Lemma 11.6 to $x_1 = T[a_2 - 8k..a_2 - 1]$ and $y_1 = T[a_2..a_3 + 4k - 1]$, as well as $x_2 = T[a_2 - 8k..a_3 - 1]$ and $y_2 = T[a_3..a_3 + 4k - 1]$ trimmed to $T[1..n]$. This takes $\mathcal{O}(|\mathsf{tail}(b_i)|)$ time and comparisons, or $\mathcal{O}\left(\sum_{|\mathsf{tail}(b_i)| \geq \Delta} |\mathsf{tail}\,(b_i)|\right)$ time and comparisons for all phrases. We need additional $\mathcal{O}(z)$ time to check if $k \geq \Delta$ for each phrase.

Now we show that the described strategy detects a square of size at least 8Δ. Let $T[s..s+2\ell-1]$ be any such square. Due to Lemma 11.12, the right-hand side $T[s+\ell..s+2\ell-1]$ of this square intersects some tail $\mathsf{tail}(b_i) = T[a_2..a_3]$ of length

$k = |\mathsf{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$. Due to the intersection, we have $a_2 \leq s+2\ell-1$ and $a_3 \geq s+\ell$. Thus, when processing b_i and applying Lemma 11.6, the starting position of x_1 and x_2 satisfies $a_2 - 8k \leq s+2\ell-1-8\frac{\ell}{4} = s-1$, while the end position of y_1 and y_2 satisfies $a_3+4k-1 \geq s+\ell+4\frac{\ell}{4}-1 = s+2\ell-1$. Therefore, the square is entirely contained in the respective fragments corresponding to x_1y_1 and x_2y_2. If $s < a_2 \leq s+2\ell-1$, we find the square with our choice of x_1 and y_1. If $s < a_3 \leq s+2\ell-1$, we find the square with our choice of x_2 and y_2. Otherwise, $T[s..s+2\ell-1]$ is entirely contained in tail $T[a_2..a_3]$, and we find another occurrence of the square further to the left. □

11.3.3 Simple Algorithm for Detecting Squares

Now we have all the tools to introduce our simple method for testing square-freeness of $T[1..n]$ using $\mathcal{O}(n(\log\sigma + \log\log n))$ comparisons, assuming that σ is known in advance. Let $\Delta = (\sigma \log n)^2$. We partition $T[1..n]$ into blocks of length 8Δ, and denote the kth block by B_k. A square of length at most 8Δ can be found by invoking Main and Lorentz's algorithm from Theorem 11.5 on B_1B_2, B_2B_3, and so on. This takes $\mathcal{O}(\Delta\log\Delta) = \mathcal{O}(\Delta(\log\sigma + \log\log n))$ comparisons for each pair of adjacent blocks, or $\mathcal{O}(n(\log\sigma + \log\log n))$ comparisons in total. It remains to test for squares of length exceeding 8Δ. This is done by first invoking Lemma 11.11 to compute a Δ-approximate LZ factorization of $T[1..n]$ using $\mathcal{O}(n\sigma\log n/\sqrt{\Delta}) = \mathcal{O}(n)$ comparisons, and then using Lemma 11.13, which adds another $\mathcal{O}(n)$ comparisons. The total number of comparisons is dominated by the $\mathcal{O}(n(\log\sigma + \log\log n))$ comparisons needed to apply Theorem 11.5 to the block pairs.

11.3.4 Improved Algorithm for Detecting Squares

We are now ready to describe the algorithm that uses only $\mathcal{O}(n\log\sigma)$ comparisons without knowing the value of σ. Intuitively, we will proceed in phases, trying to "guess" the value of σ. We first observe that Lemma 11.11 can be extended to obtain the following.

Lemma 11.14. *There is an algorithm that, given any parameter $\Delta \in [1, m]$, estimate $\tilde{\sigma}$ and fragment $T[x..y]$ of length m, uses $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma, and either computes a Δ-approximate LZ factorization of $T[x..y]$ or determines that $\sigma > \tilde{\sigma}$.*

Proof. We run the procedure described in the proof of Lemma 11.11 and keep track of the number of comparisons with negative answer. As soon as it exceeds $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ (where the constant follows from the complexity analysis) we know that necessarily $\sigma > \tilde{\sigma}$, so we can terminate. Otherwise, the algorithm obtains a Δ-approximate LZ factorization using $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ comparisons. Comparisons with positive answer are paid for globally. □

Now we describe how to find any square using $\mathcal{O}(n\log\sigma)$ comparisons. We define the sequence $\sigma_t = 2^{2^{\lceil\log\log n\rceil - t}}$, for $t = 0, 1, \ldots, \lceil\log\log n\rceil$. We observe that $\sigma_{t-1} = (\sigma_t)^2$, and proceed in phases corresponding to the values of t. In the tth phase we are guaranteed that any square of length at least $(\sigma_t)^2$ has been already detected, and we aim to detect square of length less than $(\sigma_t)^2$, and at least σ_t. We partition the whole $T[1..n]$ into blocks of length $(\sigma_t)^2$, and denote the kth block by B_k. A square of length less than $(\sigma_t)^2$ is fully contained within some two consecutive blocks $B_i B_{i+1}$, hence we consider each such pair $B_1 B_2$, $B_2 B_3$, and so on. We first apply Lemma 11.14 with $\Delta = \sigma_t/8$ and $\tilde{\sigma} = (\sigma_t)^{1/4}/\log(\sigma_t)$ to find an $(\sigma_t/8)$-approximate LZ factorization of the corresponding fragment of $T[1..n]$, and then use Lemma 11.13 to detect squares of length at least σ_t. We cannot always afford to apply Lemma 11.13 to all block pairs. Thus, we have to deactivate some of the blocks, which we explain when analyzing the number of comparisons performed by the algorithm. If any of the calls to Lemma 11.14 in the current phase detects that $\sigma > \tilde{\sigma}$, we switch to applying Main and Lorentz's algorithm from Theorem 11.5 on every pair of blocks $B_i B_{i+1}$ of the current phase and then terminate the whole algorithm.

We now analyze the total number of comparisons, ignoring the $\mathcal{O}(n)$ comparisons shared by all invocations of Lemma 11.14. Throughout the tth phase, we use $\mathcal{O}(n \cdot \tilde{\sigma} \log \sigma_t / \sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4}/\log(\sigma_t) \cdot \log(\sigma_t)/\sqrt{\sigma_t}) = \mathcal{O}(n/(\sigma_t)^{1/4})$ comparisons to construct the Δ-approximate factorizations (using Lemma 11.14) until we either process all pairs of blocks or detect that $\sigma > (\sigma_t)^{1/4}/\log(\sigma_t)$. In the latter case, we finish off the whole computation using $\mathcal{O}(n \log(\sigma_t))$ comparisons (with Theorem 11.5), and by assumption on σ this is $\mathcal{O}(n \log \sigma)$ as required. Until this happens (or until we reach phase $t = \lceil \log \log n \rceil - 3$ where $\sigma_t \le 256$), we use $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4})$ comparisons to construct the Δ-approximate factorizations, for some $t' \in [0, \lceil \log \log n \rceil]$. To analyze the sum, we need the following bound (made more general for the purpose of the next section).

Lemma 11.15. *For any $0 \le x \le y$ and $c \ge 0$ we have $\sum_{i=x}^{y} 2^{ic}/2^{2^i} = \mathcal{O}(2^{xc}/2^{2^x})$.*

Proof. The sequence of exponents 2^i is strictly increasing from $i = 0$, hence

$$\sum_{i=x}^{y} \frac{2^{ic}}{2^{2^i}} \le \sum_{i=2^x}^{2^y} \frac{i^c}{2^i} \le \sum_{i=2^x}^{\infty} \frac{i^c}{2^i} = \sum_{i=0}^{\infty} \frac{(2^x + i)^c}{2^{(2^x+i)}} \le \sum_{i=0}^{\infty} \frac{2^{xc} \cdot (i+1)^c}{2^{(2^x+i)}} = \frac{2^{xc}}{2^{2^x}} \cdot \sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}.$$

Since $\sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}$ is a series of positive terms, we can use Alembert's ratio test. The ratio $\frac{(i+2)^c}{2^{i+1}} \cdot \frac{2^i}{(i+1)^c} = \frac{1}{2} \frac{(i+2)^c}{(i+1)^c}$ of consecutive terms tends to $\frac{1}{2}$ when i goes to the infinity, thus the series converges to a constant. □

Corollary 11.16. *For $t' \in [0, \lceil \log \log n \rceil]$, it holds*

$$\sum_{t=0}^{t'} n \cdot \mathrm{polylog}(\sigma_t)/(\sigma_t)^{1/4} = \mathcal{O}(n).$$

Proof. We have to show that $\sum_{t=0}^{t'} \log^c(\sigma_t)/(\sigma_t)^{1/4}) = \mathcal{O}(1)$ for any constant $c \ge 0$. We achieve this by splitting the sum and applying Lemma 11.15.

$$\begin{aligned}\sum_{t=0}^{t'} \frac{\log^c(\sigma_t)}{(\sigma_t)^{1/4}} &\le \sum_{t=0}^{\lceil \log \log n \rceil} \frac{(2^{\lceil \log \log n \rceil - t})^c}{(2^{2^{\lceil \log \log n \rceil - t}})^{1/4}} = \sum_{t=0}^{\lceil \log \log n \rceil} \frac{(2^t)^c}{(2^{2^t})^{1/4}} \\ &= \sum_{t=0}^{\lceil \log \log n \rceil} \frac{2^{tc}}{2^{2^{t-2}}} = \frac{1}{2^{2^{-2}}} + \frac{2^c}{2^{2^{-1}}} + 4 \cdot \sum_{t=0}^{\lceil \log \log n \rceil - 2} \frac{2^{tc}}{2^{2^t}} = \mathcal{O}(1)\end{aligned}$$

□

Thus, all invocations of Lemma 11.14 cause $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ comparisons. So far, we analyzed all comparisons except for the ones issued by Lemma 11.13, and the total number of comparisons is dominated by the $\mathcal{O}(n \log \sigma)$ comparisons needed when applying Theorem 11.5 to all block pairs.

Deactivating Block Pairs

It remains to analyze the number of comparisons used by Lemma 11.13 throughout all phases. As mentioned earlier, we cannot actually afford to apply Lemma 11.13 to all block pairs. Thus, we introduce a mechanism that deactivates some of the pairs.

First, note that there are $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^2) \subseteq \mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ block pairs in all phases. For each pair, we store whether it has been deactivated or not, where being deactivated broadly means that we do not have to investigate the pair because it does not contain a leftmost distinct square. For each block pair B_iB_{i+1} in the current phase t, we first check if it has been marked as deactivated. If not, we also check if it has been *implicitly* deactivated, i.e., if any of the two pairs from the previous phase that contain B_iB_{i+1} are marked as deactivated. If B_iB_{i+1} has been implicitly deactivated, then we mark it as deactivated and do not apply Lemma 11.14 and Lemma 11.13 (the implicit deactivation serves the purpose of propagating the deactivation to all later phases). Note that if some position of the string is not contained in any active block pair in some phase, then it is also not contained in any active block pair in all later phases. This is because always $\sigma_{t-1} = (\sigma_t)^2$ (with no rounding required), which guarantees that block boundaries of earlier phases do not intersect blocks of later phases.

We only apply Lemma 11.14 and then Lemma 11.13 to B_iB_{i+1} if the pair has neither explicitly nor implicitly been deactivated. When applying Lemma 11.13, a tail $T[a..a+\ell)$ contributes $\mathcal{O}(\ell)$ comparisons if $\ell \geq \Delta = \sigma_t/8$ (and otherwise it contributes no comparisons). As the entire fragment $T[a..a+\ell)$ occurs earlier, it cannot contain the leftmost occurrence of a square within T. Thus, any block pair (of any phase) contained in $T[a..a+\ell)$ also cannot contain such an occurrence, and thus such block pairs can be deactivated.

The mechanism used for deactivation works as follows. Let $T[a..a+\ell)$ be a tail contributing $\mathcal{O}(\ell)$ comparisons with $\ell \geq \Delta = \sigma_t/8$ in phase t. We mark all block pairs of phase $t+2$ that are entirely contained in $T[a..a+\ell)$ as deactivated. Note that blocks in phase $t+2$ are of length $\sqrt{\sigma_t}$, and consider the fragment $T[a+2\sqrt{\sigma_t}..a+\ell-2\sqrt{\sigma_t})$. In phase $t+2$, and by implicit deactivation in all later phases, this fragment overlaps (either partially or fully) only block pairs that have been deactivated. Thus, after phase $t+1$, we will never inspect any of the symbols in $T[a+2\sqrt{\sigma_t}..a+\ell-2\sqrt{\sigma_t})$ again. We say that tail $T[a..a+\ell)$ deactivated the fragment of length $\ell-4\sqrt{\sigma_t} = \Omega(\ell)$, which is positive until phase $t = \lceil \log\log n \rceil - 3$ because $\sigma_t > 256$. Since the number of deactivated positions is linear in the number of comparisons that the tail contributes to Lemma 11.13, it suffices to show that each position gets deactivated at most a constant number of times. In a single phase, any position gets deactivated at most twice. This is because the tails of each factorization do not overlap by definition, but the tails of the two factorizations of adjacent block pairs B_iB_{i+1} and $B_{i+1}B_{i+2}$ can overlap. If a position gets deactivated for the first time in phase t, then (as explained earlier) we will not consider it in any of the phases $t' \geq t+2$. Thus, it can only be that we deactivate the position again in phase $t+1$, but not in any of the later phases. In total, each position gets deactivated at most four times. Hence Lemma 11.13 contributes $\mathcal{O}(n)$ comparisons in total.

We have shown:

Lemma 11.3. *Testing square-freeness of a length-n string that contains σ distinct symbols over general unordered alphabet can be done using $\mathcal{O}(n \log \sigma)$ comparisons.*

11.4 Testing Square-Freeness in $\mathcal{O}(n \log \sigma)$ Time

In this section, we show how to implement the approach described in the previous section to work in $\mathcal{O}(n \log \sigma)$ time. The main difficulty is to efficiently implement the sparse suffix tree construction algorithm, and then compute a Δ-approximate factorization. We first how to obtain an $\mathcal{O}(n \log \sigma + n \log^* n)$ time algorithm that still uses only $\mathcal{O}(n \log \sigma)$ comparisons, and then further improve its running time to $\mathcal{O}(n \log \sigma)$.

11.4.1 Constructing the Sparse Suffix Tree and the Δ-Approximate LZ Factorization

To give an efficient algorithmic construction of the sparse suffix tree from Lemma 11.8, we will use a restricted version of LCEs, where a query $\text{ShortLCE}_x(i, j)$ (for any positive integer x) returns $\min(x, \text{LCE}(i, j))$. The following result was given by Gawrychowski et al. [Gaw+16]:

Lemma 11.17 (Lemma 14 in [Gaw+16])**.** *For a length-n string over general unordered alphabet*[a]*, a sequence of q queries* $\text{ShortLCE}_{4^{k_i}}$ *for $i \in \{1, \ldots, q\}$ can be answered online in total time $\mathcal{O}(n \log^* n + s)$ and $\mathcal{O}(n + q)$ comparisons*[b]*, where $s = \sum_{i=1}^{q}(k_i + 1)$.*

[a] Lemma 14 in [Gaw+16] does not explicitly mention that it works over general unordered alphabet. However, the proof of the lemma relies solely on equality tests.

[b] Lemma 14 in [Gaw+16] does not explicitly mention that it requires $\mathcal{O}(n + q)$ comparisons. However, they use a union-find approach where there can be at most $\mathcal{O}(n)$ comparisons with outcome "equal", and each LCE query performs only one comparison with outcome "not-equal", similarly to what we describe in the proof of Lemma 11.8.

In the lemma, apart from the $\mathcal{O}(n \log^* n)$ time, each $\text{ShortLCE}_{4^{k_i}}$ query accounts for $\mathcal{O}(k_i + 1)$ time. Note that we can answer the queries online, without prior knowledge of the number and length of the queries. Also, computing an LCE in a fragment $T[x..y]$ of length m trivially reduces to a $\text{ShortLCE}_{4^{\lceil \log_4 m \rceil}}$ query on T. Thus, we have:

Corollary 11.18. *A sequence of q longest common extension queries on a fragment $T[x..y]$ of length m over general unordered alphabet can be answered in $\mathcal{O}(q \log m)$ time, plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma. The number of comparisons is $\mathcal{O}(q)$, plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

While constructing the sparse suffix tree, we will maintain a heavy-light decomposition using a rebuilding scheme introduced by Gabow [Gab90]. Let $L(u)$ denote the number of leaves in the subtree of a node u. We use the following recursive construction of a heavy-light decomposition. Starting from a node r (initially the root of the tree), we find the deepest descendant node e such that $L(e) \geq \frac{5}{6}L(r)$ (possibly $e = r$). The path p from the root $r(p) = r$ to $e(p) = e$ is a heavy path. Any edge (u, v) on this path satisfies $L(v) \geq \frac{5}{6}L(u)$, and we call those edges *heavy*. As a consequence, a node u can have at most one child v such that (u, v) is heavy. For each edge (u, v) where u is on the heavy path and v is not, we recursively build a new heavy path construction starting from v.

When inserting a new suffix in our tree, we keep track of the insertion in the following way. For every root of a heavy path, we maintain the number $I(u)$ of insertions made in the subtree of u since we built the heavy-light decomposition of this subtree. When $I(u) \geq \frac{1}{6}L(u)$ we recalculate the values of $L(v)$ for all nodes v in the subtree of u and rebuild the heavy-light decomposition for the subtree of u.

This insures that, despite insertion, for any heavy path starting at node r and a node u on that heavy path, $L(u) \geq \frac{2}{3}L(r)$. When crossing a non-heavy edge the number of nodes in the subtree reduces by a constant fraction, which leads to the following property:

Observation 11.19. *The path from any node to the root crosses at most $\mathcal{O}(\log m)$ heavy paths.*

Additionally, rebuilding a subtree of size s takes $\mathcal{O}(s)$ time, and adding a suffix $T[i_j..y]$ to the tree increases $I(r)$ for each path p from the root r to the new leaf. Those are at most $\mathcal{O}(\log m)$ nodes, and thus maintaining the heavy path decomposition takes amortized time $\mathcal{O}(\log m)$ time per insertion.

With these building blocks now clearly defined, we are ready to describe the construction of the sparse suffix tree.

Lemma 11.20. *The sparse suffix tree containing any b suffixes $T[i_1..y], \ldots, T[i_b..y]$ of $T[x..y]$ with $m = |T[x..y]|$ can be constructed using $\mathcal{O}(b\sigma \log b \log m)$ time and comparisons, plus $\mathcal{O}(n \log^* n)$ time and $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

Proof. As in the proof of Lemma 11.8, we consider the insertion of a suffix $T[i_j..y]$ into the sparse suffix tree with suffixes $T[i_1..y], T[i_2..y] \cdots T[i_{j-1}..y]$. At all times, we maintain the heavy path decomposition. Additionally, we maintain for each heavy path a predecessor data structure, where given some length ℓ, we can quickly identify the deepest explicit node on the heavy path that spells a string of length at least ℓ. The data structure can, e.g., be a balanced binary search tree with insertion and search operations in $\mathcal{O}(\log b)$ time (the final sparse suffix tree and thus each heavy path contains $\mathcal{O}(b)$ nodes). When rebuilding a subtree of the heavy path decomposition, we also have to rebuild the predecessor data structure for each of its heavy paths. Thus, rebuilding a size-q subtree takes $\mathcal{O}(q \log b)$ time (each node is on exactly one heavy path and has to be inserted into one predecessor data structure), and the amortized insertion time increases from $\mathcal{O}(\log m)$ to $\mathcal{O}(\log m \cdot \log b)$. Whenever we insert a suffix, we make at most one node explicit, and thus have to perform at most one insertion into a predecessor data structure. The time for this is $\mathcal{O}(\log b)$, which is dominated by the previous term.

When inserting $T[i_j..y]$, we look for the node u corresponding to the longest common prefix between $T[i_j..y]$ and the inserted suffixes, make u explicit if necessary and add a new leaf corresponding to $T[i_j..y]$ attached to u. Let v be the current node (initialized by the root, and always an explicit node) and $v_1, \cdots, v_d$ be its (explicit) children. If there is a heavy edge (v, v_a) for $1 \leq a \leq d$, let p be the corresponding heavy path. For each heavy path p, we store the label of one leaf (i.e., the starting position of one suffix) that is contained in the subtree of $e(p)$. Thus, we can use Corollary 11.18 to compute the longest common extension between the string spelled by $e(p)$ and $T[i_j..y]$. Now we use the predecessor data structure on the heavy path to find the deepest (either explicit or implicit) node v' on the path that spells a prefix of $T[i_j..y]$. If v' is implicit, we make it explicit and add the leaf. If v' is explicit and $v' \neq v$, we use v' as the new current node and continue. Otherwise, we have $v' = v$, i.e., the suffix does not belong to the subtree rooted in v_a. In this case, we issue d LCE queries between $T[i_j..y]$ and each of the strings spelled by the nodes $v_1, \ldots, v_d$. This either reveals that we can continue using one of the v_a as the new current node, or that we can create a new explicit node on some (v, v_a) edge and attach the leaf to it, or that we can simply attach a new leaf to v.

We spend $\mathcal{O}(b \cdot \log m \cdot \log b)$ total time for inserting $\mathcal{O}(b)$ nodes into the dynamic heavy path decomposition and the predecessor data structures. Now we analyze the time spent while inserting one suffix. In each step of the insertion process, we either (i) move as far as possible along some heavy path or (ii) move along some non-heavy edge. For (i), we issue one LCE query and one predecessor query. For (ii) we issue $\mathcal{O}(\sigma)$ LCE queries. Due to Observation 11.19, both (i) and (ii) happen at most $\mathcal{O}(\log b)$ times per suffix. Thus, for all suffixes, we perform $\mathcal{O}(b \log b)$ predecessor queries and $\mathcal{O}(b\sigma \log b)$ LCE queries. The total time is $\mathcal{O}(b \log^2 b)$ for predecessor queries, and $\mathcal{O}(b\sigma \log b \log m)$ for LCE queries (apart from the $\mathcal{O}(n \log^* n)$ time and $\mathcal{O}(n)$ comparisons shared by all invocations of Corollary 11.18). □

Lemma 11.21. *For any parameter $\Delta \in [1, m]$, a Δ-approximate LZ factorization of any fragment $T[x..y]$ of length m can be computed in $\mathcal{O}(m\sigma \log^2 m/\sqrt{\Delta})$ time and comparisons, plus $\mathcal{O}(n \log^* n)$ time and $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

Proof. Let $T' = T[x..y]$, and let $\{i_1, i_2, \ldots, i_b\}$ be a Δ-cover of $\{1, \ldots, m\}$, which implies $b = \Theta(m/\sqrt{\Delta})$. We obtain a sparse suffix tree that contains the suffixes $T'[i_1..m], \ldots, T'[i_b..m]$, which takes $\mathcal{O}(b\sigma \log b \log m) \subseteq \mathcal{O}(m\sigma \log^2 m/\sqrt{\Delta})$ time according to Lemma 11.20, plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma. Now we compute a Δ-approximate LZ factorization of T' from the spare suffix tree in $\mathcal{O}(b)$ time.

In the following proof, we use $i_1, i_2, \ldots, i_b$ interchangeably to denote both the difference cover positions, as well as their corresponding leaves in the sparse suffix tree. Assume that the order of difference cover positions is $i_1 < i_2 < \cdots < i_b$. First, we determine for each $i_k > i_1$, the position $\mathsf{src}(i_k) = i_h$ and the length $\mathsf{len}(i_k) = \textsc{lce}(i_h, i_k)$, where $i_h \in \{i_1, \ldots, i_{k-1}\}$ is a position that maximizes $\textsc{lce}(i_h, i_k)$. This is similar to what was done in [Fis+18] for the LZ77 factorization. We start by assigning labels from $\{1, \ldots, b\}$ to the nodes of the sparse suffix tree. A node has label k if and only of i_k is its smallest descendant leaf. We assign the labels as follows. Initially, all nodes are unlabeled. We assign label 1 to each node on the path from i_1 to the root. Then, we process the remaining leaves $i_2, \ldots, i_b$ in increasing order. For each i_k, we follow the path from i_k to the root. We assign label k to each unlabeled node that we encounter. As soon as we reach a node that has already been labeled, say, with label h and string-depth ℓ, we are done processing leaf i_k. It should be easy to see that i_h is also exactly the desired index that maximizes $\textsc{lce}(i_h, i_k)$, and we have $\textsc{lce}(i_h, i_k) = \ell$. Thus, we have found $\mathsf{src}(i_k) = i_h$ and $\mathsf{len}(i_k) = \ell$. The total time needed is linear in the number of sparse suffix tree nodes, which is $\mathcal{O}(b)$.

Finally, we obtain a Δ-approximate LZ factorization using src and len. The previously computed values can be interpreted as follows: i_k could become the starting position of a length-$\mathsf{len}(i_k)$ tail (with previous occurrence at position $\mathsf{src}(i_k)$). For the Δ-approximate LZ factorization, we will create the factors greedily in a left-to-right manner. Assume that we already factorized $T'[1..s-1]$, then the next phrase starts at position s, and thus the next tail starts within $T'[s..s+\Delta)$ (as a reminder, the head is by definition shorter than Δ). Let $S = \{i_1, i_2, \dots, i_b\} \cap \{s, \dots, s+\Delta-1\}$. If there is no $i_k \in S$ with $i_k + \mathsf{len}(i_k) > s + \Delta - 1$, then the next phrase is simply $T'[s.. \min(|T'|, s+\Delta-1))$ with empty tail. Otherwise, the next phrase has (possibly empty) head $T'[s..i_k)$ and tail $T'[i_k..i_k + \mathsf{len}(i_k))$ (with previous occurrence $\mathsf{src}(i_k)$), where i_k is chosen from S such that it maximizes $i_k + \mathsf{len}(i_k)$. Creating the phrase in this way clearly takes $\mathcal{O}(|S|)$ time. Since the next phrase starts at least at position $s + \Delta - 1$, none of the positions from $S \setminus \{s + \Delta - 1\}$ will ever be considered as starting positions of other tails. Thus, every i_k is considered during the creation of at most two phrases, and the total time needed to create all phrases is $\mathcal{O}(b)$.

It remains to be shown that the computed factorization is indeed a Δ-approximate LZ factorization, i.e., if we output a phrase $T'[s..e]$, then the unique (non-approximate) LZ phrase $T'[s..e']$ starting at position s satisfies $e' - 1 \leq e$. First, note that for the created approximate phrases (except possibly the last phrase of T) we have $s + \Delta - 2 \leq e$. Assume $e' < s + \Delta$, then clearly $e' - 1 \leq e$. Thus, we only have to consider $e' > s + \Delta - 1$. Since $T'[s..e']$ is an LZ phrase, there is some $s' < s$ such that $\textsc{lce}(s', s) = e' - s$. Let h be the constant-time computable function that defines the Δ-cover, and let $i_{k'} = s' + h(s', s)$ and $i_k = s + h(s', s)$. Note that $i_{k'} \in \{i_1, i_2, \dots, i_{k-1}\}$ and $i_k \in \{i_1, i_2, \dots, i_b\} \cap \{s, \dots, s + \Delta - 1\}$. Therefore, we have $\mathsf{len}(i_k) \geq \textsc{lce}(i_{k'}, i_k) = \textsc{lce}(s', s) - h(s', s) = (e' - s) - (i_k - s) = e' - i_k$. While computing the Δ-approximate phrase $T'[s..e]$, we considered i_k as the starting positions of the tail, which implies $e \geq i_k + \mathsf{len}(i_k) - 1 \geq e' - 1$. □

Lemma 11.22. *There is an algorithm that, given any parameter $\Delta \in [1, m]$, estimate $\tilde{\sigma}$ and fragment $T[x..y]$ of length m, takes $\mathcal{O}(m\tilde{\sigma}\log^2 m/\sqrt{\Delta})$ time and comparisons, and either computes a Δ-approximate LZ factorization of $T[x..y]$ or determines $\sigma > \tilde{\sigma}$. Additional $\mathcal{O}(n \log^* n)$ time and $\mathcal{O}(n)$ comparisons are shared by all invocations of the lemma.*

Proof. We simply use Lemma 11.21 to compute the factorization. In the first step, we have to construct the sparse suffix tree using the algorithm from Lemma 11.20. While this algorithm takes $\mathcal{O}(m\sigma\log^2 m/\sqrt{\Delta})$ time, it is easy to see that a more accurate time bound is $\mathcal{O}(md\log^2 m/\sqrt{\Delta})$, where d is the maximum degree of any node in the sparse suffix tree. If during construction the maximum degree of a node becomes $\tilde{\sigma} + 1$, we immediately stop and return that $\sigma > \tilde{\sigma}$. Otherwise, we finish the construction in the desired time. □

Now we can describe the algorithm that detects squares in $\mathcal{O}(n \log \sigma + n \log^* n)$ time and $\mathcal{O}(n \log \sigma)$ comparisons. We simply use the algorithm from Section 11.3, but use Lemma 11.22 instead of Lemma 11.14. Next, we analyze the time needed apart from the $\mathcal{O}(n \log^* n)$ time shared by all invocations of Lemma 11.22. Throughout the tth phase, we use $\mathcal{O}(n \cdot \tilde{\sigma} \cdot \log^2(\sigma_t)/\sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4}/\log(\sigma_t) \cdot \log^2(\sigma_t)/\sqrt{\sigma_t}) = \mathcal{O}(n \log(\sigma_t)/(\sigma_t)^{1/4})$ comparisons to construct all the Δ-approximate factorizations. As before, if at any time we discover that $\tilde{\sigma} > (\sigma_t)^{1/4}/\log(\sigma_t)$, then we use Theorem 11.5 to finish the computation in $\mathcal{O}(n \log \sigma_t) = \mathcal{O}(n \log \sigma)$ time. Until then (or until we finished all $\lceil \log \log n \rceil$ phases), we use $\mathcal{O}(\sum_{t=0}^{t'} n \log(\sigma_t)/(\sigma_t)^{1/4})$ time, and by Corollary 11.16 this is $\mathcal{O}(n)$. For detecting squares, we still use Lemma 11.13, which as explained in Section 11.3 takes $\mathcal{O}(n)$ time and comparisons in total, plus additional $\mathcal{O}(Z)$ time, where Z is the number of approximate LZ factors considered during all invocations of the lemma. We apply the lemma to each approximate LZ factorization exactly once, and by construction each factor in phase t has size at least $\Delta = \Omega(\sigma_t)$. Also, each text position is covered by at most two tails per phase. Hence $Z = \mathcal{O}(\sum_{t=0}^{t'} n/\sigma_t)$, which is $\mathcal{O}(n)$ by Corollary 11.16.

The last thing that remains to be shown is how to implement the bookkeeping of blocks, i.e., in each phase we have to efficiently deactivate block pairs as described at the end of Section 11.3. We maintain the block pairs in $\lceil \log \log n \rceil$ bitvectors of total length $\mathcal{O}(n)$, where a set bit means that a block pair has been deactivated (recall that there are $\mathcal{O}(n)$ pairs in total). Bitvector t contains at position j the bit corresponding to block pair $B_j B_{j+1} = T[i..i + 2(\sigma_t)^2)$ with $i = 1 + (\sigma_t)^2 \cdot (j-1)$. Note that translating between i and j takes constant time. For each sufficiently long tail in phase t, we simply iterate over the relevant block pairs in phase $t+2$ and deactivate them, i.e., we set the corresponding bit. This takes time linear in the number of deactivated blocks. Since there are $\mathcal{O}(n)$ block pairs, and each block pair gets deactivated at most a constant number of times, the total cost for this bookkeeping is $\mathcal{O}(n)$.

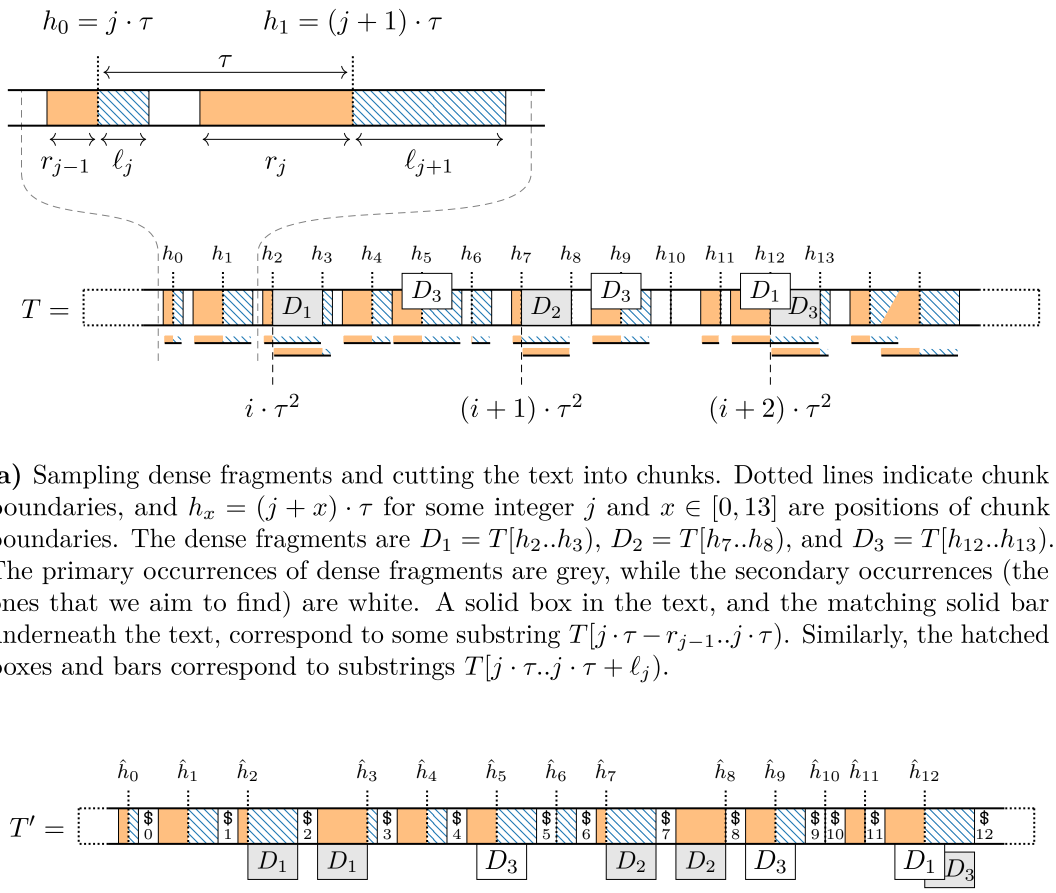
The number of comparisons is dominated by the $\mathcal{O}(n \log \sigma)$ comparisons used when finishing the computation with Theorem 11.5. The only other comparisons are performed by Lemma 11.13, which we already bounded by $\mathcal{O}(n)$, and by LCE queries via Corollary 11.18. Since we ask $\mathcal{O}(n)$ such queries in total, the number of comparisons is also $\mathcal{O}(n)$. We have shown:

Lemma 11.23. *The square detection algorithm from Section 11.3 can be implemented in $\mathcal{O}(n \log \sigma + n \log^* n)$ time and $\mathcal{O}(n \log \sigma)$ comparisons.*

11.4.2 Final Improvement

For our final improvement we need to replace the LCE queries implemented by Corollary 11.18 with our own mechanism. The goal will remain the same, that is, given a parameter Δ and estimate $\tilde{\sigma}$ of the alphabet size, find a Δ-approximate LZ factorization of any fragment $T[x..y]$ of length m in $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ time, where $m = |T[x..y]|$ (with $m = \Theta(\Delta^2)$, as otherwise we are not required to detect anything). As in the previous section, the algorithm might detect that the size of the alphabet is larger than $\tilde{\sigma}$, and in such case we revert to the divide-and-conquer algorithm. Let $\tau = \lfloor \sqrt{\Delta} \rfloor$.

Initially, we only consider some fragments of $T[x..y]$. We say that $T[i \cdot \tau^2..i \cdot \tau^2 + \tau)$ is a dense fragment. We start by remapping the symbols in all dense fragments that intersect $T[x..y]$ to a linearly-sortable alphabet. This can be done in $\mathcal{O}(\tilde{\sigma})$ time for each position by maintaining a list of the already seen distinct symbols (as in the



$h_0 = j \cdot \tau$
$h_1 = (j+1) \cdot \tau$
τ
r_{j-1}
ℓ_j
r_j
ℓ_{j+1}
h_0 h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10} h_{11} h_{12} h_{13}
$T =$
D_1 D_3 D_2 D_3 D_1 D_3
$i \cdot \tau^2$
$(i+1) \cdot \tau^2$
$(i+2) \cdot \tau^2$
$\hat{h}_0$ $\hat{h}_1$ $\hat{h}_2$ $\hat{h}_3$ $\hat{h}_4$ $\hat{h}_5$ $\hat{h}_6$ $\hat{h}_7$ $\hat{h}_8$ $\hat{h}_9$ $\hat{h}_{10}$ $\hat{h}_{11}$ $\hat{h}_{12}$
$T' =$
$\underset{0}{\$}$ $\underset{1}{\$}$ $\underset{2}{\$}$ $\underset{3}{\$}$ $\underset{4}{\$}$ $\underset{5}{\$}$ $\underset{6}{\$}$ $\underset{7}{\$}$ $\underset{8}{\$}$ $\underset{9}{\$}$ $\underset{10}{\$}$ $\underset{11}{\$}$ $\underset{12}{\$}$
D_1 D_1 D_3 D_2 D_2 D_3 D_1 D_3

(a) Sampling dense fragments and cutting the text into chunks. Dotted lines indicate chunk boundaries, and $h_x = (j+x) \cdot \tau$ for some integer j and $x \in [0, 13]$ are positions of chunk boundaries. The dense fragments are $D_1 = T[h_2..h_3)$, $D_2 = T[h_7..h_8)$, and $D_3 = T[h_{12}..h_{13})$. The primary occurrences of dense fragments are grey, while the secondary occurrences (the ones that we aim to find) are white. A solid box in the text, and the matching solid bar underneath the text, correspond to some substring $T[j \cdot \tau - r_{j-1}..j \cdot \tau)$. Similarly, the hatched boxes and bars correspond to substrings $T[j \cdot \tau..j \cdot \tau + \ell_j)$.

(b) The string T' used to find all the occurrences of dense fragments. Each position $\hat{h}_x$ maps to position h_x in Figure 11.4a. The substring indicated by the solid box preceding $h_x = (j+x)\tau$ and the hatched box succeding h_x is exactly $T[h_x - r_{j+x-1}..h_x + \ell_{j+x})$. Each $\underset{x}{\$}$ is a distinct separator symbol that is unique within T'.

Figure 11.4: Supplementary drawings for Section 11.4.2.

reduction shown in Lemma 2.1). For each position in a dense fragment, we iterate over the symbols in the list, and possibly append a new symbol to the list if it is not present. As soon as the size of the list exceeds $\tilde{\sigma}$, we terminate the procedure and revert to the divide-and-conquer algorithm. Otherwise, we replace each symbol by its position in the list. Overall, there are $\mathcal{O}(m/\sqrt{\Delta})$ positions in the dense fragments of $T[x..y]$, and the remapping takes $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$ time.

Next, we construct two generalized suffix trees [Gus97], the first one of all dense fragments, and the second one of their reversals. (The generalized suffix tree of a collection of strings is the compacted trie that contains all suffixes of all strings in the collection.) Since we now work with a linearly-sortable alphabet, this takes only $\mathcal{O}(m/\sqrt{\Delta})$ time [Far97]. We consider fragments of the form $T[i \cdot \tau..(i+1) \cdot \tau)$ having non-empty intersection with $T[x..y]$. We call such fragments chunks. We note that there are $\mathcal{O}(m/\sqrt{\Delta})$ chunks, and their total length is $\mathcal{O}(m)$. For each chunk, we find its longest prefix $T[i \cdot \tau..i \cdot \tau + \ell_i)$ and longest suffix $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$ that occur in one of the dense fragments. Figure 11.4a visualizes the dense fragments, chunks, and longest prefixes and suffixes. This can be done efficiently by following the heavy path decomposition of the generalized suffix tree of all dense fragments and their reversals, respectively. On each current heavy path, we just naively match the symbols as long as possible. In case of a mismatch, we spend $\mathcal{O}(\tilde{\sigma})$ time to descend to the appropriate subtree, which happens at most $\mathcal{O}(\log m)$ times due to the heavy path decomposition. After having found ℓ_i and r_i, we test square-freeness of $T[i \cdot \tau..i \cdot \tau + \ell_i)$ and $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$. Because they both occur in dense fragments, and we have remapped the alphabet of all dense fragments, we can use Theorem 10.1 to implement this in $\mathcal{O}(\ell_i + r_i)$ time. Thus, the total time per chunk is thus $\mathcal{O}(\tilde{\sigma} \log m)$ plus $\mathcal{O}(\ell_i + r_i)$. The former sums up to $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$, and we will later show that the latter can be amortized by deactivating blocks on the lower levels.

The situation so far is that we have remapped the alphabet of all dense fragments to linearly-sortable alphabet, and for every chunk we know its longest prefix and suffix that occur in one of the dense fragments. We concatenate all fragments of the form $T[i \cdot \tau - r_{i-1}..i \cdot \tau + \ell_i)$ (intersected with $T[x..y]$) while adding distinct separators in between to form a new string T'. We stress that, because we have remapped the alphabet of all dense fragments, and the found longest prefix and suffix of each chunk also occur in some dense fragment, T' is over linearly-sortable alphabet. Thus, we can build the suffix tree ST of T' in $\mathcal{O}(|T'|)$ time [Far97]. A visualization of T' is provided in Figure 11.4b

Let $\mathcal{D} = \{D_1, D_2, \ldots\}$ be the set of distinct dense fragments. We would like to construct the set of all occurrences of the strings from $\mathcal{D}$ in $T[x..y]$. Using the suffix tree of T', we can retrieve all occurrences of every D_j in T'. We observe that, because of how we have defined $T[i \cdot \tau..i \cdot \tau + \ell_i)$ and $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$, this will in fact give us all occurrences of every D_j in the original $T[x..y]$. To implement this efficiently, we proceed as follows. First, for every i we traverse ST starting from its root to find the (explicit or implicit) node corresponding to the dense fragment $T[i \cdot \tau^2..i \cdot \tau^2 + \tau)$. This takes only $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$ time. Then, all leaves in every subtree rooted at such a node correspond to occurrences of some D_j, and can be reported by traversing the subtree in time proportional to its size, so at most $\mathcal{O}(|T'|)$ in total. Finally, remapping the occurrences back to $T[x..y]$ can be done in constant time per occurrence by precomputing, for every position in T', its corresponding position in $T[x..y]$, which can be done in $\mathcal{O}(|T'|)$ time when constructing T'. Thus, in $\mathcal{O}(|T'|)$ time, we obtain the set S of starting positions of all occurrences of the strings in $\mathcal{D}$. We summarize the properties of S below.

Proposition 11.24. *The described set S admits the following properties:*

(1) *For every $i \in [x, y]$ such that $i = 0 \pmod{\tau^2}$, $i \in S$.*

(2) *For every $i \in [x, y - \tau]$, $i \in S$ if and only if $T[i..i + \tau) \in \mathcal{D}$.*

(3) $|S| \leq |T'|$.

We now define a parsing of $T[x..y]\$$ based on S. Let $i_1 < i_2 < \ldots i_k$ be all the positions in S, that is, $(i_j, i_{j+1}) \cap S = \emptyset$ for every $j = 1, 2, \ldots, k-1$. For every $j = 1, 2, \ldots, k-1$, we create the phrase $T[i_j..i_{j+1} + \tau)$. We add the last phrase $T[i_k..y]\$$. We stress that consecutive phrases overlap by τ symbols, and each phrase begins with a length-τ fragment starting at a position in S. This, together with Proposition 11.24(2), implies the following property.

Observation 11.25. *The set of distinct phrases is prefix-free, i.e., no phrase is a proper prefix of another phrase.*

We would like to construct the compacted trie $\mathcal{T}_{\text{phrase}}$ of all such phrases, so that we particularly identify identical phrases. We first notice that each phrase begins with a fragment $T[i_j..i_j+\tau)$ that has its corresponding occurrence in T'. We note that, given a set of positions P in T, we can find their corresponding positions in T' (if they exist) by sorting and scanning in $\mathcal{O}(|P|+|T'|)$ time.

Thus, we can assume that for each i_j we know its corresponding position i'_j in T'. Next, for each node of ST we precompute its unique ancestor at string depth τ in $\mathcal{O}(|T'|)$ time. Then, for every fragment $T[i_j..i_j+\tau)$ we can access its corresponding (implicit or explicit) node of ST. This allows us to partition all phrases according to their prefixes of length τ. In fact, this gives us the top part of $\mathcal{T}_{\text{phrase}}$ containing all such prefixes in $\mathcal{O}(m/\sqrt{\Delta})$ time, and for each phrase we can assume that we know the node of $\mathcal{T}_{\text{phrase}}$ corresponding to its length-τ prefix.

To build the remaining part of $\mathcal{T}_{\text{phrase}}$, we partition the phrases into short and long. $T[i_j..i_{j+1}+\tau)$ is short when $i_{j+1} \le i_j+\tau$ (meaning that its length is at most 2τ), and long otherwise.

We begin with constructing the compacted trie $\mathcal{T}'_{\text{phrase}}$ of all short phrases. This can be done similarly to constructing the top part of $\mathcal{T}_{\text{phrase}}$, except that now the fragments have possibly different lengths. However, every short phrase $T[i_j..i_{j+1}+\tau)$ occurs in T' as $T'[i'_j..i'_{j+1}+\tau)$. We claim that the nodes of ST corresponding to every $T'[i'_j..i'_{j+1}+\tau)$ can be found in $\mathcal{O}(|T'|)$ time. This can be done by traversing ST in the depth-first order while maintaining a stack of all explicit nodes with string depth at least τ on the current path. Then, when visiting the leaf corresponding to the suffix of T' starting at position i'_j, we iterate over the current stack to find the sought node. This takes at most $\mathcal{O}(|T'[i_j+\tau..i_{j+1}+\tau]|)$ time, which sums up to $\mathcal{O}(|T'|)$. Having found the node of ST corresponding to $T[i_j..i_{j+1}+\tau)$, we extract $\mathcal{T}'_{\text{phrase}}$ from ST in $\mathcal{O}(|T'|)$ time.

With $\mathcal{T}'_{\text{phrase}}$ in hand, we construct the whole $\mathcal{T}_{\text{phrase}}$ as follows. We begin with taking the union of $\mathcal{T}'_{\text{phrase}}$ and the already obtained top part of $\mathcal{T}_{\text{phrase}}$, this can be obtained in $\mathcal{O}(|T'|)$ time. For each long phrase $T[i_j..i_{j+1}+\tau)$, we know the node corresponding to $T[i_j..i_j+\tau)$ and would like to insert the whole string $T[i_j..i_{j+1}+\tau)$ into $\mathcal{T}_{\text{phrase}}$. We perform the insertions in increasing order of i_j (this will be crucial for amortizing the time later). This is implemented with a dynamic heavy path decomposition similarly as in Section 11.4.1, however with one important change. Namely, we fix a heavy path decomposition of the part of $\mathcal{T}_{\text{phrase}}$ corresponding to the union of $\mathcal{T}'_{\text{phrase}}$ and the top part of $\mathcal{T}_{\text{phrase}}$, and maintain a dynamic heavy path decomposition of every subtree hanging off from this part. Thanks to this change, the time to maintain the dynamic trie and all heavy path decompositions is $\mathcal{O}(m \log m/\sqrt{\Delta})$, as there are only $\mathcal{O}(m/\sqrt{\Delta})$ long phrases. Next, for each long phrase $T[i_j..i_{j+1}+\tau)$, we begin the insertion at the already known node corresponding to $T[i_j..i_j+\tau)$, and continue the insertion by following the heavy paths, first in the static heavy path decomposition in the part of $\mathcal{T}_{\text{phrase}}$ corresponding to $\mathcal{T}'_{\text{phrase}}$, second in the dynamic heavy path decomposition in the appropriate subtree. On each heavy path, we naively match the symbols as long as possible. The time to insert a single phrase $T[i_j..i_{j+1}+\tau)$ is $\mathcal{O}(\log m)$ (twice) plus the length of the longest prefix of $T[i_j+\tau..i_{j+1}+\tau)$ equal to a prefix of $T[i_{j'}+\tau..i_{j'+1}+\tau)$, for some $j' < j$.

The former sums up to another $\mathcal{O}(m \log m/\sqrt{\Delta})$, and we will later show that the latter can be amortized by deactivating blocks on the lower levels.

The trie $\mathcal{T}_{\text{phrase}}$ allows us to form metasymbols corresponding to the phrases, where every metasymbol is an integer, and two phrases are mapped to the same integer if and only if they are identical. We then transform $T[x..y]$ into a string T_{parse} of length $\mathcal{O}(|T'|)$ consisting of these metasymbols, where the strings underlying any two consecutive metasymbols overlap by τ symbols. We build a suffix tree $\mathcal{S}_{\text{parse}}$ of this string over linearly-sortable metasymbols in $\mathcal{O}(|T'|)$ time. Next, we convert it into the sparse suffix tree $\mathcal{S}'_{\text{parse}}$ of all suffixes $T[i_j..y]$ as follows. Consider an explicit node $u \in \mathcal{S}_{\text{parse}}$ with children $v_1, v_2, \ldots, v_d$, $d \geq 2$. We first compute the subtree $\mathcal{T}_u$ of $\mathcal{T}_{\text{phrase}}$ induced by the leaves corresponding to the first metasymbols on the edges (u, v_i), for $i = 1, 2, \ldots, d$, and connect every v_i to the appropriate leaf of $\mathcal{T}_u$. This can be implemented in $\mathcal{O}(d)$ time, assuming constant-time lowest common ancestor queries [BF00] on $\mathcal{T}_{\text{phrase}}$, and processing the leaves from left to right with a stack, similarly as in the Cartesian tree construction algorithm [Vui80]. We note that the order on the leaves is the same as the order on the metasymbols, and hence no extra sorting is necessary. Overall, this sums up to $\mathcal{O}(|T'|)$ time. Next, we observe that, unless u is the root of $\mathcal{S}_{\text{parse}}$, all metasymbols on the edges (u, v_i) correspond to strings starting with the same prefix of length τ (this is due to the fact that the substrings underlying the metasymbols overlap by τ symbols). We obtain the subtree $\mathcal{T}'_u$ by truncating this prefix (or taking $\mathcal{T}_u$ if u is the root). Finally, we identify the root of $\mathcal{T}'_u$ with u, and every child v_i with its corresponding leaf of $\mathcal{T}'_u$. Because we truncate the overlapping prefixes of length τ, after this procedure is executed on every node of $\mathcal{S}_{\text{parse}}$ we obtain a tree $\mathcal{S}'_{\text{parse}}$ with the property that each leaf corresponds to a suffix $T[i_j..y]$. Also, by Observation 11.25, the edges outgoing from every node start with different symbols as required.

By following an argument from the proof of Lemma 11.21, $\mathcal{S}'_{\text{parse}}$ allows us to determine, for every suffix $T[i_j..y]$, its longest prefix equal to a prefix of some $T[i'..y]$ with $i' < i_j$, as long as its length is at least τ. Indeed, in such case we must have $i' \in S$ by Proposition 11.24(2), so in fact $i' = i_{j'}$ and it is enough to maximize the length of the common prefix with all earlier positions in S, which can be done using $\mathcal{S}'_{\text{parse}}$. Thus, we either know that the length of this longest prefix is less than τ, or know its exact value (and the corresponding position $i' \in S$).

Lemma 11.26. *For any parameter $\Delta \in [1, m]$ and estimate $\tilde{\sigma}$ of the alphabet size, a $(\Delta + \tau)$-approximate LZ factorization of any fragment $T[x..y]$ of length m can be computed in $\mathcal{O}(m/\sqrt{\Delta})$ time, plus the time needed for the preprocessing described earlier in this section.*

Proof. Let $e \in [x, y]$ and suppose we have already constructed the factorization of $T[x..e-1]$ and are now trying to construct the next phrase. Let e' be the next multiple of τ^2, we have that $e' - e < \tau^2 \leq \Delta$ and $T[e'..e' + \tau)$ is a dense fragment. Thus, by Proposition 11.24(1) we have $e' \in S$.

The first possibility is that the longest common prefix between $T[e'..y]$ and any suffix starting at an earlier position is shorter than τ. In this case, we can simply set the head of the new phrase to be $T[e..e' + \tau)$ and the tail to be empty. Otherwise, we know the length ℓ of this longest prefix by the preprocessing described above. We set the head of the new phrase to be $T[e..e')$ and the tail to be $T[e'..e'' + \ell)$. This takes constant time per phrase, and each phrase is of length at least τ, giving the claimed overall time complexity. It remains to argue correctness of every step.

Let $T[e..s]$ be the longest LZ phrase starting at position e, to show that we obtain a valid $(\Delta+\tau)$-approximate phrase it suffices to show that $s \le e' + \max(\tau, \ell)$. Let the previous occurrence of $T[e..s)$ be at position $p < e$. If $s - e' < \tau$ then there is nothing to prove. Otherwise, $T[e'..s)$ is a string of length at least τ that also occurs starting earlier at position $p + e' - e < e'$. Thus, we will correctly determine that $\ell \ge \tau$, and find a previous occurrence of the string maximizing the value of ℓ. In particular, we will have $\ell \ge s - e'$ as required. □

To achieve the bound of Lemma 11.4, we now proceed as in Section 11.3.4, except that instead of Lemma 11.22 we use Lemma 11.26. For every $T[x..y]$ with $m = |T[x..y]|$ this takes $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ time plus the time used for computing the longest prefix and suffix of each chunk (the latter also accounts for constructing the suffix tree ST and other steps that have been estimated as taking $\mathcal{O}(|T'|)$ in the above reasoning) plus the time for inserting $T[i_j+\tau..i_{j+1}+\tau)$ into $\mathcal{T}_{\text{phrase}}$ when $i_{j+1} \ge i_j + \tau$.

We observe that we can deactivate any block pair fully contained in $T[i\cdot\tau..i\cdot\tau+\ell_i)$ and $T[(i+1)\cdot\tau - r_i..(i+1)\cdot\tau)$, as we have already checked that these fragments are square-free. Also, we can deactivate any block pair fully contained in the longest prefix of $T[i_j+\tau..i_{j+1}+\tau)$ equal to $T[i_{j'}+\tau..i_{j'+1}+\tau)$, for some $j' < j$, because such fragment cannot contain the leftmost occurrence of a square.

There are $\mathcal{O}(m/\sqrt{\Delta})$ chunks and long phrases. If a chunk or a long phrase contributes $x = \Omega(\sqrt[4]{\Delta})$ to the total time, then we explicitly deactivate the block pairs in phase $t+3$ that are entirely contained in the corresponding fragment. Block pairs in phase $t+3$ are of length $\mathcal{O}(\sqrt[4]{\Delta})$, and thus we deactivate $\Omega(x)$ positions. Therefore, the time spent on such chunks and long phrases in all phases sums to $\mathcal{O}(n)$. The remaining chunks and long phrases contribute $\mathcal{O}(\sqrt[4]{\Delta})$ to the total time, and there are $\mathcal{O}(m/\sqrt{\Delta})$ of them, which adds up to $\mathcal{O}(m/\sqrt[4]{\Delta})$. In every phase, this is $\mathcal{O}(n/\sqrt[4]{\Delta})$, so $\mathcal{O}(n)$ overall by Corollary 11.16. Hence we have shown the final result for testing square-freeness.

Lemma 11.4. *Testing square-freeness of a length-n string that contains σ distinct symbols over general unordered alphabet can be implemented in $\mathcal{O}(n\log\sigma)$ time.*

11.5 Computing Runs

Now we adapt the algorithm such that it computes all runs. We start with the algorithm from Sections 11.3 and 11.4 without the final improvement from Section 11.4.2. First, note that the key properties of the Δ-approximate LZ factorization, in particular Lemmas 11.12 and 11.13, also hold for the computation of runs. This is expressed by the lemmas below.

Lemma 11.27. *Let $b_1b_2\dots b_z$ be a Δ-approximate LZ factorization of a string T. For every run $\langle s,e,p\rangle$ of length $e-s+1 \ge 8\Delta$, there is at least one phrase b_i with $|tail(b_i)| \ge \frac{e-s+1}{8} \ge \Delta$ such that $tail(b_i)$ and the right-hand side $T[s+\left\lceil\frac{e-s+1}{2}\right\rceil..e]$ of the run intersect.*

Proof. Let $\ell = \frac{e-s+1}{2}$ and note that $\frac{\ell}{4} \geq \Delta$ and $e = s + 2\ell - 1$. Assume that all tails that intersect $T[s + \lceil \ell \rceil ..e]$ are of length less than $\frac{\ell}{4}$, then the respective phrases of these tails are of length at most $\frac{\ell}{4} + \Delta - 1 \leq \frac{\ell}{2} - 1$ (because each head is of length less than Δ). This means that $T[s + \lceil \ell \rceil ..e]$ (of length $\lfloor \ell \rfloor$) intersects at least $\left\lceil \lfloor \ell \rfloor / (\frac{\ell}{2} - 1) \right\rceil \geq 3$ phrases (the inequality holds for $\ell \geq 4$, which is implied by $\Delta \geq 1$). Thus there is some phrase $b_i = T[x..y]$ properly contained in $T[s + \lceil \ell \rceil ..e]$, formally $s + \lceil \ell \rceil < x \leq y < e$. However, this contradicts the definition of the Δ-approximate LZ factorization because $T[x..e+1]$ is the prefix of a standard LZ phrase (due to $T[x..e] = T[x-p..e-p]$). The contradiction implies that $T[s + \lceil \ell \rceil ..e]$ intersects a tail of length at least $\frac{\ell}{4}$. □

Before we show how to algorithmically apply Lemma 11.27, we need to explain how Lemma 11.6 extends to computing runs, and then how this implies that the approach of Main and Lorentz [ML84] easily extends to computing all runs. We do not claim this to be a new result, but the original paper only talks about finding a representation of all squares, and we aim to find runs, and hence include a description for completeness.

Lemma 11.28. *Given two strings x and y over general unordered alphabet, we can compute all runs in xy that include either the last symbol of x or the first symbol of y using $\mathcal{O}(|x| + |y|)$ time and comparisons.*

Proof. Consider a run $\langle s, e, p \rangle$ in $t = xy$ that includes either the last symbol of x or the first symbol of y, meaning that $s \leq |x| + 1$ and $e \geq |x|$. Let $\ell = \lfloor \frac{e-s+1}{2} \rfloor \geq p$. We separately compute all runs with $s + \ell \leq |x| + 1$ and $s + \ell > |x| + 1$. Below we describe the former, and the latter is symmetric.

Due to $s + \ell \leq |x| + 1$, the length-p substring $x[|x| - p + 1.. |x|]$ is fully within the run. This suggests the following strategy to generate all runs with $s + \ell \leq |x| + 1$. We iterate over the possible values of $p = 1, 2, \ldots, |x|$. For a given p, we calculate the length of the longest common prefix of $x[|x| - p + 1.. |x|]y$ and y, denoted pref, and the length of the longest common suffix of $x[1.. |x| - p]$ and x, denoted suf. It is easy to see that $t[|x| - p + 1 - \mathsf{suf}.. |x| + \mathsf{pref}]$ is a length-wise maximal p-periodic substring, and its length is $\ell' = p + \mathsf{suf} + \mathsf{pref}$. If $\mathsf{pref} + \mathsf{suf} \geq p$ and $s + \lfloor \ell'/2 \rfloor \leq |x| + 1$, then we report the substring as a run. (The latter condition ensures that each run gets reported by exactly one of the two symmetric cases.)

This procedure reports lengthwise maximal p-periodic substrings (for every p), but it is not guaranteed that p is also the *minimal* period. Hence we additionally filter the reported runs such that, whenever the same substring gets reported multiple times with different period, we only report the one with the minimal period. It is easy to see that this takes $\mathcal{O}(|x| + |y|)$ time using, e.g., radix sorting.

We use a prefix table to compute the longest common prefixes. For a given string, this table contains at position i the length of the longest substring starting at position i that is also a prefix of the string. For computing the values pref, we use the prefix table of $y\$xy$ (where $\$$ is a new symbol that does not match any symbol in x nor y). Similarly, for computing the values suf, we use the prefix table of the reversal of a new string $x\$x$. The tables can be computed in $\mathcal{O}(|x| + |y|)$ time and comparisons (see, e.g., computation of table *lppattern* in [ML84]). Then, each value of p can be checked in constant time. □

Lemma 11.29. *Computing all runs in a length-n string over general unordered alphabet can be implemented in $\mathcal{O}(n \log n)$ time and comparisons.*

Proof. Let the input string be $T[1..n]$. We apply divide-and-conquer. Let $x = T[1..\lfloor n/2 \rfloor]$ and $y = T[\lfloor n/2 \rfloor + 1..n]$. First, we recursively compute all runs in x and y. Of the reported runs, we filter out all the ones that contain either the last symbol of x or the first symbol of y, which takes $\mathcal{O}(|x| + |y|)$ time. In this way, if some reported run is a run with respect to x (or y), but not with respect to xy, then it will be filtered out. We have generated all runs except for the ones that contain the last symbol of x or the first symbol of y (or both). Thus we simply invoke Lemma 11.28 on xy, which will output exactly the missing runs in $\mathcal{O}(|x| + |y|)$ time and comparisons. There are $\mathcal{O}(\log n)$ levels of recursion, and each level takes $\mathcal{O}(n)$ time and comparisons in total. □

Lemma 11.30. *Let $T = b_1 b_2 \dots b_z$ be a Δ-approximate LZ factorization, and $\chi = \sum_{|tail(b_i)| \geq \Delta} |tail\,(b_i)|$. We can compute in $\mathcal{O}\left(\chi + z\right)$ time and $\mathcal{O}\left(\chi\right)$ comparisons a multiset R of size $\mathcal{O}(\chi)$ of runs with the property that a run $T[s..e]$ is possibly not in R only if $e - s + 1 < 8\Delta$ or there is some tail $\mathsf{tail}(b_i) = T[a_2..a_3]$ with $a_2 < s$ and $e < a_3$.*

Proof. The general idea is the same as in the proof of Lemma 11.13 for detecting squares. Let $n = |T|$. We consider each phrase $b_i = T[a_1..a_3]$ with $\mathsf{head}(b_i) = T[a_1..a_2 - 1]$ and $\mathsf{tail}(b_i) = T[a_2..a_3]$ separately. Let $k = |\mathsf{tail}(b_i)|$. If $k \geq \Delta$, we apply Lemma 11.28 to $x_1 = T[a_2 - 8k..a_2 - 1]$ and $y_1 = T[a_2..a_3 + 4k]$, as well as $x_2 = T[a_2 - 8k..a_3 - 1]$ and $y_2 = T[a_3..a_3 + 4k]$ trimmed to $T[1..n]$. This takes $\mathcal{O}(|\mathsf{tail}(b_i)|)$ time and comparisons and reports $\mathcal{O}(|\mathsf{tail}(b_i)|)$ runs with respect to $x_1 y_1 = x_2 y_2 = T[a_2 - 8k..a_3 + 4k]$ (trimmed to $T[1..n]$). Of these runs, we filter out the ones that contain any of the positions $a_2 - 8k$ (only if $a_2 - 8k > 1$) and $a_3 + 4k$ (only if $a_3 + 4k < n$), which takes $\mathcal{O}(|\mathsf{tail}(b_i)|)$ time. This way, each reported run is not only a run with respect to $x_1 y_1$, but also a run with respect to T. In total, we report $\mathcal{O}(\chi)$ runs (including possible duplicates) and spend $\mathcal{O}\left(\chi\right)$ time and comparisons when applying Lemma 11.28. Additional $\mathcal{O}(z)$ time is needed to check if $|\mathsf{tail}(b_i)| \geq \Delta$ for each phrase.

Now we show that the described strategy computes all runs of length at least 8Δ, except for the ones that are properly contained in a tail. Let $\langle s, e, p \rangle$ be a run of length 2ℓ, where $\ell \geq 4\Delta$ is a multiple of $\frac{1}{2}$. Due to Lemma 11.27, the right-hand side $T[s + \lceil \ell \rceil..e]$ of this run intersects some tail $\mathsf{tail}(b_i) = T[a_2..a_3]$ of length $k = |\mathsf{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$. Due to the intersection, we have $a_2 \leq e$ and $a_3 \geq s + \lceil \ell \rceil$. Thus, when processing b_i and applying Lemma 11.28, the starting position of x_1 and x_2 satisfies $a_2 - 8k \leq e - 8\frac{\ell}{4} < s$, while the end position of y_1 and y_2 satisfies $a_3 + 4k \geq s + \lceil \ell \rceil + 4\frac{\ell}{4} > e$. Therefore, the run is contained in the fragment $T[a_2 - 8k..a_3 + 4k]$ (trimmed to $T[1..n]$) corresponding to $x_1 y_1$ and $x_2 y_2$, and the run does not contain positions $a_2 - 8k$ and $a_3 + 4k$. If $s \leq a_2 \leq e$, we find the run when applying Lemma 11.28 to x_1 and y_1. If $s \leq a_3 \leq e$, we find the run when applying Lemma 11.28 to x_2 and y_2. Otherwise, $T[s..e]$ is entirely contained in $T[a_2 + 1..a_3 - 1]$ and we do not have to report the run. □

Now we describe how to compute all runs using $\mathcal{O}(n \log \sigma)$ comparisons and $\mathcal{O}(n \log \sigma + n \log^* n)$ time. We again use the sequence $\sigma_t = 2^{2^{\lceil \log \log n \rceil - t}}$, for $t =$

$0, 1, \dots, \lceil \log \log n \rceil$. We observe that $\sigma_{t-1} = (\sigma_t)^2$, and proceed in phases corresponding to the values of t. In the tth phase we aim to compute runs of length at least σ_t and less than $(\sigma_t)^2$. We stress that this condition depends on the length of the run and not on its period. We partition the whole $T[1..n]$ into blocks of length $(\sigma_t)^2$, and denote the kth block by B_k. A run of length less than $(\sigma_t)^2$ is fully contained within some two consecutive blocks $B_i B_{i+1}$, and there is always a pair of consecutive blocks such that the run contains neither the first nor the last position of the pair (unless the first position is $T[1]$ or the last position is $T[n]$ respectively). Hence we consider each pair $B_1 B_2$, $B_2 B_3$, and so on. We first apply Lemma 11.22 with $\Delta = \sigma_t/8$ and $\tilde{\sigma} = (\sigma_t)^{1/4}/\log(\sigma_t)$ to find an $(\sigma_t/8)$-approximate LZ factorization of the corresponding fragment of $T[1..n]$, and then use Lemma 11.30 to compute all runs of length at least σ_t, apart from possibly the ones that are properly contained in a tail. Of the computed runs, we discard the ones that contain the first or last position of the block pair (unless the first position is $T[1]$ or the last position is $T[n]$ respectively). This way, each reported run is a run not only with respect to the block pair, but with respect to the entire $T[1..n]$. If we do not report some run of length at least σ_t and less than $(\sigma_t)^2$ in this way, then it is properly contained in one of the tails.

We cannot always afford to apply Lemmas 11.22 and 11.30 to all block pairs. Thus, we have to deactivate some of the blocks. During the current phase t, for each tail $T[s..e]$ of length at least Δ, we deactivate all block pairs in phase $t+3$ that are contained in $T[s+1..e-1]$. By similar logic as in Section 11.3, if a tail contributes $e-s+1$ comparisons and time to the application of Lemma 11.30, then it permanently deactivates $\Omega(e-s+1)$ positions of the string, and thus the total time and comparisons needed for all invocations of Lemmas 11.22 and 11.30 are bounded by $\mathcal{O}(n)$ (apart from the additional $\mathcal{O}(n \log^* n)$ total time for Lemma 11.22). Whenever we apply Lemma 11.22, we add all the tails of length at least Δ to a list $\mathcal{L}$, where each tail is annotated with the position of its previous occurrence. After the algorithm terminates, $\mathcal{L}$ contains all sufficiently long tails from all phases. We have already shown that the total time needed for Lemma 11.30 is bounded by $\mathcal{O}(n)$, and thus the total length of the tails in $\mathcal{L}$ is at most $\mathcal{O}(n)$.

If any of the calls to Lemma 11.22 in the current phase detects that $\sigma > \tilde{\sigma}$, or if $\tilde{\sigma} < 256$, we immediately switch to applying Lemma 11.29 on every pair of blocks $B_i B_{i+1}$ of the current phase, which takes $\mathcal{O}(n \log \sigma)$ time (because the length of a block pair is polynomial in $\tilde{\sigma}$). Again, after applying Lemmas 11.22 and 11.30 to $B_i B_{i+1}$, we discard all runs that contain the first or last position of $B_i B_{i+1}$ (unless the first position is $T[1]$ or the last position is $T[n]$, respectively). After this procedure terminates, we have computed all runs, except for possibly some of the runs that were properly contained in a tail in list $\mathcal{L}$. We may have reported some duplicate runs, which we filter out as follows. The number of runs reported so far (including possible duplicates) is $r = \mathcal{O}(n \log \sigma)$[2]. We sort them in additional $\mathcal{O}(n + r) = \mathcal{O}(n \log \sigma)$ time, e.g., by using radix sort, and remove duplicates. The running time so far is $\mathcal{O}(n \log \sigma)$.

[2] A more careful analysis would reveal that it is $\mathcal{O}(n)$, but this is not necessary for the proof.

11.5.1 Copying Runs From Previous Occurrences

Lastly, we have to compute the runs that were properly contained in a tail in $\mathcal{L}$. Consider such a run $\langle s_r, e_r, p\rangle$, and let $T[s..e]$ be a tail in $\mathcal{L}$ with $s < s_r$ and $e_r < e$. If multiple tails match this criterion, let $T[s..e]$ be the one that maximizes e. In $\mathcal{L}$, we annotated $T[s..e]$ with its previous occurrence $T[s-d..e-d]$. Note that $\langle s_r - d, e_r - d, p\rangle$ is also a run. Thus, if we compute the runs in an appropriate order, we can simply copy the missing runs from their respective previous occurrences. For this sake, we annotate each position $i \in [1, n]$ with:

- a list of all the runs $\langle i, e, p\rangle$ that we already computed, arranged in increasing order of end position e. We already sorted the runs for duplicate elimination, and can annotate all position in $\mathcal{O}(n)$ time.
- a pair (e^*, d^*), where $e^* = d^* = 0$ if there is no tail $T[s..e]$ such that $s < i < e$. Otherwise, among all tails $T[s..e]$ with $s < i < e$, we choose the one that maximizes e. Let $T[s-d..e-d]$ be its previous occurrence, then we use $e^* = e$ and $d^* = d$. As explained earlier, the total length of all tails in $\mathcal{L}$ is $\mathcal{O}(n)$, and thus we can simply scan each tail and update the annotation pair of each contained position whenever necessary.

Observe that, if a position is annotated with $(0, 0)$, then none of the runs starting at position i is fully contained in a tail, and thus we have already annotated position i with the complete list of the runs starting at i. Now we process the positions $i \in [1, n]$ one at a time and in increasing order. We inductively assume that, at the time at which we process i, we have already annotated each $j < i$ with the complete list of runs starting at j. Hence our goal is to complete the list of i such that it contains all runs starting at i. If i is annotated with $(0, 0)$, then the list is already complete. Otherwise, i is annotated with (e, d), every missing run $\langle i, e_r, p\rangle$ satisfies $e_r < e$, and the annotation list of $i - d$ already contains the run $\langle i - d, e_r - d, p\rangle$ (due to $T[i-1..e_r+1] = T[i-d-1..e_r-d+1]$ and the inductive assumption). For each run $\langle i - d, e_r - d, p\rangle$ in the annotation list of position $i - d$, we insert the run $\langle i, e_r, p\rangle$ into the annotation list of i. We perform this step in a merging fashion, starting with the shortest runs of both lists and zipping them together. As soon as we are about to insert a run $\langle i, e_r, p\rangle$ with $e_r \geq e$, we do not insert it and abort. Thus, the time needed for processing i is linear in the number of runs starting at position i. By the runs theorem [Ban+17], the total number of runs is less than n, limiting the total time for this step by $\mathcal{O}(n)$.

Apart from the new steps in Section 11.5.1, the complexity analysis works exactly like in Section 11.3. Hence we have shown:

Lemma 11.31. *Computing all runs in a length-n string that contains σ distinct symbols over general unordered alphabet can be implemented in $\mathcal{O}(n \log \sigma)$ comparisons and $\mathcal{O}(n \log \sigma + n \log^* n)$ time.*

11.5.2 Final Improvement for Computing Runs

The goal is now to adapt the final algorithm to detect all runs. We can no longer stop as soon as we detect a square, and we cannot simply deactivate pairs of blocks that occur earlier. However, Theorem 10.1 is capable of reporting all runs in $T[i\cdot\tau..i\cdot\tau+\ell_i)$

and $T[(i+1)\cdot\tau - r_i..(i+1)\cdot\tau)$ in $\mathcal{O}(\ell_i + r_i)$ time, and we do not need to terminate the algorithm if these fragments are not square-free. Thus, we can indeed deactivate any block pair fully contained in $T[i\cdot\tau..i\cdot\tau+\ell_i)$ and $T[(i+1)\cdot\tau - r_i..(i+1)\cdot\tau)$. Next, we also deactivate block pairs fully contained in the longest prefix of $T[i_j+\tau..i_{j+1}+\tau)$ equal to a prefix of $T[i_{j'}+\tau..i_{j'+1}+\tau)$, for some $j' < j$. Denoting the length of this prefix by ℓ, we treat $T[i_j+\tau..i_j+\ell)$ as a tail and add it to the list $\mathcal{L}$ (annotated with $i_{j'}+\tau$). The total length of all fragments added to $\mathcal{L}$ is still $\mathcal{O}(n)$.

Theorem 11.2. *All the runs contained in a length-n string over general unordered alphabet can be computed in $\mathcal{O}(n\log\sigma)$ time, where σ is the number of distinct symbols in the string, which is not known in advance.*

11.6 Conclusion

We presented the first algorithm that computes all runs in optimal $\mathcal{O}(n\log\sigma)$ time over general unordered alphabet. The solution is quite complicated and requires many technical details. Considering the simplicity of the previous $\mathcal{O}(n\log n)$ time algorithm by Main and Lorentz [ML84], it seems likely that the new algorithm can be significantly simplified, which we leave as future work. Another open question is whether or not the new $\Omega(n\log\sigma)$ time lower bound also applies to randomized algorithms. Finally, it would also be interesting to consider the computation of runs in tries over unordered alphabet (which has already been done for linearly-sortable alphabet, see, e.g., [Sug+21]).

Bibliography

[AS98] Jean-Paul Allouche and Jeffrey O. Shallit. „The ubiquitous Prouhet-Thue-Morse sequence.“ In: *Proceedings of the 1st International Conference on Sequences and their Applications (SETA 1998)*. Singapore, 1998, pages 1–16. DOI: 10.1007/978-1-4471-0551-0_1 (cited on page 153).

[AEL10] Amihood Amir, Estrella Eisenberg, and Avivit Levy. „Approximate periodicity.“ In: *Proceedings of the 21st International Symposium on Algorithms and Computation, Part I (ISAAC 2010)*. Jeju Island, Korea, 2010, pages 25–36. DOI: 10.1007/978-3-642-17517-6_5 (cited on page 128).

[AEL15] Amihood Amir, Estrella Eisenberg, and Avivit Levy. „Approximate periodicity.“ In: *Information and Computation* 241 (2015), pages 215–226. DOI: 10.1016/j.ic.2015.02.004 (cited on page 128).

[ALU02] Amihood Amir, Gad M. Landau, and Esko Ukkonen. „Online timestamped text indexing.“ In: *Information Processing Letters* 82.5 (2002), pages 253–259. DOI: 10.1016/S0020-0190(01)00275-7 (cited on page 31).

[AL12] Amihood Amir and Avivit Levy. „Approximate period detection and correction.“ In: *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*. Cartagena de Indias, Colombia, 2012, pages 1–15. DOI: 10.1007/978-3-642-34109-0_1 (cited on page 128).

[And+98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. „Sorting in linear time?“ In: *Journal of Computer and System Sciences* 57.1 (1998), pages 74–93. DOI: 10.1006/JCSS.1998.1580 (cited on page 18).

[Apo92] Alberto Apostolico. „Optimal parallel detection of squares in strings.“ In: *Algorithmica* 8.1–6 (1992), pages 285–319. DOI: 10.1007/bf01758848 (cited on pages 1, 129).

[AB96] Alberto Apostolico and Dany Breslauer. „An optimal $\mathcal{O}(\log\log N)$-time parallel algorithm for detecting all squares in a string.“ In: *SIAM Journal on Computing* 25.6 (1996), pages 1318–1331. DOI: 10.1137/S0097539793260404 (cited on page 129).

[ABG95] Alberto Apostolico, Dany Breslauer, and Zvi Galil. „Parallel detection of all palindromes in a string.“ In: *Theoretical Computer Science* 141.1–2 (1995), pages 163–173. DOI: 10.1016/0304-3975(94)00083-u (cited on page 16).

[AG08] Alberto Apostolico and Raffaele Giancarlo. „Periodicity and repetitions in parameterized strings.“ In: *Discrete Applied Mathematics* 156.9 (2008), pages 1389–1398. DOI: 10.1016/j.dam.2006.11.017 (cited on page 128).

[AP83] Alberto Apostolico and Franco P. Preparata. „Optimal off-line detection of repetitions in a string.“ In: *Theoretical Computer Science* 22.3 (1983), pages 297–315. DOI: 10.1016/0304-3975(83)90109-3 (cited on page 129).

[Bab+15] Maxim A. Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. „Wavelet trees meet suffix trees.“ In: *Proceedings of the 25th Annual Symposium on Discrete Algorithms (SODA 2015)*. San Diego, CA, USA, 2015, pages 572–591. DOI: 10.1137/1.9781611973730.39 (cited on pages 15, 111).

[BC22] Golnaz Badkobeh and Maxime Crochemore. „Linear construction of a left Lyndon tree.“ In: *Information and Computation* 285.Part B (2022), page 104884. DOI: 10.1016/J.IC.2022.104884 (cited on page 67).

[Bad+22] Golnaz Badkobeh, Maxime Crochemore, **Jonas Ellert**, and Cyril Nicaud. „Back-to-front online Lyndon forest construction.“ In: *Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*. Prague, Czech Republic, 2022, 13:1–13:23. DOI: 10.4230/LIPIcs.CPM.2022.13 (cited on pages 6, 8, 68, 71, 76, 81).

[Bai15] Uwe Baier. „Linear-time suffix sorting - A new approach for suffix array construction.“ Master's thesis. Ulm University, 2015. URL: https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/baier/gsaca.pdf (cited on pages 8, 68).

[Bai16] Uwe Baier. „Linear-time suffix sorting - A new approach for suffix array construction.“ In: *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*. Tel Aviv, Israel, 2016, 23:1–23:12. DOI: 10.4230/LIPIcs.CPM.2016.23 (cited on pages 8, 68, 109).

[BE23] Hideo Bannai and **Jonas Ellert**. „Lyndon arrays in sublinear time.“ In: *Proceedings of the 31st Annual European Symposium on Algorithms (ESA 2023)*. Amsterdam, The Netherlands, 2023, 14:1–14:16. DOI: 10.4230/LIPICS.ESA.2023.14 (cited on pages 7, 71).

[Ban+23] Hideo Bannai, Mitsuru Funakoshi, Kazuhiro Kurita, Yuto Nakashima, Kazuhisa Seto, and Takeaki Uno. „Optimal LZ-End parsing is hard.“ In: *Proceedings of the 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*. Marne-la-Vallée, France, 2023. DOI: 10.4230/LIPIcs.CPM.2023.3 (cited on page 32).

[Ban+17] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. „The “runs” theorem.“ In: *SIAM Journal on Computing* 46.5 (2017), pages 1501–1514. DOI: 10.1137/15M1011032 (cited on pages 4, 68, 71, 127, 129–131, 134, 151, 176).

[BFN11] Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. „LRM-trees: Compressed indices, adaptive sorting, and compressed permutations.“ In: *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*. Palermo, Italy, 2011, pages 285–298. DOI: 10.1007/978-3-642-21458-5_25 (cited on pages 43, 68, 91).

[BCN02] Frédérique Bassino, Julien Clément, and Cyril Nicaud. „The average lengths of the factors of the standard factorization of Lyndon words.“ In: *Proceedings of the 6th International Conference on Developments in Language Theory (DLT 2002)*. Kyoto, Japan, 2002, pages 307–318. DOI: 10.1007/3-540-45005-X_27 (cited on page 67).

[BCN04] Frédérique Bassino, Julien Clément, and Cyril Nicaud. „Lyndon words with a fixed standard right factor.“ In: *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA 2004)*. New Orleans, LA, USA, 2004, pages 653–654. URL: http://dl.acm.org/citation.cfm?id=982792.982891 (cited on page 67).

[BEM79] Dwight R. Bean, Andrzej Ehrenfeucht, and George F. McNulty. „Avoidable patterns in strings of symbols.“ In: *Pacific Journal of Mathematics* 85.2 (1979), pages 261–294. DOI: pjm/1102783913 (cited on page 128).

[Bel12] Djamal Belazzougui. „Worst-case efficient single and multiple string matching on packed texts in the word-RAM model.“ In: *Journal of Discrete Algorithms* 14 (2012), pages 91–106. DOI: 10.1016/J.JDA.2011.12.011 (cited on page 14).

[Bel+21] Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. „Block trees.“ In: *Journal of Computer and System Sciences* 117 (2021), pages 1–22. DOI: 10.1016/J.JCSS.2020.11.002 (cited on page 50).

[Bel+16] Djamal Belazzougui, Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. „Lempel-Ziv decoding in external memory.“ In: *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA 2016)*. St. Petersburg, Russia, 2016, pages 63–74. DOI: 10.1007/978-3-319-38851-9_5 (cited on pages 31, 56).

[BP16] Djamal Belazzougui and Simon J. Puglisi. „Range predecessor and Lempel-Ziv parsing.“ In: *Proceedings of the 27th Annual Symposium on Discrete Algorithms (SODA 2016)*. Arlington, VA, USA, 2016, pages 2053–2071. DOI: 10.1137/1.9781611974331.ch143 (cited on pages 31, 32, 45, 46, 55).

[Ben+14] Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. „Towards optimal packed string matching.“ In: *Theoretical Computer Science* 525 (2014), pages 111–129. DOI: 10.1016/J.TCS.2013.06.013 (cited on page 14).

[BF00] Michael A. Bender and Martin Farach-Colton. „The LCA problem revisited.“ In: *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN 2000)*. Punta del Este, Uruguay, 2000, pages 88–94. DOI: 10.1007/10719839_9 (cited on page 171).

[BSV93] Omer Berkman, Baruch Schieber, and Uzi Vishkin. „Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values.“ In: *Journal of Algorithms* 14.3 (1993), pages 344–370. DOI: 10.1006/JAGM.1993.1018 (cited on page 68).

[Ber94] Jean Berstel. „Axel Thue's papers on repetitions in words: A translation.“ In: (1994). URL: https://www-igm.univ-mlv.fr/~berstel/Articles/1994ThueTranslation.pdf (cited on pages 16, 128).

[BB99] Jean Berstel and Luc Boasson. „Partial words and a theorem of Fine and Wilf.“ In: *Theoretical Computer Science* 218.1 (1999), pages 135–141. DOI: 10.1016/S0304-3975(98)00255-2 (cited on page 128).

[BEF21] Nico Bertram, **Jonas Ellert**, and Johannes Fischer. „Lyndon words accelerate suffix sorting.“ In: *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021)*. Lisbon, Portugal (Virtual Conference), 2021, 15:1–15:13. DOI: 10.4230/LIPIcs.ESA.2021.15 (cited on pages 8, 9, 68).

[Bet+23] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, Wesam Manassra, Prafulla Dhariwal, Casey Chu, Yunxin Jiao, and Aditya Ramesh. *Improving image generation with better captions.* Technical report. OpenAI, 2023. URL: https://cdn.openai.com/papers/dall-e-3.pdf (cited on pages 14, 16, 29, 67, 127).

[Bil11] Philip Bille. „Fast searching in packed strings.“ In: *Journal of Discrete Algorithms* 9.1 (2011), pages 49–56. DOI: 10.1016/J.JDA.2010.09.003 (cited on page 14).

[Bil+17] Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz. „Lempel-Ziv compression in a sliding window.“ In: *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*. Warsaw, Poland, 2017, 15:1–15:11. DOI: 10.4230/LIPIcs.CPM.2017.15 (cited on page 31).

[Bil+20] Philip Bille, **Jonas Ellert**, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. „Space efficient construction of Lyndon arrays in linear time.“ In: *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Saarbrücken, Germany (Virtual Conference), 2020, 14:1–14:18. DOI: 10.4230/LIPIcs.ICALP.2020.14 (cited on pages 6, 7, 71, 73, 131, 135).

[Bil+18] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. „Time–space trade-offs for Lempel–Ziv compressed indexing.“ In: *Theoretical Computer Science* 713 (2018), pages 66–77. DOI: https://doi.org/10.1016/j.tcs.2017.12.021 (cited on page 31).

[BGS17] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. „Deterministic indexing for packed strings.“ In: *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*. Warsaw, Poland, 2017, 6:1–6:11. DOI: 10.4230/LIPICS.CPM.2017.6 (cited on page 15).

[BGS20] Philip Bille, Inge Li Gørtz, and Teresa Anna Steiner. „String indexing with compressed patterns." In: *Proceedings of the 37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020)*. Montpellier, France, 2020, 10:1–10:13. DOI: 10.4230/LIPIcs.STACS.2020.10 (cited on page 31).

[BBS08] Francine Blanchet-Sadri, Deepak Bal, and Gautam Sisodia. „Graph connectivity, partial words, and a theorem of Fine and Wilf." In: *Information and Computation* 206.5 (2008), pages 676–693. DOI: 10.1016/j.ic.2007.11.007 (cited on page 128).

[BH02] Francine Blanchet-Sadri and Robert A. Hegstrom. „Partial words and a theorem of Fine and Wilf revisited." In: *Theoretical Computer Science* 270.1-2 (2002), pages 401–419. DOI: 10.1016/S0304-3975(00)00407-2 (cited on page 128).

[Bla+13] Francine Blanchet-Sadri, Sean Simmons, Amelia Tebbe, and Amy Veprauskas. „Abelian periods, partial words, and an extension of a theorem of Fine and Wilf." In: *RAIRO Theoretical Informatics and Applications* 47.3 (2013), pages 215–234. DOI: 10.1051/ita/2013034 (cited on page 128).

[Bre92] Dany Breslauer. „Efficient string algorithmics." PhD thesis. Columbia University, 1992. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.1251 (cited on pages 1, 5, 129, 130).

[BJJ97] Dany Breslauer, Tao Jiang, and Zhigen Jiang. „Rotations of periodic strings and short superstrings." In: *Journal of Algorithms* 24.2 (1997), pages 340–353. DOI: 10.1006/jagm.1997.0861 (cited on page 67).

[BL23] Srecko Brlek and Shuo Li. „On the number of distinct squares in finite sequences: Some old and new results." In: *Proceedings of the 14th International Conference on Combinatorics on Words (WORDS 2023)*. Umeå, Sweden, 2023, pages 35–44. DOI: 10.1007/978-3-031-33180-0_3 (cited on page 128).

[BL22] Srečko Brlek and Shuo Li. „On the number of squares in a finite word." In: *CoRR* abs/2204.10204 (2022). DOI: 10.48550/arXiv.2204.10204. arXiv: 2204.10204 (cited on page 128).

[BM13] Helen Budworth and Cynthia T. McMurray. „A brief history of triplet repeat diseases." In: *Trinucleotide Repeat Protocols*. 2nd edition. Volume 1010. Methods in Molecular Biology. Springer, 2013, pages 3–17. ISBN: 978-1-62703-411-1. DOI: 10.1007/978-1-62703-411-1_1 (cited on page 128).

[CMR99] Maria Gabriella Castelli, Filippo Mignosi, and Antonio Restivo. „Fine and Wilf's theorem for three periods and a generalization of Sturmian words." In: *Theoretical Computer Science* 218.1 (1999), pages 83–94. DOI: 10.1016/S0304-3975(98)00251-5 (cited on page 128).

[CT18] Timothy M. Chan and Konstantinos Tsakalidis. „Dynamic orthogonal range searching on the RAM, revisited." In: *Journal of Computational Geometry* 9.2 (2018), pages 45–66. DOI: 10.20382/jocg.v9i2a5 (cited on pages 48, 49, 56).

[Cha+21] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. „Faster algorithms for longest common substring.“ In: *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021)*. Lisbon, Portugal (Virtual Conference), 2021, 30:1–30:17. DOI: 10.4230/LIPICS.ESA.2021.30 (cited on page 15).

[CPR22] Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. „Longest palindromic substring in sublinear time.“ In: *Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*. Prague, Czech Republic, 2022, 20:1–20:9. DOI: 10.4230/LIPICS.CPM.2022.20 (cited on page 15).

[Cha+05] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. „The smallest grammar problem.“ In: *IEEE Transactions on Information Theory* 51.7 (2005), pages 2554–2576. DOI: 10.1109/TIT.2005.850116 (cited on page 30).

[CPS19] Émilie Charlier, Manon Philibert, and Manon Stipulanti. „Nyldon words.“ In: *Journal of Combinatorial Theory, Series A* 167 (2019), pages 60–90. DOI: 10.1016/j.jcta.2019.04.002 (cited on page 69).

[CFL58] Kuo Tsai Chen, Ralph Hartzler Fox, and Roger Conant Lyndon. „Free differential calculus, iv. The quotient groups of the lower central series.“ In: *Annals of Mathematics* 68.1 (1958), pages 81–95. DOI: 10.2307/1970044 (cited on pages 69, 71, 106).

[Cie+17] Adam Ciesiolka, Magdalena Jazurek, Karolina Drazkowska, and Wlodzimierz J. Krzyzosiak. „Structural characteristics of simple RNA repeats associated with disease and their deleterious protein interactions.“ In: *Frontiers in Cellular Neuroscience* 11 (2017). DOI: 10.3389/fncel.2017.00097 (cited on page 128).

[CH97] Richard Cole and Ramesh Hariharan. „Tighter upper bounds on the exact complexity of string matching.“ In: *SIAM Journal on Computing* 26.3 (1997), pages 803–856. DOI: 10.1137/S009753979324694X (cited on page 16).

[Col+95] Richard Cole, Ramesh Hariharan, Mike Paterson, and Uri Zwick. „Tighter lower bounds on the exact complexity of string matching.“ In: *SIAM Journal on Computing* 24.1 (1995), pages 30–45. DOI: 10.1137/S0097539793245829 (cited on page 16).

[CI06] Sorin Constantinescu and Lucian Ilie. „Fine and Wilf's theorem for abelian periods.“ In: *Bulletin of the EATCS* 89 (2006), pages 167–170. URL: https://eatcs.org/images/bulletin/beatcs89.pdf (cited on page 128).

[Cor+22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. 4th edition. MIT Press, 2022. ISBN: 9780262367509 (cited on pages 13, 17–19, 91).

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. 2nd edition. Wiley, 2006. ISBN: 978-0-471-24195-9. URL: http://www.elementsofinformationtheory.com/ (cited on page 155).

[Cro81] Maxime Crochemore. „An optimal algorithm for computing the repetitions in a word." In: *Information Processing Letters* 12.5 (1981), pages 244–250. DOI: 10.1016/0020-0190(81)90024-7 (cited on pages 127, 129).

[Cro86] Maxime Crochemore. „Transducers and repetitions." In: *Theoretical Computer Science* 45.1 (1986), pages 63–86. DOI: 10.1016/0304-3975(86)90041-1 (cited on pages 50, 129, 148).

[Cro+12] Maxime Crochemore, Laura Giambruno, Alessio Langiu, Filippo Mignosi, and Antonio Restivo. „Dictionary-symbolwise flexible parsing." In: *Journal of Discrete Algorithms* 14 (2012), pages 74–90. DOI: 10.1016/j.jda.2011.12.021 (cited on page 31).

[CI08a] Maxime Crochemore and Lucian Ilie. „Computing longest previous factor in linear time and applications." In: *Information Processing Letters* 106.2 (2008), pages 75–80. DOI: 10.1016/j.ipl.2007.10.006 (cited on pages 3, 17, 30, 31, 40, 42, 43, 148).

[CI08b] Maxime Crochemore and Lucian Ilie. „Maximal repetitions in strings." In: *Journal of Computer and System Sciences* 74.5 (2008), pages 796–807. DOI: 10.1016/j.jcss.2007.09.003 (cited on page 129).

[CIT11] Maxime Crochemore, Lucian Ilie, and Liviu Tinta. „The "runs" conjecture." In: *Theoretical Computer Science* 412.27 (2011), pages 2931–2941. DOI: 10.1016/j.tcs.2010.06.019 (cited on page 129).

[Cro+16] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. „Near-optimal computation of runs over general alphabet via non-crossing LCE queries." In: *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*. Beppu, Japan, 2016, pages 22–34. DOI: 10.1007/978-3-319-46049-9_3 (cited on page 130).

[Cro+14] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. „Extracting powers and periods in a word from its runs structure." In: *Theoretical Computer Science* 521 (2014), pages 29–41. DOI: 10.1016/j.tcs.2013.11.018 (cited on page 134).

[CLM13] Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. „The rightmost equal-cost position problem." In: *Proceedings of the 2013 Data Compression Conference (DCC 2013)*. Snowbird, UT, USA, 2013, pages 421–430. DOI: 10.1109/DCC.2013.50 (cited on page 31).

[CLR21] Maxime Crochemore, Thierry Lecroq, and Wojciech Rytter. *125 problems in text algorithms*. 334 pages. Cambridge University Press, 2021. ISBN: 9781108869317. DOI: 10.1017/9781108835831 (cited on page 71).

[CP91] Maxime Crochemore and Dominique Perrin. „Two-way string-matching." In: *Journal of the ACM* 38.3 (1991), pages 651–675. DOI: 10.1145/116825.116845 (cited on page 67).

[CR20] Maxime Crochemore and Luís M. S. Russo. „Cartesian and Lyndon trees." In: *Theoretical Computer Science* 806 (2020), pages 1–9. DOI: 10.1016/j.tcs.2018.08.011 (cited on page 67).

[CR91] Maxime Crochemore and Wojciech Rytter. „Efficient parallel algorithms to test square-freeness and factorize strings.“ In: *Information Processing Letters* 38.2 (1991), pages 57–60. DOI: 10.1016/0020-0190(91)90223-5 (cited on page 30).

[Cur05] James D. Currie. „Pattern avoidance: Themes and variations.“ In: *Theoretical Computer Science* 339.1 (2005), pages 7–18. DOI: 10.1016/j.tcs.2005.01.004 (cited on page 128).

[Day+18] Jacqueline W. Daykin, Frantisek Franek, Jan Holub, A. S. M. Sohidull Islam, and W. F. Smyth. „Reconstructing a string from its Lyndon arrays.“ In: *Theoretical Computer Science* 710 (2018), pages 44–51. DOI: 10.1016/J.TCS.2017.04.008 (cited on pages 68, 71).

[DFT15] Antoine Deza, Frantisek Franek, and Adrien Thierry. „How many double squares can a string contain?“ In: *Discrete Applied Mathematics* 180 (2015), pages 52–69. DOI: 10.1016/j.dam.2014.08.016 (cited on page 128).

[Duv83] Jean-Pierre Duval. „Factorizing words over an ordered alphabet.“ In: *Journal of Algorithms* 4.4 (1983), pages 363–381. DOI: 10.1016/0196-6774(83)90017-2 (cited on pages 71, 106, 107).

[DLL14] Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre. „Linear computation of unbordered conjugate on unordered alphabet.“ In: *Theoretical Computer Science* 522 (2014), pages 77–84. DOI: 10.1016/j.tcs.2013.12.008 (cited on page 17).

[Eli75] Peter Elias. „Universal codeword sets and representations of the integers.“ In: *IEEE Transactions on Information Theory* 21.2 (1975), pages 194–203. DOI: 10.1109/TIT.1975.1055349 (cited on pages 31, 51, 52).

[Ell22] **Jonas Ellert**. „Lyndon arrays simplified.“ In: *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*. Potsdam, Germany, 2022, 48:1–48:14. DOI: 10.4230/LIPICS.ESA.2022.48 (cited on pages 7, 131, 135).

[Ell23] **Jonas Ellert**. „Sublinear time Lempel-Ziv (LZ77) factorization.“ In: *Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)*. Pisa, Italy, 2023, pages 171–187. DOI: 10.1007/978-3-031-43980-3_14 (cited on pages 6, 31).

[EF21] **Jonas Ellert** and Johannes Fischer. „Linear time runs over general ordered alphabets.“ In: *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Glasgow, Scotland (Virtual Conference), 2021, 63:1–63:16. DOI: 10.4230/LIPIcs.ICALP.2021.63 (cited on pages 8, 17).

[EFP23] **Jonas Ellert**, Johannes Fischer, and Max Rishøj Pedersen. „New advances in rightmost Lempel-Ziv.“ In: *Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)*. **Winner of the SPIRE 2023 Best Paper Award**. Pisa, Italy, 2023, pages 188–202. DOI: 10.1007/978-3-031-43980-3_15 (cited on pages 6, 31).

[EFS20] **Jonas Ellert**, Johannes Fischer, and Nodari Sitchinava. „LCP-aware parallel string sorting.“ In: *Proceedings of the 26th International Conference on Parallel and Distributed Computing (Euro-Par 2020)*. Warsaw, Poland (Virtual Conference), 2020, pages 329–342. DOI: 10.1007/978-3-030-57675-2_21 (cited on page 9).

[EGG23a] **Jonas Ellert**, Paweł Gawrychowski, and Garance Gourdel. „Optimal square detection over general alphabets.“ In: *Proceedings of the 34th Annual Symposium on Discrete Algorithms (SODA 2023)*. Florence, Italy, 2023, pages 5220–5242. DOI: 10.1137/1.9781611977554.ch189 (cited on pages 6, 8, 17).

[EGG23b] **Jonas Ellert**, Paweł Gawrychowski, and Garance Gourdel. „Optimal square detection over general alphabets.“ In: *CoRR* abs/2303.07229 (2023). DOI: 10.48550/ARXIV.2303.07229. arXiv: 2303.07229 (cited on pages 6, 8).

[Far97] Martin Farach. „Optimal suffix tree construction with large alphabets.“ In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*. Miami Beach, FL, USA, 1997, pages 137–143. DOI: 10.1109/SFCS.1997.646102 (cited on pages 53, 151, 168, 169).

[FM95] Martin Farach and S. Muthukrishnan. „Optimal parallel dictionary matching and compression (extended abstract).“ In: *Proceedings of the 7th Annual Symposium on Parallel Algorithms and Architectures (SPAA 1995)*. Santa Barbara, CA, USA, 1995, pages 244–253. DOI: 10.1145/215399.215451 (cited on page 30).

[FFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. „On the sorting-complexity of suffix tree construction.“ In: *Journal of the ACM* 47.6 (2000), pages 987–1011. DOI: 10.1145/355541.355547 (cited on page 53).

[Fer+14] Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J. Puglisi. „Hybrid indexes for repetitive datasets.“ In: *Philosophical Transactions of the Royal Society A* 372.2016 (2014). DOI: 10.1098/rsta.2013.0137 (cited on page 31).

[FNV13] Paolo Ferragina, Igor Nitto, and Rossano Venturini. „On the bit-complexity of Lempel-Ziv compression.“ In: *SIAM Journal on Computing* 42.4 (2013), pages 1521–1541. DOI: 10.1137/120869511 (cited on pages 31, 57).

[FW65] Nathan J. Fine and Herbert S. Wilf. „Uniqueness theorems for periodic functions.“ In: *Proceedings of the American Mathematical Society* 16.1 (1965), pages 109–114. DOI: 10.1090/s0002-9939-1965-0174934-9 (cited on pages 119, 128, 133).

[Fis10] Johannes Fischer. „Optimal succinctness for range minimum queries.“ In: *Proceedings of the 9th Latin American Symposium Theoretical Informatics (LATIN 2010)*. Oaxaca, Mexico, 2010, pages 158–169. DOI: 10.1007/978-3-642-12200-2_16 (cited on page 91).

[Fis+15a] Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. „Approximating LZ77 via small-space multiple-pattern matching.“ In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA 2015)*. Patras, Greece, 2015, pages 533–544. DOI: 10.1007/978-3-662-48350-3_45 (cited on page 31).

[Fis+15b] Johannes Fischer, Stepan Holub, Tomohiro I, and Moshe Lewenstein. „Beyond the runs theorem.“ In: *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*. London, UK, 2015, pages 277–286. DOI: 10.1007/978-3-319-23826-5_27 (cited on page 129).

[FIK15] Johannes Fischer, Tomohiro I, and Dominik Köppl. „Lempel Ziv computation in small space (LZ-CISS).“ In: *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*. Ischia Island, Italy, 2015, pages 172–184. DOI: 10.1007/978-3-319-19929-0_15 (cited on page 31).

[Fis+18] Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. „Lempel-Ziv factorization powered by space efficient suffix trees.“ In: *Algorithmica* 80.7 (2018), pages 2048–2081. DOI: 10.1007/s00453-017-0333-1 (cited on pages 31, 53, 165).

[FK17] Johannes Fischer and Florian Kurpicz. „Dismantling DivSufSort.“ In: *Proceedings of the 2017 Prague Stringology Conference (PSC 2017)*. Prague, Czech Republic, 2017, pages 62–76. URL: http://www.stringology.org/event/2017/p07.html (cited on page 108).

[FK96] Aviezri S. Fraenkel and Shmuel T. Klein. „Robust universal complete codes for transmission and compression.“ In: *Discrete Applied Mathematics* 64.1 (1996), pages 31–55. DOI: 10.1016/0166-218X(93)00116-H (cited on page 52).

[FS98] Aviezri S. Fraenkel and Jamie Simpson. „How many squares can a string contain?“ In: *Journal of Combinatorial Theory, Series A* 82.1 (1998), pages 112–120. DOI: 10.1006/jcta.1997.2843 (cited on page 128).

[Fra+16] Frantisek Franek, A. S. M. Sohidull Islam, Mohammad Sohel Rahman, and William F. Smyth. „Algorithms to compute the Lyndon array.“ In: *Proceedings of the 2016 Prague Stringology Conference (PSC 2016)*. Prague, Czech Republic, 2016, pages 172–184. URL: http://www.stringology.org/event/2016/p15.html (cited on pages 68, 71–73, 89, 108, 132).

[FL19] Frantisek Franek and Michael Liut. „Algorithms to compute the Lyndon array revisited.“ In: *Proceedings of the 2019 Prague Stringology Conference (PSC 2019)*. Prague, Czech Republic, 2019, pages 16–28. URL: http://www.stringology.org/event/2019/p03.html (cited on pages 68, 71).

[FL20] Frantisek Franek and Michael Liut. „Computing maximal Lyndon substrings of a string.“ In: *Algorithms* 13.11 (2020). DOI: 10.3390/a13110294 (cited on pages 68, 71–73, 132).

[FLS18] Frantisek Franek, Michael Liut, and William F. Smyth. „On Baier's sort of maximal Lyndon substrings.“ In: *Proceedings of the 2018 Prague Stringology Conference (PSC 2018)*. Prague, Czech Republic, 2018, pages 63–78. URL: http://www.stringology.org/event/2018/p07.html (cited on page 68).

[FPS17] Frantisek Franek, Asma Paracha, and William F. Smyth. „The linear equivalence of the suffix array and the partially sorted Lyndon array.“ In: *Proceedings of the 2017 Prague Stringology Conference (PSC 2017)*. Prague, Czech Republic, 2017, pages 77–84. URL: http://www.stringology.org/event/2017/p08.html (cited on page 68).

[FY08] Frantisek Franek and Qian Yang. „An asymptotic lower bound for the maximal number of runs in a string.“ In: *International Journal of Foundations of Computer Science* 19.1 (2008), pages 195–203. DOI: 10.1142/S0129054108005620 (cited on page 129).

[Fre60] Edward Fredkin. „Trie memory.“ In: *Communications of the ACM* 3.9 (1960), pages 490–499. DOI: 10.1145/367390.367400 (cited on page 53).

[FW93] Michael L. Fredman and Dan E. Willard. „Surpassing the information theoretic bound with fusion trees.“ In: *Journal of Computer and System Sciences* 47.3 (1993), pages 424–436. DOI: 10.1016/0022-0000(93)90040-4 (cited on page 95).

[Gab90] Harold N. Gabow. „Data structures for weighted matching and nearest common ancestors with linking.“ In: *Proceedings of the 1st Annual Symposium on Discrete Algorithms (SODA 1990)*. San Francisco, California, USA, 1990, pages 434–443. URL: http://dl.acm.org/citation.cfm?id=320176.320229 (cited on page 163).

[Gag22] Travis Gagie. „Space-efficient RLZ-to-LZ77 conversion.“ In: *CoRR* abs/2211.13254 (2022). DOI: 10.48550/arXiv.2211.13254. arXiv: 2211.13254 (cited on page 31).

[Gag+14] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. „LZ77-based self-indexing with faster pattern matching.“ In: *Proceedings of the 11th Latin American Symposium on Theoretical Informatics (LATIN 2014)*. Montevideo, Uruguay, 2014, pages 731–742. DOI: 10.1007/978-3-642-54423-1_63 (cited on page 31).

[GGP15] Travis Gagie, Paweł Gawrychowski, and Simon J. Puglisi. „Approximate pattern matching in LZ77-compressed texts.“ In: *Journal of Discrete Algorithms* 32 (2015), pages 64–68. DOI: 10.1016/j.jda.2014.10.003 (cited on page 31).

[GNP18] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. „On the approximation ratio of Lempel-Ziv parsing.“ In: *Proceedings of the 13th Latin American Theoretical Informatics Symposium (LATIN 2018)*. Buenos Aires, Argentina, 2018, pages 490–503. DOI: 10.1007/978-3-319-77404-6_36 (cited on page 30).

[GP92] Zvi Galil and Kunsoo Park. „Truly alphabet-independent two-dimensional pattern matching.“ In: *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS 1992)*. Pittsburgh, PA, USA, 1992, pages 247–256. DOI: 10.1109/SFCS.1992.267767 (cited on page 16).

[GS83] Zvi Galil and Joel I. Seiferas. „Time-space-optimal string matching.“ In: *Journal of Computer and System Sciences* 26.3 (1983), pages 280–294. DOI: 10.1016/0022-0000(83)90002-8 (cited on page 16).

[Gaw+16] Paweł Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. „Faster longest common extension queries in strings over general alphabets.“ In: *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*. Tel Aviv, Israel, 2016, 5:1–5:13. DOI: 10.4230/LIPIcs.CPM.2016.5 (cited on pages 130, 150, 163).

[GKM23] Pawel Gawrychowski, Maria Kosche, and Florin Manea. „On the number of factors in the LZ-End factorization.“ In: *Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)*. Pisa, Italy, 2023, pages 253–259. DOI: 10.1007/978-3-031-43980-3_20 (cited on page 32).

[GGF13] Emanuele Giaquinta, Szymon Grabowski, and Kimmo Fredriksson. „Approximate pattern matching with k-mismatches in packed text.“ In: *Information Processing Letters* 113.19-21 (2013), pages 693–697. DOI: 10.1016/J.IPL.2013.07.002 (cited on page 15).

[Gir08] Mathieu Giraud. „Not so many runs in strings.“ In: *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications (LATA 2008)*. Tarragona, Spain, 2008, pages 232–239. DOI: 10.1007/978-3-540-88282-4_22 (cited on page 129).

[Gir09] Mathieu Giraud. „Asymptotic behavior of the numbers of runs and microruns.“ In: *Information and Computation* 207.11 (2009), pages 1221–1228. DOI: 10.1016/j.ic.2009.02.007 (cited on page 129).

[Gol66] Solomon W. Golomb. „Run-length encodings (corresp.)“ In: *IEEE Transactions on Information Theory* 12.3 (1966), pages 399–401. DOI: 10.1109/TIT.1966.1053907 (cited on page 52).

[Gol07] Alexander Golynski. „Optimal lower bounds for rank and select indexes.“ In: *Theoretical Computer Science* 387.3 (2007), pages 348–359. DOI: 10.1016/j.tcs.2007.07.041 (cited on page 92).

[GB13] Keisuke Goto and Hideo Bannai. „Simpler and faster Lempel Ziv factorization.“ In: *Proceedings of the 2013 Data Compression Conference (DCC 2013)*. Snowbird, UT, USA, 2013, pages 133–142. DOI: 10.1109/DCC.2013.21 (cited on page 31).

[GB14] Keisuke Goto and Hideo Bannai. „Space efficient linear time Lempel-Ziv factorization for small alphabets.“ In: *Proceedings of the 2014 Data Compression Conference (DCC 2014)*. Snowbird, UT, USA, 2014, pages 163–172. DOI: 10.1109/DCC.2014.62 (cited on page 31).

[Gou+20] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny M. Shur, and Tomasz Walen. „String periods in the order-preserving model.“ In: *Information and Computation* 270 (2020). DOI: 10.1016/j.ic.2019.104463 (cited on page 128).

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences - Computer science and computational biology.* Cambridge University Press, 1997. ISBN: 0-521-58519-8. DOI: 10.1017/cbo9780511574931 (cited on pages 151, 168).

[Hag98] Torben Hagerup. „Sorting and searching on the word RAM.“ In: *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998).* Paris, France, 1998, pages 366–398. DOI: 10.1007/BFb0028575 (cited on pages 1, 13).

[HR03] Christophe Hohlweg and Christophe Reutenauer. „Lyndon words, permutations and trees.“ In: *Theoretical Computer Science* 307.1 (2003), pages 173–178. DOI: 10.1016/S0304-3975(03)00099-9 (cited on pages 67, 72, 73, 132).

[Hol17] Stepan Holub. „Prefix frequency of lost positions.“ In: *Theoretical Computer Science* 684 (2017), pages 43–52. DOI: 10.1016/j.tcs.2017.01.026 (cited on page 129).

[HRB23] Aaron Hong, Massimiliano Rossi, and Christina Boucher. „LZ77 via prefix-free parsing.“ In: *Proceedings of the 25th Symposium on Algorithm Engineering and Experiments (ALENEX 2023).* Florence, Italy, 2023, pages 123–134. DOI: 10.1137/1.9781611977561.ch11 (cited on page 31).

[HC08] Jin-Ju Hong and Gen-Huey Chen. „Efficient on-line repetition detection.“ In: *Theoretical Computer Science* 407.1-3 (2008), pages 554–563. DOI: 10.1016/j.tcs.2008.08.038 (cited on page 129).

[IS14] Lidia A. Idiatulina and Arseny M. Shur. „Periodic partial words and random bipartite graphs.“ In: *Fundamenta Informaticae* 132.1 (2014), pages 15–31. DOI: 10.3233/FI-2014-1030 (cited on page 128).

[Ili07] Lucian Ilie. „A note on the number of squares in a word.“ In: *Theoretical Computer Science* 380.3 (2007), pages 373–376. DOI: 10.1016/j.tcs.2007.03.025 (cited on page 128).

[JSS15] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. „Linked dynamic tries with applications to LZ-compression in sublinear time and space.“ In: *Algorithmica* 71.4 (2015), pages 969–988. DOI: 10.1007/S00453-013-9836-6 (cited on page 31).

[Jus00] Jacques Justin. „On a paper by Castelli, Mignosi, Restivo.“ In: *RAIRO Theoretical Informatics and Applications* 34.5 (2000), pages 373–377. DOI: 10.1051/ita:2000122 (cited on page 128).

[KKP13a] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. „Lightweight Lempel-Ziv parsing.“ In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013).* Rome, Italy, 2013, pages 139–150. DOI: 10.1007/978-3-642-38527-8_14 (cited on page 31).

[KKP13b] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. „Linear time Lempel-Ziv factorization: Simple, fast, small." In: *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*. Bad Herrenalb, Germany, 2013, pages 189–200. DOI: 10.1007/978-3-642-38905-4_19 (cited on page 31).

[KKP14] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. „Lempel-Ziv parsing in external memory." In: *Proceedings of the 2014 Data Compression Conference (DCC 2014)*. Snowbird, UT, USA, 2014, pages 153–162. DOI: 10.1109/DCC.2014.78 (cited on page 31).

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. „Linear work suffix array construction." In: *Journal of the ACM* 53.6 (2006), pages 918–936. DOI: 10.1145/1217856.1217858 (cited on page 68).

[KS98] Juha Kärkkäinen and Erkki Sutinen. „Lempel-Ziv index for q-grams." In: *Algorithmica* 21.1 (1998), pages 137–154. DOI: 10.1007/PL00009205 (cited on page 31).

[Kat+21] Kenneth Katz, Oleg Shutov, Richard Lapoint, Michael Kimelman, J Rodney Brister, and Christopher O'Sullivan. „The sequence read archive: A decade more of explosive growth." In: *Nucleic Acids Research* 50.D1 (2021), pages D387–D390. DOI: 10.1093/nar/gkab1053 (cited on page 29).

[Kem19] Dominik Kempa. „Optimal construction of compressed indexes for highly repetitive texts." In: *Proceedings of the 30th Annual Symposium on Discrete Algorithms (SODA 2019)*. San Diego, CA, USA, 2019, pages 1344–1357. DOI: 10.1137/1.9781611975482.82 (cited on pages 31, 39).

[KK19] Dominik Kempa and Tomasz Kociumaka. „String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure." In: *Proceedings of the 51st Annual Symposium on Theory of Computing (STOC 2019)*. Phoenix, AZ, USA, 2019, pages 756–767. DOI: 10.1145/3313276.3316368 (cited on pages 15, 40, 42, 61, 113).

[KK22] Dominik Kempa and Tomasz Kociumaka. „Resolution of the Burrows-Wheeler transform conjecture." In: *Communications of the ACM* 65.6 (2022), pages 91–98. DOI: 10.1145/3531445 (cited on pages 30, 31, 39).

[KK23] Dominik Kempa and Tomasz Kociumaka. „Breaking the O(n)-barrier in the construction of compressed suffix arrays and suffix trees." In: *Proceedings of the 34th Annual Symposium on Discrete Algorithms (SODA 2023)*. Florence, Italy, 2023, pages 5122–5202. DOI: 10.1137/1.9781611977554.CH187 (cited on page 15).

[KK17] Dominik Kempa and Dmitry Kosolobov. „LZ-End parsing in linear time." In: *Proceedings of the 25th Annual European Symposium on Algorithms (ESA 2017)*. Vienna, Austria, 2017, 53:1–53:14. DOI: 10.4230/LIPIcs.ESA.2017.53 (cited on page 32).

[KP18] Dominik Kempa and Nicola Prezza. „At the roots of dictionary compression: String attractors." In: *Proceedings of the 50th Annual Symposium on Theory of Computing (STOC 2018)*. Los Angeles, CA, USA, 2018, pages 827–840. DOI: 10.1145/3188745.3188814 (cited on page 30).

[KS22] Dominik Kempa and Barna Saha. „An upper bound and linear-space queries on the LZ-End parsing.“ In: *Proceedings of the 33rd Annual Symposium on Discrete Algorithms (SODA 2022)*. Alexandria, VA, USA (Virtual Conference), 2022, pages 2847–2866. DOI: 10.1137/1.9781611977073.111 (cited on page 32).

[Kid+03] Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. „Collage system: A unifying framework for compressed pattern matching.“ In: *Theoretical Computer Science* 298.1 (2003), pages 253–272. DOI: https://doi.org/10.1016/S0304-3975(02)00426-7 (cited on page 121).

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. „Fast pattern matching in strings.“ In: *SIAM Journal on Computing* 6.2 (1977), pages 323–350. DOI: 10.1137/0206024 (cited on pages 1, 16).

[KNP20] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. „Towards a definitive measure of repetitiveness.“ In: *Proceedings of the 14th Latin American Symposium on Theoretical Informatics (LATIN 2020)*. São Paulo, Brazil, 2020, pages 207–219. DOI: 10.1007/978-3-030-61792-9_17 (cited on page 30).

[Koc+22] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. „A periodicity lemma for partial words.“ In: *Information and Computation* 283 (2022), page 104677. DOI: 10.1016/j.ic.2020.104677 (cited on page 128).

[KBK03] Roman M. Kolpakov, Ghizlane Bana, and Gregory Kucherov. „mreps: Efficient and flexible detection of tandem repeats in DNA.“ In: *Nucleic Acids Research* 31.13 (2003), pages 3672–3678. DOI: 10.1093/NAR/GKG617 (cited on page 128).

[KK99] Roman M. Kolpakov and Gregory Kucherov. „Finding maximal repetitions in a word in linear time.“ In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*. New York, NY, USA, 1999, pages 596–604. DOI: 10.1109/SFFCS.1999.814634 (cited on pages 50, 129).

[Köp21] Dominik Köppl. „Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees.“ In: *Algorithms* 14.2 (2021), page 44. DOI: 10.3390/a14020044 (cited on page 31).

[KNP22] Dominik Köppl, Gonzalo Navarro, and Nicola Prezza. „HOLZ: High-order entropy encoding of Lempel-Ziv factor distances.“ In: *Proceedings of the 2022 Data Compression Conference (DCC 2022)*. Snowbird, UT, USA, 2022, pages 83–92. DOI: 10.1109/DCC52660.2022.00016 (cited on page 31).

[KS16] Dominik Köppl and Kunihiko Sadakane. „Lempel-Ziv computation in compressed space (LZ-CICS).“ In: *Proceedings of the 2016 Data Compression Conference (DCC 2016)*. Snowbird, UT, USA, 2016, pages 3–12. DOI: 10.1109/DCC.2016.38 (cited on page 31).

[Kos94] S. Rao Kosaraju. „Computation of squares in a string (preliminary version).“ In: *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM 1994)*. Asilomar, CA, USA, 1994, pages 146–150. DOI: 10.1007/3-540-58094-8_13 (cited on page 129).

[Kos14] Dmitry Kosolobov. „Online square detection.“ In: *CoRR* abs/1411.2022 (2014). DOI: 10.48550/arXiv.1411.2022. arXiv: 1411.2022 (cited on page 129).

[Kos15a] Dmitry Kosolobov. „Faster lightweight Lempel-Ziv parsing.“ In: *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015)*. Milan, Italy, 2015, pages 432–444. DOI: 10.1007/978-3-662-48054-0_36 (cited on page 31).

[Kos15b] Dmitry Kosolobov. „Lempel-Ziv factorization may be harder than computing all runs.“ In: *Proceedings of the 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*. Garching, Germany, 2015, pages 582–593. DOI: 10.4230/LIPICS.STACS.2015.582 (cited on pages 17, 20, 35, 130).

[Kos15c] Dmitry Kosolobov. „Online detection of repetitions with backtracking.“ In: *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*. Ischia Island, Italy, 2015, pages 295–306. DOI: 10.1007/978-3-319-19929-0_25 (cited on page 129).

[Kos16a] Dmitry Kosolobov. „Computing runs on a general alphabet.“ In: *Information Processing Letters* 116.3 (2016), pages 241–244. DOI: 10.1016/j.ipl.2015.11.016 (cited on pages 5, 130).

[Kos16b] Dmitry Kosolobov. „Finding the leftmost critical factorization on unordered alphabet.“ In: *Theoretical Computer Science* 636 (2016), pages 56–65. DOI: 10.1016/j.tcs.2016.04.037 (cited on page 17).

[Kos+20] Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. „Lempel-Ziv-like parsing in small space.“ In: *Algorithmica* 82.11 (2020), pages 3195–3215. DOI: 10.1007/s00453-020-00722-6 (cited on page 31).

[KN10] Sebastian Kreft and Gonzalo Navarro. „LZ77-like compression with fast random access.“ In: *Proceedings of the 2010 Data Compression Conference (DCC 2010)*. Snowbird, UT, USA, 2010, pages 239–248. DOI: 10.1109/DCC.2010.29 (cited on pages 32, 51).

[KN13] Sebastian Kreft and Gonzalo Navarro. „On compressing and indexing repetitive sequences.“ In: *Theoretical Computer Science* 483 (2013), pages 115–133. DOI: 10.1016/j.tcs.2012.02.006 (cited on pages 31, 32, 51).

[Lar14] N. Jesper Larsson. „Most recent match queries in on-line suffix trees.“ In: *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*. Moscow, Russia, 2014, pages 252–261. DOI: 10.1007/978-3-319-07566-2_26 (cited on page 31).

[LZ76] Abraham Lempel and Jacob Ziv. „On the complexity of finite sequences.“ In: *IEEE Transactions on Information Theory* 22.1 (1976), pages 75–81. DOI: 10.1109/TIT.1976.1055501 (cited on pages 3, 17, 29, 30, 39, 40, 148).

[Lev+07] Samuel Levy, Granger Sutton, Pauline C Ng, Lars Feuk, Aaron L Halpern, Brian P Walenz, Nelson Axelrod, Jiaqi Huang, Ewen F Kirkness, Gennady Denisov, Yuan Lin, Jeffrey R MacDonald, Andy Wing Chun Pang, Mary Shago, Timothy B Stockwell, Alexia Tsiamouri, Vineet Bafna, Vikas Bansal, Saul A Kravitz, Dana A Busam, et al. „The diploid genome sequence of an individual human.“ In: *PLoS Biology* 5.10 (2007), pages 2113–2144. DOI: 10.1371/journal.pbio.0050254 (cited on page 29).

[LPR22] Shuo Li, Jakub Pachocki, and Jakub Radoszewski. „A note on the maximum number of k-powers in a finite word.“ In: *CoRR* abs/2205.10156 (2022). DOI: 10.48550/arXiv.2205.10156. arXiv: 2205.10156 (cited on page 129).

[Lot83] M. Lothaire. *Combinatorics on words.* 1st edition. (2nd edition with minor corrections in 1997). Cambridge University Press, 1983. ISBN: 9780511566097. DOI: 10.1017/CBO9780511566097 (cited on pages 67, 71, 106).

[Lou+19] Felipe A. Louza, Sabrina Mantaci, Giovanni Manzini, Marinella Sciortino, and Guilherme P. Telles. „Inducing the Lyndon array.“ In: *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019).* Segovia, Spain, 2019, pages 138–151. DOI: 10.1007/978-3-030-32686-9_10 (cited on page 68).

[Lou+18] Felipe A. Louza, William F. Smyth, Giovanni Manzini, and Guilherme P. Telles. „Lyndon array construction during Burrows-Wheeler inversion.“ In: *Journal of Discrete Algorithms* 50 (2018), pages 2–9. DOI: 10.1016/J.JDA.2018.08.001 (cited on pages 68, 89).

[Lyn54] Roger C. Lyndon. „On Burnside's problem.“ In: *Transactions of the American Mathematical Society* 77.2 (1954), pages 202–215. DOI: 10.2307/1990868 (cited on pages 16, 67, 69, 70).

[Mae85] Mamoru Maekawa. „A square root N algorithm for mutual exclusion in decentralized systems.“ In: *ACM Transactions on Computer Systems* 3.2 (1985), pages 145–159. DOI: 10.1145/214438.214445 (cited on page 157).

[ML84] Michael G. Main and Richard J. Lorentz. „An $\mathcal{O}(n \log n)$ algorithm for finding all repetitions in a string.“ In: *Journal of Algorithms* 5.3 (1984), pages 422–432. DOI: 10.1016/0196-6774(84)90021-X (cited on pages 1, 5, 16, 129, 130, 148, 150, 173, 177).

[Man75] Glenn Manacher. „A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string.“ In: *Journal of the ACM* 22.3 (1975), pages 346–351. DOI: 10.1145/321892.321896 (cited on pages 16, 17, 86).

[MM93] Udi Manber and Eugene W. Myers. „Suffix arrays: A new method for on-line string searches.“ In: *SIAM Journal on Computing* 22.5 (1993), pages 935–948. DOI: 10.1137/0222058 (cited on pages 1, 46, 53, 67, 68).

[Man+13] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. „Sorting suffixes of a text via its Lyndon factorization.“ In: *Proceedings of the 2013 Prague Stringology Conference (PSC 2013)*. Prague, Czech Republic, 2013, pages 119–127. URL: http://www.stringology.org/event/2013/p11.html (cited on page 68).

[Mat+09] Wataru Matsubara, Kazuhiko Kusano, Hideo Bannai, and Ayumi Shinohara. „A series of run-rich strings.“ In: *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications (LATA 2009)*. Tarragona, Spain, 2009, pages 578–587. DOI: 10.1007/978-3-642-00982-2_49 (cited on pages 129, 144).

[Mat+08] Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara. „New lower bounds for the maximum number of runs in a string.“ In: *Proceedings of the 2008 Prague Stringology Conference (PSC 2008)*. Prague, Czech Republic, 2008, pages 140–145. URL: http://www.stringology.org/event/2008/p13.html (cited on page 129).

[Mat+16] Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. „Generalized pattern matching and periodicity under substring consistent equivalence relations.“ In: *Theoretical Computer Science* 656.Part B (2016), pages 225–233. DOI: 10.1016/j.tcs.2016.02.017 (cited on page 128).

[MS19] Oleg Merkurev and Arseny M. Shur. „Searching runs in streams.“ In: *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019)*. Segovia, Spain, 2019, pages 203–220. DOI: 10.1007/978-3-030-32686-9_15 (cited on page 129).

[MS22] Oleg Merkurev and Arseny M. Shur. „Computing the maximum exponent in a stream.“ In: *Algorithmica* 84.3 (2022), pages 742–756. DOI: 10.1007/s00453-021-00883-y (cited on page 129).

[MST97] Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. „An improved pattern matching algorithm for strings in terms of straight-line programs.“ In: *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997)*. Aarhus, Denmark, 1997, pages 1–11. DOI: 10.1007/3-540-63220-4_45 (cited on page 121).

[Mor06] Christian Worm Mortensen. „Fully dynamic orthogonal range reporting on RAM.“ In: *SIAM Journal on Computing* 35.6 (2006), pages 1494–1525. DOI: 10.1137/S0097539703436722 (cited on page 48).

[MNN20a] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. „Fast compressed self-indexes with deterministic linear-time construction.“ In: *Algorithmica* 82.2 (2020), pages 316–337. DOI: 10.1007/S00453-019-00637-X (cited on page 15).

[MNN20b] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. „Text indexing and searching in sublinear time.“ In: *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*. Copenhagen, Denmark, 2020, 24:1–24:15. DOI: 10.4230/LIPICS.CPM.2020.24 (cited on page 15).

[MNV16] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. „Fast construction of wavelet trees.“ In: *Theoretical Computer Science* 638 (2016), pages 91–97. DOI: 10.1016/J.TCS.2015.11.011 (cited on pages 15, 111).

[MR01] J. Ian Munro and Venkatesh Raman. „Succinct representation of balanced parentheses and static trees.“ In: *SIAM Journal on Computing* 31.3 (2001), pages 762–776. DOI: 10.1137/s0097539799364092 (cited on pages 59, 91, 111).

[Nak+17] Yuto Nakashima, Takuya Takagi, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. „On reverse engineering the Lyndon tree.“ In: *Proceedings of the 2017 Prague Stringology Conference (PSC 2017)*. Prague, Czech Republic, 2017, pages 108–117. URL: http://www.stringology.org/event/2017/p11.html (cited on page 68).

[Nak+19] Yuto Nakashima, Takuya Takagi, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. „On the size of the smallest alphabet for Lyndon trees.“ In: *Theoretical Computer Science* 792 (2019), pages 131–143. DOI: 10.1016/J.TCS.2018.06.044 (cited on page 68).

[Nao91] Moni Naor. „String matching with preprocessing of text and pattern.“ In: *Proceedings of the 18th International Colloquium on Automata, Languages, and Programming (ICALP 1991)*. Madrid, Spain, 1991, pages 739–750. DOI: 10.1007/3-540-54233-7_179 (cited on page 30).

[Nav16] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016. ISBN: 9781316588284. DOI: 10.1017/CBO9781316588284 (cited on page 59).

[NS14] Gonzalo Navarro and Kunihiko Sadakane. „Fully functional static and dynamic succinct trees.“ In: *ACM Transactions on Algorithms* 10.3 (2014), pages 1–39. DOI: 10.1145/2601073 (cited on pages 91, 92).

[Nek09] Yakov Nekrich. „Orthogonal range searching in linear and almost-linear space.“ In: *Computational Geometry* 42.4 (2009), pages 342–351. DOI: 10.1016/j.comgeo.2008.09.001 (cited on page 48).

[Nis+20] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. „Dynamic index and LZ factorization in compressed space.“ In: *Discrete Applied Mathematics* 274 (2020), pages 116–129. DOI: 10.1016/j.dam.2019.01.014 (cited on page 31).

[NT22] Takaaki Nishimoto and Yasuo Tabei. „LZRR: LZ77 parsing with right reference.“ In: *Information and Computation* 285.Part B (2022). DOI: 10.1016/j.ic.2021.104859 (cited on page 31).

[OG11] Enno Ohlebusch and Simon Gog. „Lempel-Ziv factorization revisited.“ In: *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*. Palermo, Italy, 2011, pages 15–26. DOI: 10.1007/978-3-642-21458-5_4 (cited on page 31).

[OS08] Daisuke Okanohara and Kunihiko Sadakane. „An online algorithm for finding the longest previous factors.“ In: *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008)*. Karlsruhe, Germany, 2008, pages 696–707. DOI: 10.1007/978-3-540-87744-8_58 (cited on pages 30, 31).

[OOB22] Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. „On the optimisation of the GSACA suffix array construction algorithm.“ In: *Proceedings of the 29th International Symposium on String Processing and Information Retrieval (SPIRE 2022)*. Concepción, Chile, 2022, pages 99–113. DOI: 10.1007/978-3-031-20643-6_8 (cited on pages 68, 109).

[Pat08] Mihai Patrascu. „Succincter.“ In: *Proceedings of the 49th Annual Symposium on Foundations of Computer Science (FOCS 2008)*. Philadelphia, PA, USA, 2008, pages 305–313. DOI: 10.1109/FOCS.2008.83 (cited on page 92).

[PT14] Mihai Patrascu and Mikkel Thorup. „Dynamic integer sets with optimal rank, select, and predecessor search.“ In: *Proceedings of the 55th Annual Symposium on Foundations of Computer Science (FOCS 2014)*. Philadelphia, PA, USA, 2014, pages 166–175. DOI: 10.1109/FOCS.2014.26 (cited on pages 95, 96).

[PP15] Alberto Policriti and Nicola Prezza. „Fast online Lempel-Ziv factorization in compressed space.“ In: *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*. London, UK, 2015, pages 13–20. DOI: 10.1007/978-3-319-23826-5_2 (cited on page 31).

[PSS08] Simon J. Puglisi, Jamie Simpson, and William F. Smyth. „How many runs can a string contain?“ In: *Theoretical Computer Science* 401.1-3 (2008), pages 165–171. DOI: 10.1016/j.tcs.2008.04.020 (cited on page 129).

[Ras+13] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam Smith. „Sublinear algorithms for approximating string compressibility.“ In: *Algorithmica* 65.3 (2013), pages 685–709. DOI: 10.1007/s00453-012-9618-6 (cited on page 30).

[Ryt03] Wojciech Rytter. „Application of Lempel–Ziv factorization to the approximation of grammar-based compression.“ In: *Theoretical Computer Science* 302.1 (2003), pages 211–222. DOI: 10.1016/S0304-3975(02)00777-6 (cited on page 30).

[Ryt06] Wojciech Rytter. „The number of runs in a string: Improved analysis of the linear upper bound.“ In: *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006)*. Marseille, France, 2006, pages 184–195. DOI: 10.1007/11672142_14 (cited on page 129).

[SN10] Kunihiko Sadakane and Gonzalo Navarro. „Fully-functional succinct trees.“ In: *Proceedings of the 21st Annual Symposium on Discrete Algorithms (SODA 2010)*. Austin, TX, USA, 2010, pages 134–149. DOI: 10.1137/1.9781611973075.13 (cited on page 91).

[SR03] Joe Sawada and Frank Ruskey. „Generating Lyndon brackets. An addendum to: Fast algorithms to generate necklaces, unlabeled necklaces and irreducible polynomials over GF(2).“ In: *Journal of Algorithms* 46.1 (2003), pages 21–26. DOI: 10.1016/S0196-6774(02)00286-9 (cited on page 89).

[SI22] Masaki Shigekuni and Tomohiro I. „Converting RLBWT to LZ77 in smaller space.“ In: *Proceedings of the 2022 Data Compression Conference (DCC 2022).* Snowbird, UT, USA, 2022, pages 242–251. DOI: 10.1109/DCC52660.2022.00032 (cited on page 31).

[Shi58] Anatoly Illarionovich Shirshov. „On free lie rings.“ In: *Matematicheskii Sbornik.* Novaya Seriya 45(87).2 (1958), pages 113–122 (cited on pages 16, 67).

[Shi09] Anatoly Illarionovich Shirshov. „On free lie rings.“ In: *Selected Works of A.I. Shirshov.* 1st edition. (Translated by M.R. Bremner and M.V. Kochetov). Birkhäuser Basel, 2009, pages 77–87. ISBN: 9783764388584. DOI: 10.1007/978-3-7643-8858-4_8 (cited on page 67).

[Shu18] Julian Shun. „Parallel Lempel-Ziv factorization.“ In: *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable.* Association for Computing Machinery and Morgan & Claypool, 2018. Chapter 13. ISBN: 9781970001914. DOI: 10.1145/3018787.3018801 (cited on page 30).

[SZ13] Julian Shun and Fuyao Zhao. „Practical parallel Lempel-Ziv factorization.“ In: *Proceedings of the 2013 Data Compression Conference (DCC 2013).* Snowbird, UT, USA, 2013, pages 123–132. DOI: 10.1109/DCC.2013.20 (cited on page 30).

[SG04] Arseny M. Shur and Yulia V. Gamzova. „Partial words and the interaction property of periods.“ In: *Izvestiya: Mathematics* 68.2 (2004), pages 405–428. DOI: 10.1070/im2004v068n02abeh000480 (cited on page 128).

[SK01] Arseny M. Shur and Yulia V. Konovalova. „On the periods of partial words.“ In: *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001).* Marianske Lazne, Czech Republic, 2001, pages 657–665. DOI: 10.1007/3-540-44683-4_57 (cited on page 128).

[Sim10] Jamie Simpson. „Modified Padovan words and the maximum number of runs in a word.“ In: *Australasian Journal of Combinatorics* 46 (2010), pages 129–146. URL: http://ajc.maths.uq.edu.au/pdf/46/ajc_v46_p129.pdf (cited on page 129).

[Sta12] Tatiana Starikovskaya. „Computing Lempel-Ziv factorization online.“ In: *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS 2012).* Bratislava, Slovakia, 2012, pages 789–799. DOI: 978-3-642-32589-2_68 (cited on page 31).

[SS82] James A. Storer and Thomas G. Szymanski. „Data compression via textual substitution.“ In: *Journal of the ACM* 29.4 (1982), pages 928–951. DOI: 10.1145/322344.322346 (cited on pages 29, 30, 49, 148).

[Sug+21] Ryo Sugahara, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. „Efficiently computing runs on a trie.“ In: *Theoretical Computer Science* 887 (2021), pages 143–151. DOI: 10.1016/j.tcs.2021.07.011 (cited on pages 144, 177).

[Sun+21] Xiuwen Sun, Di Wu, Da Mo, Jie Cui, and Hong Zhong. „Accelerating Knuth-Morris-Pratt string matching over LZ77 compressed text.“ In: *Proceedings of the 2021 Data Compression Conference (DCC 2021)*. Snowbird, UT, USA, 2021, page 372. DOI: 10.1109/DCC50243.2021.00070 (cited on page 31).

[Tak+17] Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. „Packed compact tries: A fast and efficient data structure for online string processing.“ In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 100-A.9 (2017), pages 1785–1793. DOI: 10.1587/TRANSFUN.E100.A.1785 (cited on page 15).

[Thi20] Adrien Thierry. „A proof that a word of length n has less than $1.5n$ distinct squares.“ In: *CoRR* abs/2001.02996 (2020). DOI: 10.48550/arXiv.2001.02996. arXiv: 2001.02996 (cited on page 128).

[Thu06] Axel Thue. „Über unendliche Zeichenreihen.“ In: *Norske Vid. Selsk. Skr., I Mat.–Nat. Kl., Christiania* 7 (1906), pages 1–22 (cited on pages 16, 128).

[TZ03] Rob Tijdeman and Luca Zamboni. „Fine and Wilf words for any periods.“ In: *Indagationes Mathematicae* 14.1 (2003), pages 135–147. DOI: https://doi.org/10.1016/S0019-3577(03)90076-0 (cited on page 128).

[Tre12] Ronald J Trent. „Forensic science and medicine.“ In: *Molecular Medicine*. 4th edition. Elsevier, 2012. Chapter 9, pages 275–299. DOI: 10.1016/b978-0-12-381451-7.00009-8 (cited on page 128).

[Tsu+20] Kazuya Tsuruta, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. „Grammar-compressed self-index with Lyndon words.“ In: *CoRR* abs/2004.05309 (2020). DOI: 10.48550/arXiv.2004.05309. arXiv: 2004.05309 (cited on page 68).

[Udo+20] Nwawuba Stanley Udogadi, Mohammed Khadija Abdullahi, Adams Tajudeen Bukola, Omusi Precious Imose, and Ayevbuomwan Davidson Esewi. „Forensic DNA profiling: Autosomal short tandem repeat as a prominent marker in crime investigation.“ In: *Malaysian Journal of Medical Sciences* 27.4 (2020), pages 22–35. DOI: 10.21315/mjms2020.27.4.3 (cited on page 128).

[Val16] Daniel Valenzuela. „CHICO: A compressed hybrid index for repetitive collections.“ In: *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA 2016)*. St. Petersburg, Russia, 2016, pages 326–338. DOI: 10.1007/978-3-319-38851-9_22 (cited on page 31).

[Ven+01] J. Craig Venter, Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, Robert A. Holt, Jeannine D. Gocayne, Peter Amanatides, Richard M. Ballew, Daniel H. Huson, Jennifer Russo Wortman, Qing Zhang, Chinnappa D. Kodira, Xiangqun H. Zheng, Lin Chen, Marian Skupski, et al. „The sequence of the human genome.“ In: *Science* 291.5507 (2001), pages 1304–1351. DOI: 10.1126/science.1058040 (cited on page 29).

[Vui80] Jean Vuillemin. „A unifying look at data structures.“ In: *Communications of the ACM* 23.4 (1980), pages 229–239. DOI: 10.1145/358841.358852 (cited on pages 67, 171).

[Weba] Website. *100000 Genomes Project*. Accessed: 24 Nov 2023. URL: https://www.genomicsengland.co.uk/initiatives/100000-genomes-project (cited on page 29).

[Webb] Website. *Faster diagnosis from 'transformational' gene project*. Accessed: 24 Nov 2023. URL: https://www.bbc.com/news/health-46456984 (cited on page 29).

[Webc] Website. *Frequently asked questions on CODIS and NDIS*. Accessed: 24 Nov 2023. URL: https://www.fbi.gov/how-we-can-help-you/dna-fingerprint-act-of-2005-expungement-policy/codis-and-ndis-fact-sheet (cited on page 128).

[Webd] Website. *GNU Gzip*. Accessed: 28 Dec 2023. URL: https://www.gnu.org/software/gzip/ (cited on page 51).

[Webe] Website. *The sequence read archive*. Accessed: 24 Nov 2023. URL: https://www.ncbi.nlm.nih.gov/sra (cited on page 29).

[Webf] Website. *The ternary Thue-Morse sequence in the on-line encyclopedia of integer sequences (OEIS)*. Accessed: 08 Jan 2024. URL: https://oeis.org/A036577 (cited on page 153).

[Wei73] Peter Weiner. „Linear pattern matching algorithms.“ In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*. Iowa City, IA, USA, 1973, pages 1–11. DOI: 10.1109/SWAT.1973.13 (cited on page 53).

[Wu21] Cody Yingquan Wu. „Improved LZ77 compression.“ In: *Proceedings of the 2021 Data Compression Conference (DCC 2021)*. Snowbird, UT, USA, 2021, page 377. DOI: 10.1109/DCC50243.2021.00066 (cited on page 31).

[WBM20] Nicole Wyner, Mark Barash, and Dennis McNevin. „Forensic autosomal short tandem repeats and their potential association with phenotype.“ In: *Frontiers in Genetics* 11 (2020). DOI: 10.3389/fgene.2020.00884 (cited on page 128).

[Yam+14] Jun'ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. „Faster compact on-line Lempel-Ziv factorization.“ In: *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Lyon, France, 2014, pages 675–686. DOI: 10.4230/LIPIcs.STACS.2014.675 (cited on page 31).

[ZL77] Jacob Ziv and Abraham Lempel. „A universal algorithm for sequential data compression.“ In: *IEEE Transactions on Information Theory* 23.3 (1977), pages 337–343. DOI: 10.1109/TIT.1977.1055714 (cited on page 30).

[ZL78] Jacob Ziv and Abraham Lempel. „Compression of individual sequences via variable-rate coding.“ In: *IEEE Transactions on Information Theory* 24.5 (1978), pages 530–536. DOI: 10.1109/TIT.1978.1055934 (cited on page 31).